

## Excel Scientific and Engineering Cookbook

By David M. Bourg

.....  
Publisher: O'Reilly

Pub Date: January 2006

Print ISBN-10: 0-596-00879-1

Print ISBN-13: 978-0-59-600879-6

Pages: 442

[Table of Contents](#) | [Index](#)

## Overview

Given the improved analytical capabilities of Excel, scientists and engineers everywhere are using it--instead of FORTRAN--to solve problems. And why not? Excel is installed on millions of computers, features a rich set of built-in analyses tools, and includes an integrated Visual Basic for Applications (VBA) programming language. No wonder it's today's computing tool of choice.

Chances are you already use Excel to perform some fairly routine calculations. Now the *Excel Scientific and Engineering Cookbook* shows you how to leverage Excel to perform more complex calculations, too, calculations that once fell in the domain of specialized tools. It does so by putting a smorgasbord of data analysis techniques right at your fingertips. The book shows how to perform these useful tasks and others:

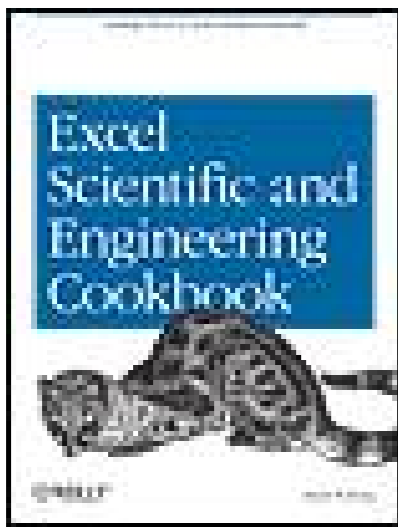
- Use Excel and VBA in general
- Import data from a variety of sources
- Analyze data
- Perform calculations
- Visualize the results for interpretation and presentation
- Use Excel to solve specific science and engineering problems

Wherever possible, the *Excel Scientific and Engineering Cookbook* draws on real-world examples from a range of scientific disciplines such as biology, chemistry, and physics. This way, you'll be better prepared to solve the problems you face in your everyday scientific or engineering tasks.

High on practicality and low on theory, this quick, look-up reference provides instant solutions, or "recipes," to problems both basic and advanced. And like other books in O'Reilly's popular Cookbook format, each recipe also includes a discussion on how and why it works. As a result, you can take

comfort in knowing that complete, practical answers are a mere page-flip away.

[← PREV](#)



## Excel Scientific and Engineering Cookbook

By David M. Bourg

.....  
Publisher: O'Reilly

Pub Date: January 2006

Print ISBN-10: 0-596-00879-1

Print ISBN-13: 978-0-59-600879-6

Pages: 442

[Table of Contents](#) | [Index](#)

[Copyright](#)

[Preface](#)

[Recipe Who Should Read This Book](#)

[Recipe What's in This Book](#)

[Recipe Conventions in This Book](#)

[Recipe Using Code Examples](#)

[Recipe Safari Enabled](#)

[Recipe We'd Like Your Feedback!](#)

[Recipe Acknowledgments](#)

[Chapter 1. Using Excel](#)

[Section 1.0. Introduction](#)

[Recipe 1.1. Navigating the Interface](#)

[Recipe 1.2. Entering Data](#)

[Recipe 1.3. Setting Cell Data Types](#)

[Recipe 1.4. Selecting More Than a Single Cell](#)

[Recipe 1.5. Entering Formulas](#)

[Recipe 1.6. Exploring the R1C1 Cell Reference Style](#)

[Recipe 1.7. Referring to More Than a Single Cell](#)

[Recipe 1.8. Understanding Operator Precedence](#)

[Recipe 1.9. Using Exponents in Formulas](#)

[Recipe 1.10. Exploring Functions](#)

[Recipe 1.11. Formatting Your Spreadsheets](#)

[Recipe 1.12. Defining Custom Format Styles](#)

[Recipe 1.13. Leveraging Copy, Cut, Paste, and Paste Special](#)

[Recipe 1.14. Using Cell Names \(Like Programming Variables\)](#)

[Recipe 1.15. Validating Data](#)

[Recipe 1.16. Taking Advantage of Macros](#)

[Recipe 1.17. Adding Comments and Equation Notes](#)

[Recipe 1.18. Getting Help](#)

[Chapter 2. Getting Acquainted with Visual Basic for Applications](#)

[Section 2.0. Introduction](#)

[Recipe 2.1. Navigating the VBA Editor](#)

- [Recipe 2.2. Writing Functions and Subroutines](#)
- [Recipe 2.3. Working with Data Types](#)
- [Recipe 2.4. Defining Variables](#)
- [Recipe 2.5. Defining Constants](#)
- [Recipe 2.6. Using Arrays](#)
- [Recipe 2.7. Commenting Code](#)
- [Recipe 2.8. Spanning Long Statements over Multiple Lines](#)
- [Recipe 2.9. Using Conditional Statements](#)
- [Recipe 2.10. Using Loops](#)
- [Recipe 2.11. Debugging VBA Code](#)
- [Recipe 2.12. Exploring VBA's Built-in Functions](#)
- [Recipe 2.13. Exploring Excel Objects](#)
- [Recipe 2.14. Creating Your Own Objects in VBA](#)
- [Recipe 2.15. VBA Help](#)
- [Chapter 3. Collecting and Cleaning Up Data](#)
  - [Section 3.0. Introduction](#)
  - [Recipe 3.1. Importing Data from Text Files](#)
  - [Recipe 3.2. Importing Data from Delimited Text Files](#)
  - [Recipe 3.3. Importing Data Using Drag-and-Drop](#)
  - [Recipe 3.4. Importing Data from Access Databases](#)
  - [Recipe 3.5. Importing Data from Web Pages](#)
  - [Recipe 3.6. Parsing Data](#)
  - [Recipe 3.7. Removing Weird Characters from Imported Text](#)
  - [Recipe 3.8. Converting Units](#)
  - [Recipe 3.9. Sorting Data](#)
  - [Recipe 3.10. Filtering Data](#)
  - [Recipe 3.11. Looking Up Values in Tables](#)
  - [Recipe 3.12. Retrieving Data from XML Files](#)
- [Chapter 4. Charting](#)
  - [Section 4.0. Introduction](#)
  - [Recipe 4.1. Creating Simple Charts](#)
  - [Recipe 4.2. Exploring Chart Styles](#)
  - [Recipe 4.3. Formatting Charts](#)
  - [Recipe 4.4. Customizing Chart Axes](#)
  - [Recipe 4.5. Setting Log or Semilog Scales](#)
  - [Recipe 4.6. Using Multiple Axes](#)
  - [Recipe 4.7. Changing the Type of an Existing Chart](#)
  - [Recipe 4.8. Combining Chart Types](#)
  - [Recipe 4.9. Building 3D Surface Plots](#)
  - [Recipe 4.10. Preparing Contour Plots](#)
  - [Recipe 4.11. Annotating Charts](#)
  - [Recipe 4.12. Saving Custom Chart Types](#)
  - [Recipe 4.13. Copying Charts to Word](#)
  - [Recipe 4-14. Displaying Error Bars](#)
- [Chapter 5. Statistical Analysis](#)
  - [Section 5.0. Introduction](#)

- [Recipe 5.1. Computing Summary Statistics](#)
- [Recipe 5.2. Plotting Frequency Distributions](#)
- [Recipe 5.3. Calculating Confidence Intervals](#)
- [Recipe 5.4. Correlating Data](#)
- [Recipe 5.5. Ranking and Percentiles](#)
- [Recipe 5.6. Performing Statistical Tests](#)
- [Recipe 5.7. Conducting ANOVA](#)
- [Recipe 5.8. Generating Random Numbers](#)
- [Recipe 5.9. Sampling Data](#)
- [Chapter 6. Time Series Analysis](#)
  - [Section 6.0. Introduction](#)
  - [Recipe 6.1. Plotting Time Series Data](#)
  - [Recipe 6.2. Adding Trendlines](#)
  - [Recipe 6.3. Computing Moving Averages](#)
  - [Recipe 6.4. Smoothing Data Using Weighted Averages](#)
  - [Recipe 6.5. Centering Data](#)
  - [Recipe 6.6. Detrending a Time Series](#)
  - [Recipe 6.7. Estimating Seasonal Indices](#)
  - [Recipe 6.8. Deseasonalization of a Time Series](#)
  - [Recipe 6.9. Forecasting](#)
  - [Recipe 6.10. Applying Discrete Fourier Transforms](#)
- [Chapter 7. Mathematical Functions](#)
  - [Section 7.0. Introduction](#)
  - [Recipe 7.1. Using Summation Functions](#)
  - [Recipe 7.2. Delving into Division](#)
  - [Recipe 7.3. Mastering Multiplication](#)
  - [Recipe 7.4. Exploring Exponential and Logarithmic Functions](#)
  - [Recipe 7.5. Using Trigonometry Functions](#)
  - [Recipe 7.6. Seeing Signs](#)
  - [Recipe 7.7. Getting to the Root of Things](#)
  - [Recipe 7.8. Rounding and Truncating Numbers](#)
  - [Recipe 7.9. Converting Between Number Systems](#)
  - [Recipe 7.10. Manipulating Matrices](#)
  - [Recipe 7.11. Building Support for Vectors](#)
  - [Recipe 7.12. Using Spreadsheet Functions in VBA Code](#)
  - [Recipe 7.13. Dealing with Complex Numbers](#)
- [Chapter 8. Curve Fitting and Regression](#)
  - [Section 8.0. Introduction](#)
  - [Recipe 8.1. Performing Linear Curve Fitting Using Excel Charts](#)
  - [Recipe 8.2. Constructing Your Own Linear Fit Using Spreadsheet Functions](#)
  - [Recipe 8.3. Using a Single Spreadsheet Function for Linear Curve Fitting](#)
  - [Recipe 8.4. Performing Multiple Linear Regression](#)
  - [Recipe 8.5. Generating Nonlinear Curve Fits Using Excel Charts](#)
  - [Recipe 8.6. Fitting Nonlinear Curves Using Solver](#)
  - [Recipe 8.7. Assessing Goodness of Fit](#)
  - [Recipe 8.8. Computing Confidence Intervals](#)

- [Chapter 9. Solving Equations](#)
  - [Section 9.0. Introduction](#)
  - [Recipe 9.1. Finding Roots Graphically](#)
  - [Recipe 9.2. Solving Nonlinear Equations Iteratively](#)
  - [Recipe 9.3. Automating Tedious Problems with VBA](#)
  - [Recipe 9.4. Solving Linear Systems](#)
  - [Recipe 9.5. Tackling Nonlinear Systems of Equations](#)
  - [Recipe 9.6. Using Classical Methods for Solving Equations](#)
- [Chapter 10. Numerical Integration and Differentiation](#)
  - [Section 10.0. Introduction](#)
  - [Recipe 10.1. Integrating a Definite Integral](#)
  - [Recipe 10.2. Implementing the Trapezoidal Rule in VBA](#)
  - [Recipe 10.3. Computing the Center of an Area Using Numerical Integration](#)
  - [Recipe 10.4. Calculating the Second Moment of an Area](#)
  - [Recipe 10.5. Dealing with Double Integrals](#)
  - [Recipe 10.6. Numerical Differentiation](#)
- [Chapter 11. Solving Ordinary Differential Equations](#)
  - [Section 11.0. Introduction](#)
  - [Recipe 11.1. Solving First-Order Initial Value Problems](#)
  - [Recipe 11.2. Applying the Runge-Kutta Method to Second-Order Initial Value Problems](#)
  - [Recipe 11.3. Tackling Coupled Equations](#)
  - [Recipe 11.4. Shooting Boundary Value Problems](#)
- [Chapter 12. Solving Partial Differential Equations](#)
  - [Section 12.0. Introduction](#)
  - [Recipe 12.1. Leveraging Excel to Directly Solve Finite Difference Equations](#)
  - [Recipe 12.2. Recruiting Solver to Iteratively Solve Finite Difference Equations](#)
  - [Recipe 12.3. Solving Initial Value Problems](#)
  - [Recipe 12.4. Using Excel to Help Solve Problems Formulated Using the Finite Element Method](#)
- [Chapter 13. Performing Optimization Analyses in Excel](#)
  - [Section 13.0. Introduction](#)
  - [Recipe 13.1. Using Excel for Traditional Linear Programming](#)
  - [Recipe 13.2. Exploring Resource Allocation Optimization Problems](#)
  - [Recipe 13.3. Getting More Realistic Results with Integer Constraints](#)
  - [Recipe 13.4. Tackling Troublesome Problems](#)
  - [Recipe 13.5. Optimizing Engineering Design Problems](#)
  - [Recipe 13.6. Understanding Solver Reports](#)
  - [Recipe 13.7. Programming a Genetic Algorithm for Optimization](#)
- [Chapter 14. Introduction to Financial Calculations](#)
  - [Section 14.0. Introduction](#)
  - [Recipe 14.1. Computing Present Value](#)
  - [Recipe 14.2. Calculating Future Value](#)
  - [Recipe 14.3. Figuring Out Required Rate of Return](#)
  - [Recipe 14.4. Doubling Your Money](#)
  - [Recipe 14.5. Determining Monthly Payments](#)
  - [Recipe 14.6. Considering Cash Flow Alternatives](#)
  - [Recipe 14.7. Achieving a Certain Future Value](#)

[Recipe 14.8. Assessing Net Present Worth](#)

[Recipe 14.9. Estimating Rate of Return](#)

[Recipe 14.10. Solving Inverse Problems](#)

[Recipe 14.11. Figuring a Break-Even Point](#)

[About the Author](#)

[Colophon](#)

[Index](#)

[← PREVIOUS](#)



## Excel Scientific and Engineering Cookbook™

by David M. Bourg

Copyright © 2006 O'Reilly Media, Inc. All rights reserved. Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles ([safari.oreilly.com](http://safari.oreilly.com)). For more information, contact our corporate/institutional sales department: (800) 998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

Editor:	Simon St.Laurent
Production Editor:	A. J. Fox
Copyeditor:	Derek Di Matteo
Proofreader:	A. J. Fox
Indexer:	John Bickelhaupt
Cover Designer:	Karen Montgomery
Interior Designer:	David Futato
Illustrators:	Robert Romano, Jessamyn Read, and Lesley Borash

## Printing History:

January 2006:	First Edition.
---------------	----------------

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The *Cookbook* series designations, *Excel Scientific and Engineering Cookbook*, the image of a common genet, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-00879-1

[M]





# Preface

The purpose of this book is to show you how to leverage Excel as a capable computing tool to solve a variety of both routine and complex problems. As a practicing engineer and researcher, I've used Excel extensively for everything from routine data analysis to sophisticated design optimization problems. There's little doubt Excel spreadsheets are quick and easy to use. Excel's convenient spreadsheet interface and functionality combined with its integrated Visual Basic for Applications programming language make it a powerful, versatile computing tool. Its ready availability makes it even more attractive.

It's tough to dismiss Excel as a computing tool when it's very likely sitting right there on your work and home computers. You probably already use Excel to some extent for routine calculations. I'd like to show you how you can leverage Excel for some calculations that are not so routine. Don't worry if you're new to Excel, because I cover the basics as well.

This book is full of real-world uses of Excel, many of which I have personally performed either for my own work or for clients. Throughout this book, I try to use easily recognizable, yet real-world, examples wherever possible. To this end, I draw examples from a variety of scientific disciplines, ranging from biology and chemistry to physics to structural engineering. I hope to show you how to leverage Excel to solve problems you may face in your everyday scientific or engineering tasks.

## Who Should Read This Book

This book is for anyone who has scientific or engineering calculations to perform.

If you are a science or engineering student, then this book is for you. As a student and instructor myself, I know that sometimes you face a lack of courses designed to give you real-world computing experience, leaving you to fend for yourself. Or you are taught how to use computing tools that are available only in special labs on campus. My personal experience suggests that students often gravitate toward the most readily accessible tool, which in many cases is Excel. You probably have Excel on your home computer, your parents' computer, or at your place of work. The trouble you often face, however, is that most resources on Excel don't address the particular types of problems you as a science or engineering student must solve; the vast majority of Excel books deal with financial calculations.

If you are already a veteran scientist or engineer who's used to using programming languages such as Fortran and are looking for an alternative, then this book is for you too. I personally know several professionals who are looking at Excel to replace, to some extent, Fortran and other computing tools such as Mathcad and Mathematica. One of the primary reservations these scientists and engineers have with migrating to Excel is they feel Excel is not powerful enough to solve the sort of problems they deal with. I hope to dispel some of those reservations with this book.

Finally, if you are a scientist or engineer who has already embraced Excel as your primary computing tool but aren't yet harnessing its full computing potential, then this book is also for you. As a professional you probably don't have time to wade through disparate Excel and VBA books to piece together the information you need to solve specific problems. This book provides a convenient, concise guide focused on scientific and engineering problems. The recipe format of this book allows you to quickly find the type of problem you're solving and get the answer without having to read the entire book sequentially. Moreover, I provide many cross-references throughout the book to point you toward relevant background material.

## What's in This Book

This book aims to present Excel as a "cradle to grave" computing environment for scientific and engineering computations. What I mean by this is that I want to show how to use Excel and VBA in general, how to import data from a variety of sources, analyze data, perform calculations, and finally visualize the results for interpretation and presentation. Further, since I'm assuming you have limited time for learning a computing language, I show you in a very concise way how to immediately use Excel to solve science and engineering problems. In order to demonstrate Excel's power and versatility, I show you how to solve some relatively complicated problems that are not immediately recognizable as suitable problems for Excel.

This book consists of 14 chapters. Here is a brief overview of each chapter:

### [Chapter 1, \*Using Excel\*](#)

Is meant for those new to Excel. It covers a variety of fundamental topics, including navigating Excel's interface, entering data and formulas, and formatting cells. Experienced Excel users can safely skip this chapter.

### [Chapter 2, \*Getting Acquainted with Visual Basic for Applications\*](#)

Is also meant for those new to Excel's Visual Basic for Applications (VBA) features. This chapter covers a variety of fundamental VBA programming topics, including VBA syntax, using the VBA editor, and integrating VBA code with Excel spreadsheets. Experienced VBA users can safely skip this chapter.

### [Chapter 3, \*Collecting and Cleaning Up Data\*](#)

Shows you how to import data into Excel from a variety of sources, including text files, databases, and the Web.

### [Chapter 4, \*Charting\*](#)

Shows you how to visualize data in Excel. This chapter shows a variety of example charts, including line, bar, 3D surface, and combination charts.

### [Chapter 5, \*Statistical Analysis\*](#)

Shows you how to perform standard statistical calculations in Excel, such as computing summary statistics, calculating confidence intervals, and performing analysis of variance, among other calculations.

### [Chapter 6](#), *Time Series Analysis*

Is an extension of the data analysis topics covered in [Chapter 5](#), but specifically focusing on conducting various time series analysis tasks, including forecasting and Fourier transforms in Excel.

### [Chapter 7](#), *Mathematical Functions*

Introduces you to many of Excel's useful math functions.

### [Chapter 8](#), *Curve Fitting and Regression*

Continues discussion of data analysis, specifically focusing on curve fitting and regression analysis.

### [Chapter 9](#), *Solving Equations*

Shows you how to leverage Excel to solve linear and nonlinear equations, as well as linear and nonlinear systems.

### [Chapter 10](#), *Numerical Integration and Differentiation*

Shows you how to perform various numerical integration calculations, such as computing areas and moments of areas, among others. This chapter also shows you how to perform numerical differentiation using standard difference formulas and how to improve the results.

### [Chapter 11](#), *Solving Ordinary Differential Equations*

Shows you how to solve initial value and boundary value problems using standard numerical techniques such as the Runge-Kutta and shooting methods.

### [Chapter 12](#), *Solving Partial Differential Equations*

Shows you how to leverage Excel to help solve problems involving partial differential equations

### [Chapter 13](#), *Performing Optimization Analyses in Excel*

Shows you how to perform optimization analysis using Excel's built-in optimization tools. This chapter also shows you how to implement a not so standard method a genetic algorithm using VBA.

### [Chapter 14](#), *Introduction to Financial Calculations*

Introduces you to some of Excel's extensive financial analysis capabilities.



## Conventions in This Book

The following typographical conventions are used in this book:

### `Constant width`

Indicates VBA and Excel functions; spreadsheet formulas and cell data; user-assigned names for cells, rows, and columns; and other literals. Also used for code examples and for text that you type in literally.

### *Constant width italic*

Indicates placeholders (for which you substitute an actual name) in examples, cell formulas, and Excel function calls.

### **Constant width bold**

Used to highlight segments of code.

### *Italic*

Used for URLs, file extensions, filenames, and directory names. Also used for emphasis and to introduce new terms.

Indicates a tip, suggestion, or general note.



## Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book doesn't require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Excel Scientific and Engineering Cookbook* by David M. Bourg. Copyright 2006 O'Reilly Media, Inc., 0-596-00879-1."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).



← PREY

## Safari Enabled



When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answer: when you need the most accurate, current information. Try it for free at <http://safari.oreilly.com>.

← PREY

 PREVIOUS

## We'd Like Your Feedback!

The information in this book has been tested and verified to the best of our ability, but mistakes and oversights do occur. Please let us know about errors you may find, as well as your suggestions for future editions, by writing to:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-998-9938 (in the U.S. or Canada)  
707-829-0515 (international or local)  
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

<http://www.oreilly.com/excelseckbk>

To ask technical questions or comment on the book, send email to:

[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

<http://www.oreilly.com>

← PREV

## Acknowledgments

Thanks to O'Reilly for taking on this project with me. This is my third book project with O'Reilly, and can say the staff at O'Reilly are true professionals and a joy to work with. Special thanks go to my editor, Simon St. Laurent, for his skillful editing and flexibility in allowing me to pursue my vision for this book.

I want to extend thanks to the respectable Dr. Dongming Wei at the University of New Orleans for his thoughtful technical review. And thanks to my friend and Excel power user extraordinaire Kenneth Humphreys for his meticulous review and suggestions.

Finally, I want to dedicate this book to my lovely wife, Helena, and beautiful daughter, Natalia, and thank them for their unwavering support.

← PREV

# Chapter 1. Using Excel

[Section 1.0. Introduction](#)

[Recipe 1.1. Navigating the Interface](#)

[Recipe 1.2. Entering Data](#)

[Recipe 1.3. Setting Cell Data Types](#)

[Recipe 1.4. Selecting More Than a Single Cell](#)

[Recipe 1.5. Entering Formulas](#)

[Recipe 1.6. Exploring the R1C1 Cell Reference Style](#)

[Recipe 1.7. Referring to More Than a Single Cell](#)

[Recipe 1.8. Understanding Operator Precedence](#)

[Recipe 1.9. Using Exponents in Formulas](#)

[Recipe 1.10. Exploring Functions](#)

[Recipe 1.11. Formatting Your Spreadsheets](#)

[Recipe 1.12. Defining Custom Format Styles](#)

[Recipe 1.13. Leveraging Copy, Cut, Paste, and Paste Special](#)

[Recipe 1.14. Using Cell Names \(Like Programming Variables\)](#)

[Recipe 1.15. Validating Data](#)

[Recipe 1.16. Taking Advantage of Macros](#)

[Recipe 1.17. Adding Comments and Equation Notes](#)

[Recipe 1.18. Getting Help](#)

[← PREV](#)

## 1.0. Introduction

This chapter is necessarily basic and is intended for readers new to Excel or with little Excel experience. Experienced users may skip this chapter without loss of continuity. The aim of this chapter is to get you acquainted with Excel's interface and the main features and capabilities that we'll use throughout the remainder of this book. I'm also going to show you several techniques that will help you create presentable spreadsheets and allow you to work efficiently in Excel. The techniques and features summarized in this chapter will be put to use in the recipes throughout the remainder of this book.

[← PREV](#)

## Recipe 1.1. Navigating the Interface

### Problem

You've never used Excel before and don't know where to start.

### Solution

Start Excel and begin poking around the interface to become familiar with it.

### Discussion

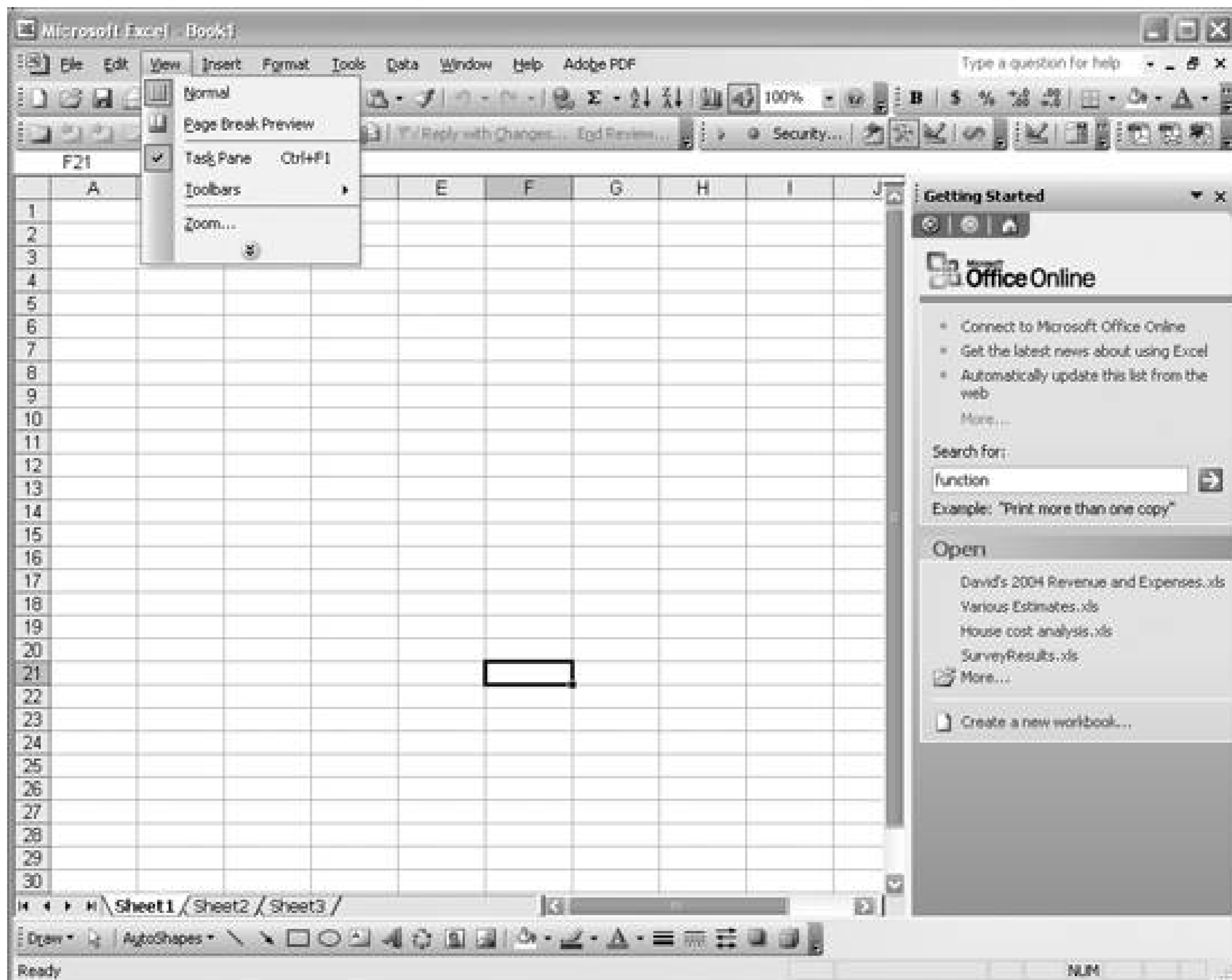
This solution sounds simple and it is. I believe there's no better way to learn something on the computer than to just start exploring and experimenting. Don't worry about breaking anything. The worst you can do is create a nonsensical spreadsheet in which case, you can exit Excel without saving anything and start over fresh. I have a four-year-old daughter who loves playing around in Excel because of the paperclip office assistant. If she can't ruin anything with all her random clicking and typing, then I think you're in good shape to do some purposeful exploring. That said, I won't just throw you to the wolves, but will point you in the right direction to begin your journey.

Excel is just like any other standard Windows application: it consists of a multiple-document interface main window containing a main menu bar, toolbars, and a status bar.

*A multiple-document interface main window is just a window that acts as a container for several of the same type of files, so to speak. For example, in Excel you can have more than one spreadsheet open at once and switch between them. Word is another example; you can have many different Word documents open at once and switch between them as desired. By contrast, an application based on a single-document interface window is one that allows you to open up and work on only one file at a time. Many specialized programs are based on this paradigm.*

[Figure 1-1](#) shows a screenshot of Excel as installed on my computer. You'll notice that it looks like any other Windows application, with one distinct difference.

Figure 1-1. Excel's main window



In addition to the standard menu bar, toolbars, and status bar, the window contains a large grid of *cells*. These cells are arranged in an organized matrix, with letters identifying columns and numbers identifying rows. This is the *spreadsheet*. Notice that there are a few tabs toward the bottom of the spreadsheet labeled Sheet 1, Sheet 2, and Sheet 3. By default, Excel starts with a blank *workbook*. A workbook contains multiple sheets, which can be linked together. You can add or remove spreadsheets from a workbook, rename the sheets, put data and formulas on one sheet and refer to them from another sheet, and so on. We'll explore the details of working with spreadsheets and workbooks throughout the remainder of this book. For now, let's continue getting acquainted with the window itself.

## Main menu bar

The main menu bar gives you access to all of Excel's features and functionality. Excel's main menu bar contains the following menus:

### *File*

Access the usual file operations such as open, save, print, and exit.

### *Edit*

Access the standard cut, copy, paste, find, replace, and other operations.

### *View*

Customize Excel's appearance by changing options such as page layout, zoom, toolbars, status bar, and task pane.

### *Insert*

Add various spreadsheet elements such as cells, rows, and columns, as well as other objects such as notes, comments, and pictures.

### *Format*

Customize the formatting of text, cells, rows, columns, and other aspects of your spreadsheets

### *Tools*

Access various useful tools including spelling tools, change tracking, formula auditing, and a host of useful analysis tools such as goal seek, solver, and data analysis tools. The tools menu also provides access to the Customize and Options windows, allowing you to tailor Excel to your specific needs and preferences.

### *Data*

Access various data management tools such as sorting, validation, and tables.

### *Window*

Manage Excel's windows by using features such as arranging windows and splitting panes.

### *Help*

Access Excel's built-in help system.

These nine menus comprise the standard menus built into Excel. I encourage you to click through these menus in your version of Excel to familiarize yourself with them a little. We're going to use these menus a great deal throughout the remainder of this book.

Take another look at [Figure 1-1](#) and notice that my version of Excel has one more menu option in addition to the nine just summarized. In my case, I have Adobe PDF converter installed, so a custom menu option for it appears in Excel's main menu bar after the standard menus. (Such custom menus will not appear on your version of Excel unless you've installed third-party software that adds them.)



I should mention that you can also add your own menus using techniques that we'll discuss in other recipes throughout this book.

As with most standard Windows programs, you can access the menu using the mouse or the keyboard. Accessing the menu with the mouse is, of course, as simple as pointing and clicking. Notice in [Figure 1-1](#) that the View menu is currently visible. Also notice the little down arrow icon at the bottom of the menu. Whenever you see this icon on a menu, it means there are more menu items available to you, but they are hidden. By default, MS Office applications hide less frequently used menu items from view, showing only your most frequently used items in an effort to simplify the menus. If you click the down arrow icon, the full menu will appear. I personally don't like this feature and prefer to see the menus in their entirety all the time.

You can change this behavior by going to the Tools → Customize... menu and clicking the Options tab (see [Figure 1-2](#)).

Figure 1-2. Customize window



On the Options tab you'll see a checkbox labeled "Always show full menus." If you check that checkbox, full menus will be shown all the time. Otherwise, you'll get the abbreviated menus, in which case you can select an option to show the full menus after a short delay. Press the Close button to save your changes and get rid of this window.

There are two ways to access menu items with the keyboard. The first way involves using the Alt key on your keyboard to set input focus on the main menu bar. In Excel, or any other Windows application for that matter, press the Alt key and you will notice that the first menu on the main

menu bar gets highlighted. This means the menu has focus. Also notice that each menu File, Edit, View, and so on has one letter underlined. The underlined characters are called *accelerator* characters. After pressing the Alt key and giving focus to the main menu bar, you can press a key corresponding to one of those accelerator characters to open the corresponding menu. For example, the key sequence Alt-V will open the View menu, as shown in [Figure 1-1](#).

You'll notice each menu item in the View menu also contains an accelerator character. If you now press the key corresponding to one of these accelerators, you'll activate the corresponding menu item. For example, the key sequence Alt-V-Z will activate the Zoom menu item, opening a new window that allows you to customize the zoom level of your spreadsheet. The zoom feature is helpful for those of us whose eyesight needs a little help.

The second way to access menu items from the keyboard involves the use of shortcut keys. Take another look at the View menu shown in [Figure 1-1](#). Notice the Task Pane menu item has a key combination denoted to the right. In this case the key combination is Ctrl-F1. This is the shortcut key combination to activate the Task Pane menu item. (This applies to Excel 2003. Earlier versions may not have this shortcut.) So, if you press and hold the Ctrl key and then press the F1 key (while still holding the Ctrl key down), you'll activate the Task Pane menu item. When you execute this key combination, the task pane on the right side of the window will disappear. If you execute this shortcut again, the task pane will reappear. (We'll come back to the task pane in a moment.)

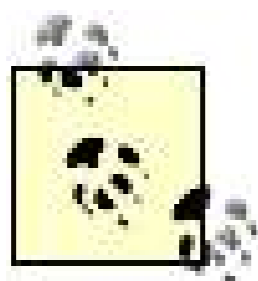
Both of these keyboard techniques are designed to speed up your productivity by providing quick access to Excel's functionality: you don't have to keep taking your hand off the keyboard to navigate with the mouse. These quick access features are not unique to Excel and, in fact, are standard Windows interface features. Most well-designed Windows applications provide one or both of these features.

## Task pane

I've already mentioned the task pane in the context of accessing menus using shortcuts (i.e., toggling the task pane's visibility by using Ctrl-F1); however, I haven't mentioned what the task pane actually does. The task pane provides another portal to features included in Excel, giving you access to Excel's built-in and online help systems, a searchable dictionary and other reference material, and various spreadsheet templates. With Excel open as shown in [Figure 1-1](#), click the little down arrow icon located to the right of the Getting Started title on the task pane to view the various tools available to you.

The one I find most useful is the Help tool. The Help tool is easy to use and provides access to both built-in and online sources of documentation on Excel. If you click the Help option, you'll be presented with a text box at the top of the pane that allows you to type in search terms for topics.

For example, if you type in `keyboard shortcuts`, the task pane will switch to the Search Results view, displaying a list of topics related to your search terms. In my case, the topic Keyboard Shortcuts is at the top of the list. Find this in your version and click it. Upon doing so, you'll be presented with a new help window showing a list of topics specifically related to keyboard shortcuts.



Excel has keyboard shortcuts for many features in addition to the menu shortcuts we discussed earlier. The help documentation describes all of those available to you and is a convenient reference for learning all sorts of useful shortcuts to help you work more efficiently. I encourage you to explore this documentation.

While the task pane does provide convenient access to some useful features, I find it sometimes gets in the way of my spreadsheet. So, when I'm not using it, I Ctrl-F1 it away or reduce its width by clicking and dragging the task pane's left edge over toward the right, making more space for the spreadsheet.

## Toolbars

Referring back to [Figure 1-1](#), or your version of Excel if you have it open now, you'll notice that the Excel window contains many toolbars and tool buttons. These are standard, dockable toolbars that provide yet another means to access Excel's features. Many of the toolbars duplicate menu functionality, giving you another way of selecting menus; other toolbars provide access to features buried in various dialog boxes or windows that are themselves accessible from the menu or other toolbars. These toolbars are provided for convenience and it's up to you to develop your own preferences, i.e., whether you prefer navigating the menu, using shortcut keys, or clicking tool buttons. I do all three.

You may have noticed that the toolbars in my version of Excel, shown in [Figure 1-1](#), are different from those shown in your version of Excel. Certain toolbars that I use frequently are visible, while others are hidden. You can customize your toolbars to your liking. To do so, open the View menu and select the Toolbars submenu. This will display a list of toolbars available to you for display. The ones shown with checkmarks to their left are visible. You may select whichever ones you'd like to show or hide.

You can also customize the toolbars by using the Customize window shown in [Figure 1-2](#). To activate this window, select Tools → Customize.... Select the Toolbars tab to display a list of available toolbars that you can show or hide.

The Commands tab in the Customize window allows you to further customize your toolbars. Using the Commands tool, you can select specific commands and then drag-and-drop a button corresponding to each command onto Excel's toolbar area. This allows you to tailor your toolbars to contain only those command buttons you use most frequently, without cluttering your window with rarely used buttons.

Toolbars in Excel are *dockable*. This means you can drag them around to rearrange them. To do so, click (and hold) the four dots located to the left of a toolbar and drag it around. You can rearrange all your toolbars in this manner, placing some at the top of the window (below the main menu bar), some toward the bottom of the window (above the status bar), and some to the left or right of the window.

## See Also

The aim of this recipe is not to provide a treatise on Excel's interface, but to acquaint you with its layout so we can explore Excel further throughout this book. As I said earlier, I think the best way to become familiar and comfortable with Excel is to explore it directly. I've shown you how to access Excel's online help and documentation already, and they are a good source for additional reading material on Excel.

If you're interested in browsing Excel's documentation rather than searching it, then click the "Table of Contents" link just below the search term entry box on the Excel Help task pane (see [Figure 1-3](#)).



← PREV

## Recipe 1.2. Entering Data

### Problem

You've familiarized yourself with Excel's layout and now you're ready to enter data into a spreadsheet but don't know how.

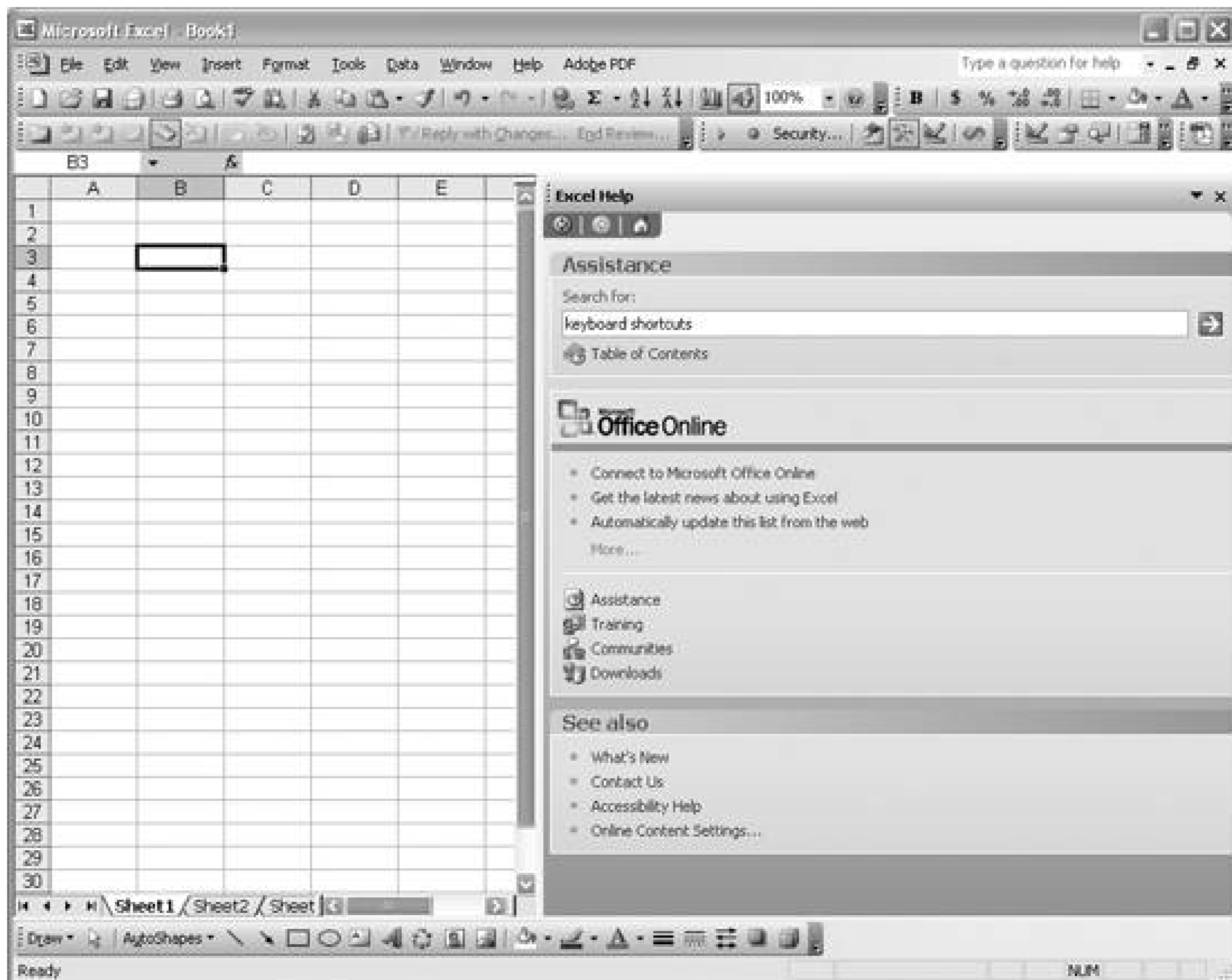
### Solution

Select a cell in the spreadsheet and start typing. Press Enter when done.

### Discussion

When you select a cell in a spreadsheet, it becomes highlighted with a thick black border, as shown in [Figure 1-3](#). In that figure the cell in row 3 and column B is the selected cell. Notice the row and column headings of the currently selected cell are highlighted. Once a cell is selected in this manner, you can simply start typing on the keyboard to insert text in the cell. Press Enter when you are finished and notice that the cell below the one within which you entered text is now automatically selected. This allows you to type and enter text in a column of cells rapidly, without having to select the next cell using the mouse.

Figure 1-3. Excel Help task pane

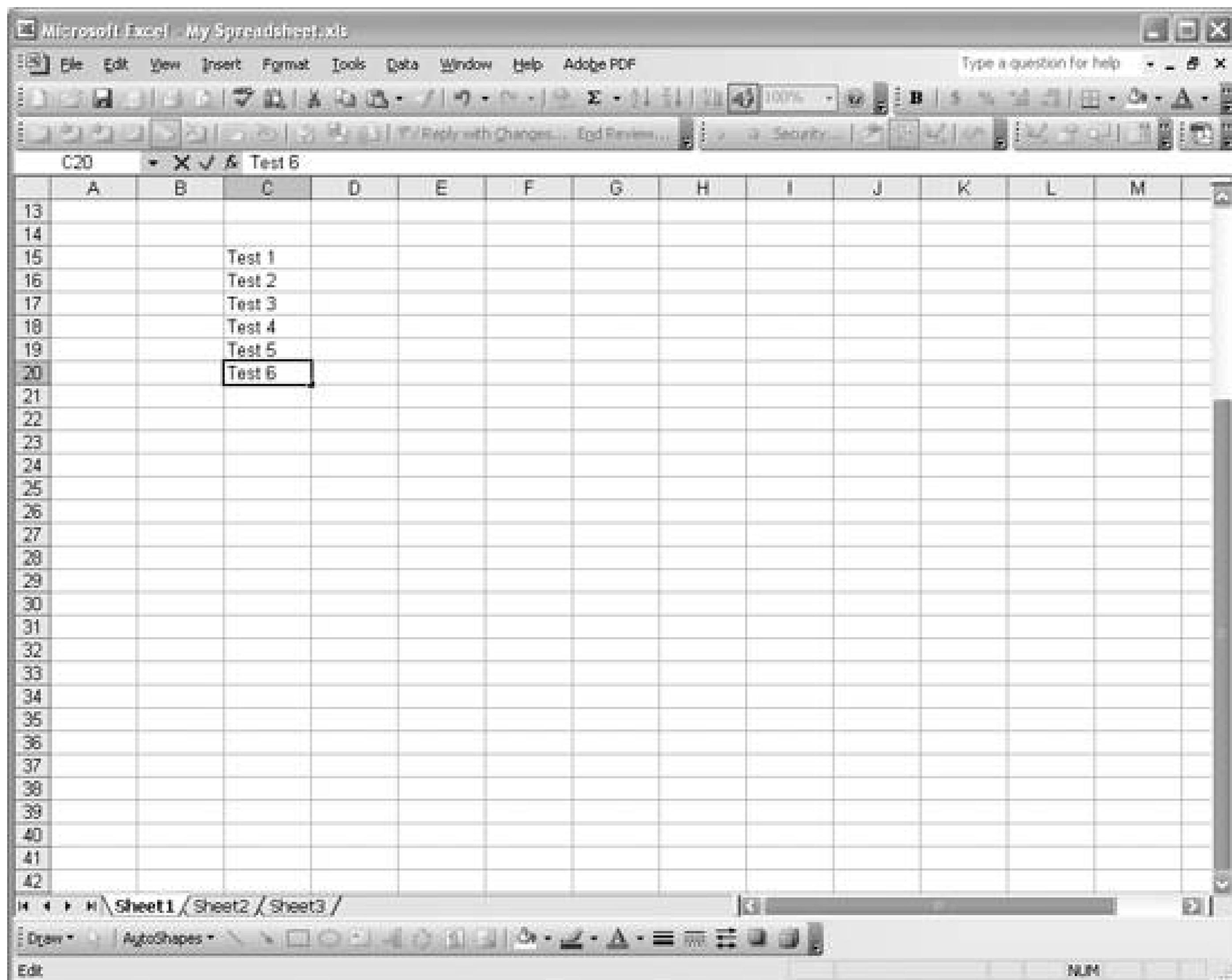


Upon entering text, you can also press the Tab key to commit your entry and move the cell selection to the next cell to the right. Alternatively, you can use the arrow keys on your keyboard to commit an entry and move to any cell adjacent to the cell within which you've entered text. Of course you can always use the mouse to click on a cell, selecting it for input; however, doing this may slow you down if you are trying to enter text in a contiguous group of cells, since you'll have to remove your hands from the keyboard very often.

You can also enter text into a cell using the formula bar (located just above the spreadsheet grid and just below the toolbars), as shown in [Figure 1-4](#). The formula bar has an  $\times$  icon, adjacent to a long white rectangular area.

Click anywhere in the white rectangular area of the formula bar to give it the input focus and then start typing. Your text will appear in both the formula bar and the currently selected cell. To commit the entry, press the Enter key or press the green checkmark icon on the formula bar. To cancel your entry, press the red x icon or press the Esc key on your keyboard.

Figure 1-4. Using the formula bar



Let's say you want to make changes to some text already entered in a cell. To make changes to a cell entry, select the cell with the mouse or use the arrow keys to navigate the grid and then press the F2 shortcut key to put the cell in edit mode, allowing you to edit the contents of the cell. Alternatively, double-clicking on a cell will automatically select it and put it in edit mode. Finally, you can always select a cell and simply start typing and press Enter to completely overwrite the contents of the cell.

[← PREVIOUS](#)

## Recipe 1.3. Setting Cell Data Types

### Problem

You've learned how to enter text from the previous recipe, but now you want to enter data other than text (e.g., numbers, dates, and currency).

### Solution

Enter the data just as you would text and let Excel automatically figure out its type, or use the Format → Cells... menu to open a dialog box allowing you to manually format the data type for a cell.

### Discussion

In the previous recipe, I had you simply enter text in cells. Whether you entered a word or a number Excel automatically figured out what type of data you entered. In general, input starting with letters is automatically interpreted as text, and input starting with numbers is automatically interpreted as numeric. There are some special cases worth noting here. Preceding any string of characters with the ' symbol forces Excel to interpret the data as text. Preceding a string of numbers with the \$ symbol forces Excel to interpret the data as currency. Using E (or e) while entering a number in scientific notation forces Excel to interpret the string as a number in scientific notation. Entering numbers with dashes between them will cause Excel to interpret the number as a date. I encourage you to try entering various types of data like those I describe here to see how Excel handles the data. In some cases you'll notice that Excel will reformat your data a little. For example, if you type `1.2345e3` in a cell, it will appear as `1.23E+3` in the cell and `1234.56` in the formula bar (when the cell is selected).

In general, Excel is pretty smart about interpreting the data type you intend; however, sometimes it does need a little help. Also, sometimes you may want to change the format of the data to give it an appearance other than the default appearance assigned to it by Excel. In these cases you can manually specify the type and format of data contained in cells by accessing the Format Cells dialog box. You can do so by selecting Format → Cells... from the menu or by using the shortcut key combination, Ctrl-1.

[Figure 1-5](#) shows the Format Cells dialog box. Notice here I had a cell selected that contained the entry `1.2345e3`, which appears in the Sample area of the dialog box. This allows you to preview the results while making format changes.

I find that one of the most common uses of formatting tasks is specifying the number of decimal places to show for numbers. You can specify the number of decimal places to show by entering a



value in the "Decimal places" field. Changing this setting is a format change only; it does not change the data itself. For example, if you enter `123.45678` in a cell and set the number of decimal places to 3, the cell will show `123.457` but the actual number is still stored as `123.45678`, as can be seen in the formula bar when the cell is selected.

You can also select one of the categories shown in the Format Cells dialog box to set the type of data. If you click each category in the list, a short description and other formatting options will appear in the Format Cells dialog box. For example, if you select the Number category, the Format Cells dialog box will appear (see [Figure 1-6](#)).

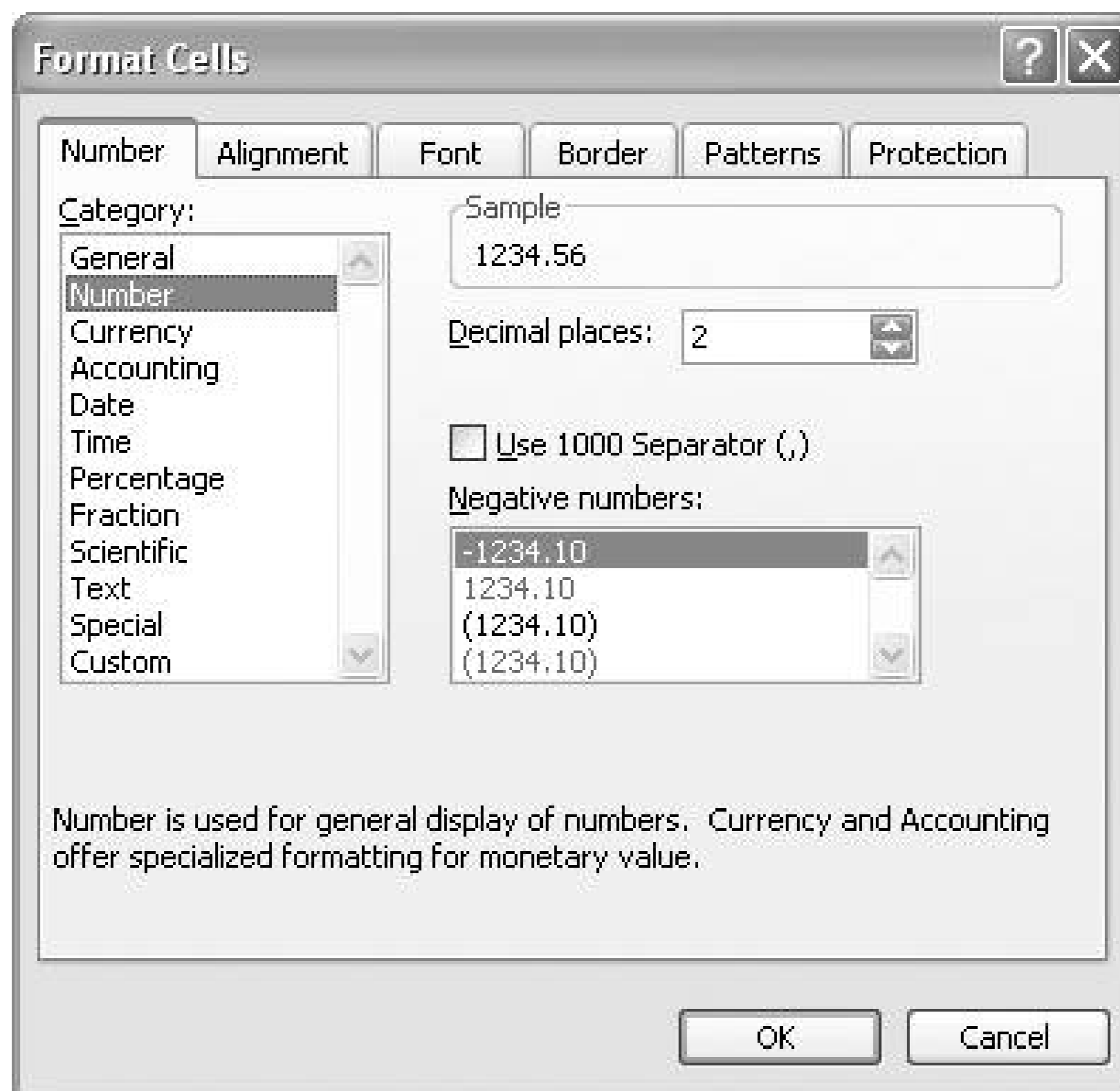
The new formatting options appear on the right side of the dialog box, just under the "Decimal places" setting.

Format changes affect the currently selected cell and remain in effect until you change the format. If you set a cell to display Currency and then enter a new value in that cell, it will also be interpreted as currency.

Figure 1-5. Format cells dialog box

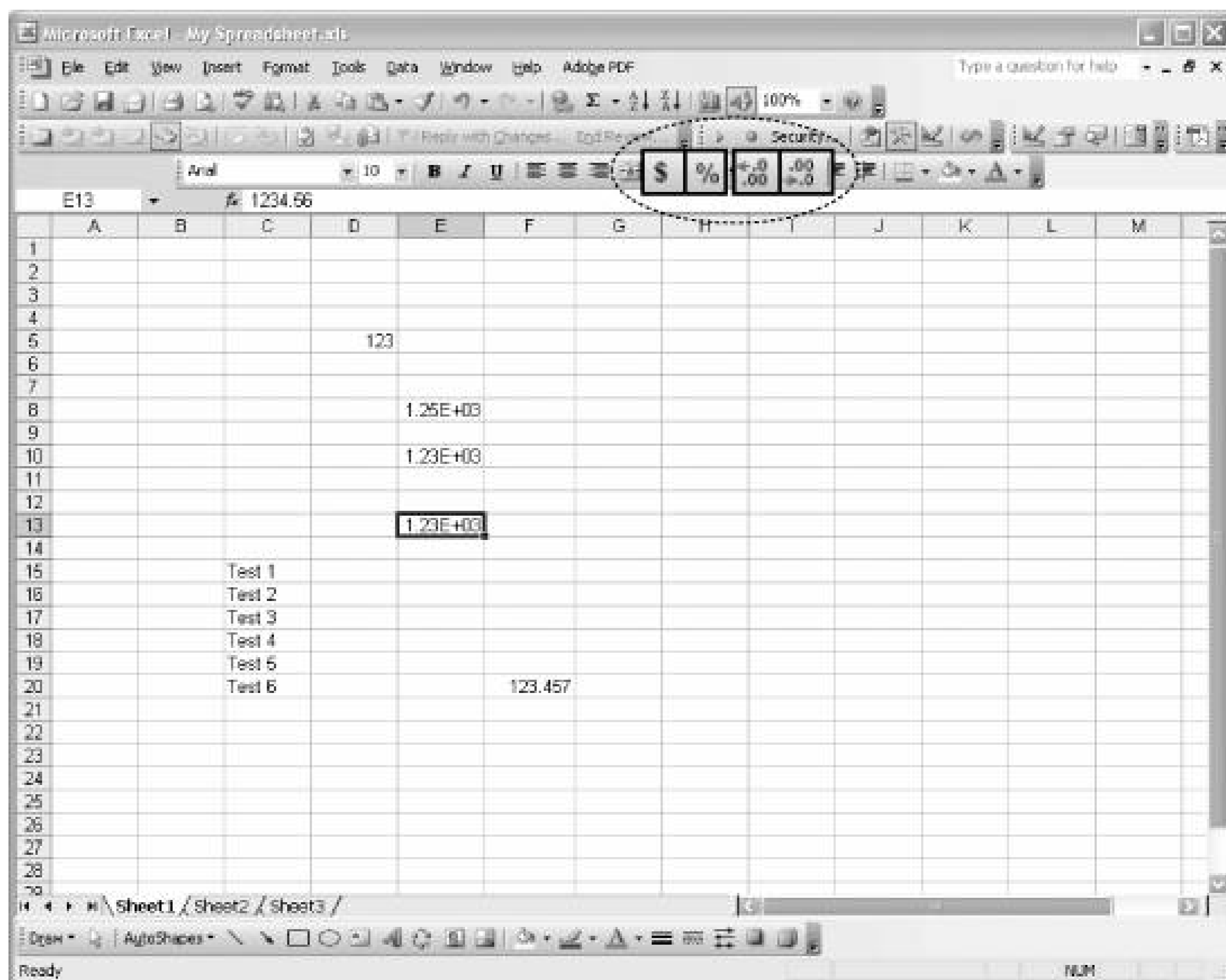


Figure 1-6. Format Cells with the Number category selected



You need not always use the Format Cells dialog box to change cell formatting. Some of these formatting operations have been assigned to toolbar buttons for convenience. Take a look at [Figure 1-7](#) and notice the toolbar buttons that I highlighted. These specific buttons allow you to change some cell format properties with the click of a button.

Figure 1-7. Cell format toolbar



The \$ button changes the cell format to currency while the % button changes the cell format to percentage; e.g., if you enter `0.38`, it will show up as `38%`. The two other highlighted buttons respectively increase and decrease the number of decimal places to show.

I should also point out that formatting changes take effect for all of the cells currently selected. So far, I've only discussed selecting a single cell; however, there are times when you may want to make changes to multiple cells all at one time. The next recipe explains how to select more than one cell at a time.

## See Also

I find the Number, Percentage, and Scientific format categories more than adequate for most scientific and engineering computing tasks; however, there may be occasions when you want to set your own specific data format for a unique application. In this case, you can actually specify your own format template in the Format Cells dialog box by setting up format codes using the Custom format category. I won't go into the details here since it's fairly well documented in Excel's online help. To learn more, open the Excel Help task pane and do a search using the key phrase "Custom Formatting." In your search results, look for the topics entitled "Create or delete a custom number format" and "Number format code."



## Recipe 1.4. Selecting More Than a Single Cell

### Problem

You want to select more than a single cell at one time; for example, so you can set the format of a group of cells at once rather than individually.

### Solution

The easiest way to select a contiguous group of cells is to click and drag with the mouse. Specifically, press the left mouse button to select a cell at one corner of the group and then hold down the mouse button while dragging to an opposite corner of the group of cells. The whole block of cells will be selected. You can click and drag to select a column of cells, a row of cells, or a block of cells, as described here.

### Discussion

As with most things in Excel, there are number of alternative methods for selecting a group of cells. Clicking and dragging with the mouse is probably the most common way, but you can also perform the same operation using the keyboard. With a cell selected, hold down the Shift key and press one of the arrow keys to select a range of cells.

You can also use the Shift key in conjunction with the mouse. For example, select a cell and then hold down the Shift key and select another cell. This selects the range of cells between the two corner cells.

If you want to select a group of cells that are not contiguous, you can do so by clicking each desired cell while holding down the Ctrl key.

To select all of the cells in the spreadsheet, click the small rectangle in the grid heading bar just above the first row heading and to the left of the first column heading. This is called the Select All button (the Select All shortcut is Ctrl-A).

To select an entire row, click the row heading. Likewise, to select an entire column, click the column heading. You can use the Shift and Control keys to select groups of rows or columns from their headings, analogously to how you select groups of cells.

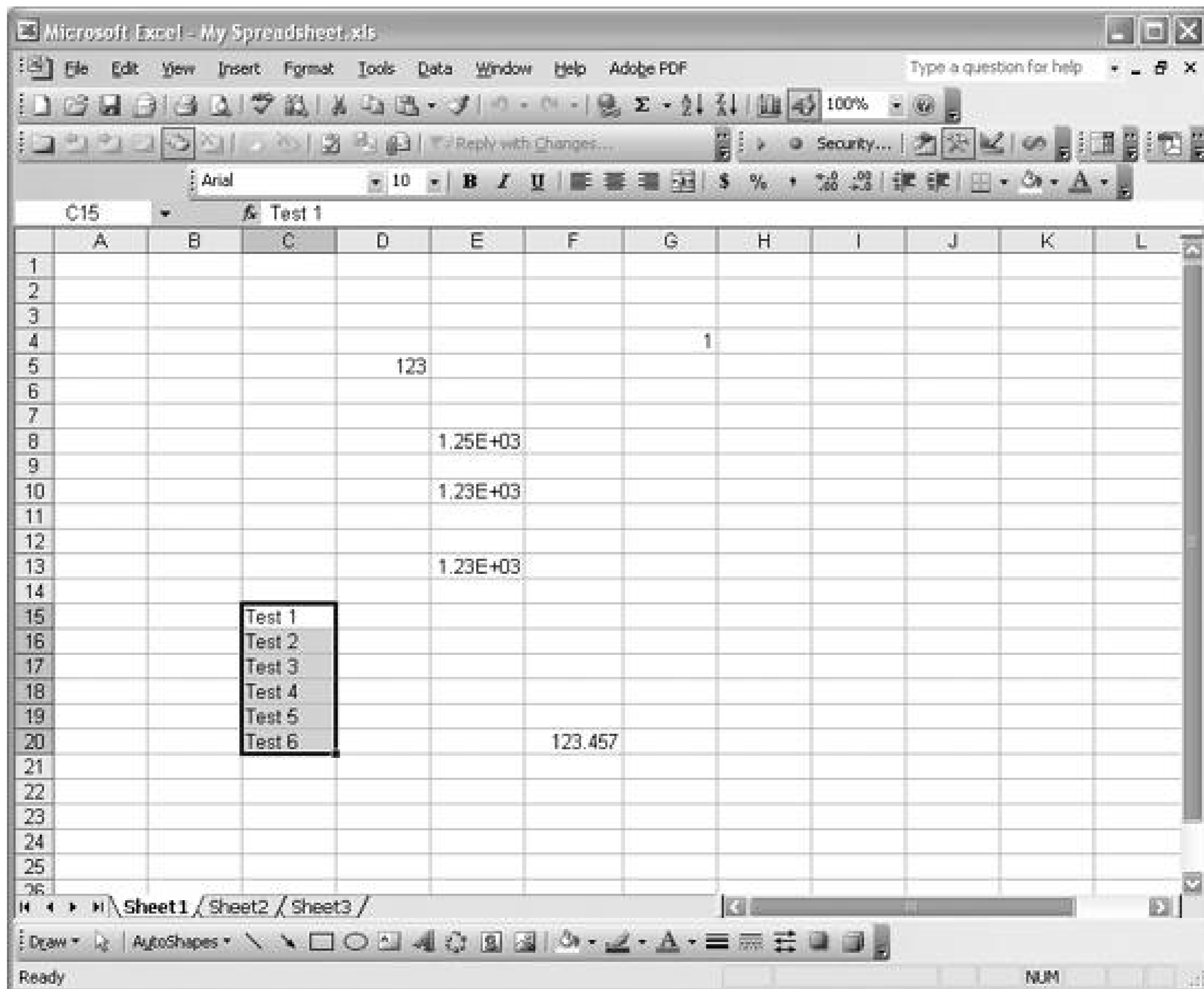
When a group of cells is selected, all of the selected cells are highlighted as shown in [Figure 1-8](#).

The selection methods described here are fairly common selection tasks that I use all the time. In some cases, even more control over selection is required. In this case, Excel has a feature that allow:

you to select cells based on some specific criterion (for example, cells that contain text or cells that contain numbers).

Press Ctrl-G or select Edit → Go To... from the main menu bar to open up the Go To dialog box. Then press the button labeled Special to open the Go To Special dialog box, as shown in [Figure 1-9](#).

Figure 1-8. Group of selected cells



The Go To Special dialog box allows you to select or go to cells that correspond to the selected criterion. For the text selected in [Figure 1-8](#), I selected the Constants criterion, which enabled the checkboxes below Formulas, at which point I made sure only Text was checked. You can do the same with numbers; for example, this is a convenient way to select all cells that contain only numbers, so that in a single shot you can change the number of decimal places shown for all numeric cells.

## See Also

To learn more about selecting cells, do a search in Excel Help using the phrase "selecting cells." In

your search results, click the topic "Select data or cells in a worksheet."

To learn more about using the Go To and Go To Special feature, do a search in Excel Help using the phrase "Go To." In your search results, click the topic entitled "Use the Go To command to find special cells."



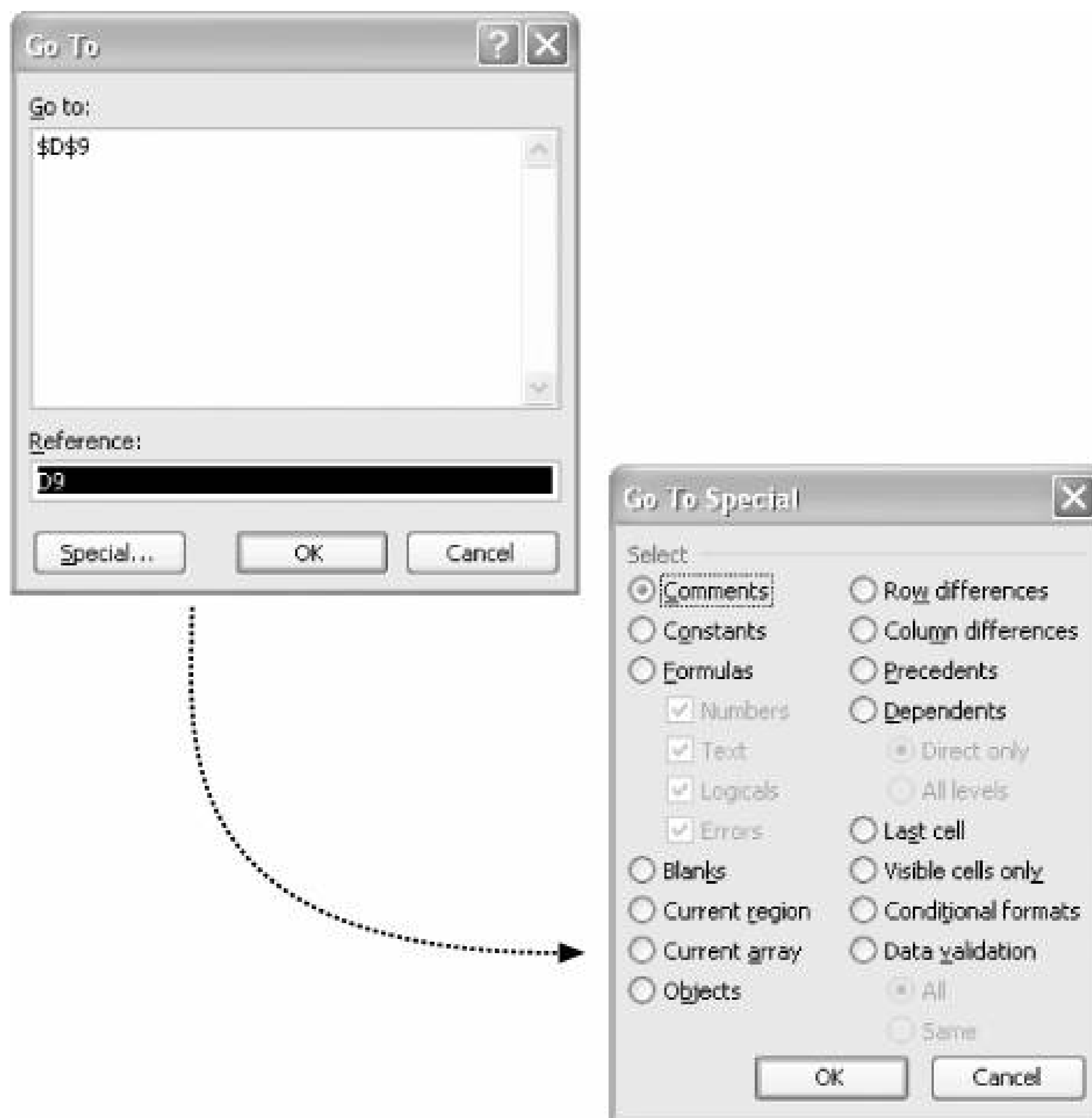
← PREY

## Recipe 1.5. Entering Formulas

### Problem

You've entered data and are ready to perform some calculations, but don't know how.

Figure 1-9. Group of selected cells



### Solution



You need to enter cell formulas in order to perform calculations on data contained in other cells. Entering a formula is simple enough: simply select a cell to hold the formula and then type the equal sign followed by your formula, which can refer to other cells that contain data. A formula to add two numbers contained in cells A1 and A2 would look like this: `=A1+A2`.

## Discussion

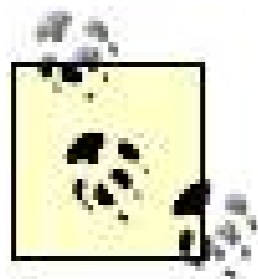
All formulas start with an equals sign, as in the `=A1+A2` example. You can enter a formula in the same way you enter text, as discussed in [Recipe 1.2](#). You can either enter the formula directly in a cell (pressing Enter when done), or you can use the formula bar, as discussed earlier. The cell containing the formula will display the result of the formula and not the formula itself. To see the formula, just look at the formula bar when the cell is selected. Or press the F2 shortcut key to edit the formula directly in the cell, as shown in [Figure 1-10](#).

Figure 1-10. Entering formulas

The cell in column C row 4 contains a formula to divide the number contained in column A row 1 by that in column A row 2. I pressed F2 to display the formula for editing directly in the cell.

Formulas may contain the usual mathematical operators such as +, -, /, and \*, as well as any number of other built-in functions, as discussed in [Recipe 1.10](#).

I should mention here that spreadsheets are a bit different from procedural programs with which you may be used to working. With a procedural program, you have to explicitly run it after writing the code (we'll do this when we write custom macros and functions using Visual Basic for Applications in [Chapter 2](#)). However, once you write a formula in a spreadsheet cell, it executes and remains up-to-date automatically. Therefore, if you change the data in cells referred to by a formula, the results of the formula are updated immediately. Excel takes care of making sure your spreadsheet calculations are updated whenever you change anything.



Excel performs calculations on the stored values of cells (not the displayed values), using 15 digits of precision.

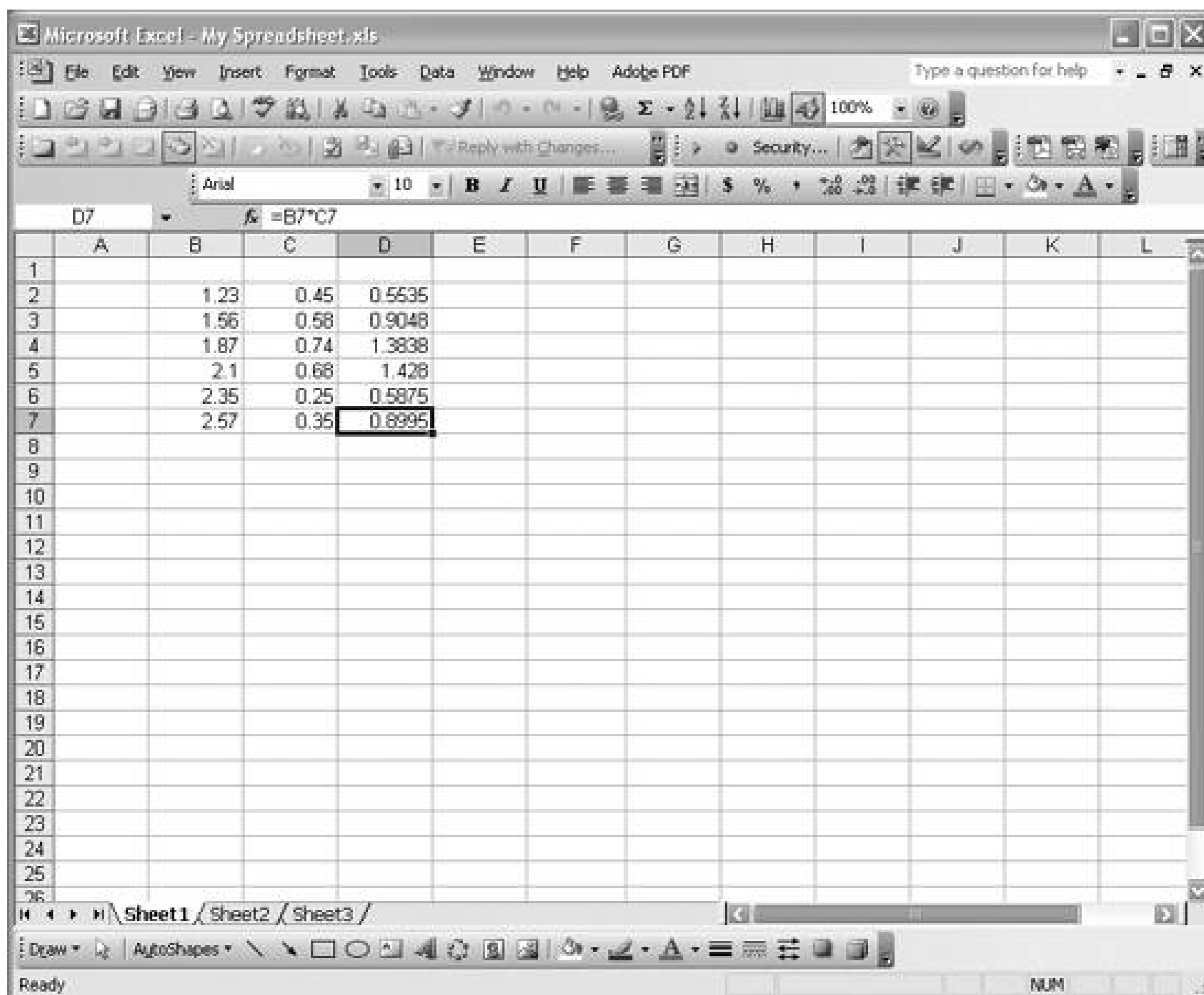
To really leverage formulas, you must understand *cell references*. In the formula `=A1+A2`, the data contained in the cells in rows 1 and 2 of column A are referenced as `A1` and `A2`, respectively. In Excel such references are called A1-style references. The letter refers to the column; the number refers to the row number.

References to cells such as `A1` are by default *relative references*. When a relative reference to a cell appears in a formula, that cell is referred to in terms of its relative position to the cell containing the formula. If you cut or copy the formula and then paste it someplace else, the cell references in the formula will automatically change so that they refer to cells in the same relative position to the formula as before. This is better explained by way of example.

In [Figure 1-10](#), the cell `A4` contains the formula `=A1+A2`, which uses relative references. If I copy and paste the formula from cell `A4` to cell `A5`, then the new formula will be `=A2+A3`. I moved the formula one row down, so the relative cell references were also adjusted by one row. If I moved the formula to column B, then the cell references would have changed from A to B.

This behavior may seem a bit unusual at first, but the advantage of it is clear when you are setting up tables of data and formulas to manipulate the data. For example, say I had two columns of data, B and C, as shown in [Figure 1-11](#).

Figure 1-11. Relative cell references

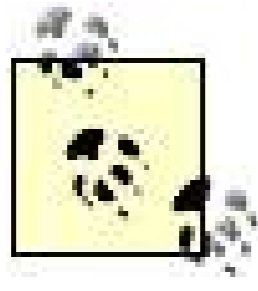


Now in column D, I want to enter a formula to multiply each number in column B by the corresponding number (on the same row) in column C. To do this, I can enter the formula `=B2*C2` in cell D2 to yield the product of numbers in cells B2 and C2. For all the remaining rows, I can simply copy and paste the formula in cell D2 into cells from D3 to D7, and Excel will automatically adjust the row numbers in the formulas. For example, after such copying and pasting, cell D7 contains the formula `=B7*C7`. This saves you the time of having to rewrite each formula directly. Relative cell references are also important when performing certain database filtering operations, as we'll discuss later.

As useful as relative references are, they aren't always what you want. Let's say you have the data as shown in [Figure 1-11](#), but instead of just taking the product of the two numbers in columns B and C, you want to also multiply this product by a single number contained in cell A1. Let's say you want every product shown in column D to be multiplied by that same factor in cell A1. You might enter a formula in cell D2 like `=B2*C2*A1`. This works for the first result contained in cell D2, but if you copy and paste this formula into cells D3 through D7, the cell reference A1 will also be adjusted by 1 each time, resulting in the formula `=(B7*C7)*A6` in cell D7. This isn't what you want, since there are no values in cells A2 through A6. To force Excel to refer to a specific cell you need to use an *absolute cell reference*. This essentially fixes the cell reference to always refer to a specific cell no matter how the formula is moved or copied. The syntax for an absolute reference to cell A1 in our example looks like `=(B2*C2)*$A$1`.

The `$` symbol in the cell reference indicates to Excel that you want to use absolute references. In this

case we fixed both the column and the row in the example cell reference. If we wanted to fix just the column, we would write  $\$A1$ . Likewise, we could fix just the row by writing  $A\$1$ .



When entering a formula in a cell, you can select another cell from the spreadsheet to include that cell's reference in the formula. You can also press F4 to cycle through the reference styles for the cell being referred to.

I use absolute references most often when performing calculations on datasets using formulas that refer to common scientific or engineering constants.



Formulas cannot refer to themselves. For example, if you put a formula in cell A3, it cannot contain a reference to cell A3. That's called a *circular reference*, and Excel will give you a warning message if one occurs.

## See Also

The A1 style of cell reference is the default reference style, but it isn't the only style; [Recipe 1.6](#) discusses others.

## Recipe 1.6. Exploring the R1C1 Cell Reference Style

### Problem

You've seen the use of the A1 style of cell reference in [Recipe 1.5](#) and are wondering if there are other ways to refer to cells.

### Solution

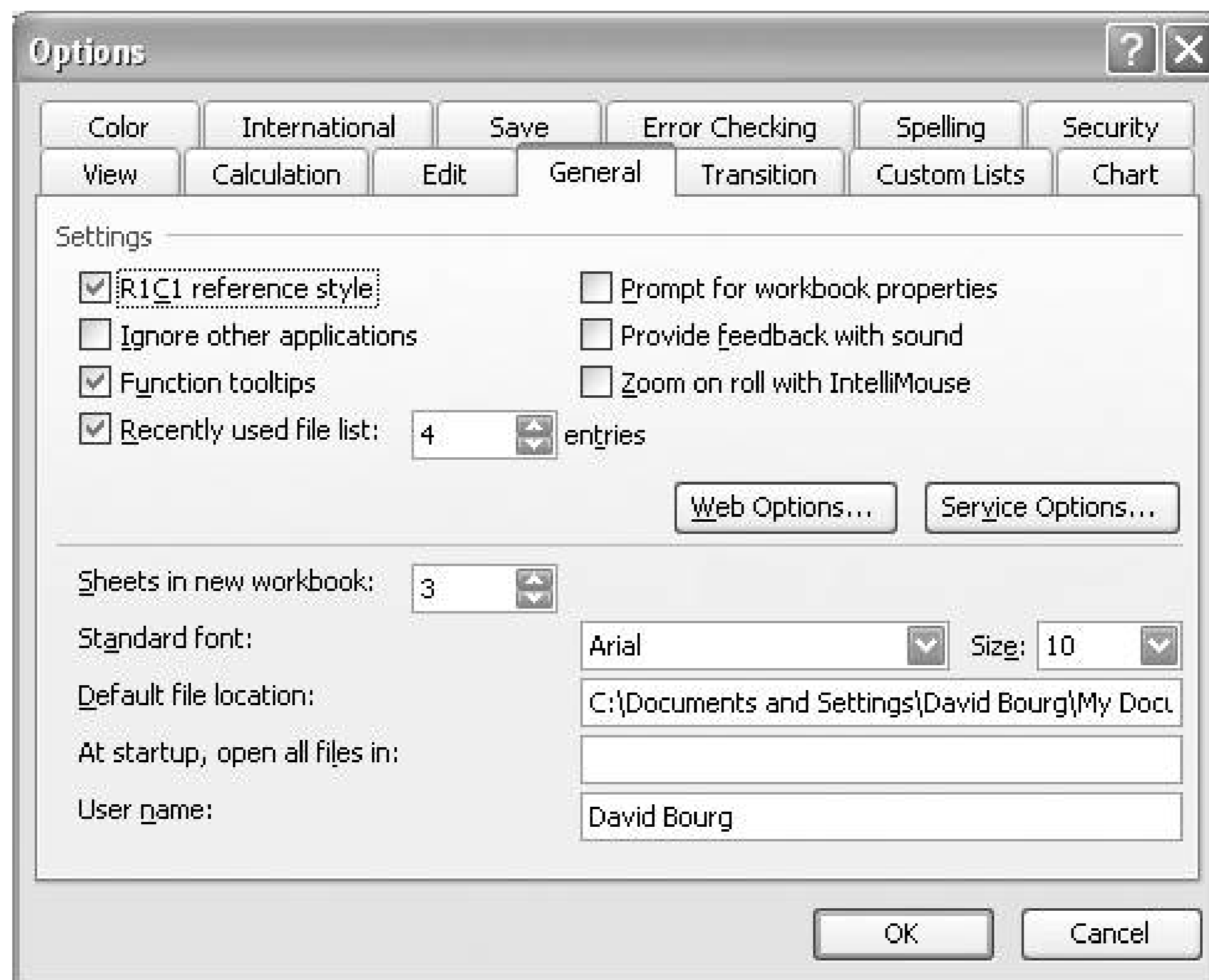
Yes, there are other ways to refer cells, such as the R1C1 style.

### Discussion

Besides using the A1 style of cell reference, you can also use the so-called R1C1 style of cell reference. This is not the default style, but in some cases it can be more intuitive or conducive to matrix operations and programming, as we'll see later.

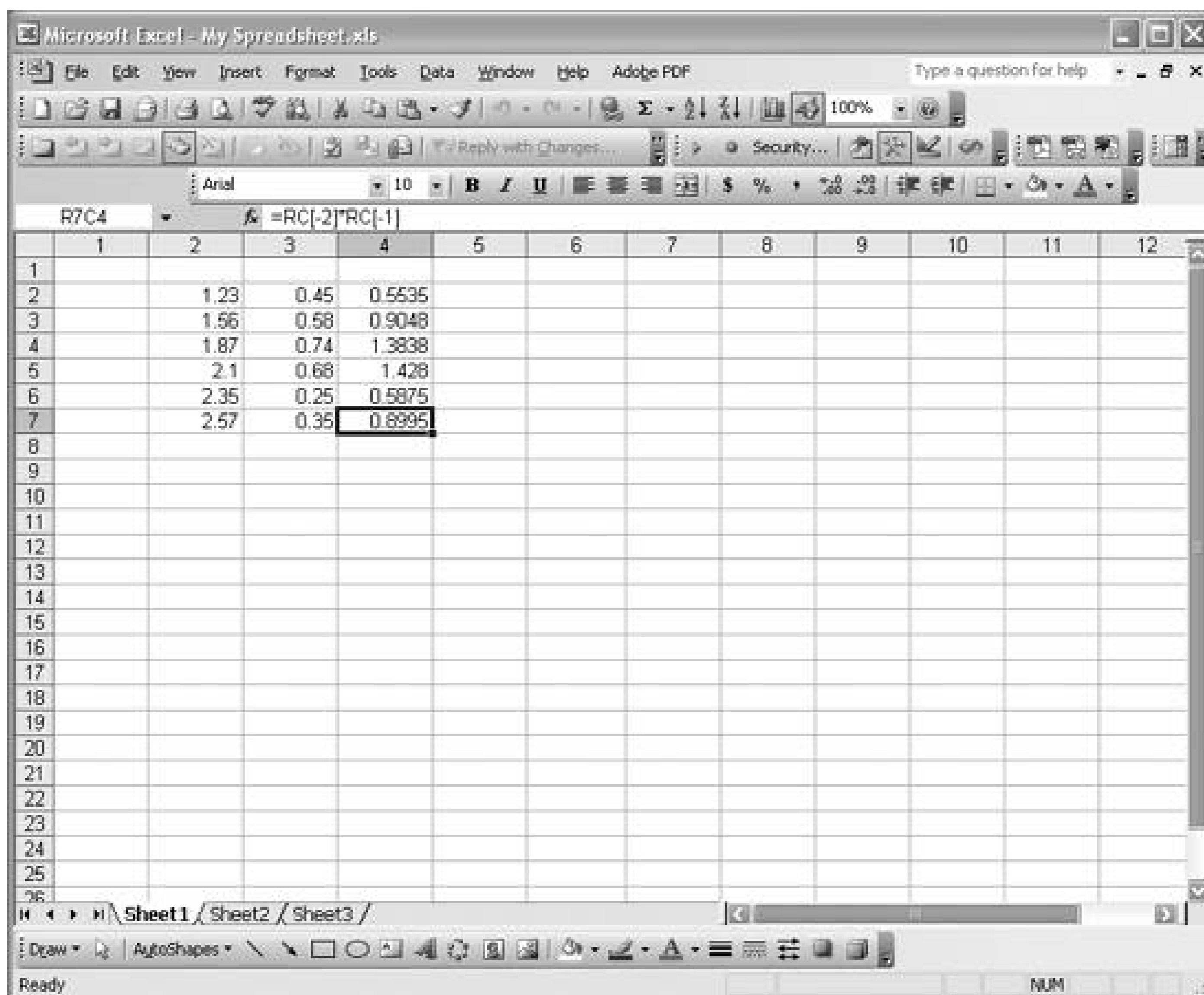
The R1C1 style uses numbers to identify both rows and columns in a spreadsheet. For example, R1C1 refers to the cell in row 1 column 1. To use the R1C1 style you must first activate it. To do so, go to the Tools → Options... menu to open the Options dialog box. Once it's open, press the General tab (see [Figure 1-12](#)).

Figure 1-12. General tab in the Options dialog box



Check the box next to R1C1 reference style under the Settings heading to activate the R1C1 style. When you press OK and return to your spreadsheet, you'll see that the column headings have changed from letters to numbers, as shown in [Figure 1-13](#).

Figure 1-13. R1C1 reference style



Also notice that Excel automatically changed the formulas. For example, the formula in cell D7 (now R7C4) was `=B7*C7`; now it's `=RC[-2]*RC[-1]`, which is a relative reference in R1C1 style.

When using the R1C1 style, if you enter a cell reference like `R3C5` (i.e., `R` followed by a number and `C` followed by another number), you are using absolute cell references. The equivalent in A1 style would be `$E$3`. Using brackets around the number following either `R` or `C` indicates relative cell references. For example, `R[1]C[2]` refers to the cell one row down and two rows to the right of the cell containing that reference. The cell two rows up and one row to the left would be referred to as `R[-2]C[-1]`. An `R` or `C` not followed by a number or a number in brackets refers to the same row or column as the cell containing the reference.

## See Also

To learn more about cell references, see [Recipe 1.7](#) and [Recipe 1.14](#). You can also check out the Excel Help topic "About cell and range references."



## Recipe 1.7. Referring to More Than a Single Cell

### Problem

Sometimes you need to refer to a group of cells, not just a single cell. For example, some formulas discussed in [Recipe 1.10](#) take more than a single cell as an argument. Thus you need to know the syntax for referring to more than one cell.

### Solution

Use *cell ranges*.

### Discussion

A cell range is simply a contiguous group of cells in rows or columns, or both. For example, the cell reference `A1:A10` refers to the range of cells in column A from row 1 to row 10. The colon character (`:`) is used to indicate a range reference. The reference `A1:B10` refers to the range of cells from column A row 1 to column B row 10. Technically speaking, the cell reference `A1` is itself a range of only a single cell; thus, in a sense, all cell references can be thought of as ranges.

### See Also

See [Recipe 1.10](#) for examples on where ranges are required as function arguments.



## Recipe 1.8. Understanding Operator Precedence

### Problem

You want to learn the specific order in which Excel executes operations in formulas.

### Solution

Excel performs operations in formulas from left to right following the leading equals sign. In doing so Excel performs specific operations, if they are encountered in your formula, in the following order of precedence: negation, exponentiation, multiplication and division, and addition and subtraction.

### Discussion

You can change operator precedence using parentheses. For example, if you enter the formula `=A1+B2/C3` in a cell, Excel will perform the division first and then the addition. This may be what you want that is, you want to add the result of B2 divided by C3 to A1. However, if you intend to divide the sum of A1 and B2 by C3, then you need to write `=(A1+B2)/C3`. The parentheses force Excel to perform the addition operation first, followed by the division.

You can also nest parentheses. For example you could write `=((A1+B2)/C3)*C4`, and so on. I find it is always good practice to use parentheses liberally to be sure your formula is executed as intended.

[← PREV](#)

## Recipe 1.9. Using Exponents in Formulas

### Problem

You need to write a formula that raises some number to some power but you don't know the syntax for exponentiation in Excel.

### Solution

Use the caret (^) operator.

### Discussion

Raising a number to some power is a common calculation task. In Excel, the caret operator (^) is used for exponentiation. For example, to raise the number contained in cell A1 to the third power, you could enter the formula `=A1^3`. You can use whole number exponents as well as decimal numbers or even other operations. For example, the formulas `=A1^0.25` and `=A1^(1/4)` both raise the value in cell A1 to the one-fourth power.

You need not hardcode exponents. You could use a number contained in another cell as an exponent: e.g., `=A1^C5`, or `=A1^(C5+D10)`, or `=(A1+A2)^(C5/E8)`, and so on.

### See Also

See [Recipe 1.10](#) for other common mathematical functions.

## Recipe 1.10. Exploring Functions

### Problem

The basic mathematical operators (+, -, /, \*, and ^) are not enough to perform all the calculations you need.

### Solution

Use Excel's built-in functions, such as `ABS( )`, `SQRT( )`, and `SIN( )`, as needed.

### Discussion

Throughout this book, I'm going to use all sorts of built-in functions in various calculations from data analysis to unit conversions to various engineering calculations. In this recipe I want to make you aware of the wide variety of built-in functions and how to access them in your spreadsheets.

Excel has many built-in functions, which can be organized in the following categories:

#### *Database functions*

Database functions include functions that allow you to get information from database entries and perform some statistical analyses of data contained in databases.

#### *Date and time functions*

Date and time functions include functions that allow you work with and perform calculations using dates and times. For example, you may want to calculate the number of working days between two dates, in which case you can use the `NETWORKDAYS` function. There are many others, including functions to get the current date and time.

#### *Engineering functions*

Engineering functions include functions that allow you to work with complex numbers, convert between number systems, and convert between systems of measurement. There are also many other specialized functions for working with Bessel and Delta functions, among others.

### *Financial functions*

Financial functions include many functions that help you analyze things like accrued interest, depreciation, annuities, and investments, among many others.

### *Logical functions*

Logical functions include a small set of useful logical functions. For example, the **IF** function allows you to evaluate a logical expression and return one value if the expression is true and another value if it is false. There are other functions including **AND**, **OR**, and **NOT**, among others, that help you construct logical expressions.

### *Lookup functions*

Lookup functions are useful for looking up data in tables, matching values, returning rows or columns for cell references, and other tasks.

### *Math functions*

Math functions help you perform calculations, and include operations such as taking the absolute value of a number, finding square roots, working with logarithms and exponential functions, summing values of data contained in rows or columns, and working with trigonometric functions. This is perhaps the most used set of functions for the sorts of calculations discussed in this book.

### *Statistical Functions*

Statistical functions are useful for calculating averages, standard deviations, variances, among many other calculations. There is also a set of functions useful for working with probability and distributions, including binomial, Weibull, and normal distributions, among others.

### *Text and data functions*

Text and data functions include a set of functions for manipulating text and converting numeric data to text. For example, there are functions to compare text strings, search for a substring within a larger text string, convert a string to all upper- or lowercase, and replace characters within a string.

There are a few other categories of built-in functions that I didn't summarize here. These functions are rather specialized and are useful for working with external programs and databases and collecting information from the host operating system, among other tasks.

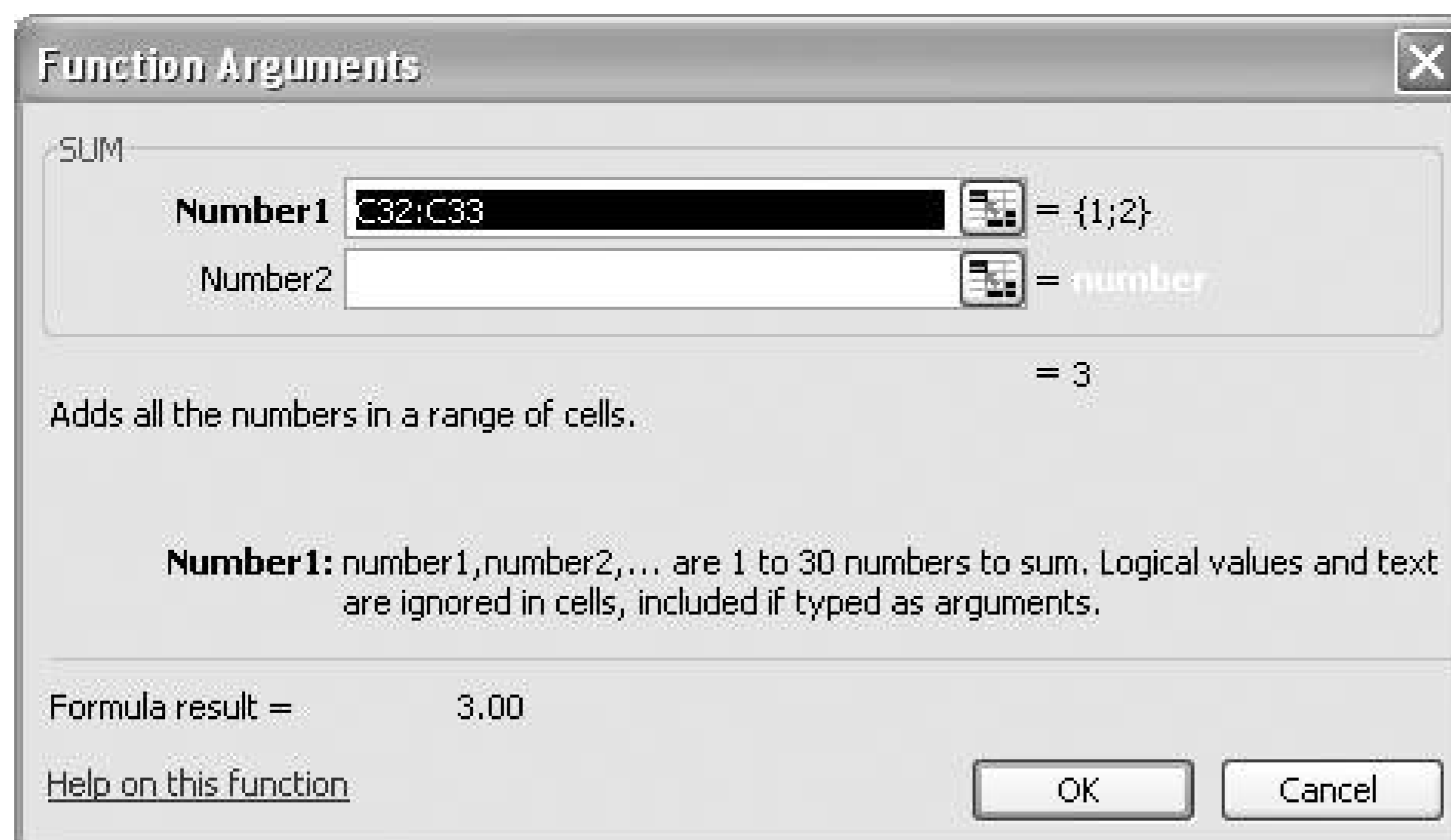
The recipes throughout the remainder of this book will use many built-in functions, most notably, but not limited to, the math, statistical, engineering, and logical functions. I'm not going to list all of the available functions here, since they are already documented fairly well within Excel itself and there are also many books that serve as complete Excel function references. I do want to show you how to access these built-in functions now so you can be familiar with the process when you read the more specific recipes to follow in later chapters of this book.

The easiest way to use a function in a formula is to simply type the function. For example, to sum the values contained in a column consisting of the cell range C1:C10, you can write a formula using the **SUM** function like this: `=SUM(C1:C10)`. In this case, **SUM** is the function and the cell range is the *argument* of the function.

When typing such a formula in a cell, as soon as you type the opening parenthesis after the function name, Excel will display a small, helpful information box below your formula, showing what arguments are expected (assuming you've typed a valid built-in function and spelled it correctly). At this point, you can type the arguments directly or, if the argument calls for a cell range, you can actually point and click to select the cell range, and Excel will automatically complete the cell reference argument for you. This reduces working with cell references in formulas to a point-and-click operation rather than typing everything out manually.

When typing a function, if you'd like more help on the function aside from the argument list that's automatically displayed, you can press the  $f_x$  icon (called the Insert Function button) in the formula bar to display the Function Arguments dialog box (see [Figure 1-14](#)).

Figure 1-14. Function Arguments dialog box

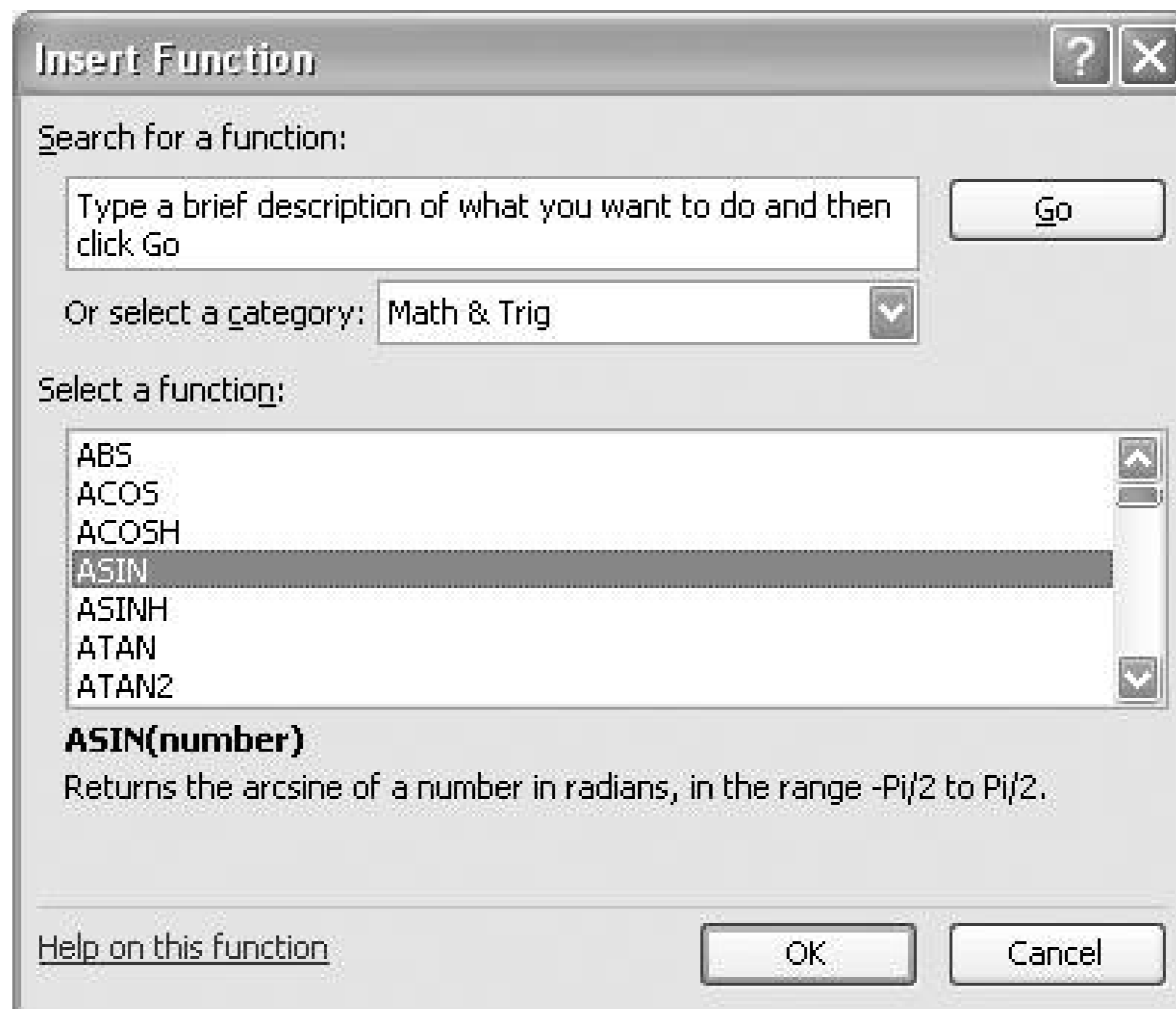


The Function Arguments dialog box displays controls and notes to help you construct the arguments for the function. There's also a link to more help on the specific function, as shown in the lower-left corner of the dialog box in [Figure 1-14](#).

Of course, the method of using functions discussed so far requires that you at least know the function name that you'd like to use. If you don't know or can't remember the name, you can either look it up in Excel Help or you can use the formula bar and the Insert Function dialog box to help you find appropriate functions.

To use the Insert Function dialog, select the formula bar to begin writing a formula and then press the  $f_x$  icon when you need to look up a function. This will bring up the Insert Function dialog box (see [Figure 1-15](#)).

Figure 1-15. Insert Function dialog box



With the Insert Function dialog box open, you may type a short description of what you're looking for in the "Search for a function" field. Or you can select a category from the drop-down list to view a list of functions within that category. From there you can select the specific function you need and press OK. When you press OK, the Function Argument dialog box will open to assist you with supplying function arguments. When you press OK in the Function Argument dialog box, you'll see your selected function, including arguments, appear in the formula bar; at this point, you can continue writing your formula.

You can nest functions. For example, you can use a formula like `=SQRT(ABS(C5))` to return the square root of the absolute value of the value contained in cell C5.

## See Also

Excel Help contains a handy function reference. I usually have the task pane opened up with the Excel Help page displayed and open to the function reference section of the help guide. To open the function reference, open the task pane (Ctrl-F1) and select Excel Help from the drop-down list at the top of the pane. Click the "Table of Contents" link, look for the topic "Working with Data," and click it. Now look for the subtopic "Function Reference" and click it to see the various function categories, which you can click to reveal specific topics on each built-in function.

There are also many useful Excel function reference books on the market, such as *Excel 2000 in a Nutshell* (O'Reilly), by Jinjer Simon. It contains a concise reference to all of the menu options and built-in functions in Excel.



## Recipe 1.11. Formatting Your Spreadsheets

### Problem

You've got data and some calculations, but the spreadsheet looks unpresentable.

### Solution

Format your spreadsheet to make it more presentable and better organized.

### Discussion

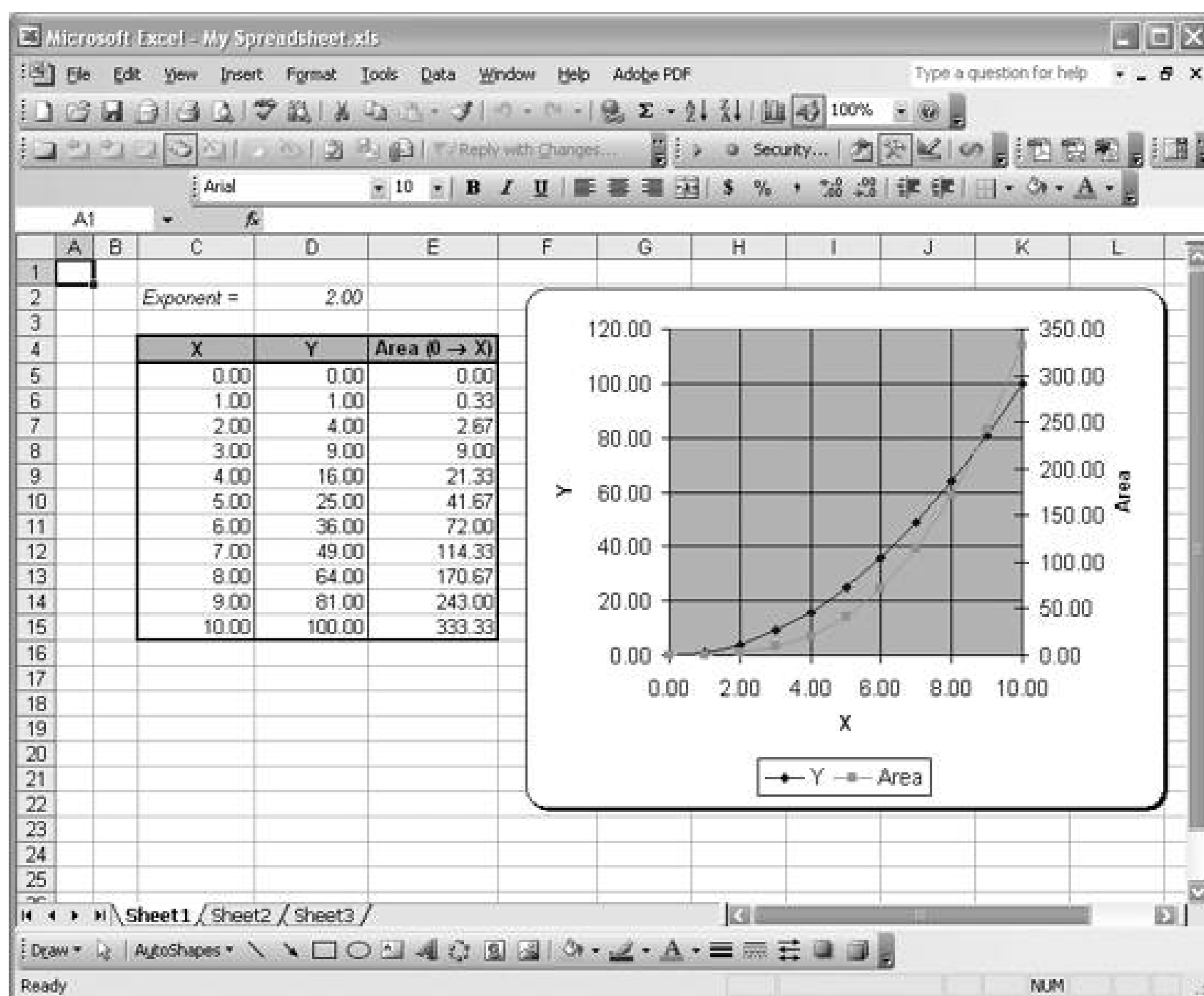
In [Recipe 1.3](#), I discussed how to format cells to specify the type of data (e.g., text, numbers, currency, etc.). In this recipe, I'll show you a few other formatting techniques. Take a look at the spreadsheet in [Figure 1-16](#).

This simple spreadsheet merely calculates values for a function of the form  $y = x^n$ , where  $n$  is the exponent shown on the spreadsheet. It also calculates the cumulative area under the curve from 0 to  $x$ . The results are plotted on a chart adjacent to the calculation table. [Chapter 4](#) covers charting in detail, so I won't discuss the chart here. Instead I want to focus on the calculation table.

I could have simply filled in a column of  $x$  values, entered formulas for the  $y$  and area values, and left it at that. However, that would look messy. More than likely you'll want to include your calculations in reports or share them with others, so making your spreadsheets presentable is a good idea. Formatting also serves as a form of documentation so that you can come back to a spreadsheet weeks or years later and quickly see what you did. There's (almost) nothing worse than opening an old spreadsheet and seeing just a grid of scattered numbers.

Figure 1-16. Formatted spreadsheet





There are many ways to format your spreadsheet. In this example, I added a text label, in italics, in the cell C2 to indicate the purpose of the value in cell D2. I also added some borders around the table of calculations and delineated the column headings with a filled background and bold text. I also centered the column labels above the data instead of using the default left justification for text. If you're a keen observer, you may have noticed that I also changed the column width of several columns. I reduced the width of columns A and B so they wouldn't take up too much space and I increased the width of column E to accommodate the area column label. These are some of the most common formatting tasks I make on virtually every spreadsheet I write. They are simple and effective.

You can access Excel's formatting functions via the Format menu. There you'll find options for formatting cells, whole rows or columns, and even the entire spreadsheet. I personally use the cell formatting options the most. You've already seen how to access the Format Cells dialog box in [Recipe 1.3](#), and as shown in [Figure 1-6](#). You can use the main menu bar or the shortcut key combination, Ctrl-1, to open the Format Cells dialog box.

The Number tab displays the data type formatting options discussed earlier. The other tabs give you control of other cell aspects:

### *Alignment*

Set the position, orientation, and justification of text within a cell. You can also specify whether long text should wrap around, creating multiple lines of text in a single cell.

### *Font*

Specify the font type, style, and size of text to appear in a cell. For example, you could specify Arial font type with a bold style and a size of 10 points.

### *Border*

Add borders to any edge of a cell. You can select from a variety of line styles, including solid, dashed, and dotted, among others, and even set the line thickness.

### *Patterns*

Set the background color or pattern for a cell.

### *Protection*

Specify whether a cell is protected (meaning it can't be changed without a password) and whether a cell is hidden.

While the Format Cells dialog box presents you with many cell formatting options, I prefer to use the toolbars for quick formatting tasks. If you look closely at [Figure 1-16](#), you'll see the formatting toolbar visible just above the formula bar. This toolbar contains buttons to change a cell's font, justification, data type, number of decimal places, indentation, borders, fill patterns, text color, and more. You can customize this toolbar as discussed in [Recipe 1.1](#).

There may be times when you want to clear all the formats applied to a cell. To do so, select the cell whose formats you want to clear and then select Edit > Clear > Formats from the main menu bar.

## **See Also**

The formatting options discussed here are common and useful. There are other formatting options that may come in handy from time to time, although you probably will not use them too often. These options include the row, column, and spreadsheet formatting options mentioned earlier, as well as others such as conditional formatting, which allows you to set formatting for a cell contingent on the value of the cell relative to some criterion you specify. We'll make use of these other formatting options throughout the remainder of the book. For the time being, you can read more about formatting options in Excel's built-in help system. To do so, open Excel Help by pressing Ctrl-F1 and click the "Table of Contents" link. Look for the "Working with Data" topic and click on it to reveal a list of subtopics. Look for the "Formatting Data" subtopic and click it to see a list of specific formatting-related topics.

← PREV

## Recipe 1.12. Defining Custom Format Styles

### Problem

You find yourself applying the same formats repeatedly to different cells and would like to save time when doing so.

### Solution

Define your own custom style and apply it to cells, taking care of many format settings all in one step.

### Discussion

Select Format Style from the main menu bar to open the Style dialog box as shown in [Figure 1-17](#).

Figure 1-17. Style dialog box

The drop-down combo box next to the "Style name" label contains a list of predefined styles that you

can apply to cells. Some of these are familiar, including the normal cell style along with currency and percent styles. You can add your own style to this list. Click the combo box, type in a unique name for your new style, and then press the Add button. Now you can check off the format options you want to include in your style and modify the format settings by pressing the Modify button. Pressing the Modify button opens the Format Cells dialog box (see [Figure 1-6](#) in [Recipe 1.3](#)). You can make any format selection you desire and close this dialog box when done. The settings you specified will be applied to your custom style.

Once your style is defined, you have a few options for actually using it to format cells. One way to apply your custom style is to select the desired cell (or cells) and then select Format > Style... from the main menu bar to open the Style dialog box again. There you can select your desired style and press OK to close the dialog box. Your style will then be applied to the selected cell (or cells).

A faster approach involves adding a style drop-down list component to one of the toolbars on the main window so that you can simply select cells and then select a style to apply from the toolbar.

To add a style drop-down list to a toolbar, select View > Toolbar > Customize... from the main menu bar to open the Customize dialog box. Select the Commands tab (shown in [Figure 1-18](#)).

### Figure 1-18. Customize dialog box

Now select Format from the Categories list to show the format commands. From the Commands list look for the Style drop-down listbox shown in [Figure 1-18](#). Don't confuse it with the Style... dialog box command. Click and drag the Style listbox control from the Commands list over to any toolbar, preferably the format toolbar, on Excel's main window. After you close the Customize dialog box, the

listbox will be available to you on the toolbar.

To use the Style listbox, select the cells to which you want to apply a style and then select the desired style from the Style listbox that you added to the toolbar.



## Recipe 1.13. Leveraging Copy, Cut, Paste, and Paste Special

### Problem

You'd like to take advantage of standard Windows copy-and-paste functionality but aren't familiar with the caveats for doing so within Excel.

### Solution

Read the following discussion.

### Discussion

Under Excel's Edit menu, you'll find the usual Cut, Copy, and Paste menu items. For the most part, these work just as they do in any other Windows program. To move the contents of a cell from one place to another, use the Cut and Paste operations. To copy a cell into other cells, use the Copy and Paste operations. There are a few things to be aware of when performing these operations in Excel:

- When you cut or copy and then paste, all of the cell's attributes are copied, along with the data or formula that it contains. This means things like font, alignment, borders, and patterns will be copied as well.
- When you cut and paste data to a new location, any formulas referring to its original cell location are automatically updated to refer to its new location. Copying a cell to another location has no effect on formulas that refer to the original cell.
- When you copy a cell containing a formula, relative cell references in the formula will be adjusted by the relative distance of the pasted cell from the copied cell.
- When you cut a formula and paste it to a new location, the formula will remain unchanged; i.e., it should still refer to the same cells and give the same results as before the cut-and-paste operation.

Sometimes you may want to cut or copy and then paste all of the format settings and data from one cell to another. In cases when you don't, you can use the Paste Special option, which is also under the Edit menu. When you select the Paste Special option, a dialog box like that shown in [Figure 1-19](#) appears.

Figure 1-19. Paste Special dialog box



Here you can select exactly what it is you want to copy and paste. For example, if you want to copy and paste only cell format settings, you can select the Formats option. Similarly, if you want to copy and paste just a formula and not the cell's formatting, then you can select the Formulas option, and so on.

The Operation options allow you to specify some basic mathematical operations to perform on the data being copied and the contents of the destination cell. For example, say you copy a number from one cell and Paste Special it to a cell that already contains another number. If you select the Add option, the sum of the two cell values will be entered in the destination cell.

The "Skip blanks" option excludes empty cells from the selected range of cells being copied. The Transpose option is similar to a matrix transpose operation. For example, if you copy a column of numbers and paste it with the Transpose option, it will be converted to a row of numbers.

Here are some handy keyboard shortcuts for the basic cut-and-paste operations:

*Cut*

Ctrl-X

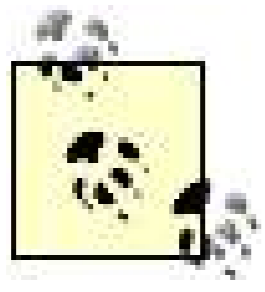
*Copy*

Ctrl-C

*Paste*

## Ctrl-V

There is no shortcut for the Paste Special operation; however, you can right-click anywhere in your spreadsheet to bring up a pop-up menu containing the Paste Special option. I find this quicker than accessing the main menu bar.



Cut, Copy, Paste, and other options (e.g., Format Cells) are also shown on the right-click pop-up menu.

[← PREV](#)





## Recipe 1.14. Using Cell Names (Like Programming Variables)

### Problem

You frequently use a particular cell in formulas and are tired of typing the reference. You'd like a more descriptive syntax, similar to that used for variable or constant names in traditional programming languages.

### Solution

Use *cell names*.

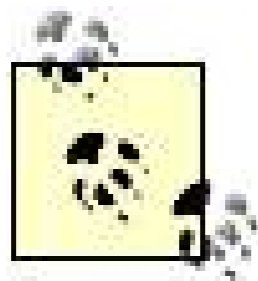
### Discussion

Select the cell or cell range for which you want to define a name. Next, select Insert    Name Define... from the main menu bar to open the Define Name dialog box (see in [Figure 1-20](#)).

Figure 1-20. Define Name dialog box

Enter a name for the selected cell range in the edit field under "Names in workbook." Make sure the "Refers to" field does indeed refer to the cell range you'd like to name. If you had the range selected when you opened this dialog box, then the cell range should already be correct. If you didn't have it selected, press the icon in the lower-right corner; this will allow you to select the proper range without leaving the Define Name dialog box. Once your name is entered and the cell reference is correct, press the Add button. You'll now be able to use that name to refer to the associated cell range in any formulas in your workbook.

Notice that the cell reference in the "Refers to" field in [Figure 1-20](#) includes the sheet name, `Sheet1`, followed by the exclamation mark (!), which precedes the absolute A1-style cell reference. This format, using the sheet name followed by !, makes the reference refer to the specified cell on that specific sheet. Thus, you can use the name in a formula on any other sheet and it will still point to the cell on Sheet1.



I find names quite useful. Typically, if I'm performing calculations that require the use of empirical constants or commonly used data, I'll set up a specific sheet in my workbook that contains only constants and commonly used data (or formulas). Then I name these so I can refer to them throughout the workbook. I find descriptive names more intuitive and easier to remember than cryptic cell references that span sheets in a workbook.

## Recipe 1.15. Validating Data

### Problem

You want to make sure users of your spreadsheet don't enter inappropriate data.

### Solution

Use Excel's Data Validation feature.

### Discussion

Excel allows you to specify what constitutes valid data for any given cell. For example, say you had a spreadsheet that performed some calculations given a value that was input in a particular cell by a user other than yourself. Let's say you want to restrict the range of values the user can enter in the input cell, in order to minimize the possibility of misuse. Such a situation could arise if, say, you write a spreadsheet that allows you to interpolate data based on a regression equation. In such a case, you would want to restrict the independent value input by the user to within the allowable bounds of the regression analysis. Basically, you don't want the user to attempt to extrapolate beyond the data range used in the regression analysis, as such extrapolations can sometimes yield dangerously inaccurate results.

To specify validation for a cell, select the cell you want to set validation for and then open the Data Validation dialog box by selecting Data      Validation... from the main menu bar. [Figure 1-21](#) shows the Data Validation dialog box.

Figure 1-21. Data Validation dialog box



The Allow drop-down listbox allows you to select the type of data to allow (for example, a decimal number, a whole number, a date, or text). Once you select the Allow type, the other controls will present additional qualifiers. In the example shown in [Figure 1-21](#), I set 0 to 560 as the valid data range. You can also specify ranges greater than some value, less than some value, and so on. Once you press OK, these changes take effect. If the user attempts to enter data that does not fit the valid data criteria, a message box will appear indicating bad data entry.

You can specify exactly what you want the error message to display by entering your message under the Error Alert tab of the Data Validation dialog box. Further, clicking the Input Message tab allows you to specify an information message to be displayed to the user whenever a cell using data validation is selected. This is useful for giving your users a clue as to what constitutes valid data for any particular cell.

## Recipe 1.16. Taking Advantage of Macros

### Problem

You find yourself repeating the same actions over and over and would like to automate that process.

### Solution

Use Excel's macro-recording feature to record your actions, which can then be executed again using a keyboard shortcut.

### Discussion

Let's say you find yourself repeatedly applying the same format settings to cells; for example, you select a cell, set the font style to bold, set justification to center, and apply a pattern and a border. You could define a custom style reflecting these format settings and use the style as discussed in [Recipe 1.12](#), or you could record a macro to automate the process of setting these formats.

Although I'm using formats as an example, you should be aware that the macro-recording feature lets you record any sequence of actions taken in Excel, thus allowing you to automate almost anything. In [Chapter 2](#), I discuss macros and other programming tasks in much greater detail in the context of using Visual Basic for Applications. That said, you can record simple macros to automate common tasks as discussed here, without using Visual Basic.

Take these steps to record a macro:

1. Select Tools    Macro    Record New Macro... from the main menu bar to open the Record Macro Dialog box.
2. In the Macro Dialog box, enter a descriptive name for your new macro in the Macro Name field. Also enter a shortcut letter for the shortcut key combination (Ctrl plus the letter you specify), which will be used to execute the macro on demand. In the Store Macro In field, select where you want the macro stored. The default is This Workbook, which stores the macro in the current workbook. However, if you want to be able to use the macro in any workbook, you should select Personal Macro Workbook. Press OK when you're done; this will return you to the spreadsheet in macro-recording mode.
3. Execute the actions you want recorded.
4. When you've finished your actions, select Tools    Macro    Stop Recording from the main

menu bar.

5. Use the shortcut key you specified in the Macro Dialog box to execute your macro.

Let's say you recorded a series of cell-formatting actions in a macro. Then, anytime you select a cell and press the shortcut key for that macro, the macro will run and you'll see that formatting applied to the cell.

If you want the macro to execute relative to the position of the active cell when you start recording, make sure the Relative Reference button is selected on the Record Macro toolbar that appears when you start recording a macro. The Relative Reference button is shown in [Figure 1-22](#).

Figure 1-22. Record Macro toolbar

You may also press the Stop Recording button to stop recording a macro (instead of using the Stop Recording menu item as mentioned earlier).

To manage macros, you can access the Macro dialog box via the Tools    Macro    Macros... menu or the Alt-F8 shortcut. The Macro dialog box allows you to run selected macros, delete macros, or redefine their shortcut keys.

## See Also

See Excel's "Create a Macro" help topic or open the Excel Help task pane and search for the phrase "recording macros" to view a list of relevant topics. Or open the Excel Help task pane, click the "Table of Contents" link, and then click on the "Automating Tasks and Programmability" topic to reveal a list of help topics related to macros.

← PREV

## Recipe 1.17. Adding Comments and Equation Notes

### Problem

You'd like to add comments to your spreadsheet, just as you'd comment your code in a traditional programming language.

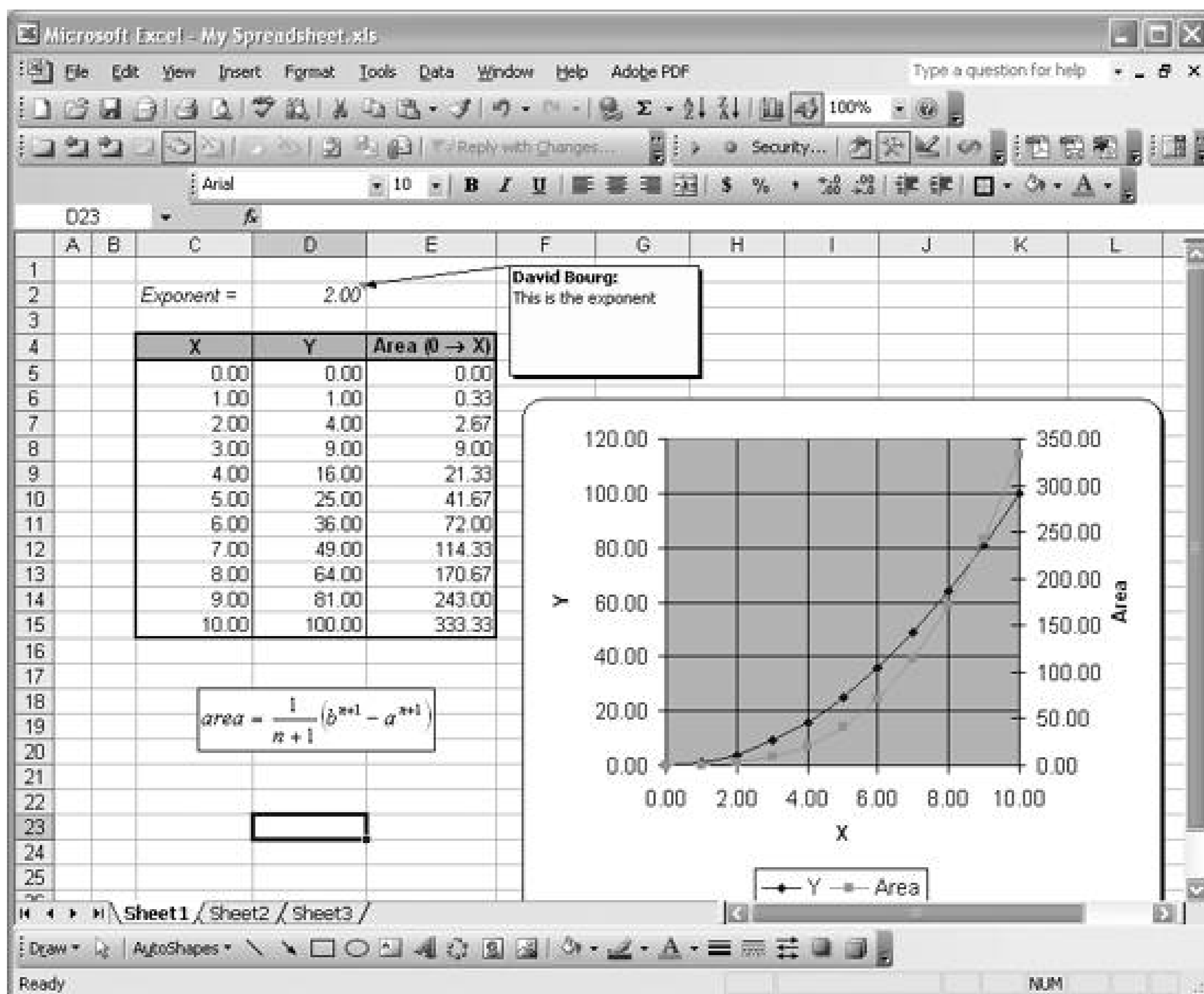
### Solution

Aside from simply inserting text in cells adjacent to cells containing data or formulas, you can use comments or equation objects to document your spreadsheets.

### Discussion

[Figure 1-23](#) shows a sample spreadsheet that includes a comment and an equation object documenting parts of the spreadsheet.

Figure 1-23. Spreadsheet with comment and equation



The comment is the rectangle containing the text "David Bourg: This is the exponent," with an arrow pointing to cell D2. The equation object is the rectangle containing the mathematical expression for the area calculation. Both comments and equations are useful devices for documenting your spreadsheets. I often use comments to leave myself notes or reminders or to-do lists within my spreadsheets. I use equations to document formulas used in my spreadsheets that will be used by others or included in reports or other presentations. Equations in standard mathematical form are much clearer and far easier to comprehend than cell formulas containing a bunch of cell references and operators all strung together.

To add a comment, first select the cell to which you want the comment attached, then select Insert Comment from the main menu bar. A comment box will appear (with input focus, so you can immediately begin typing your comment). When you're done typing, use the mouse to select where on the spreadsheet you want the comment box to be placed. To edit an existing comment, select the cell containing the comment and select Insert Edit Comment from the main menu bar. This will give input focus to the comment, allowing you to edit its message. To delete a comment, select the cell containing the comment and right-click to reveal a pop-up menu where you can select Delete Comment to actually delete the comment.

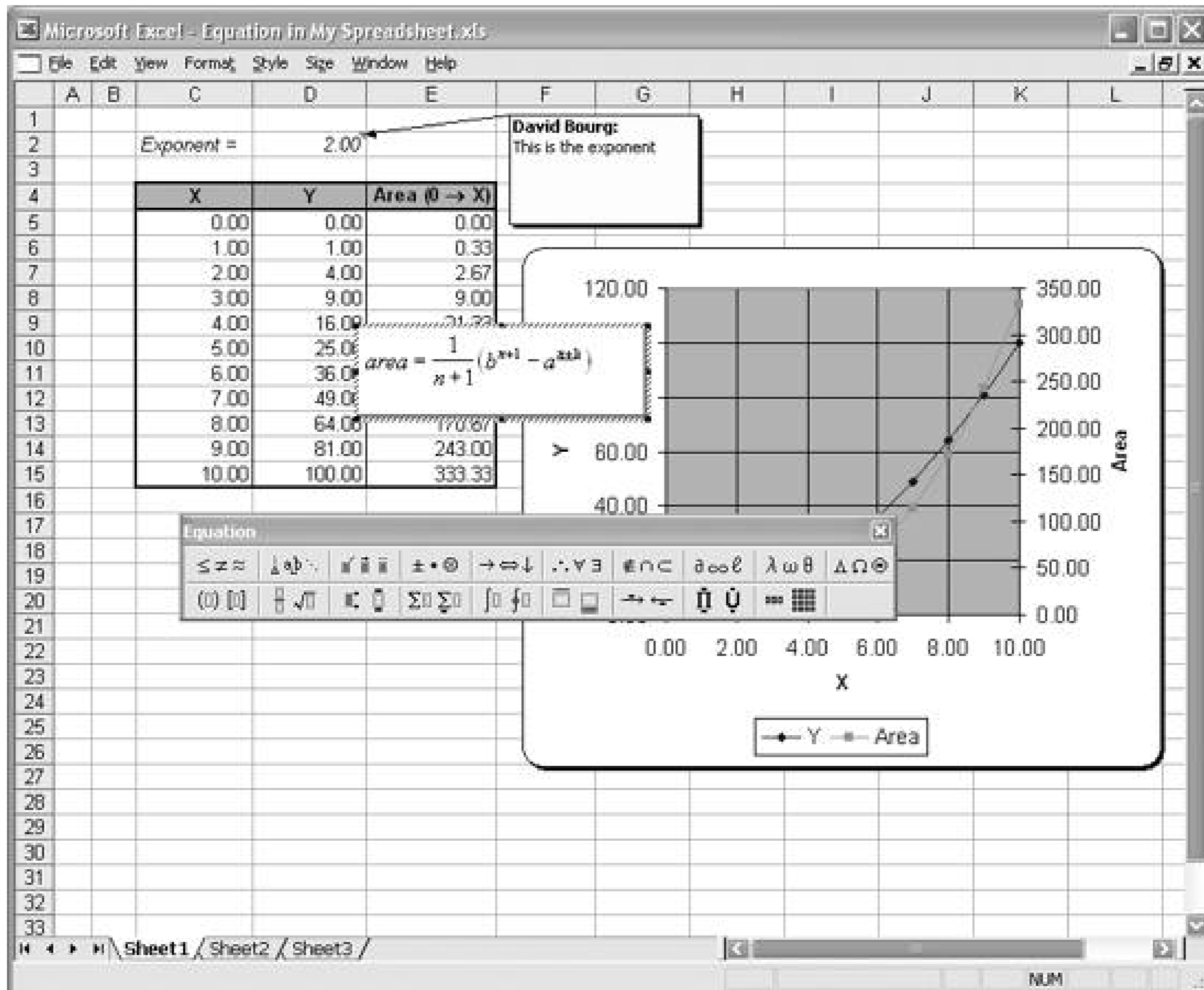
You can also use the Reviewing toolbar (by selecting View Toolbars Reviewing) to access comment features such as adding, editing, deleting, or temporarily hiding comments.



To add an equation, you need to use the Microsoft Equation Editor to insert an equation object into your spreadsheet. Select Insert → Object... to show the Object Dialog box. From the list of Object Types in this dialog box, choose Microsoft Equation 3.0. When you press OK, the dialog will close, returning you to the spreadsheet in equation-editing mode (see [Figure 1-24](#)).

The equation rectangle will appear, allowing you to enter an equation using the equation toolbar shown in [Figure 1-24](#). You can type text in the equation using the keyboard. To insert mathematical symbols, select them from the equation toolbar. Use the arrow keys on the keyboard to navigate an equation while editing. The Equation Editor is a bit cumbersome at first; I suggest you try it and practice until you get the hang of it. When you're done entering an equation, simply click anywhere in your spreadsheet to close the Equation Editor. You can click and drag the resulting equation to place it anywhere in your spreadsheet. To edit an existing equation, select it and then right-click on it to reveal a pop-up menu. Select Equation Object → Edit from this pop-up menu to edit the equation. To delete an equation, simply select it and press the Delete key on your keyboard.

Figure 1-24. Editing an equation



## See Also

There's an alternative to editing equations directly on your spreadsheet as illustrated in [Figure 1-24](#). If you select an equation object and then right-click to open the pop-up menu, you can select the Equation Object  $\rightarrow$  Open menu option. This will open a new Equation Editor window showing your equation along with the Equation Editor toolbar. You can edit your equation here and select File Exit to return to your spreadsheet.

The Equation Editor also contains a help menu option. Select Help  $\rightarrow$  Equation Editor Help Topics to open the Equation Editor help file and read more about using the Equation Editor.



The Equation Editor discussed here is the same one that's accessible from Microsoft Word . You can insert equations in your Word documents in the same way that you add equations to a spreadsheet, and you don't have to have Excel open in order to do so. This is useful for preparing technical reports in Word.

[← PREV](#)



## Recipe 1.18. Getting Help

### Problem

You're off to a great start and would like to know where you can go to find help within Excel.

### Solution

You can access Excel's online help system by selecting Help Microsoft Excel Help from the main menu bar. Or you can use the shortcut Ctrl-F1 to access the Help task pane, enabling you to select the Excel Help page, where you can click the "Table of Contents" link or enter a search phrase to search for help topics.

# Chapter 2. Getting Acquainted with Visual Basic for Applications

[Section 2.0. Introduction](#)

[Recipe 2.1. Navigating the VBA Editor](#)

[Recipe 2.2. Writing Functions and Subroutines](#)

[Recipe 2.3. Working with Data Types](#)

[Recipe 2.4. Defining Variables](#)

[Recipe 2.5. Defining Constants](#)

[Recipe 2.6. Using Arrays](#)

[Recipe 2.7. Commenting Code](#)

[Recipe 2.8. Spanning Long Statements over Multiple Lines](#)

[Recipe 2.9. Using Conditional Statements](#)

[Recipe 2.10. Using Loops](#)

[Recipe 2.11. Debugging VBA Code](#)

[Recipe 2.12. Exploring VBA's Built-in Functions](#)

[Recipe 2.13. Exploring Excel Objects](#)

[Recipe 2.14. Creating Your Own Objects in VBA](#)

[Recipe 2.15. VBA Help](#)

## 2.0. Introduction

I started using Visual Basic in the early 1990s, way back in the days of DOS and Windows 3.1. At the time, I was not all that impressed with Visual Basic and preferred Visual C and a few Borland products over Visual Basic. I still prefer C/C++ for most of the commercial applications I develop for clients; however, when I need to perform quick calculations for my own research or when a client needs something done quickly, I do now turn to Visual Basic for Applications (VBA).

When I first used Visual Basic, I saw no advantage over the other languages I was using at the time. All that changed, however, when Microsoft integrated Visual Basic with its Office suite of applications. This opened the door to greater flexibility, more control, and the possibility of developing very powerful applications very quickly using VBA while leveraging the interface, features, and functionality of the host Office application (Excel in our case).

Indeed, I've since started integrating standalone applications developed for my own clients using C/C++ with Office applications such as Word, Outlook, and Excel. For example, a standalone application written in C/C++ can perform calculations and then automatically send the output to Word in a nicely formatted report, which can then be emailed automatically using Outlook. VBA allows you to do this sort of thing too. We'll cover some of these cool things, such as making Excel talk to Word, later in the book. But before we do, you should be familiar with the VBA development tools, language, and interface to Excel.

This chapter is meant to give you that familiarity if you don't already have it. Just like the overview of Excel presented in [Chapter 1](#), the material covered in this chapter is necessarily at an introductory level. If you are already familiar with VBA, you can skip this chapter without loss of continuity.

## Recipe 2.1. Navigating the VBA Editor

### Problem

You want to start exploring the Visual Basic for Applications (VBA) language and don't know where to begin.

### Solution

Open Excel and select Tools → Macro → Visual Basic Editor from the main menu bar to open the VBA *integrated development environment* (IDE). Start exploring the IDE as described in the following discussion.

### Discussion

The VBA IDE is basically the VBA editor. It's where to write VBA code, manage code modules, and debug your code. The editor allows you to perform all these development tasks in a single, integrated application. You can access the IDE using the menu as mentioned earlier or you can use the shortcut Alt-F11. You can also use the Visual Basic toolbar, shown in [Figure 2-1](#).

Figure 2-1. Visual Basic toolbar

The IDE looks very much like any other Windows application, with a menu at the top, toolbars, and child windows in the client area of the window, as shown in [Figure 2-2](#).

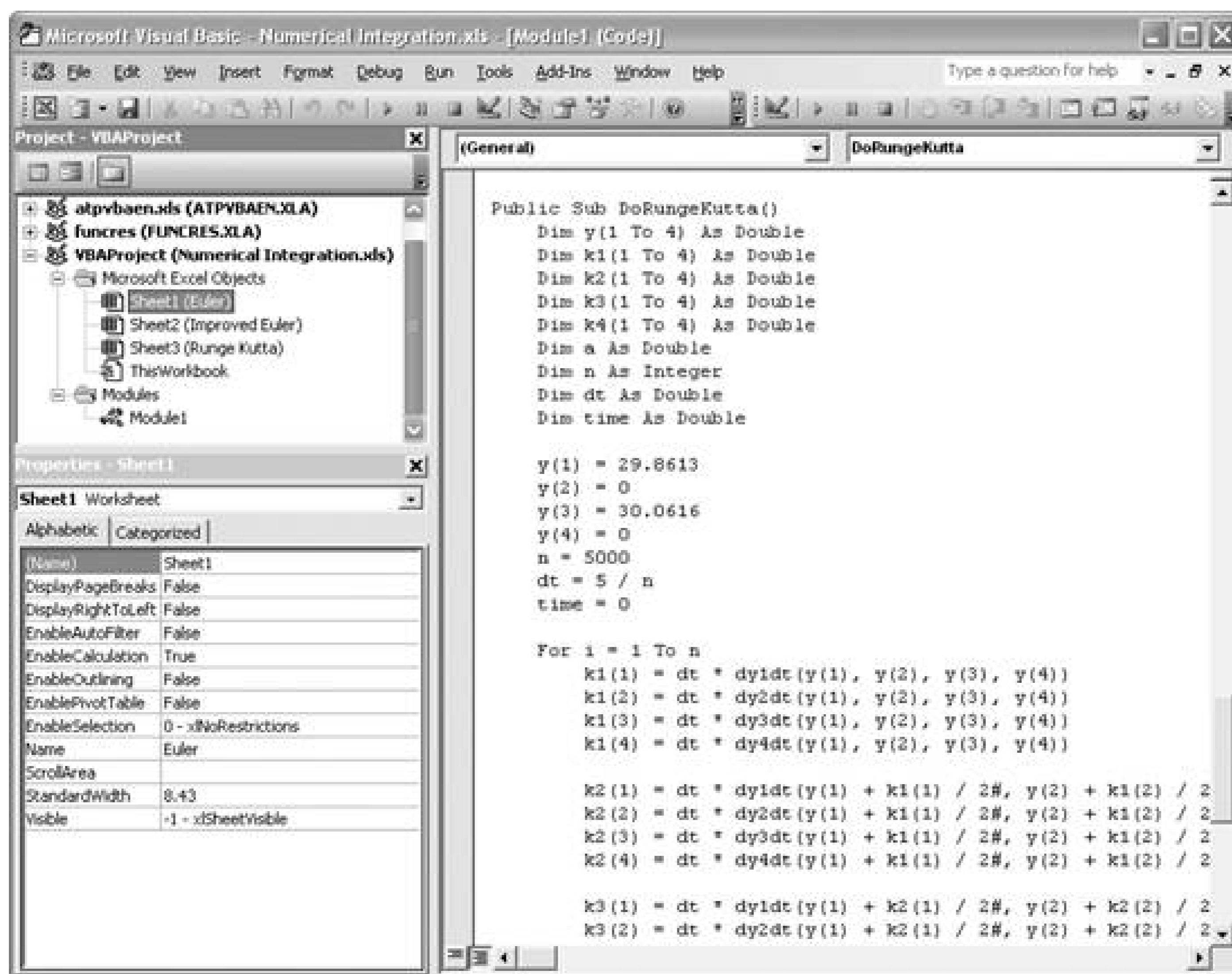
The VBA IDE is a multiple-document interface style of window. The area below the toolbars and menu is called the client area, which acts as a container for all the other (child) windows to be displayed within the IDE.

Within the client area of the VBA IDE shown in [Figure 2-2](#), there are three *panels* or windows. The panel in the upper left is called the *project panel*, or *project explorer* window. It shows an Explorer-like view of all the currently open VBA projects.

Each Excel workbook has a corresponding VBA project with a default name of the form *VBAProject (spreadsheet name)*. You can change this name by editing the **Name** property in the properties panel (more on that panel in a moment). Each project consists of multiple objects and code modules. Every sheet in a workbook has an associated object shown in the project window, as does the workbook itself. I'll come back to these *Excel objects* in [Recipe 2.13](#).

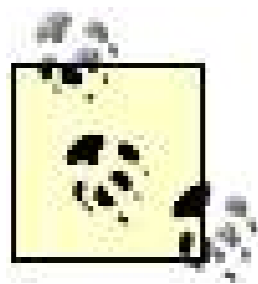
For the most part, we'll add our own code modules to a project and put our custom code in these modules. This has the advantage of giving our custom code global scope so that our custom function will be accessible by any sheet in the workbook.

Figure 2-2. VBA IDE



To add a module, select **Insert > Module** from the main menu bar (make sure your project is selected first in the project panel). Upon doing so, you'll see a module object appear under your selected project in the project panel. Double-click the module in the project panel to open its code window if it isn't already opened. [Figure 2-2](#) shows an example project with an open code window containing some VBA code.

When you select any object or module in the project panel, its *properties* are displayed in the properties panel.



In object-oriented programming lingo, an *object* is a construct that contains properties and methods. *Properties* define certain attributes of the object, while *methods* define the object's functionality. For example, let's say we write a code object to represent your car. We'd give it properties such as make, model, color, engine type, and seat covering type. These properties describe what the object looks like (i.e., its *attributes*, or characteristics). Further, we could give it *methods* such as move forward, reverse, and stop. These are actions (i.e., things the car can do when we invoke these methods). A more realistic programming example would be an object that represents, say, a text file. The properties could be things like size and number of words, while methods would be things like save and spellcheck.

For the example module shown in [Figure 2-2](#), the only property available is its name. You can change the module's name by editing the `Name` field in the properties panel.

Unless I'm doing work with Excel objects themselves or writing my own custom objects, I usually close the properties panel to clean up the editor a bit. Press F4 or select View Properties Window to show it when you need it. (We'll come back to Excel objects in [Recipe 2.13](#).)

Throughout this book, we'll spend most of our time in the VBA editor working with module code windows. Take another look at the code window shown in [Figure 2-2](#). At the top of the window you should see two drop-down listboxes. The listbox on the left is the *object list* and the one on the right is the *procedure list*. The object list shows a list of objects in the selected module, while the procedure list shows a list of procedures associated with the selected object.

*Procedures* in VBA context refer to functions and subroutines. We can write both functions and subroutines to perform actions and calculations for us, but there's a distinct difference between the two as discussed in [Recipe 2.2](#).

For Excel objects, the object list may contain several entries; however, for the code modules that we'll work with the most, the object list will show only General, while the procedure list will show a list of all our custom procedures. You can jump to any procedure in the code window by selecting it from the procedure list. This is a handy way to get around your code window when it fills up with a lot of code.

You may have also noticed two little icons in the lower-left corner of the code window shown in [Figure 2-2](#). (These icons consist of a few horizontal lines.) The icon on the left with fewer lines puts the code window in *Procedure View* when clicked, while the one on the right puts the code window in *Full Module View*. Procedure View shows only one procedure at a time in the code window, which is convenient if you find your code window too cluttered. In Procedure View, you have to use the procedure list to navigate from procedure to procedure. On the other hand, Full Module View shows the entire contents of the module. I usually leave the window in full view and navigate using both normal window scrolling and the procedure list. I have no particular reason for doing so other than that's how all my other development tools work and it's how I'm used to working with code.

You write VBA *statements* (instructions) in code windows, as illustrated in [Figure 2-2](#). As you can see, the code illustrated in [Figure 2-2](#) is indented in blocks. This is not a VBA requirement, so you're free to space and indent your code as you desire. However, it's considered good practice to indent logical



blocks of code, making the code more readable. You'll see this practice throughout all the code presented in this book.

I'll talk more about the VBA language and statements throughout the rest of this chapter, and the remainder of this book for that matter, so I won't get into code details in this recipe. I do want to mention that when you're typing code in a code window, VBA will automatically complete some code for you. For example, if you type a function declaration, VBA will automatically complete an `End Function` statement for you. VBA has other code completion features that you'll encounter as you use it. These are designed to save you time and help reduce errors.

I should also mention that it's a good idea to save your work often while working in the code window. Simply press the Save button on the VBA IDE toolbar, select File → Save..., or press Ctrl-S. Saving in either the VBA IDE or Excel saves both your VBA project and its associated workbook.

To return to Excel from the VBA IDE, press the Excel icon in the VBA IDE toolbar. Or you can switch to Excel via the normal application switching methods in Windows; e.g., press the Excel button on the Windows Start bar at the bottom of the screen, click directly on the Excel window if it's visible, or use Alt-Tab.

## See Also

This recipe is merely an overview of the VBA IDE. Granted, it isn't that complicated an application, but there are other aspects to its interface that I have not addressed. The VBA help guide contains a complete reference entitled "Visual Basic User Interface Help" that you may consult for more details on the interface. I can't say it's the most captivating reading, but it's there if you need it. Press F1 in the VBA IDE to open the VBA help document and then press the Contents tab. The first topic is the user interface reference. As I mentioned in [Chapter 1](#), I think the best way to learn the VBA IDE is to explore it and try things out. You'll also learn more as you go through other recipes in this book.

[← PREV](#)

## Recipe 2.2. Writing Functions and Subroutines

### Problem

You want to write VBA code to perform some task or calculation but you're not sure where to begin.

### Solution

Open the VBA IDE, create a code module for your workbook, and then start writing your custom procedures (functions and subroutines).

### Discussion

When working with Excel and VBA, you write custom code in functions and subroutines. Unlike in traditional application programming, we're not going to write a main program from which we manage the application loop, making calls to other functions and subroutines, and so on. We will, however, make heavy use of custom VBA procedures and call them from other VBA procedures. Moreover, we're going to call our VBA procedures from within Excel itself. Therefore, we're sort of attaching our custom code to Excel's main program and invoking our code within cell formulas or in response to certain events (for example, when a user presses a button). Essentially, the VBA procedures you write are extensions of Excel. This extensibility is what makes Excel, in my opinion, such a powerful computation tool.

You can actually write a subroutine and call it `main`, using it as a starting point for subsequent code and calls to other procedures.

### Subroutines

VBA subroutines have the basic form shown in [Example 2-1](#).

### Example 2-1. VBA Subroutine

```
Public Sub MySubroutineName(Param1 As Integer, Param2 As Double)
    ' Your code goes here...
End Sub
```

Subroutines start with a scope qualifier, `Public` or `Private`, followed by the `Sub` keyword, followed by the subroutine name, which is then followed by an optional parameter list. After the declaration comes the body of the subroutine, which is a collection of code statements that you supply. A subroutine definition is closed with `End Sub`. Let's consider each part of the declaration in more detail.

The scope qualifier specifies whether the subroutine has *global* or *local* scope. If you specify `Public`, then the subroutine has global scope and can be called from any code module. On the other hand, if you specify `Private`, then the subroutine has local scope and can only be called from the code module within which it is defined. If you don't specify anything, VBA assumes `Public`.

The `Sub` keyword is mandatory and lets VBA know you are indeed declaring a subroutine procedure. Note that VBA will show keywords in a different color from that used for your code statements. I have my version set to display keywords in blue and code statements in black. This is the default, but you can change this if you'd like. Select `Tools` → `Options...` from the main menu bar and select the `Editor Format` tab to customize the editor.

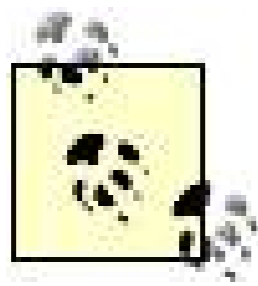
After the `Sub` keyword, you must supply a subroutine name. I recommend you use descriptive names wherever possible; it doesn't matter if the names are long. Gone are the programming days of trying to compress names down to a meager eight characters! Just keep the following naming rules in mind when making up subroutine names:

- The name must start with a letter.
- Names can't include the characters `@`, `&`, `$`, `#`, or `!`. They can't include spaces or periods either.
- Names can be up to 255 characters in length.
- You should avoid names that are the same as built-in VBA functions and keywords.
- You can't reuse names in a single code module. For example, you can't give two different subroutines the same name if they reside in the same code file.

By the way, these naming rules apply to all names in VBA, including function names, variable names, constants, and parameters. I should also point out that VBA is not case sensitive. It is, however, considered good style to use both upper- and lowercase characters to make your code more readable. For example, `MySubroutineName` is more readable than `mysubroutinename`. To make things even clearer, you could use the underscore character (e.g., `My_Subroutine_Name`).

Parameters are optional for subroutines. If you don't include parameters you still have to include the open and closing parenthesis (i.e., `(` and `)`). Parameters may be passed into the subroutine and used in your code. A parameter list must include the parameter name followed by a type specifier in the form `ParamName As DataType`. You must separate the parameter name from the data type specifier using the `As` keyword. Separate multiple parameters in the list with commas as illustrated in [Example 2-1](#). Data types are types such as `Integer`, `Double`, `Boolean`, and `Byte`. See [Recipe 2.3](#) for more

information on data types .



By default, arguments passed into procedures as parameters are passed by reference. This means the actual value of the argument is not passed to the function; rather, a *pointer* to the argument is passed. A pointer is a data type that stores the memory address of the object being pointed to. This gives the procedure direct access to the argument, which means the procedure can change the value of the argument. This may be what you want, but you need to be careful not to inadvertently change the value of an argument. To pass an argument by value, use the `ByVal` keyword in the parameter list. For example, to force `Param1` to be passed by value in [Example 2-1](#), you would write `ByVal Param1 As Integer`. Doing so ensures the argument passed in as `Param1` will be preserved irrespective of how its value is used in the procedure. The drawback of passing parameters by value is performance degradation, since the actual value has to be copied in memory when the procedure is called.

After you've defined a subroutine, you can call it from other places in your code simply by writing the name of the subroutine followed by any parameters. Alternatively, you may use the `Call` keyword. Both approaches are illustrated in [Example 2-2](#).

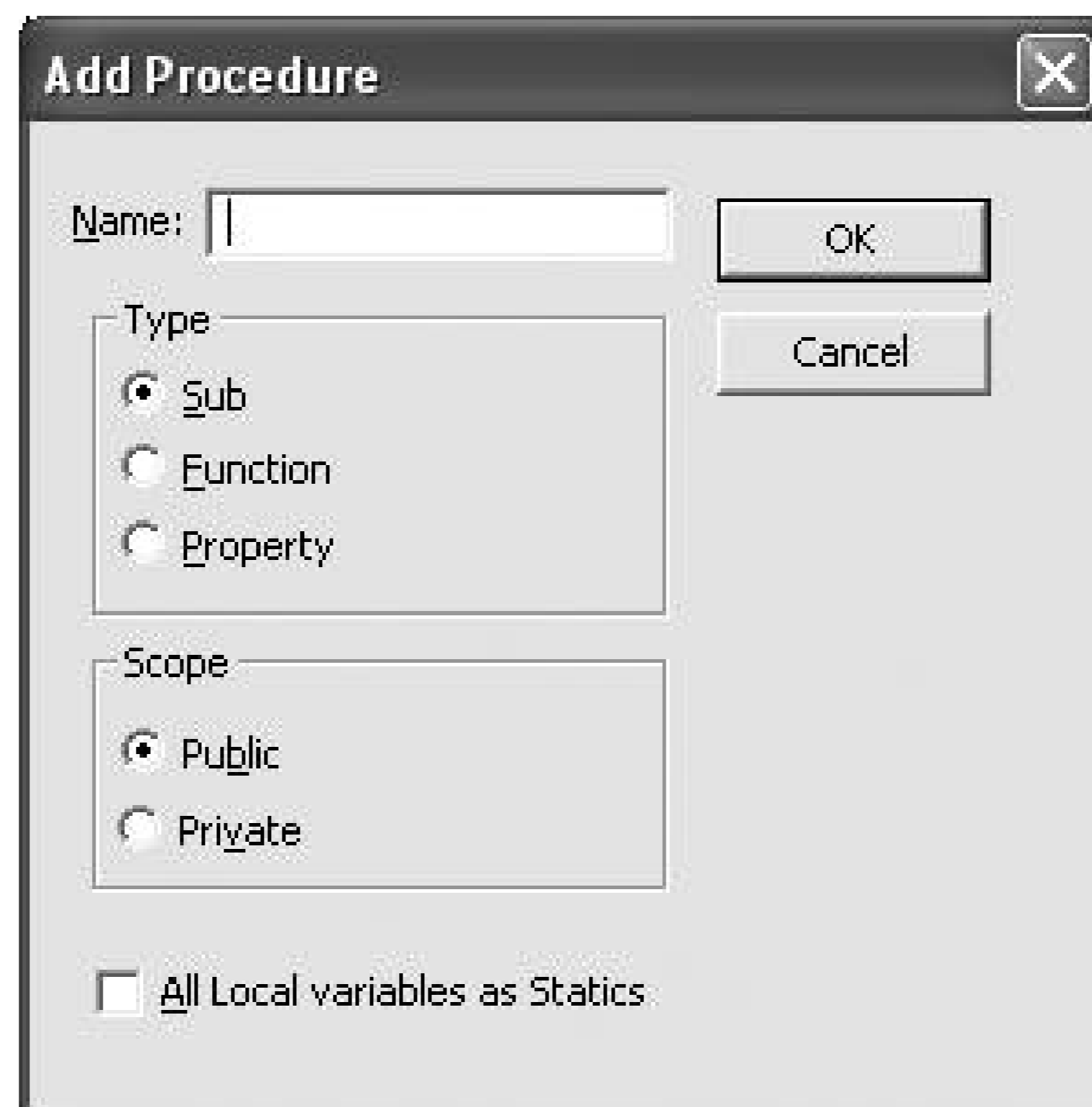
### Example 2-2. Calling subroutines

```
MySubroutineName 4, 2.87
Call MySubroutineName(4, 2.87)
```

When using the `Call` approach, you must enclose the parameter list in parentheses. In many cases you won't actually call subroutines directly in code. Instead, you'll assign them to buttons or other controls included in a spreadsheet (see [Recipe 9.3](#)). Note that you cannot call subroutines from cell formulas. You can do so with functions, as discussed in a moment.

To actually add a new subroutine to a code module, you can either use the Add Procedure dialog box or simply start typing the subroutine in a code window. Personally, I find it more efficient to just type the subroutine directly. The Add Procedure dialog box, shown in [Figure 2-3](#), allows you to specify what type of procedure you want to add, along with a few options.

Figure 2-3. Add Procedure dialog box



To access the Add Procedure dialog box, select Insert      Procedure... from the main menu bar.

## Functions

VBA functions are similar to subroutines except for two key differences. First, functions are declared using the `Function` keyword instead of `Sub`. Second, functions actually return a value and can therefore be used within expressions. [Example 2-3](#) shows an example function declaration, along with how the function may be called as part of an expression.

### Example 2-3. Example VBA function

```
Public Function Calc_q(y1 As Double, y3 As Double) As Double
    Calc_q = 1 / ((Abs(y3 - y1)) ^ 0.74)
End Function
```

•  
•  
•

```
x = 1.24 * Calc_q(3.75, 2.87) + y1
```

•  
•  
•

In this example, the function is named `Calc_q` and it takes two `Double` parameters. Since this is a function that returns a value, you need to specify the data type of the value returned by the function. Thus, you have to include the `As DataType` qualifier at the end of the function declaration. In this example, the return type is a `Double`. The example statement setting the value for the variable `x` shows how the `Calc_q` function may be included in an arithmetic expression. Moreover, you can include function calls in cell formulas, whereas you can't with subroutines.

You can, of course, include other statements in the body of a function, in addition to the single statement illustrated in [Example 2-3](#). The important thing to remember is that the function exits, returning a value, on a code line where the function name is set to some value.

The naming rules that apply to subroutines also apply to functions.

Adding a function is similar to adding a subroutine. You can either type it in directly as I prefer to do, or you can use the Add Procedure dialog.



## Recipe 2.3. Working with Data Types

### Problem

You are unfamiliar with VBA data types and their syntax.

### Solution

Visual Basic supports the usual data types you'd expect to see in any programming language. [Table 2-1](#) summarizes some of the VBA data types that I use most often for the sorts of calculations covered in this book.

Table 2-1. VBA data types

Data type	Storage	Values
<code>Boolean</code>	2 bytes	<code>True</code> or <code>False</code>
<code>Byte</code>	1 byte	0 to 255
<code>Integer</code>	2 bytes	-32,768 to 32,767
<code>Double</code>	8 bytes	-1.79769313486231 E 308 to +1.79769313486231 E 308
<code>String</code>	Variable	ASCII and special characters

### Discussion

There are other data types, including user-defined data types, supported by Visual Basic. For the most part, the types shown in [Table 2-1](#) are sufficient for the sorts of programming tasks discussed throughout this book.

I mentioned in [Chapter 1](#) that Excel performs calculations using double precision by default. To ensure double-precision calculations in Visual Basic, be sure to use the `Double` data type. When writing expressions using literal numeric values, you can use Visual Basic type-declaration characters to let Visual Basic know what data type you want it to consider when evaluating your expression. For example, if you have an expression that includes an operation like `2*3`, Visual Basic will interpret these numbers as integers. However, if you write `2#*3#`, then Visual Basic will treat each number as a `Double`. The `#` character is the type-declaration character for `Double` data types. In fact, the VBA IDE will help you out here. If you type a number like `3.0`, VBA will replace it with `3#` automatically. I say

this now so you're not surprised when your expressions change before your eyes as you type them.

There are other type-declaration characters recognized by Visual Basic. For example, the `%` character indicates `Integer`, and the `$` character indicates a string.

Speaking of strings, while this book focuses mainly on scientific and engineering calculations where doubles and integers are more prevalent, strings are sometimes useful for formatting output. Such output may go to spreadsheet cells in the form of text or datafiles written directly to disk.





## Recipe 2.4. Defining Variables

### Problem

You're ready to write some code and would like to declare some variables but are not sure of the proper VBA syntax for doing so.

### Solution

Use the `Dim` statement to declare a variable, as illustrated in [Example 2-4](#).

#### Example 2-4. Declaring variables using the `Dim` statement

```
Dim a As Double
Dim n As Integer
```

`Dim` statements are of the general form `Dim VariableName As DataType`, where `Dim` and `As` are VBA keywords, `DataType` is one of the VBA data types as discussed in [Recipe 2.3](#), and `VariableName` is a name you supply that conforms to the naming rules discussed in [Recipe 2.2](#).

### Discussion

You can use `Dim` statements to declare local variables within procedures by including the `Dim` statements in the body of the procedure. Such local variables can be used only by the procedures within which they are declared. To declare global variables that have module-level scope, place the `Dim` statements at the beginning of the code module before any procedure declarations. Such global variables can be used by any procedure in the module within which the variables are declared.

To declare global variables that can be used by any code module within a project, use a `Public` statement like this `Public VariableName As DataType`.

You can also declare module-level variables using the `Private` statement, but this is the same as using the `Dim` statement.

You can declare multiple variables on a single line by separating the variable declarations with commas, as shown in [Example 2-5](#).

### Example 2-5. Multiple declarations

```
Dim a As Double, Dim b As Double, Dim c As Double  
Dim d, e, f As Double
```

The first line in [Example 2-5](#) declares three `Double` type variables, `a`, `b`, and `c`. It's tempting to think that the second line in [Example 2-5](#) declares three more `Double` type variables, `d`, `e`, and `f`, but this is not the case. The fact is, only variable `f` in the second line is a `Double`; `d` and `e` are of `Variant` type. So what gives?

VBA requires an explicit data type to follow each variable name; otherwise, it assumes a `Variant` type. So to make `d` and `e` `Doubles`, you'd have to put `As Double` after *each one*. Given this requirement, in my opinion there isn't much use in putting multiple declarations on a single line in VBA. You're better off putting each `Dim` statement on a separate line to make your code more readable.

By now you're probably wondering what a `Variant` data type is. A `Variant` is a generic data type that can represent any of the other explicit data types. VBA determines the appropriate type to use given the context within which a `Variant` variable is used. While this may seem like a flexible way to use variables, it's less efficient in terms of code execution performance and makes your code less readable. I recommend using explicit data types as much as possible and using `Variants` only when absolutely necessary (for example, when a built-in VBA procedure or function requires `Variant` parameters).

The VBA IDE provides a useful way to quickly find the declaration of any variable in your code. Say you're reading through a long procedure and come across a variable whose type you've forgotten or which isn't clear given the context in which it is used. You can select the variable name in your code and use the shortcut Shift-F2 to automatically jump to the statement that declares the variable.

[← PREVIOUS](#)

## Recipe 2.5. Defining Constants

### Problem

You'd like to declare some constants and need to know the correct VBA syntax.

### Solution

VBA constants are declared using the syntax `Const Name As DataType = Value`, as illustrated in [Example 2-6](#).

#### Example 2-6. Declaring constants

```
Const MyConstant As Integer = 14  
  
Public MyConstant2 As Double = 1.025
```

### Discussion

Just as with variables, you can declare constants with local or global scope. Constants declared within procedures have local scope; that is, they are available only within the procedure. Constants declared outside of any procedures at the beginning of a code module have module-level scope and can be used by any procedure within the module. As with variables, you can use the `Public` keyword, as illustrated in [Example 2-6](#), when declaring constants at the beginning of a code module to make the constant available to all procedures in the project.

A constant is just that, constant. Once a constant is declared with its value set, you cannot change its value. Constants are useful for defining descriptive names to commonly used values such as scientific or engineering coefficients. For example, you could define a constant to represent  $\pi$  and use the constant `Pi` throughout your calculations rather than write out the numerical value.

Excel includes a built-in worksheet function `PI` that returns the value of  $\pi$ . You can actually use that function in your VBA code by accessing the function through the `WorksheetFunction` object, as in `WorksheetFunction.Pi`. See [Recipe 2.13](#) for more information on Excel Objects.





## Recipe 2.6. Using Arrays

### Problem

You don't know the VBA syntax for declaring arrays.

### Solution

You declare arrays just as you do variables, except you specify the size of the array in the declaration, as shown in [Example 2-7](#).

#### Example 2-7. Declaring arrays

```
Dim y(1 To 4) As Double
Dim x(4) As Double

Dim M(1 To 8, 1 To 8) As Double
Dim N(8, 8) As Double
```

Arrays are declared with their size specified in parentheses following the array name.

### Discussion

When you use the `To` keyword in an array size specification, the first number is the lower bound and the second number (the one following `To`) is the upper bound of the array. You can skip the `To` keyword and simply put the desired size (i.e., the number of array elements) in parentheses following the array name. In this latter approach, the first array index, the base, is 0 unless you use the `Option Base` statement.

Use the `Option Base` statement at the beginning of a code module (before your procedures) to specify the default base for arrays. For example, `Option Base 1` sets the base to 1, while `Option Base 0` sets the base to 0.

You can declare multidimensional arrays by separating each dimension's size, or upper and lower bounds, by commas in the array declaration, as shown in the last two lines of [Example 2-7](#). VBA supports multidimensional arrays of up to 60 dimensions.

To actually access an array's elements in your code statements, write the array name followed by the index to the array element enclosed in parentheses. [Example 2-8](#) shows how to access array elements.

### Example 2-8. Accessing array elements

```
y(3) = 2.983  
M(1, 2) = 4.321
```

Separate the array indices by commas for multidimensional arrays.

### See Also

The array declarations discussed here are *static* declarations in that the size of the arrays is specified in the `Dim` statement. In some special cases, you may not know the required size of an array and may need to allocate memory for the array dynamically. In such cases you can declare the array as discussed earlier but leave the parentheses following the array name blank (include the parentheses, just don't put anything between them). Then in your code you'll have to dynamically allocate memory for the array of the size you desire. Use `ReDim` statements to set, or reset, the size of an array. For the most part, I'll stay away from dynamically allocated arrays except in special circumstances. You can learn more about dynamic arrays by reading the VBA help topic "Declaring Arrays." Press F1 in the VBA IDE to access the VBA help guide.

## Recipe 2.7. Commenting Code

### Problem

You want to follow good coding practice by documenting your code with comments.

### Solution

There are two ways to comment code in VBA: use the `Rem` keyword or the apostrophe (`'`) character to denote a comment, as illustrated in [Example 2-9](#).

#### Example 2-9. Comments

```
Public Sub DoRungeKutta( )
    ' Local variables

    Dim y(1 To 4) As Double
    Dim k1(1 To 4) As Double
    Dim k2(1 To 4) As Double
    Dim k3(1 To 4) As Double
    Dim k4(1 To 4) As Double
    Dim a As Double
    Dim n As Integer
    Dim dt As Double
    Dim time As Double

    Rem Initialize Variables

    y(1) = 29.8613
    y(2) = 0
    y(3) = 30.0616
    y(4) = 0
    n = 5000
    dt = 5 / n: Rem calculate time step
    time = 0 ' set starting time to zero
    .
    .
    .

End Sub
```



[Example 2-9](#) shows a portion of a custom subroutine to perform numerical integration using the Runge-Kutta method. This code contains four comments, using both commenting approaches.

You may place comments on the same line as code statements; however, if you use `Rem` for comments when doing so, you must precede the `Rem` keyword with a colon (`:`). You need not use the colon when using the apostrophe comment indicator.





[← PREV](#)

## Recipe 2.8. Spanning Long Statements over Multiple Lines

### Problem

Some of your code statements are very long and you'd like to make them more readable by splitting them over multiple lines.

### Solution

Use the underscore character (`_`) to break statements over multiple lines, as shown in [Example 2-10](#).

#### Example 2-10. Spanning multiple lines

```
k2(1) = dt * dyldt(y(1) + k1(1) / 2#, y(2) + _  
        k1(2) / 2#, y(3) + k1(3) / 2#, y(4) + _  
        k1(4) / 2#)
```

You must include a space between the last character of a line and the underscore character.

## Recipe 2.9. Using Conditional Statements

### Problem

You need to make some logical tests and branch your code accordingly.

### Solution

Use `If/Then` statements as illustrated in [Example 2-11](#).

#### Example 2-11. If statements

```
If (time = 32000) then
    ' Statements go here
End If

If (MyCondition = True) Then
    ' Statements
Else
    ' More statements
End If

If (count < 10) Then
    ' Statements
ElseIf (count < 20) Then
    ' Statements
ElseIf (count < 30) Then
    ' Statements
End If
```

`If` statements start with the `If` keyword followed by a logical expression, which is followed by the `Then` keyword. If the logical expression is true, the code between the `If` statement and the `End If` statement gets executed. You can also use the `Else` keyword to define a block of code to execute if the logical expression is false. Further, you can use the `ElseIf` keyword to test other conditions in a series of `If-Then-ElseIf-Then` statements.

You'll notice I enclose the logical expressions being evaluated in parentheses. VBA does not require this; I do so only because it is more readable to me. (It's also kind of a habit I picked up from

programming in C.)

## See Also

Sometimes you may encounter situations where you have a lot of conditions to test and you find yourself with long, strung out If-Then-Else-Else-Then statements. In these situations, you may find VBA's `Select Case` statement more appropriate. `Select Case` statements are handy when you're testing for different values of a numerical expression or text. See the VBA help topic "Select Case Statement" for more information.



## Recipe 2.10. Using Loops

### Problem

You need to loop through some code and don't know the VBA syntax for loops.

### Solution

Use a `For` loop like the one shown in [Example 2-12](#).

#### Example 2-12. For loop

```
For i = 1 To n
    ' Statements go here
Next i
```

`For` loops are sometimes called `For Next` loops because you use the `Next` statement to bound the code over which you want to loop. Also, the `Next` statement increments the counter variable, `i` in this case, by 1 each time through the loop. In this example, the loop starts with the counter equal to 1 and ends when the counter equals the value stored in the variable `n`. You can hardcode start and end values or use variables.

### Discussion

To increment the counter in a `For` loop by a something other than 1, use the `Step` keyword as shown in [Example 2-13](#).

#### Example 2-13. For loop

```
For i = 1 To n Step 2
    ' Statements go here
Next i
```

In this case, the counter variable gets incremented by 2 each time through the loop. You may also use a negative step value to count down from a start value to an end value.

Sometimes it's necessary to exit a `For` loop before the counter actually reaches the end value. To exit a `For` loop, use the `Exit For` statement. You can put this in a conditional statement to exit only under specific conditions, as shown in [Example 2-14](#).

### Example 2-14. Exit For statement

```
For i = 1 To n
    ' Statements go here

    ' Test an exit condition
    If time > 32000 Then
        Exit For
    End If

    ' May have more statements here

Next i
```

When an `Exit For` statement is encountered, code execution will jump from the `Exit For` statement to the first statement after the `Next` statement of the loop.

For most of the calculations discussed in this book that require looping, plain old `For` loops will work just fine. In fact, I use `For` loops probably 90% of the time, but in some special cases `Do` loops are a better choice.

`Do` loops are useful when you're not actually iterating a counter but are testing some condition (a logical statement) instead. In VBA there are actually four different ways of writing `Do` loops, as shown in [Example 2-15](#).

### Example 2-15. Four different Do loops

```
Do While (dtime < 32000)
    ' Statements go here
Loop

Do
    ' Statements go here
Loop While (dtime < 32000)

Do Until (dtime > 32000)
    ' Statements go here
Loop

Do
    ' Statments go here
Loop Until (dtime > 32000)
```

The first two loops in [Example 2-15](#) are `Do While` loops, and the last two are `Do Until` loops.

`Do While` loops execute the loop as long as the condition following the `While` keyword is `True`. In this example, the condition is enclosed in parentheses. Note the difference between the two `Do While` loops. In the first case the condition is tested before the code in the loop is executed, while in the second case the code in the loop is executed once before the condition is tested. This means that in the first case it's possible the code within the loop may not get executed, while in the second case the code within the loop will get executed at least once.

`Do Until` loops execute the loop as long as the condition following the `Until` keyword is `false` (that is, until the condition becomes `true`). As with `Do While` loops, there are two ways to write `Do Until` loops. In the first case shown in [Example 2-15](#) the condition is tested before the code within the loop is executed, while in the second case the code in the loop is executed before the condition is tested.

As with `For` loops, you can exit a `Do` loop anytime before the looping condition is satisfied, using an `Exit Do` statement.

## See Also

VBA also has a `For Each` loop that is used to loop over collections of objects. These loops are useful when interfacing with Excel objects, such as cells. For example, you can use a `For Each` construct to loop over a range of selected cells in a spreadsheet without actually knowing the number of cells in the range. See the help topic "Using For Each...Next Statements" in the VBA help for more information on these loops. Press F1 in the VBA IDE to access the VBA help guide.

## Recipe 2.11. Debugging VBA Code

### Problem

You want to know how to debug your code in VBA.

### Solution

Use VBA's built-in debugger by accessing the Debug menu in the VBA IDE or working with the Debug toolbar (select View → Toolbars → Debug to show the Debug toolbar).

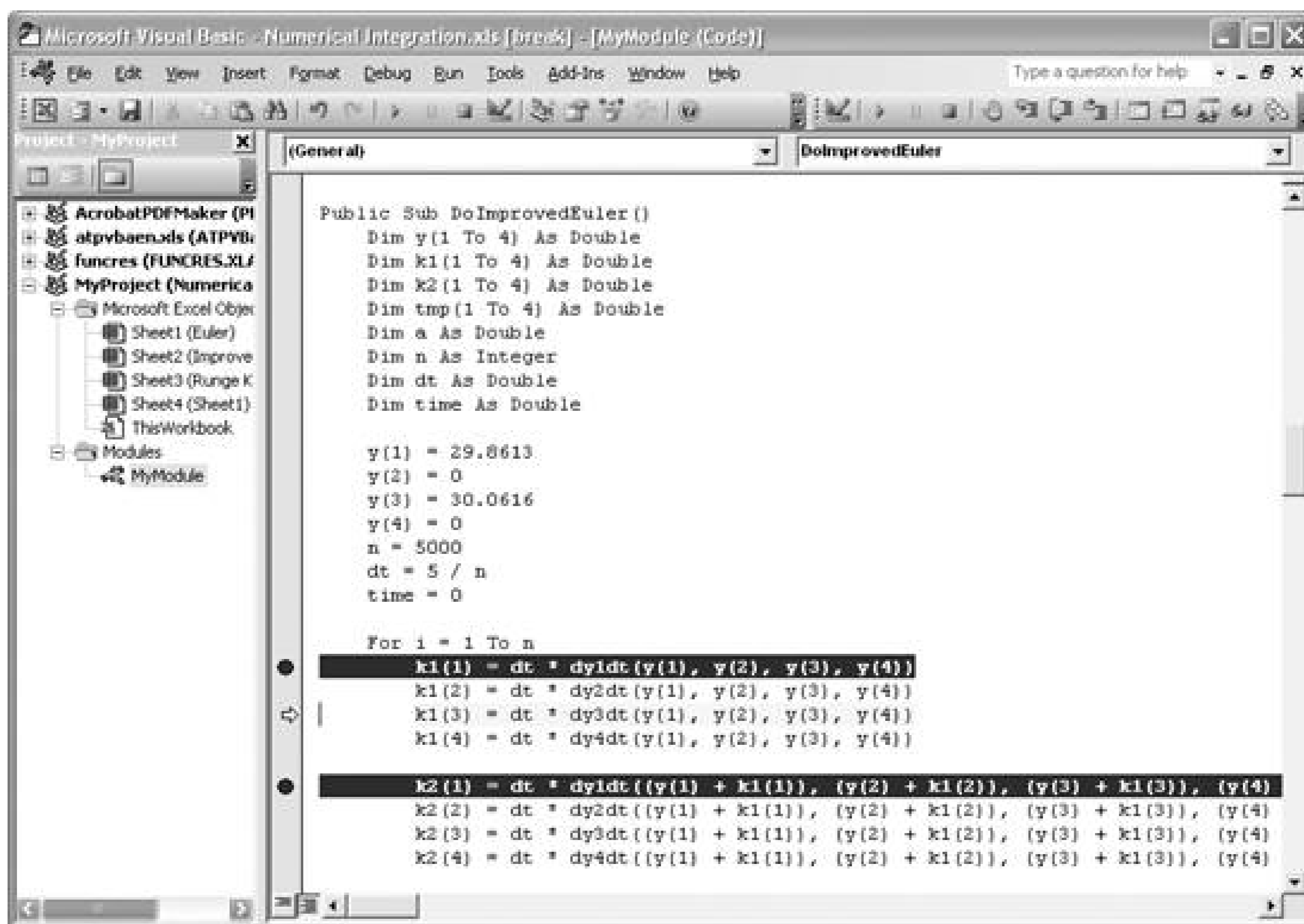
### Discussion

When you are writing code, VBA will automatically check for syntax errors as soon as you press Enter following each code line. If VBA finds no syntax errors, your code will be accepted and it will be highlighted to denote comments and keywords. If VBA finds an error, it will immediately display an error message box with information on the error. It will also highlight your code statement and indicate which part is causing the error. You can then edit the statement in the code window until you get it right.

Alas, VBA cannot automatically detect logical errors in your code. That technology is a long way off for any computing language. In the meantime, we do have integrated debuggers to help us find and correct logical errors. Logical errors are the sort of error where your code, though syntactically correct, does not do what you intend. For example, you might have written an integration routine based on the Improved Euler method and it does not appear to work correctly; e.g., it's giving you garbage results. Such poor results could be due to a time step that's too small, an incorrect formula in your code, or a particularly nasty function that you're trying to integrate.

For tracking down and fixing logical errors, the debugger included with the VBA IDE is an invaluable tool. Let's say you're trying to debug the procedure shown in [Figure 2-4](#).

Figure 2-4. Debugging a procedure



The first thing you should do is display the Debug toolbar by selecting View → Toolbars → Debug from the main menu bar. The Debug toolbar is shown in [Figure 2-4](#) in the upper-right side of the window. [Figure 2-5](#) shows a close-up of the Debug toolbar.

Figure 2-5. Debug toolbar

The next thing you need to do is set some *breakpoints* in the procedure you're trying to debug. Breakpoints are lines in your code that you tag to let the debugger know you want to stop at that line while debugging, in order to inspect what's going on during execution. To set a breakpoint, simply click in the margin to the left of the code window adjacent to any executable line upon which you'd like to break. The breakpoints will be highlighted as in [Figure 2-4](#). The two breakpoints shown there have a small circle in the margin and a dark rectangle highlighting the code line where the debugger will break.

Now you can start the debugger and inspect what's happening in your code. Place the cursor in the procedure you want to debug and press the Continue button on the toolbar (see [Figure 2-5](#)) to start the debugger. The debugger will execute your procedure and break when it encounters a breakpoint.



Once a breakpoint has been reached, you can take several actions to inspect your code and follow its execution. One thing you might do is step line by line through your code to follow the execution path. To step through your code, you can press the Step Into, Step Over, or Step Out button on the Debug toolbar (see [Figure 2-5](#)). Step Over will execute the current code line and stop on the next line. The current line, the one upon which execution has paused, is also highlighted with an arrow icon in the margin (see [Figure 2-4](#)). Step Into is useful if you want to follow code execution line by line into other procedures that may be called from within the one you're trying to debug. Step Out is useful for jumping out of a procedure and getting back to the calling procedure to continue line by line stepping.

While stepping through your code, it's helpful to actually inspect the values of variables used in your algorithms. This is how you can check to see if things are being calculated correctly or where things may be going wrong. The easiest way to inspect a value is to hold the mouse cursor over a variable name and wait a second or so for VBA to display a small rectangle showing the current value of the selected variable.

You may also display the Locals window to show all local variables within the procedure you're debugging. Press the Locals button on the Debug toolbar (see [Figure 2-5](#)) to open the Locals window. The Locals window will show not only the variables, but their values as well. This is very handy when stepping through to monitor changes in values of some key variables.

Alternatively, you can open the Watch window (use the toolbar button shown in [Figure 2-5](#)) to monitor select variables. Initially the Watch window will be empty. To add a variable to watch, highlight or place the cursor on the desired variable in the code window, and then right-click to reveal a pop-up menu. Select Add Watch to add the variable to the Watch window.

With either of these windows open, you can continue to step through your code, monitoring how key variables change. Watching variables this way and in the context of executing statements might illuminate problems in your code.

VBA also includes a window called the Immediate window. This window can be used for typing in commands or writing output using the `Print` statement. When you are debugging, the Immediate window is useful for writing debugging text so you can trace the execution of the code or follow the value of a variable as the procedure executes. To display the Immediate window, press the toolbar button shown in [Figure 2-5](#). You can send output to the Immediate window using the `Print` method. When debugging, you have to use the Debug object's `Print` method, as in `Debug.Print Str(k1(1))`. In this example, the value stored in element 1 of the `k1` array is converted to a string, using the `Str` function, and then printed to the Immediate window.

I use the Immediate window in this way for long procedures or complex programs that are simply too long or complicated to step through line by line. I strategically place print statements in the code that will send write messages informing me of the status of certain things of interest. Such things may include variable values or whether or not a particular line or block of code was executed.

I often use the `Beep` statement to play a short tone during execution of a procedure. I'll place it strategically if I want to determine whether a particular block of code is reached. For quick checks, using `Beep` is faster than typing in a message to display via the `Print` method.

At any point during debugging, you can press the Continue button to continue code execution from

the current line. Execution will proceed until another breakpoint is encountered or the end of the procedure is reached. You can also terminate a debugging session by pressing the Reset button (see [Figure 2-5](#)).



## Recipe 2.12. Exploring VBA's Built-in Functions

### Problem

You need to get familiar with VBA's built-in functions and use them in your code, but you're not sure where to begin.

### Solution

Check out the Visual Basic Language Reference in the VBA help guide. There's a section in the language reference entitled "Functions," which includes an alphabetically organized list of built-in functions. For your convenience, I've summarized some of the more commonly used functions for scientific and engineering calculations in [Table 2-2](#).

Table 2-2. Some useful VBA functions

Function	Comment
<code>Abs(number)</code>	Returns the absolute value of the number argument. The data type returned is the same as that of the argument.
<code>Atn(number)</code>	Returns the arctangent of the number argument. The returned data type is a <code>Double</code> . The result is in radians from $-\pi/2$ to $\pi/2$ .
<code>Cos(number)</code>	Returns the cosine of the number argument, which represents an angle in radians. The return type is a <code>Double</code> in the range from -1 to 1.
<code>Exp(number)</code>	Returns e raised to the <code>number</code> power. The return type is a <code>Double</code> .
<code>Int(number)</code>	Returns the integer part of <code>number</code> , where <code>number</code> is a decimal number. The return type is an <code>Integer</code> . <code>Int</code> essentially truncates the number. To round a number use the <code>Round</code> function.
<code>Log(number)</code>	Returns the natural logarithm (to base $e$ ) of <code>number</code> . The return type is a <code>Double</code> . To calculate the base 10 log of a number, use the expression <code>Log(number) / Log(10)</code> .
<code>Round(number)</code>	Returns <code>number</code> rounded to the nearest integer. The return type is an <code>Integer</code> .
<code>Round(number, n)</code>	Returns <code>number</code> rounded to the nearest decimal place specified by <code>n</code> . The return type is a <code>Double</code> .
<code>Sgn(number)</code>	Returns the sign of <code>number</code> . <code>Sgn</code> returns -1 if number is negative, 1 if number is positive, and 0 if <code>number</code> equals zero.

Function	Comment
<code>Sin(number)</code>	Returns the sine of <i>number</i> , which represents an angle in radians. The return type is a <code>Double</code> in the range from -1 to 1.
<code>Sqr(number)</code>	Returns the square root of <i>number</i> . The return type is a <code>Double</code> . Taking the <code>Sqr</code> of a negative number causes an error.
<code>Tan(number)</code>	Returns the tangent of <i>number</i> , which represents an angle in radians. The return type is a <code>Double</code> .

## Discussion

You can call VBA functions in your own expressions, as illustrated in [Example 2-16](#).

### Example 2-16. Calling VBA functions

```
num1 = 1 - cos(1.571)
num2 = 1 - Atn(a/b)
```

You need not hardcode arguments as shown in the first statement. You can use an expression like that shown in the second statement as an argument to a function so long as the expression evaluates to an acceptable value permitted by the particular function.

As discussed in [Recipe 1.10](#), Excel has a great many useful functions that you can use in conjunction with the VBA functions discussed here. You can actually call Excel's built-in functions from within a VBA procedure. See [Recipe 2.13](#) for more information on calling Excel functions from VBA code. You have to be a little careful when working with both VBA and Excel functions. VBA and Excel both have some functions that perform the same (or a similar) operation, but in some cases functions with the same name are different, and this can be confusing. For example, both Excel and VBA have a `Log` function; however, the VBA version of the function returns the base  $e$  logarithm while the Excel version returns the base 10 logarithm unless you specify otherwise. This is just something to keep in mind when mixing functions.

Some programming languages have a function that raises a number to some exponent. VBA does not; however, you can use the `^` operator to raise a number to an exponent just as you do when raising a number to a power in an Excel formula. For example, the expression `num^3` raises the variable `num` to the third power. You can also include an expression as the exponent, as in `num^(k*1.3)`. See [Recipes 1.8](#) and [1.9](#) for more information.



## Recipe 2.13. Exploring Excel Objects

### Problem

You've heard of Excel objects (in fact I've already mentioned them several times) and want to know what they are and how they'll benefit you.

### Solution

Excel exposes its features and functionality through objects. Excel's object model, as it's called, includes nearly 200 objects that represent everything from cell ranges to charts to worksheets, workbooks, and the Excel application. Each object has properties that allow access and control over its attributes, as well as methods that expose the object's functionality. When working with VBA, you access Excel and the spreadsheets included in your project through Excel objects. There are far too many objects to discuss here. Further, not all of them are directly relevant to the sort of calculations in which we're interested in this book. The purpose of this recipe is to introduce key objects to you now, so we can use them in later recipes. The discussion to follow gives an overview of these key objects.

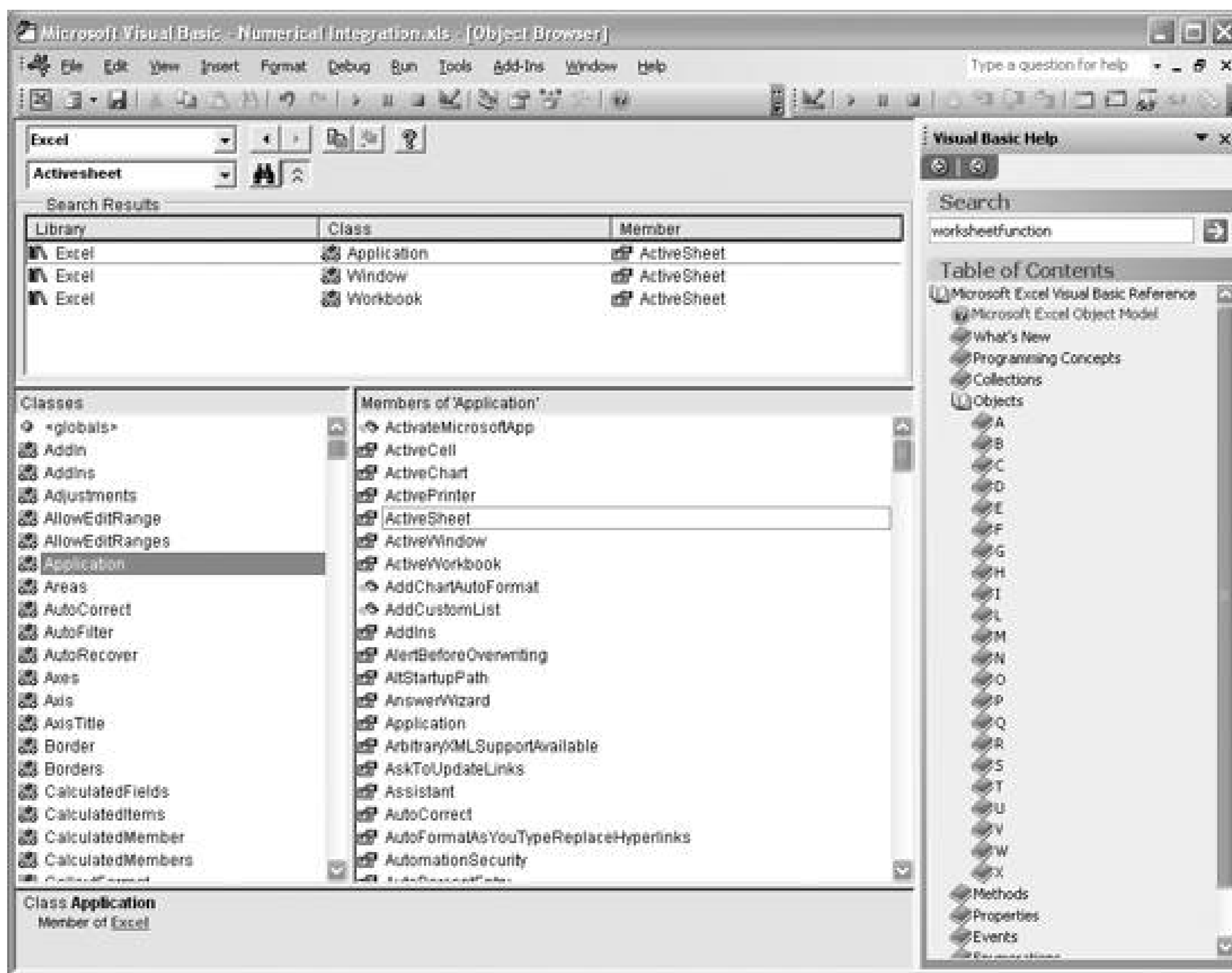
### Discussion

Before looking at specific objects, I want to discuss the Object Browser available in the VBA development environment. The Object Browser is a handy tool that allows you to browse through available objects, inspecting their properties and methods.

### The Object Browser

As I mentioned earlier, there are nearly 200 Excel objects that you can access using VBA. Each object also contains numerous properties and methods. Thus, it's very difficult to keep track of all the objects, their properties, and methods without the aid of a good reference book, the VBA help guide, or the Object Browser. The Object Browser is a window that you can open within the VBA IDE by pressing the F2 key or selecting View → Object Browser from the main menu bar. [Figure 2-6](#) shows the Object Browser opened in the VBA IDE.

Figure 2-6. Object Browser window



The Object Browser displays a list of objects. The upper-left drop-down listbox allows you to select the scope of the objects to view. For example, in [Figure 2-6](#) I've selected Excel; therefore, the Object Browser is displaying only those objects belonging to Excel. Other options include selecting only those objects in your project or only objects in the Microsoft Office Suite object model; there are also many others. We're going to focus on objects in Excel.

The list labeled Classes is the list of objects. You can select any object in this list to display more information on the object. For example, the very bottom panel of the Object Browser window is telling us that the selected Application object is a member of the Excel object model. Also, the list to the right of the Classes list displays all *members* of the selected object. Members include both properties and methods. You can select a member in this list and the bottom panel will display relevant information, such as whether it is a property or a method. If you press the button at the top of the Object Browser window with the question mark icon, a new window will open that contains context-sensitive help specific to the selected object or member. Thus the Object Browser is a convenient way to navigate the object reference guide, too.

The second drop-down listbox at the top of the browser window allows you to enter a term (e.g., a property name), and search for it in the object list. The search results panel will display a list of findings for your search term.

I encourage you to play around with the Object Browser and explore the myriad objects, properties, and methods included in the Excel object model. That said, for the most part we'll use only a small subset of these for our purposes.

## Application object

The `Application` object represents the Excel program. It's a top-level object that gives access to top-level features in Excel such as setting options, among others. Some of the most common uses I have for the `Application` object involve accessing an object that represents the currently active worksheet, accessing an object that represents the currently active cell, and controlling Excel's window updating while VBA procedures are running.

[Example 2-17](#) shows how to access the currently active worksheet in Excel by using the `ActiveSheet` property of the `Application` object. You can type the statement shown in [Example 2-17](#) directly in a VBA procedure.

### Example 2-17. ActiveSheet property

```
Application.ActiveSheet.Cells(1, 2) = time
```

You access members of objects by placing a period after the object name, followed by the property or method you want to access. In [Example 2-17](#) I'm accessing the `Cells` property of the `ActiveSheet`, which itself is a property of the `Application` object. Note that properties can be objects with members of their own. This sort of cell access is useful when writing calculation output directly into cells in a worksheet, in this case the currently active worksheet. Note the cell reference style used here. `Cells(1, 2)` represents the cell in row 1 column 2.

[Example 2-18](#) shows how to access the `ScreenUpdating` property of the `Application` object.

### Example 2-18. ScreenUpdating property

```
Application.ScreenUpdating = false
```

The `ScreenUpdating` property stores a `Boolean` value, indicating whether the Excel window should be updated while VBA code is running. In most cases I leave `ScreenUpdating` set to `True` so I can follow my VBA code as it runs. However, for some intensive calculations, especially with a lot of looping, keeping `ScreenUpdating` set to `true` can really slow things down. In those cases, it may be desirable to turn `ScreenUpdating` off (i.e., set it to `false`). You can set it back to `true` after the intensive block of code.

## Workbook object



The `Workbook` object represents an Excel workbook. You can control many aspects of workbooks, such as gaining access to the active sheet in a workbook through its `ActiveSheet` property, obtaining the name and path to the workbook, and saving the workbook.

I often use the `Workbook` object's `Save` method to save the workbook. For example, if I write a particular VBA procedure that makes changes to a workbook by writing calculation output to cells in a worksheet, then I'll have the VBA procedure also save the workbook. [Example 2-19](#) shows how to call the `Save` method.

### Example 2-19. Save method

```
Application.ActiveWorkbook.Save
```

In this example, I'm gaining access to the currently active workbook through the `Application` object's `ActiveWorkbook` property.

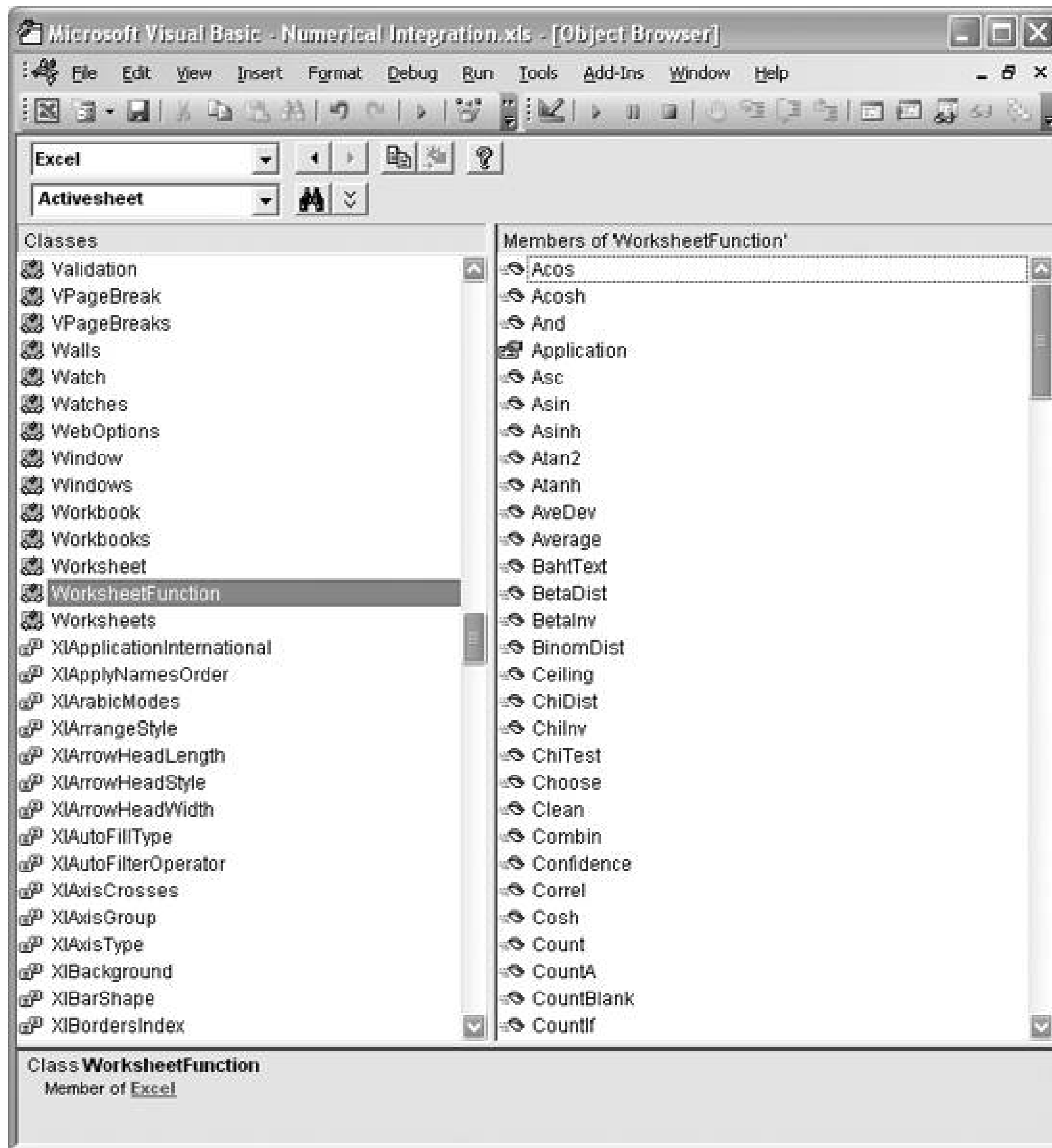
### Worksheet object

The `Worksheet` object represents an Excel worksheet. It provides access to sheet-level features and functionality such as copying and pasting, printing the sheet, saving the sheet, and showing or hiding the sheet. I use it primarily to access cells, as shown earlier in [Example 2-17](#) (in that example, `ActiveSheet` is a `Worksheet` object that refers to the currently active worksheet).

### WorksheetFunction object

As discussed in [Recipes 2.5](#) and [2.12](#), the `WorksheetFunction` object allows you to access Excel functions from within VBA. [Figure 2-7](#) shows a screenshot of the Object Browser with the `WorksheetFunction` object selected.

Figure 2-7. WorksheetFunction object in the Object Browser



In the figure, you can see a partial list of functions to which you have access through the `WorksheetFunction` object. [Example 2-20](#) shows a few ways to call Excel functions using the `WorksheetFunction` object.

Example 2-20. Calling functions through the `WorksheetFunction` object

```

area = WorksheetFunction.Pi * r ^ 2

a = WorksheetFunction.Acos(b)

Set TestRange = Worksheets("Sheet1").Range("A1:A5")
TestSum = WorksheetFunction.Sum(TestRange)

```

The first line uses the worksheet function `Pi` in an expression for calculating the area of a circle given its radius (stored in the variable `r`).

The second line calculates the arccosine of the variable `b` and returns the result in the variable `a`.

The last pair of lines sets a range consisting of the first five cells in column A of `Sheet1` in the open workbook and then uses Excel's `Sum` function to get the sum of the values stored in the range.

This last pair of lines uses the `Worksheets` collection, which is simply a collection of all the open `Worksheet` objects in a workbook. In this case, we access a specific sheet in the collection using the sheet's name. We then access the `Range` property of the selected sheet and set it to include cells A1 through A5.

I use the `Set` statement to set the `Range` object to a temporary variable called `TestRange`. This saves a bit of typing when using the range in multiple statements. Finally, `TestRange` is passed in as an argument to the `Sum` function, and the result is returned in the `TestSum` variable.

## Range objects

`Range` objects represent cell ranges. Earlier I showed you how to access cells using the `Cells` property of the `ActiveSheet` object. You can use `Range` objects as another means of accessing cells, as shown previously in [Example 2-20](#).

[Example 2-21](#) shows a few more ways to use the `Range` object to access cells.

### Example 2-21. Using Ranges

```

Set TestRange = Worksheets("Sheet1").Range("A1:A5")
TestSum = WorksheetFunction.Sum(TestRange)

Worksheets("Sheet1").Range("A7").Value = TestSum
Worksheets("Sheet1").Range("B7").Formula = "=Sum(A1:A5)"

```

This example includes the last two statements from the earlier example, with the addition of two new

statements. The second to last statement takes the sum calculated in the previous statement and puts it in cell A7. As a comparison, the final statement adds a cell formula to cell B7, which also calculates the sum of values in cells A1 to A5. When these statements are executed, both cells A7 and B7 should show the same value.

These are just a few common tasks that involve working with `Range` objects. Another very useful member of the `Range` object is the `GoalSeek` function, which we can use for iterative calculations (see [Chapter 9](#) for some examples).

## See Also

The summary presented in this recipe only scratches the surface of Excel's object model. Granted, we don't really need much in the way of Excel objects for our purposes; however, it's good to be familiar with the other objects in case you decide to expand your use of VBA and Excel; for example, if you decide you want to write and distribute your own add-ins.

For more information on Excel objects, you can always consult the VBA help guide. Look for the "Objects" topic under Microsoft Excel Visual Basic Reference in the table of contents. There you'll find an alphabetically organized object reference.

## Recipe 2.14. Creating Your Own Objects in VBA

### Problem

You know VBA is an object-oriented language and you'd like to create your own objects for use in other subroutines and functions.

### Solution

Create a new *Class Module* in VBA and add your object properties and methods.

### Discussion

Let me show you how to do this by way of example. First, for those of you not familiar with object-oriented lingo, an *object* is a code construct comprised of *properties* and *methods*. Properties are variables that describe the object in some way; they store attributes of the object. Methods are functions or subroutines that actually do something. They can manipulate properties, perform some action on given parameters, or do whatever other programming task you can devise. Objects provide a convenient way of packaging code that allows easy reuse.

In [Recipe 7.11](#), I show you how you might add support for manipulating vectors using VBA. You might also approach the same problem by writing a vector object to encapsulate all the properties and functionality of a vector. This way you can define vector type variables and perform computations with these variables. I'll show you step by step how to get started.

Open Excel and the VBA editor. In the VBA editor, select Insert → Class Module from the main menu bar. In VBA (as in some other languages) an object is called a *class*. Under the VBA project browser panel, you should see a folder called Class Modules and under that a new module called something like `class1`. Now press F4 (or select View → Properties Window from the main menu bar) to open the properties panel. Click on your new class and you should see its properties displayed in the properties panel.

For the moment you won't see much. You should, however, see a Name property. Click the field to the right of Name and type in `MyVector`. This is the new name for your vector object, which should also be reflected in the VBA project browser window.

Find the module now named `MyVector` in the project browser and double-click it to open a code window. It will be blank at first, so you need to add the properties and methods for this new vector class.

First, let's add some properties. At the very least, for a 2D vector, you know you need an `x` and a `y` component. So add the code shown in [Example 2-22](#).

## Example 2-22. MyVector properties

```
'MyVector Object Properties:  
Public x As Double  
Public y As Double
```

These look similar to local variable declarations. Notice the `Public` keyword, however. You need to use this keyword to make your properties available to you outside of this module. You can add any number of properties in this manner, but for now these two will suffice.

When you use a custom object in other code, you have to create the object by allocating memory for it. I'll show you how to do that later, but you should be aware that when an object is created, VBA will automatically call an initializing subroutine to initialize the object for you, setting its properties to default values. If you want your object's properties set to some specific default values, then you should create a custom initialize method.

Look at the top of the code window in your class module. You should see a drop-down listbox with the word `General` displayed in it. Click this listbox and select `Class` from the drop-down list. Over to the right, you'll see another listbox. Click it and select "initialize" from the list. Now you should see that a new subroutine has been created in your code module. It should look like that shown in [Example 2-23](#).

## Example 2-23. Empty initialize method

```
Private Sub Class_Initialize( )  
  
End Sub
```

You can add your custom initialization code inside this subroutine. For example, in [Example 2-24](#) I added two lines to initialize this object's properties to 0.

## Example 2-24. Empty initialize method

```
Private Sub Class_Initialize( )  
    x = 0  
    y = 0  
End Sub
```

It isn't quite necessary in this simple case, but does illustrate how to initialize objects.

So far, this vector object only stores two values. To make it actually do something, you have to add methods. As I said earlier, methods are just subroutines and functions. [Example 2-25](#) shows one I added to compute and return the magnitude of the vector.

### Example 2-25. Magnitude method

```
Public Function Magnitude( ) As Double  
    Magnitude = sqrt(x * x + y * y)  
End Function
```

This method is simply a function. You add it just as you would a regular function, as shown earlier in this chapter. The `Public` keyword makes this function available outside of this module. As you can see, the vector magnitude is computed using the object's `x` and `y` properties. Since these properties have module-level scope, you can access them from within any method you write.

You can add any number of methods in a similar manner. For example, you could add a subroutine that normalizes the vector, or you could write a subroutine that adds another vector given as a parameter to the vector, and so on.

Now we have a real object that can do something. To use this object in other code, you must first declare a variable with your object name as the type. Then you have to call VBA's `Set` and `New` statements to actually create the memory for the object, which also initializes the object by calling its initialize method if one exists. Once the object is created, you can use it in code using the `.` syntax to access its methods and properties. [Example 2-26](#) illustrates all these tasks.

### Example 2-26. Using the MyVector class

```
Public Sub MySubroutine( )
    Dim v As MyVector
    Dim mag As Double

    Set v = New MyVector

    v.x = 2
    v.y = 4

    mag = v.Magnitude
End Sub
```

The line `Dim v As MyVector` declares a local variable of type `MyVector`. You can't use it just yet (unlike `mag`, which is a regular variable of type `Double`). The line `Set v = New MyVector` is required to actually create the `MyVector` object, assigning it to the variable `v`.

The lines `v.x = 2` and `v.y = 4` set the `x` and `y` properties of the vector, `v`, to 2 and 4, respectively. They'll stay that way until you change them. A method is called in a similar manner. The line `mag = v.Magnitude` calls the vector's `Magnitude` method, returning the vector magnitude and storing it in the local variable `mag`.

## See Also

You can create fairly complex objects encapsulating a wide variety of functionality using these simple techniques. In [Recipe 13.7](#), I use these same techniques to create a `Chromosome` object for a genetic algorithm.



← PREV

## Recipe 2.15. VBA Help

### Problem

You'd like more information on Visual Basic for Applications.

### Solution

Look in the VBA help guide. Select Help → Microsoft Visual Basic Help from the main menu bar. For context-sensitive help, you can select a keyword in a code window and press F1 to open up a new help window containing information specific to the selected keyword.

Further, if you'd like to learn more about programming Excel in general, I can recommend Steven Roman's book *Writing Excel Macros with VBA* (O'Reilly). Steven's book covers most of the topics summarized in this chapter. He also covers topics I've left out that, while useful for general Excel programming, aren't immediately relevant to the calculations I discuss in this book.

# Chapter 3. Collecting and Cleaning Up Data

[Section 3.0. Introduction](#)

[Recipe 3.1. Importing Data from Text Files](#)

[Recipe 3.2. Importing Data from Delimited Text Files](#)

[Recipe 3.3. Importing Data Using Drag-and-Drop](#)

[Recipe 3.4. Importing Data from Access Databases](#)

[Recipe 3.5. Importing Data from Web Pages](#)

[Recipe 3.6. Parsing Data](#)

[Recipe 3.7. Removing Weird Characters from Imported Text](#)

[Recipe 3.8. Converting Units](#)

[Recipe 3.9. Sorting Data](#)

[Recipe 3.10. Filtering Data](#)

[Recipe 3.11. Looking Up Values in Tables](#)

[Recipe 3.12. Retrieving Data from XML Files](#)

[← PREV](#)

## 3.0. Introduction

Chances are you're going to have to deal with data that's not already in the form of an Excel file. This data may be standard scientific and engineering data (such as element properties or thermodynamic tables), or it could be the results of other computer simulations (such as computation fluid dynamics results or galaxy formation simulation results), or it could be experimental data obtained from any of a wide variety of scientific and engineering measurements. In many cases such data would be in the form of text files that could be imported into Excel for further analysis. Text files are common because they are universal, they are easy to generate in custom programs, and you don't have to have specialized software to read them. Still other data that you may want to process in Excel could reside in nontext files such as database files. This chapter will show you how to import data into Excel from these and other sources. Further, I'll show you how to massage that data once it's in Excel, making it easier to manipulate and analyze.

[← PREV](#)

← PREV

## Recipe 3.1. Importing Data from Text Files

### Problem

You have some data you'd like to analyze in Excel but it's located in a separate text file.

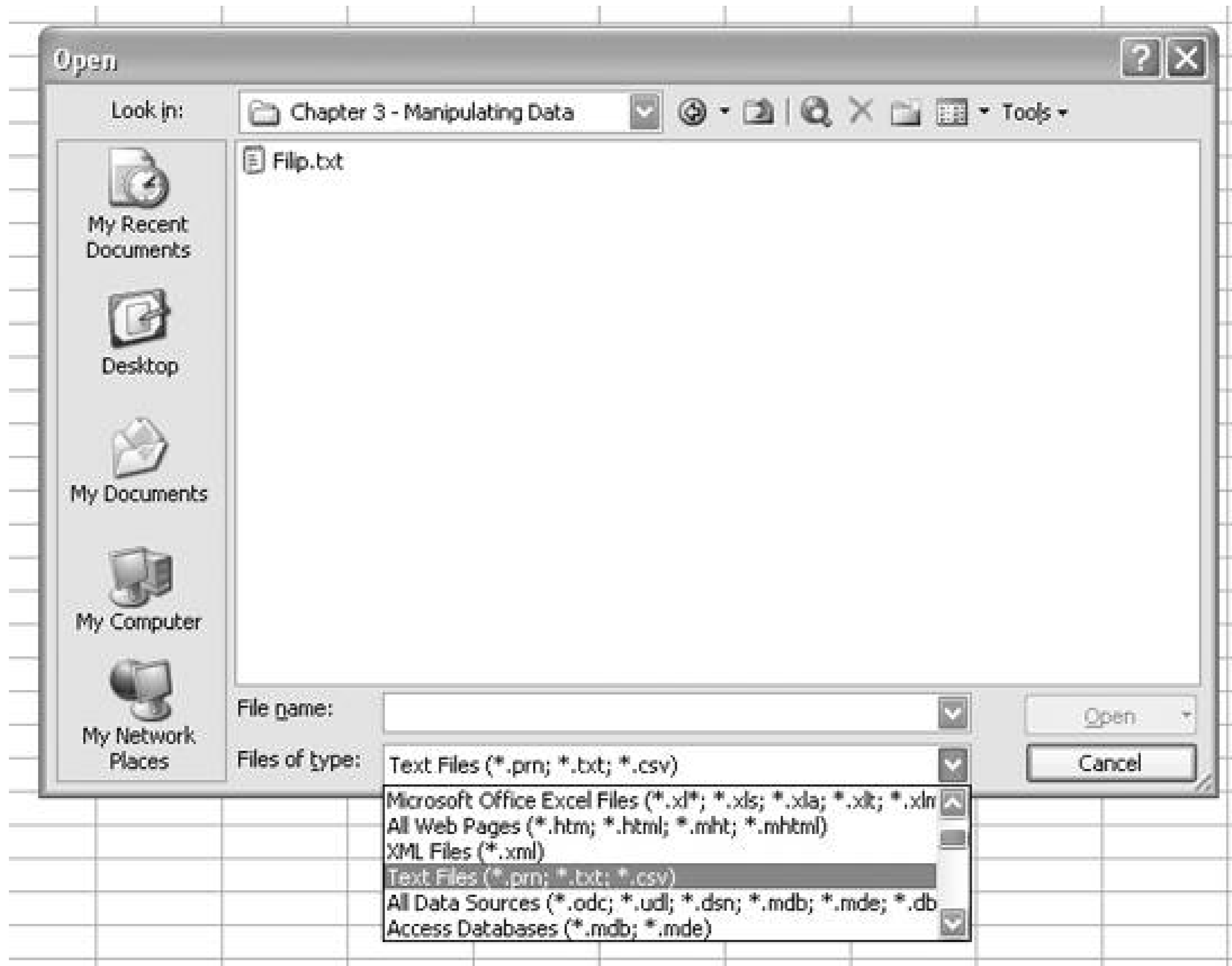
### Solution

Open the text file in Excel using File → Open....

### Discussion

Excel is capable of reading many different file formats, not just Excel spreadsheet files. Opening a text file is very much like opening a spreadsheet file: simply use File → Open... (or use the shortcut Ctrl-O) to pull up the Open dialog box and select the file you wish to open. [Figure 3-1](#) illustrates the Open dialog box.

Figure 3-1. Open dialog box



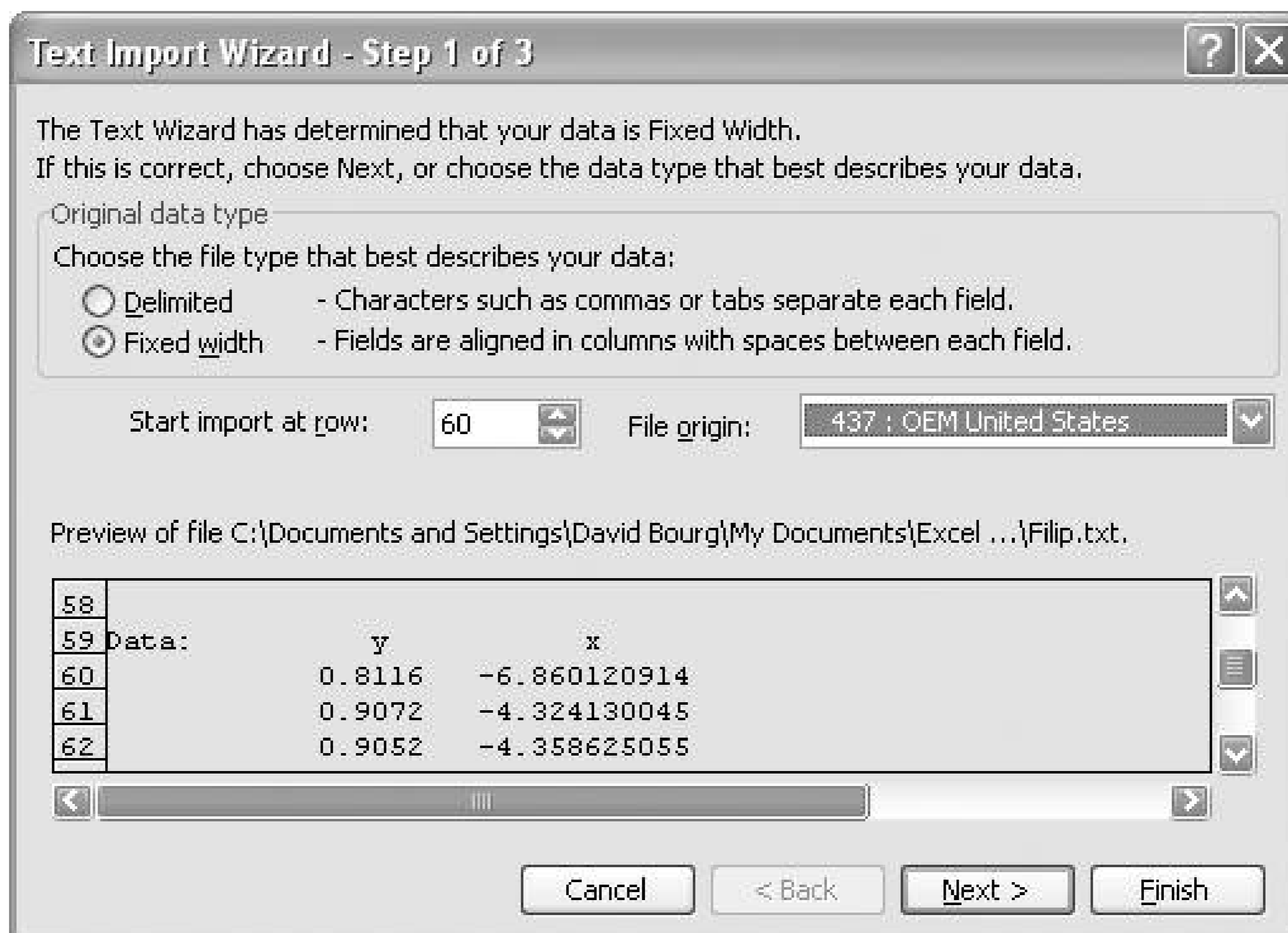
The Open dialog box defaults to listing only Microsoft Excel files (with extensions *.xls*, *.xla*, and so on). To list text files (with the *.txt* extension), you'll have to select Text Files in the "Files of type" drop-down listbox, as shown in [Figure 3-1](#). Press the Open button after selecting the file you wish to open.

The sample file I'm using here, *Filip.txt*, was obtained from the National Institute of Standards and Technology (NIST) online statistical reference dataset archive. This particular dataset was prepared by A. Filippelli of NIST for polynomial least-squares regression. [\[\\*\]](#)

[\*] You can find all sorts of useful data, a lot of which is freely available, on the NIST web site at <http://www.nist.gov>.

Upon pressing the Open button, you'll see the Text Import Wizard dialog box shown in [Figure 3-2](#). Since text files aren't necessarily in a format consistent with tidy rows and columns, Excel provides this import wizard to assist with parsing text, so the data you're after ends up in rows and columns in the newly formed spreadsheet. The wizard treats every line in the text file as a row of data. It then allows you to specify where column breaks should go so your data ends up in rows and columns. If you skip the column specification part, you could end up with rows of data in a single column, with each cell in the column potentially containing long strings of text.

Figure 3-2. Text Import Wizard: Step 1



Specifying column breaks is fairly painless. The first thing you need to do is specify whether your data is in *delimited* or *fixed-width* form. *Delimited* means the data on each row is separated by characters such as commas, spaces, tabs, or semicolons. *Fixed width* means the sets of data on each row are already aligned in columns, with spaces separating the columns. The data I'm using in this example is fixed width, as shown in [Figure 3-2](#). See [Recipe 3.2](#) to learn how to work with delimited data.

Notice the row numbers to the left of the data previewed in [Figure 3-2](#). The text file actually contains some descriptive information at the beginning of the file that, while useful, is not what I'm really interested in importing. Therefore, I've scrolled down to row 60 in the preview window so I can see the actual data. I've also set "Start import at row" to 60 so all the lines above the actual data will be skipped.

At this point, you should press the Next button to advance to the next step in the import wizard. Doing so opens the dialog box for step 2 shown in [Figure 3-3](#).

In this step you actually specify your desired column breaks. The import wizard usually makes a pretty good guess for you, and often you don't have to do anything else. In some cases, a few tweaks are in order. Specifying column breaks is easy and the dialog box shown in [Figure 3-3](#) actually includes instructions. Simply click at a point along the ruler just above the data preview window to set a column break. Double click a break to delete it. Or, click and drag to reposition a break. It's that simple. Click Next to proceed to the third step once your breaks are set. [Figure 3-4](#) shows the dialog box for step 3.

Figure 3-3. Text Import Wizard: Step 2

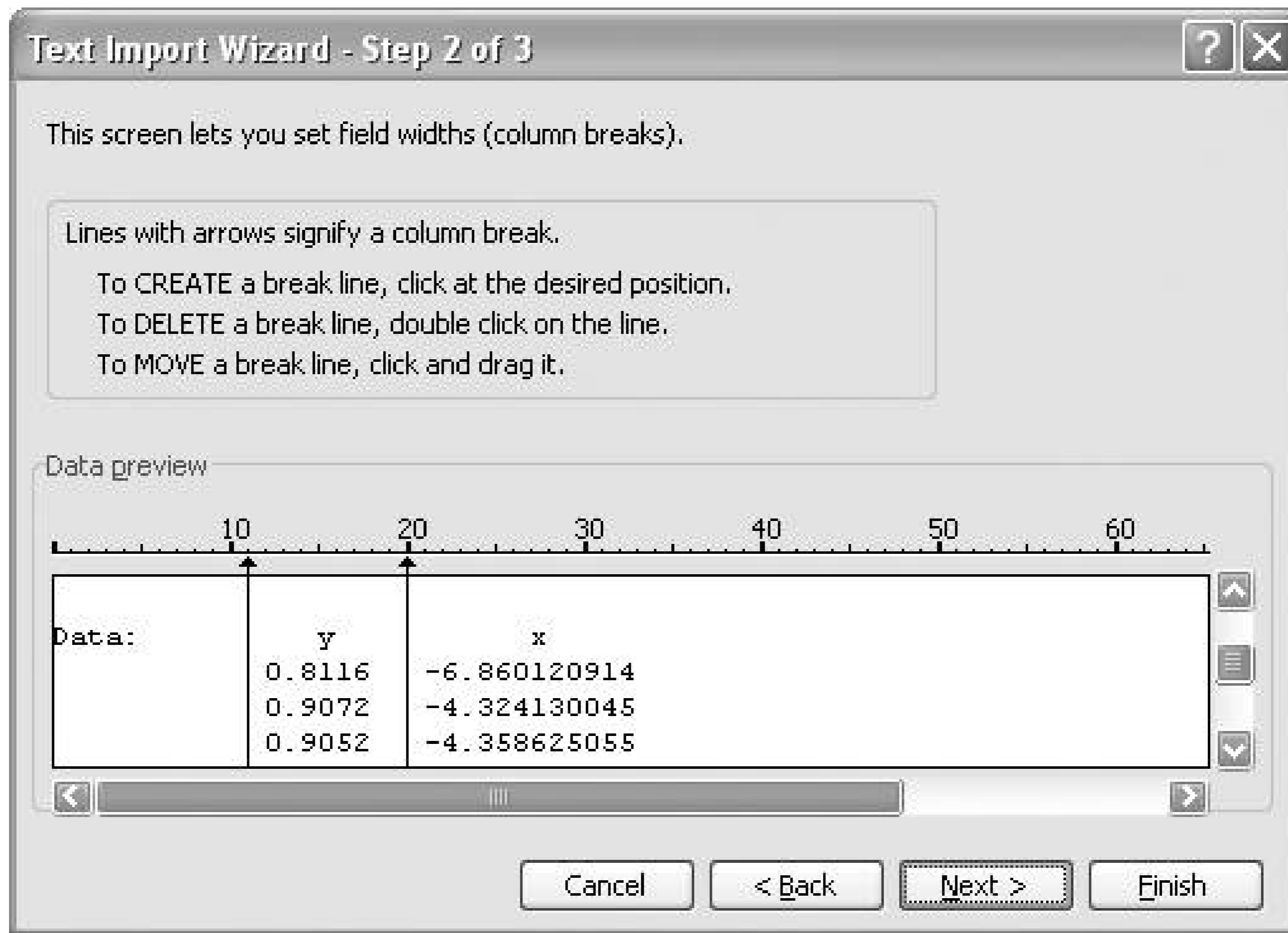
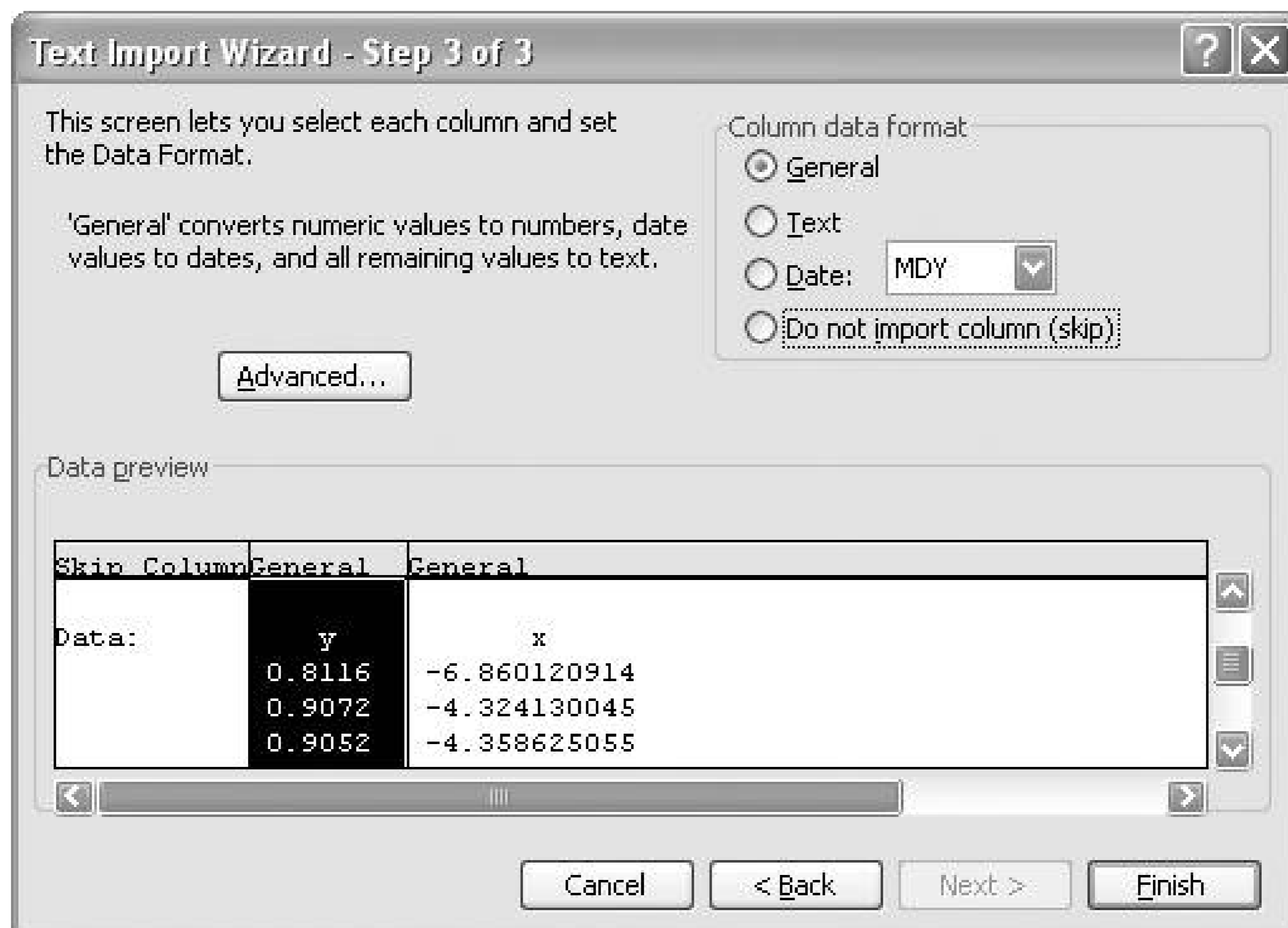


Figure 3-4. Text Import Wizard: Step 3

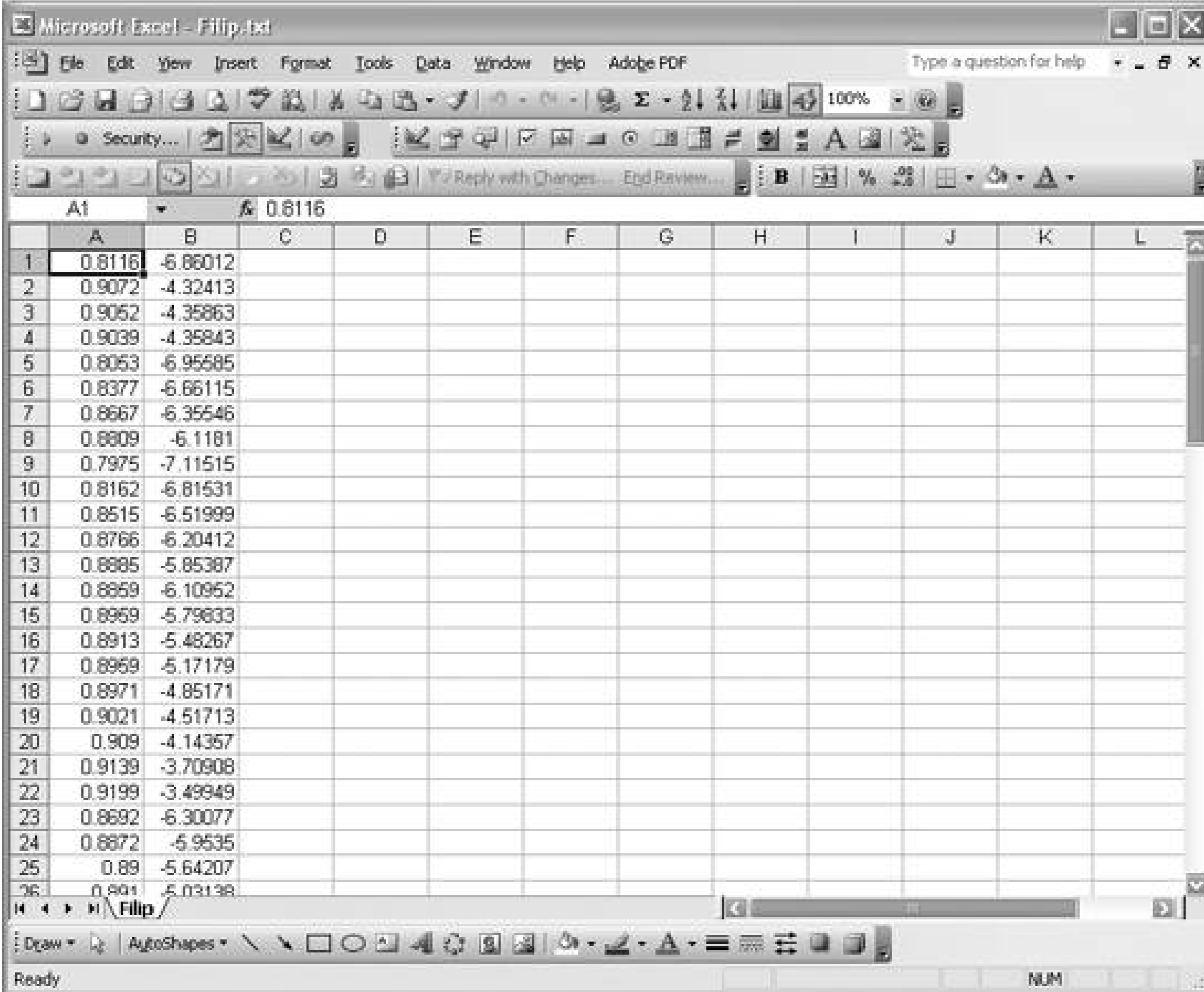


The third step allows you to specify the data type, or format, of the data contained in each column you specified in the previous step. Simply click any column in the data preview window to select the column. Next, specify the data format for that column by pressing one of the "Column data format" radio buttons. To be honest, I usually skip this step and use the defaults set by the wizard. The wizard usually does a fine job recognizing text versus numbers. I find this step most useful when I want to skip a column. For example, the first column in [Figure 3-4](#) does not contain any data; it contains only one row of text. Therefore, I select it and specify "Do not import column."

At this point you should press Finish to exit the wizard. Excel will then import the text per your specifications. The results for this example are shown in [Figure 3-5](#).

Figure 3-5. Imported text





The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Filip.txt". The spreadsheet contains the following data:

	A	B	C	D	E	F	G	H	I	J	K	L
1	0.8116	-6.86012										
2	0.9072	-4.32413										
3	0.9052	-4.35863										
4	0.9039	-4.35843										
5	0.8053	-6.95585										
6	0.8377	-6.66115										
7	0.8667	-6.35546										
8	0.8809	-6.1181										
9	0.7975	-7.11515										
10	0.8162	-6.81531										
11	0.8515	-6.51999										
12	0.8766	-6.20412										
13	0.8885	-5.85387										
14	0.8859	-6.10952										
15	0.8959	-5.79633										
16	0.8913	-5.48267										
17	0.8959	-5.17179										
18	0.8971	-4.85171										
19	0.9021	-4.51713										
20	0.909	-4.14357										
21	0.9139	-3.70908										
22	0.9199	-3.49949										
23	0.8692	-6.30077										
24	0.8672	-5.9535										
25	0.89	-5.64207										
26	0.891	-5.03138										

As you can see, Excel imported only the data I specified, placing it in columns A and B of a new spreadsheet. Now you can perform any of the usual spreadsheet operations on this data and save the resulting spreadsheet as you normally would. Before going too far, I recommend you do a Save As (File → Save As...) and save the new spreadsheet as a spreadsheet file with the .xls extension. This way, you can be sure to avoid overwriting your original text file. (Excel will warn you anyway if you attempt to overwrite it.)

## See Also

See [Recipes 3.2](#), [3.3](#), [3.4](#), and [3.5](#) for alternative data-importing techniques.

## Recipe 3.2. Importing Data from Delimited Text Files

### Problem

You want to import data from a text file, but the data isn't in fixed-width format as in [Recipe 3.1](#). This time, the data is delimited, i.e., separated by commas, spaces, tabs, or some other character.

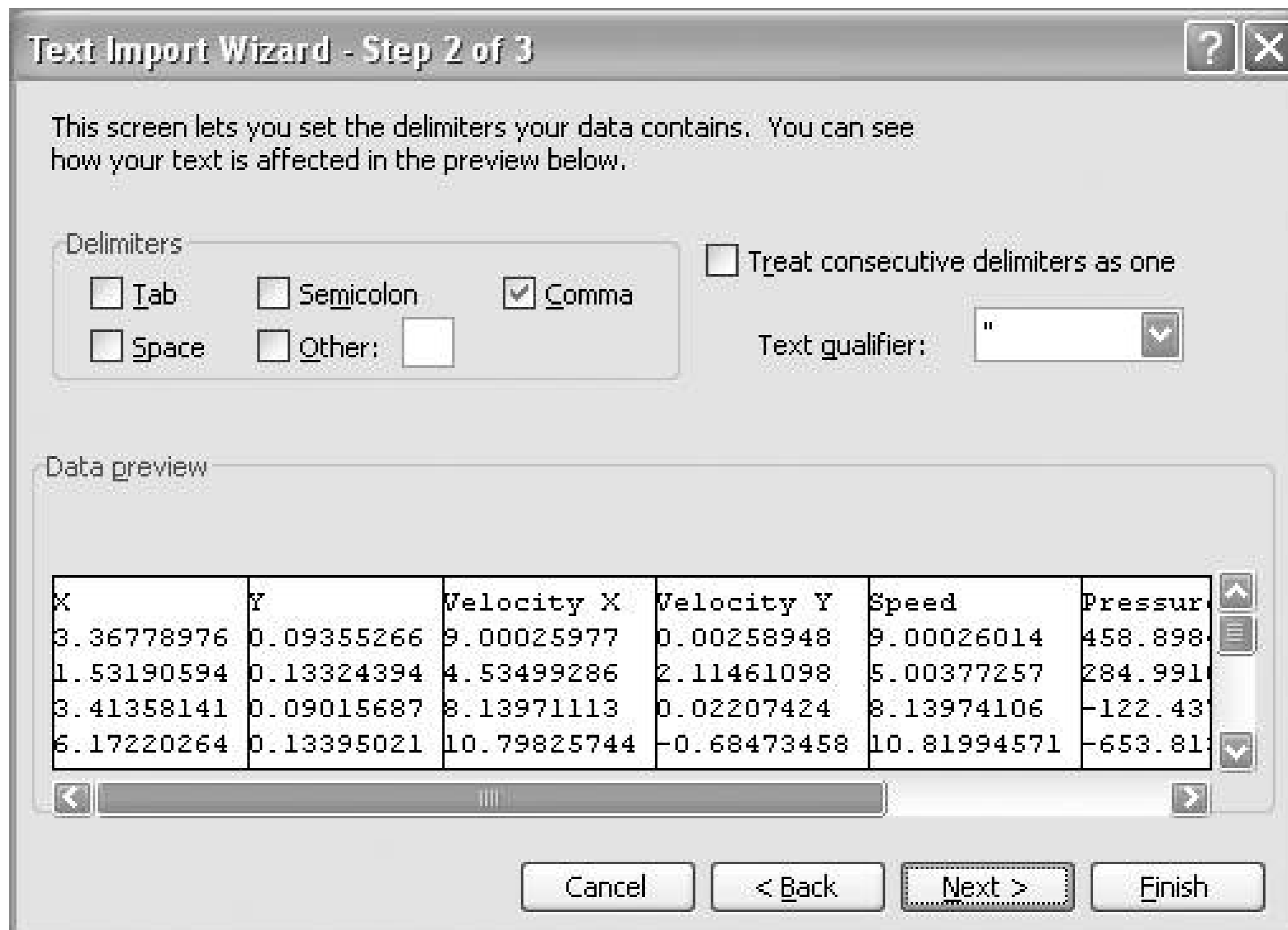
### Solution

Follow the import procedure described in [Recipe 3.1](#), but select Delimited instead of "Fixed width" in step 1 of the Text Import Wizard (see [Figure 3-2](#)).

### Discussion

When you select Delimited instead of "Fixed width," you don't specify column breaks during step 2 of the import procedure. Instead, you specify the delimiter that separates the data on each row being imported. In this case, step 2 of the Text Import Wizard looks like [Figure 3-6](#) instead of [Figure 3-3](#).

Figure 3-6. Text Import Wizard: Step 2 (Delimited file type)



The example data shown in the data preview window in [Figure 3-6](#) was taken from one of my fluid dynamics simulations. It shows fluid field variables (velocity, pressure, density) as functions of position on a 2D grid. The output file consists of rows of data, with each number separated by a comma. I checked the comma delimiter in the import wizard dialog and the wizard automatically set the column break points. Pressing Next at this point takes you to step 3, as already discussed in [Recipe 3.1](#).

## See Also

See [Recipe 3.1](#), [Recipe 3.3](#), and [Recipe 3.4](#) for alternative data importing techniques.

← PREV

## Recipe 3.3. Importing Data Using Drag-and-Drop

### Problem

You routinely use drag-and-drop to open files in Windows and are wondering if you can do the same with text files in Excel.

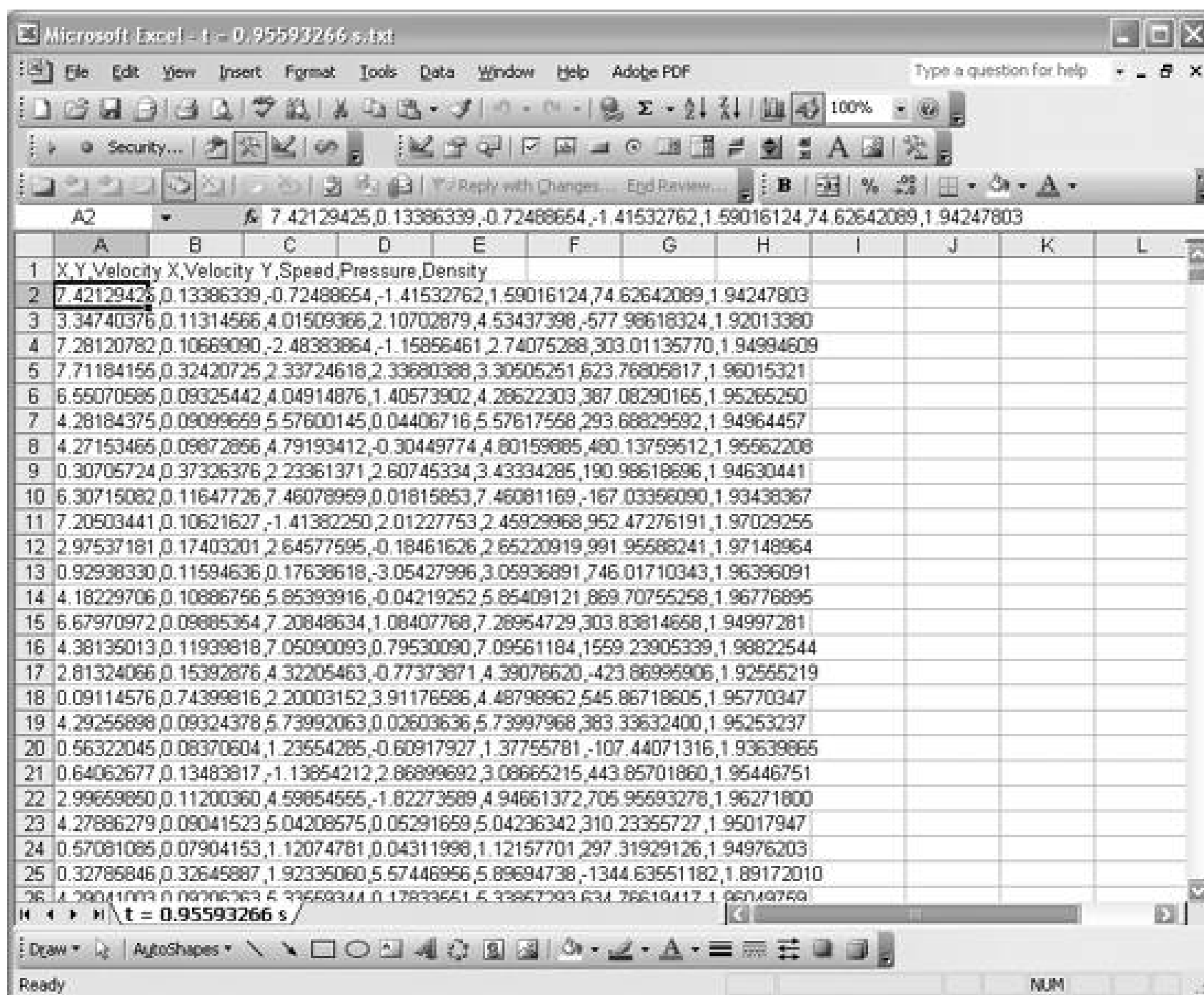
### Solution

Yes, you can. Simply drag the text file and drop it on Excel's window.

### Discussion

While this sounds trivial, you'll quickly realize that dragging and dropping a text file into a spreadsheet does not always yield the results you'd like. Take a look at the example shown in [Figure 3-7](#).

Figure 3-7. Text imported by drag-and-drop



The data contained in the text file was indeed loaded into the spreadsheet; however, all of the data on each line in the text file was lumped into a single cell (in column A) on each row in the spreadsheet. In this form, the data is treated as text and you can't perform analyses on the data or plot it on a chart. You need to parse the data in order to put it in a useful form, i.e., separate the data into columns. See [Recipe 3.6](#) to learn how to convert text like that shown in [Figure 3-7](#) to columns of data.

## See Also

See [Recipe 3.6](#) to learn how to convert strings of text to useful columns of data.

Also, take a look at [Recipes 3.1](#), [3.2](#), [3.4](#), and [3.5](#) for alternative data-importing techniques.

## Recipe 3.4. Importing Data from Access Databases

### Problem

You have data in a Microsoft Access database that you'd like to import into Excel for analysis.

### Solution

Use File → Open... to open the Access database file.

### Discussion

When you select File → Open..., you'll get the familiar Open dialog box. In the "Files of type" drop-down listbox, look for and select the Access Databases file type, as shown in [Figure 3-8](#). Access database files have the file extension *.mdb*.

Locate and select the database file you'd like to open and then press the Open button. Upon doing so you'll see the Select Table dialog box shown in [Figure 3-9](#).

The example database file I'm using here contains standard structural material properties and standard structural shape data. It includes two distinct tables: Materials and Sections. In this case, I've selected the Materials table for import into Excel. When I make the table selection and press the OK button, the data contained in the table is imported into Excel, as shown in [Figure 3-10](#).

That's all there is to it. You can now perform operations on the imported data as you wish.

I should point out that the data imported into Excel is linked to the original Access database. This means that when data is changed in the original database, it can automatically be updated in your spreadsheet. You have some control over how the data is updated. Notice the External Data toolbar, shown in [Figure 3-10](#), to the right of the imported data. This toolbar appears automatically when the data is imported. (You can also show this toolbar from the main menu bar via View → Toolbars → External Data.)

Figure 3-8. Opening an Access database

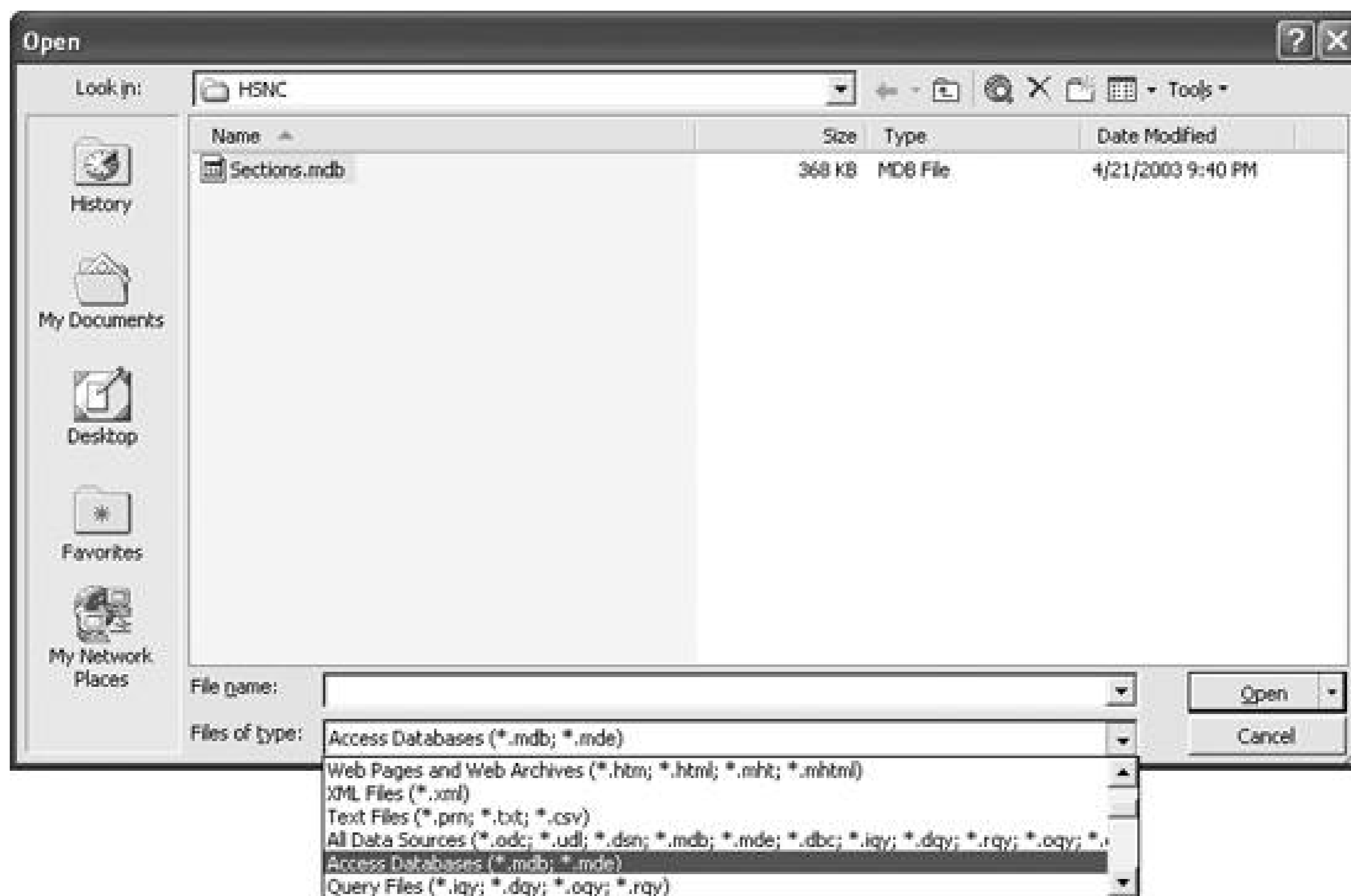


Figure 3-9. Select Table dialog box

The second button from the left of the External Data toolbar is the Data Range Properties button. Pressing this button opens the External Data Range Properties dialog box, which allows you to specify how often you want Excel to automatically refresh the imported data. You can also specify that you want the data automatically refreshed every time you open your spreadsheet. The button with the exclamation point on it forces Excel to refresh the data instantly.

Figure 3-10. Imported Access table

ID	Description	UT	Sigma YW	Tau	Sigma Y	Type	Units
1	5083 - H111	269	145	83	165.62	Aluminum	Metric
2	5083 - H116	276	165	96	199.92	Aluminum	Metric
3	5083 - H321	276	165	96	215.6	Aluminum	Metric
4	5083 - H323	276	165	96	234.22	Aluminum	Metric
5	5083 - H343	276	165	96	268.52	Aluminum	Metric
6	5086 - H111	241	124	69	142.1	Aluminum	Metric
7	5086 - H112 6 mm (0.25 in) -12 mm (0.50 in)	241	117	65	124.46	Aluminum	Metric
8	5086 - H112 12 mm (0.5 in) -25 mm (1.0 in)	241	110	62	109.76	Aluminum	Metric
9	5086 - H112 25 mm (1.0 in) - 51 mm (2.0 in)	241	96.5	55	96.04	Aluminum	Metric
10	5086 - H116	241	131	76	193.06	Aluminum	Metric
11	5086 - H32	241	131	76	193.06	Aluminum	Metric
12	5086 - H34	241	131	76	234.22	Aluminum	Metric
13	5383 - O	290	145	83	142.1	Aluminum	Metric
14	5383 - H111	290	145	83	142.1	Aluminum	Metric
15	5383 - H116	290	145	83	215.6	Aluminum	Metric
16	5383 - H321	290	145	83	215.6	Aluminum	Metric
17	5383 - H34	290	145	83	264.6	Aluminum	Metric
18	5454 - H111	214	110	65	131.32	Aluminum	Metric
19	5454 - H112	214	83	48	82.32	Aluminum	Metric
20	5454 - H32	214	110	65	179.34	Aluminum	Metric
21	5454 - H34	214	110	65	199.92	Aluminum	Metric
22	5456 - H111	283	165	96	131.32	Aluminum	Metric
23	5456 - H112	283	131	76	124.46	Aluminum	Metric
24	5456 - H116, H321	290	179	103	199.92	Aluminum	Metric
25	5456 - H323	290	179	103	247.94	Aluminum	Metric
26	5456 - H343	290	179	103	282.24	Aluminum	Metric
27	6061-T6 under 9.5 mm (0.375 in)	165	138	83	241.08	Aluminum	Metric
28	6061-T6 over 9.5 mm (0.375 in)	165	103	62	241.08	Aluminum	Metric
29	5083 - H111	39000	21000	12000	24000	Aluminum	English
30	5083 - H116	40000	24000	14000	29000	Aluminum	English
31	5083 - H321	40000	24000	14000	31900	Aluminum	English
32	5083 - H323	40000	24000	14000	34000	Aluminum	English

You can import data from other databases as well. For example, Excel allows you to create new database queries facilitating access to such databases as Oracle, FoxPro, and dBase. Select Data Import External Data New Database Query... to create a new database query or use an existing data source. See the help topics "About importing data," "Create an OLAP query file to connect to an OLAP database," and "Import or connect to data in Excel's online help" for more information.

## See Also

Take a look at the help topic "Data sources you can access" for more information on the types of databases you can access from within Excel and for specific information on connecting to those databases.





## Recipe 3.5. Importing Data from Web Pages

### Problem

You've found a web page containing useful data that you'd like to use in a spreadsheet; however, you don't feel like retyping it all in.

### Solution

There are a few ways to get the data from a web page into Excel without having to retype it all. You could try copying and pasting, but this may give you format problems. You could also try saving the web page as a text file, but the formatting of the web page may make this cumbersome.

The most direct way to import data from a web page uses Excel's built-in capability to import data from tables and preformatted areas contained in web pages. From within Excel, select Data > Import External Data > New Web Query....

### Discussion

Upon selecting Data > Import External Data > New Web Query... from the main menu bar, you'll see the New Web Query window shown in [Figure 3-11](#). The New Web Query window is essentially a web browser. You can type in a URL in the address bar to go to the page containing the data you'd like to import. You can also navigate using the forward and back buttons (the circular ones with arrows, to the right of the Go button in the toolbar).

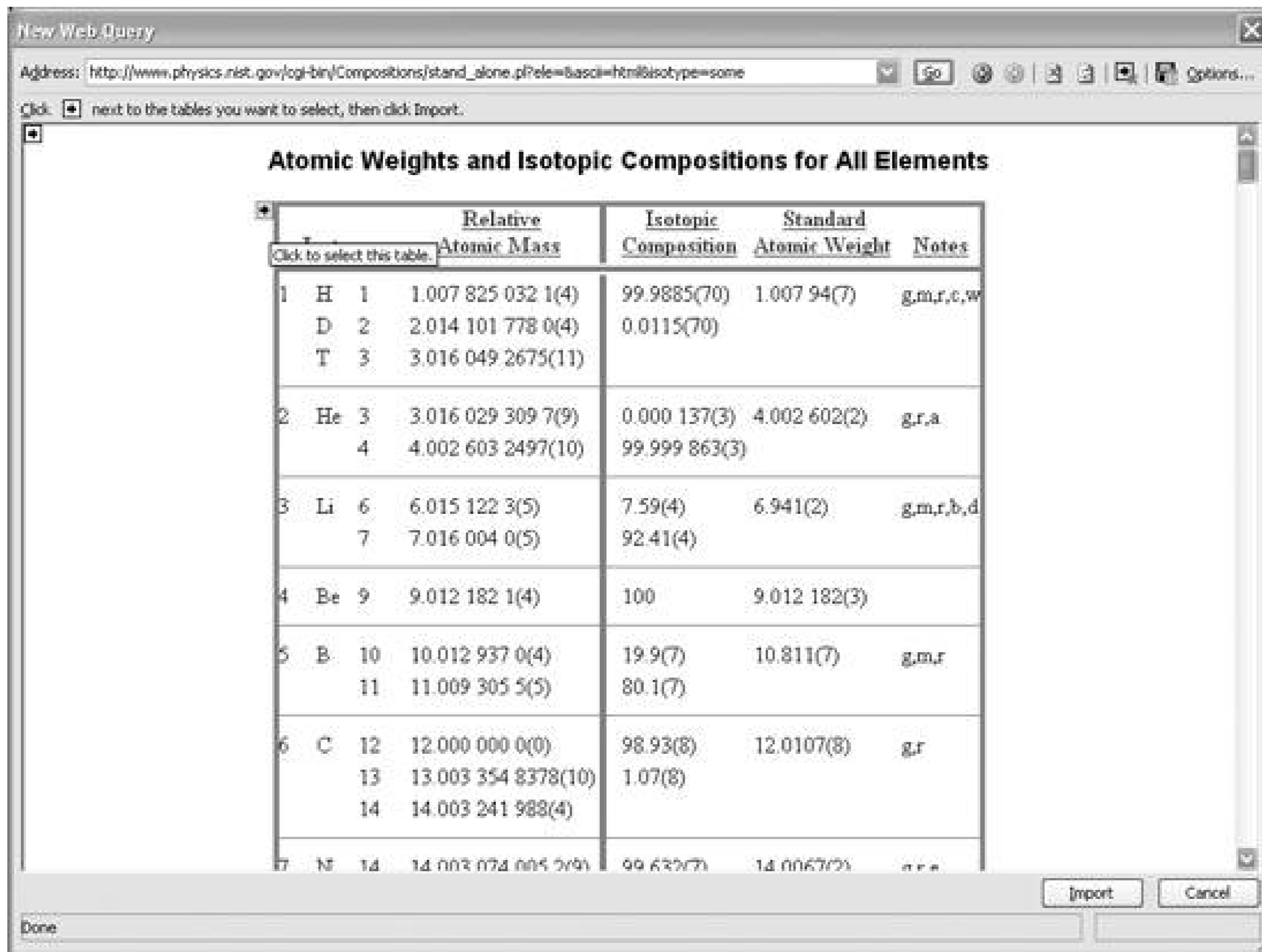
A key difference between this window and your usual web browser is that this window analyzes the web page to determine which data you can import. Each block of data that can be imported is marked with a little arrow icon in the block's upper-left corner, as shown in [Figure 3-11](#). If you drag your mouse over one of these icons, the corresponding data block will be highlighted. Click the icon to select that block of data (the arrow will turn into a checkbox). Click the arrow icon in the upper-left corner of the page to import the entire page. Once you've selected the blocks of data you'd like to import, press the Import button.

The results of this action for the page displayed in [Figure 3-11](#) are shown in [Figure 3-12](#). Notice that the External Data toolbar is shown when the web query is completed, just as in the earlier example showing how to import data from an Access database. You can use the External Data toolbar to manipulate the web query. For example, if you press the refresh button, Excel will attempt to refresh the data obtained from the web page by reaccessing the page over the Internet. This can be useful if the data you've imported changes frequently.

Once you've imported the data, you can use it in your own analyses or manipulate it further using

other techniques discussed in this chapter.

Figure 3-11. New Web Query window



## See Also

You can save your web queries and rerun them at later times from other workbooks if you'd like. This is useful if you want to use the data in more than one workbook. You can save the query by pressing the save icon in the New Web Query window (see [Figure 3-11](#)). For more information see the help topic "Query for data from a Web page" in Excel's online help.

← PREV

## Recipe 3.6. Parsing Data

### Problem

You've imported data from a text file as discussed in [Recipe 3.3](#), but all data on each row is lumped into a single cell. You'd like to separate the data on each row into columns.

### Solution

Select the data you'd like to parse. Then select Data > "Text to Columns..." from the main menu bar to open the "Convert Text to Columns" Wizard.

Figure 3-12. Data imported using web query

1	Isotope		Relative	Isotopic	Standard	Notes
2			Atomic Mass	Composition	Atomic Weight	
6	1 H	1	1.007 825 032 1(4)	99.9885(70)	1.007 94(7)	g,m,r,c,w
7		2	2.014 101 778 0(4)	0.0115(70)		
8		3	3.016 049 2675(11)			
11	2 He	3	3.016 029 309 7(9)	0.000 137(3)	4.002 602(2)	g,r,a
12		4	4.002 603 2497(10)	99.999 863(3)		
15	3 Li	6	6.015 122 3(5)	7.59(4)	6.941(2)	g,m,r,b,d
16		7	7.016 004 0(5)	92.41(4)		
19	4 Be	9	9.012 182 1(4)		100 9.012 182(3)	
22	5 B	10	10.012 937 0(4)	19.9(7)	10.811(7)	g,m,r
23		11	11.009 305 5(5)	80.1(7)		

## Discussion

The data shown in [Figure 3-7](#) (in [Recipe 3.3](#)) was imported from a text file via drag-and-drop. The problem with this approach is that all the data on each row is lumped into a single cell. This is less of a problem with the import method in [Recipe 3.1](#), because using File → Open... automatically launches the Text Import Wizard, which allows you to set columns for your data either manually or by specifying delimiters.

The "Convert Text to Columns" Wizard, accessible via Data → "Text to Columns ...", is essentially the Text Import Wizard. The three steps in the "Convert Text to Columns" Wizard are identical to those in the Text Import Wizard. First you specify whether the text to be parsed is in fixed-width form or delimited. Then, depending on your selection in step 1, you either manually set column breaks or select the delimiting character. Finally, you can specify data formats and choose which columns to ignore.

Performing these steps on the data shown in [Figure 3-7](#) yields the parsed data shown in [Figure 3-13](#).

Figure 3-13. Parsed data

Microsoft Excel - t = 0.95593266 s.txt

File Edit View Insert Format Tools Data Window Help Adobe PDF Type a question for help

100%

Security...

Reply with Changes... End Review...

A2 7.42129425

	A	B	C	D	E	F	G	H	I	J	K	L
1	X	Y	Velocity X	Velocity Y	Speed	Pressure	Density					
2	7.421294	0.133863	-0.72489	-1.41533	1.590161	74.62642	1.942478					
3	3.347404	0.113146	4.015094	2.107029	4.534374	-577.986	1.920134					
4	7.281208	0.106691	-2.48384	-1.15856	2.740753	303.0114	1.949946					
5	7.711842	0.324207	2.337246	2.336804	3.305053	623.7681	1.960153					
6	6.550706	0.093254	4.049149	1.405739	4.286223	387.0829	1.952653					
7	4.281844	0.090997	5.576001	0.044067	5.576176	293.6883	1.949645					
8	4.271535	0.098729	4.791934	-0.3045	4.801599	480.1376	1.956622					
9	0.307057	0.373264	2.233614	2.607453	3.433343	190.9862	1.946304					
10	6.307151	0.116477	7.46079	0.018159	7.460812	-167.034	1.934384					
11	7.205034	0.106216	-1.41382	2.012278	2.4593	952.4728	1.970293					
12	2.975372	0.174032	2.645776	-0.18462	2.652209	991.9559	1.97149					
13	0.929383	0.115946	0.176386	-3.05428	3.059369	746.0171	1.963961					
14	4.182297	0.108868	5.853939	-0.04219	5.854091	869.7076	1.967769					
15	6.67971	0.098854	7.208486	1.084078	7.289547	303.8381	1.949973					
16	4.38135	0.119398	7.050901	0.795301	7.095612	1559.239	1.988225					
17	2.813241	0.153929	4.322055	-0.77374	4.390766	-423.87	1.925552					
18	0.091146	0.743998	2.200032	3.911766	4.48799	545.8672	1.957703					
19	4.292559	0.093244	5.739921	0.028036	5.73998	383.3363	1.952532					
20	0.56322	0.083706	1.235543	-0.60918	1.377558	-107.441	1.936399					
21	0.640627	0.134838	-1.13854	2.868997	3.086652	443.857	1.954468					
22	2.996599	0.112004	4.598546	-1.82274	4.946614	705.9559	1.962718					
23	4.278863	0.090415	5.042086	0.052917	5.042363	310.2336	1.950179					
24	0.570811	0.079042	1.120748	0.04312	1.121577	297.3193	1.949762					
25	0.327858	0.326459	1.923351	5.57447	5.896947	-1344.64	1.89172					
26	1.590161	0.093254	5.336503	0.178336	5.338573	634.7662	1.960498					

t = 0.95593266 s

Draw AutoShapes

Ready NUM

## Recipe 3.7. Removing Weird Characters from Imported Text

### Problem

You've imported data from a text file but find weird (i.e., nonprintable) characters in the imported text. You'd like to remove the nonprintable characters.

### Solution

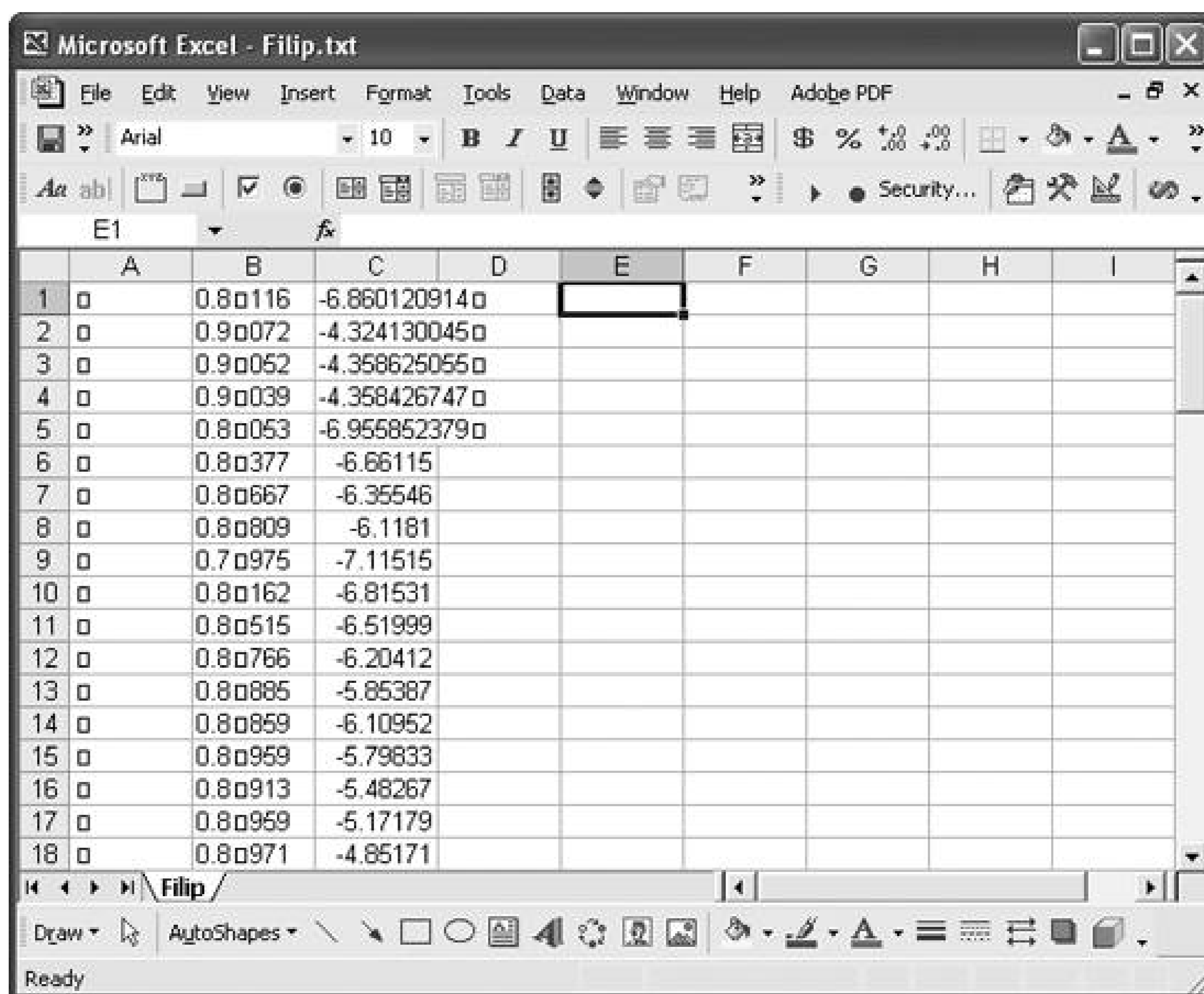
Use the `CLEAN` worksheet function.

### Discussion

If you've tried some of the text-importing recipes in this chapter, you may have come across some unusual characters in your imported data, as illustrated in [Figure 3-14](#).

Notice the unusual rectangle characters mixed with the data. These characters are so-called nonprintable characters because under normal circumstances you can't see them. They include control characters such as line-feed and carriage-return characters, tabs, and others. You may or may not encounter such characters, depending on where your text files originate. I find I'm more likely to encounter such characters when importing data from text files that originated on a non-Windows system. This is because there are different standards for specifying formatting and control instructions on different systems. For example, Windows uses both a line feed and carriage return to denote the end of a line of text, whereas Macintosh systems use a single carriage return.

Figure 3-14. Nonprintable characters



You can remove these nonprintable characters using Excel's built-in worksheet function **CLEAN**. Simply type `=CLEAN(cell reference)` in a cell to remove nonprintable characters from the text contained in the referenced cell. You should replace *cell reference* with a valid cell reference, such as C1. [Figure 3-15](#) shows how **CLEAN** deals with the first several rows of data.

Columns G and H show the cleaned data. Notice the cell formula for the selected cell, G1, displayed in the formula bar. In this case, the formula is `=CLEAN(B1)`. The result is that the nonprintable character was removed from the number 0.8116.



← PREVIOUS

## Recipe 3.8. Converting Units

### Problem

You want to convert a number from one unit to another directly within Excel.

Figure 3-15. CLEAN

The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Filip.txt". The spreadsheet has columns A through I and rows 1 through 18. Column B contains numbers with trailing zeros, and column C contains negative numbers with trailing zeros. Column G shows the result of the CLEAN function applied to column B, which removes the trailing zeros. The formula bar shows the formula =CLEAN(B1) for cell G1.

	A	B	C	D	E	F	G	H	I
1		0.80116	-6.8601209140				0.8116	-6.860120914	
2		0.90072	-4.3241300450				0.9072	-4.324130045	
3		0.90052	-4.3586250550				0.9052	-4.358625055	
4		0.90039	-4.3584267470				0.9039	-4.358426747	
5		0.80053	-6.9558523790				0.8053	-6.955852379	
6		0.800377	-6.66115				0.8377	-6.661145254	
7		0.800667	-6.35546				0.8667	-6.355462942	
8		0.800809	-6.1181						
9		0.700975	-7.11515						
10		0.800162	-6.81531						
11		0.800515	-6.51999						
12		0.800766	-6.20412						
13		0.800885	-5.85387						
14		0.800859	-6.10952						
15		0.800959	-5.79833						
16		0.800913	-5.48267						
17		0.800959	-5.17179						
18		0.800971	-4.85171						

### Solution

Use Excel's built-in worksheet function `CONVERT`. Alternatively, you can write your own conversion

function.

## Discussion

The sections below describe how to use `CONVERT`, how to write your own conversion function (or other functions), and how to create an add-in to make your own collection of functions accessible from any spreadsheet.

## Using CONVERT

The Analysis ToolPak add-in included in Excel contains a worksheet function called `CONVERT` that allows you to convert values from one unit to another. To use `CONVERT`, you must ensure the Analysis ToolPak is installed in your version of Excel. To do this, select Tools → Add-Ins ... from the main menu bar to open the Add-Ins dialog box. Select Analysis ToolPak from the list of available add-ins and then press OK.

The syntax for `CONVERT` is `=CONVERT(cell reference, from units, to units)`, where *cell reference* is a valid cell reference, *from units* is a string indicating the units of the value being converted, and *to units* is a string indicating the units to which the value is being converted.

Valid strings for units are listed in the Excel help topic "Convert." You can find this topic by searching for "Convert" in Excel's online help, or you can browse this topic starting from the help table of contents (look for the topic Working with Data → Function Reference → Engineering Functions → Convert).

`CONVERT` supports a basic set of units, including length, mass, time, force, pressure, energy, power, magnetism, temperature, and volume. It also supports a range of prefixes for tera-, giga-, centi-, pica-, and so on.

You use `CONVERT` in a worksheet by entering the function in a cell, as illustrated in the following examples:

```
=CONVERT(A1, "in", "ft")
=CONVERT(A2, "in", "cm")
=CONVERT(A3, "HP", "MW")
```

In the first example, the value in cell A1 is converted from inches to feet. Notice how the *from units* and *to units* strings must be included in quotes. In the second example, the value in A2 is converted from inches to centimeters. In this case, the units string for meters, *m*, is prefixed with the centi-prefix, *c*, to yield the centimeters string, *cm*. In the third example, the value in A3 is converted from horsepower to megawatts. Here, *MW* is formed by prefixing the watts units string, *w*, with the mega-prefix, *M*. I should point out that these units strings are case-sensitive. For example, *MW* yields megawatts; however, *mW* yields milliwatts.

## Writing your own functions

While `CONVERT` is a handy conversion function, it is limited to a set of very common units. You'll probably find that units common to your specialty are not included in `CONVERT`'s lexicon. For example, in the field of naval architecture, the standard weight unit used for ships and cargo is the *long ton*, or LT. `CONVERT` does not support long tons. In fact, it does not support tons of any form.

Since converting units is a common task when pre- and post-processing data, you may find it helpful to write your own custom unit-conversion functions to supplement those supported by `CONVERT`. [Example 3-1](#) shows a few custom conversion functions I use quite often.

### Example 3-1. Custom unit-conversion functions

```
Public Function LbsToLT(value As Double) As Double
    LbsToLT = value / 2240#
End Function

Public Function LbsToTons(value As Double) As Double
    LbsToTons = value / 2000#
End Function

Public Function cuFtTocum(value As Double) As Double
    cuFtTocum = value * 0.028316846592
End Function
```

You can add these or similar functions to your own spreadsheets as explained in [Recipe 2.2](#). Be sure to put these in a new code module as explained in [Chapter 2](#). You'll then be able to call these functions as cell formulas anywhere in your spreadsheet.

### Creating an add-in

If you're like me, you'll find yourself building a custom library of useful functions and subroutines that you'd like to use repeatedly in different spreadsheets. To make your library of custom functions accessible from any spreadsheet, convert it to an add-in. Creating an add-in is easy. Here are the steps:

1. Open a workbook and add your custom functions to a code module as discussed earlier.
2. Save your workbook as an add-in. Select `File` → `Save As ...` to open the `Save As` dialog box. Then select "Microsoft Excel Add-in (\*.xla)" from the "Save as type" drop-down listbox. Doing so will automatically change the `Save In` folder to the default Excel add-ins folder on your machine (it's usually something like `C:\Documents and Settings\User\Application Data\Microsoft\AddIns`). Close your add-in.

3. Open a new workbook.
4. Select Tools → Add-Ins ... from the main menu bar to open the Add-Ins dialog box.
5. Locate your add-in in the list of available add-ins and make sure it's checked. Press OK.

From now on, your custom add-in will be available to all of your workbooks. To add new capabilities to your add-in, go to the Visual Basic editor and locate your add-in in the Project window as discussed in [Chapter 2](#). Double-click the code module for your add-in and make your changes. Save your changes when you're done.



[← PREV](#)

## Recipe 3.9. Sorting Data

### Problem

You have unsorted data in your spreadsheet and you'd like it sorted.

### Solution

Select the data you want to sort, then choose Data → Sort ... from the main menu bar to open the Sort dialog box. The Sort dialog box will allow you to specify how to sort the data, as discussed below.

### Discussion

[Figure 3-16](#) shows a sample spreadsheet containing hypothetical student grades. The student names are not in alphabetical order, so I'll use Excel's sort feature to sort them from A to Z. Moreover, when I sort the names, the grades associated with each student will remain so; that is, we're not just going to sort the first column, but will also keep grade associations intact for each student name.

Figure 3-16. Unsorted grade report

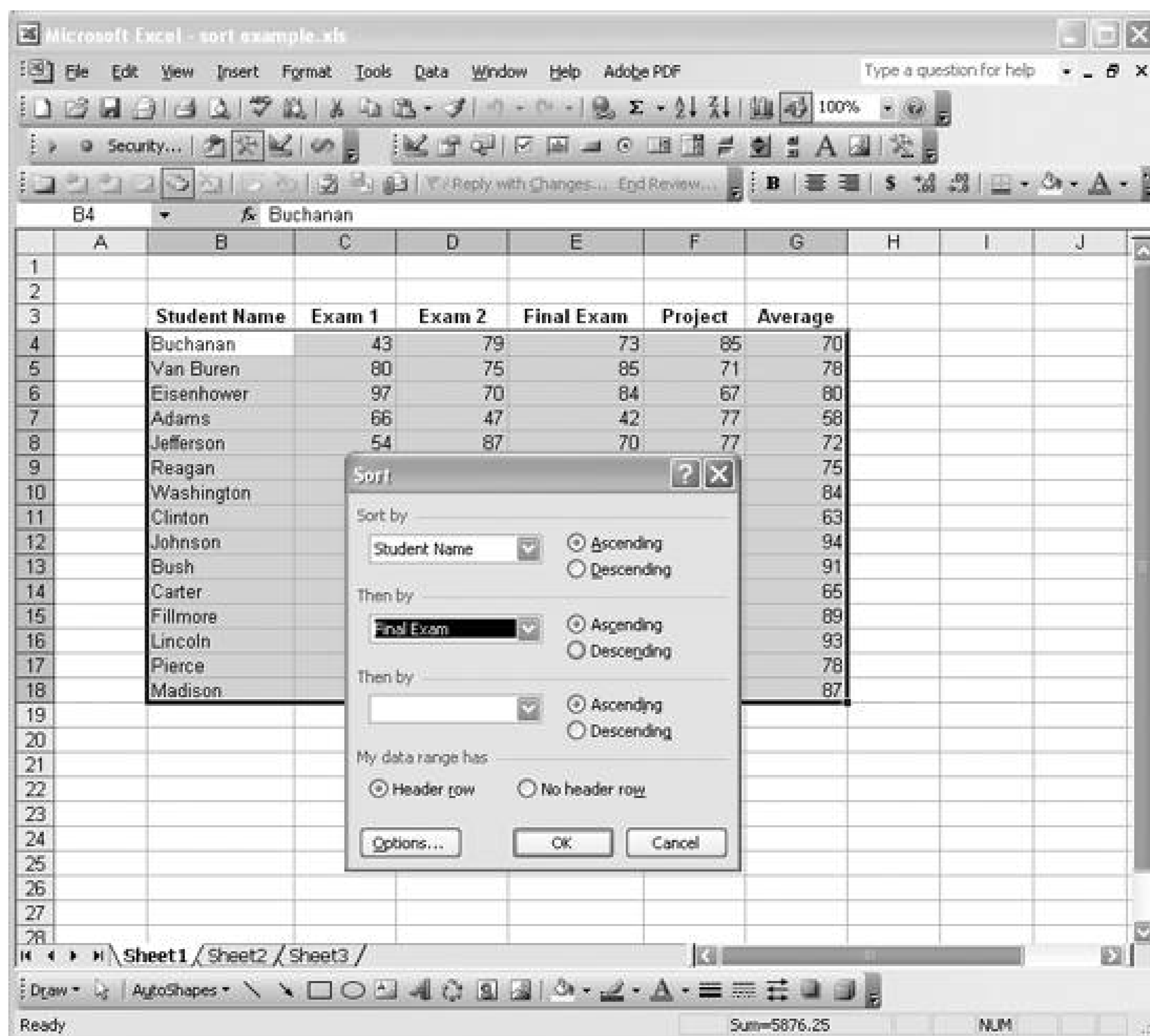
	A	B	C	D	E	F	G	H	I	J
1										
2										
3		<b>Student Name</b>	<b>Exam 1</b>	<b>Exam 2</b>	<b>Final Exam</b>	<b>Project</b>	<b>Average</b>			
4		Buchanan	43	79	73	85	70			
5		Van Buren	80	75	85	71	78			
6		Eisenhower	97	70	84	67	80			
7		Adams	66	47	42	77	58			
8		Jefferson	54	87	70	77	72			
9		Reagan	79	71	66	82	75			
10		Washington	94	87	78	75	84			
11		Clinton	56	43	65	86	63			
12		Johnson	100	100	88	88	94			
13		Bush	94	93	86	90	91			
14		Carter	77	42	50	92	65			
15		Fillmore	91	90	79	97	89			
16		Lincoln	100	88	92	92	93			
17		Pierce	66	90	72	85	78			
18		Madison	77	93	84	94	87			
19										
20										
21										
22										
23										
24										
25										
26										
27										
28										

To sort the data keeping everything intact, you must select the entire data range. Click the first name in the list and drag the selection to the lower-right of the range. Now select Data > Sort ... from the main menu bar. You should see the Sort dialog shown in [Figure 3-17](#).

In this example, let's sort by student names in ascending order. Select Student Name in the "Sort by" drop-down listbox, and select Ascending next to it. In the event two names are identical, let's use the associated final exam grade as the tiebreaker, with the name of the student who has the lower final exam grade appearing first. To do this, select Final Exam from the "Then by" drop-down listbox and select Ascending next to it.

You might be wondering how Excel knows the names of the columns to include in the "Sort by" and "Then by" listboxes. Excel detects whether or not you have text in the row above the selected data. If so, it sets "My data range has" to "Header row" and uses the header row labels in the drop-down listboxes. If you don't have a header row, Excel will use the column labels in the drop-down listboxes instead.

Figure 3-17. Sort dialog box



Pressing the OK button at this point causes Excel to perform the sort. Results for this example are shown in [Figure 3-18](#).

As you can see, the names are now in alphabetical order and their associated grades were kept intact. You need not sort on the first column of data as we did in this example. You could have just as easily selected the average grade column to sort by in order to see grades reported in ascending or descending order of each student's final score. These capabilities give you a fair amount of flexibility when sorting data for further processing or plotting using Excel's charting features.

## See Also

O'Reilly's Excel Hacks book, written by David and Raina Hawley, contains a couple of sorting hacks you might find useful. Hack #29 shows you how to sort using more than three columns, while Hack #30 shows you how to perform random sorts.

Figure 3-18. Sorted grades

The screenshot shows a Microsoft Excel window titled "Microsoft Excel - sort example.xls". The spreadsheet contains a table with the following data:

	A	B	C	D	E	F	G	H	I	J
1										
2										
3		<b>Student Name</b>	<b>Exam 1</b>	<b>Exam 2</b>	<b>Final Exam</b>	<b>Project</b>	<b>Average</b>			
4		Adams	66	47	42	77	58			
5		Buchanan	43	79	73	85	70			
6		Bush	94	93	86	90	91			
7		Carter	77	42	50	92	65			
8		Clinton	56	43	65	86	63			
9		Eisenhower	97	70	84	67	80			
10		Fillmore	91	90	79	97	89			
11		Jefferson	54	87	70	77	72			
12		Johnson	100	100	88	88	94			
13		Lincoln	100	88	92	92	93			
14		Madison	77	93	84	94	87			
15		Pierce	66	90	72	85	78			
16		Reagan	79	71	66	82	75			
17		Van Buren	80	75	85	71	78			
18		Washington	94	87	78	75	84			
19										
20										
21										
22										
23										
24										
25										
26										
27										
28										



← PREV

## Recipe 3.10. Filtering Data

### Problem

You have a spreadsheet that contains a large table of data and you'd like to be able to filter the data so you can view smaller subsets at a time.

### Solution

Use Excel's built-in filter capability.

### Discussion

[Figure 3-19](#) shows a spreadsheet that contains atomic weights for all elements and their isotopes.

The data used in this example was downloaded from the freely available Atomic Weights and Isotopic Compositions database on the National Institute of Standards (NIST) and Technology web site, <http://www.nist.gov>. The data shown here is an abridged version of the full dataset prepared by J. S. Coursey, D. J. Schwab, and R. A. Dragoset, which is available on the NIST web site.

Figure 3-19. Unfiltered data

	A	B	C	D	E	F	G	H	I
1	Isotope		Relative Atomic Mass						
2	H	1	1.0078250321(4)						
3	D	2	2.0141017780(4)						
4	T	3	3.0160492675(11)						
5	H	4	4.02783(12)						
6	H	5	5.03954(102)						
7	H	6	6.04494(28)						
8	He	3	3.0160293097(9)						
9	He	4	4.0026032497(10)						
10	He	5	5.012220(50)						
11	He	6	6.0188881(11)						
12	He	7	7.028030(30)						
13	He	8	8.033922(8)						
14	He	9	9.043820(70)						
15	He	10	10.052400(80)						
16	Li	4	4.02718(23)						
17	Li	5	5.012540(50)						
18	Li	6	6.0151223(5)						
19	Li	7	7.0160040(5)						
20	Li	8	8.0224867(5)						
21	Li	9	9.0267891(21)						

This dataset consists of nearly 3,000 records. Clearly it would be cumbersome to search for a particular element by having to manually scroll through all of the data. To make looking up specific records easier, you can filter the data.

The easiest way to filter data like this is to simply select Data Filter AutoFilter from the main menu bar. Excel will automatically scan your data and set up drop-down listboxes above each column, allowing you to filter the dataset. [Figure 3-20](#) shows the spreadsheet with autofiltering turned on.

Notice the little down arrow buttons located adjacent to the column headings. Excel populates these drop-down listboxes with filter options automatically determined from your data. For example, [Figure 3-21](#) shows what's in the listbox for the **Isotope** column.

Figure 3-20. Autofiltering active

	A	B	C	D	E	F	G	H	I
1	Isotop		Relative Atomic Mass						
2	H	1	1.0078250321(4)						
3	D	2	2.0141017780(4)						
4	T	3	3.0160492675(11)						
5	H	4	4.02783(12)						
6	H	5	5.03954(102)						
7	H	6	6.04494(28)						
8	He	3	3.0160293097(9)						
9	He	4	4.0026032497(10)						
10	He	5	5.012220(50)						
11	He	6	6.0188881(11)						
12	He	7	7.028030(30)						
13	He	8	8.033922(8)						
14	He	9	9.043820(70)						
15	He	10	10.052400(80)						
16	Li	4	4.02718(23)						
17	Li	5	5.012540(50)						
18	Li	6	6.0151223(5)						
19	Li	7	7.0160040(5)						
20	Li	8	8.0224867(5)						
21	Li	9	9.0267891(21)						

You can select any element listed to apply a filter such that only records with the selected element in the **Isotope** column will be shown. [Figure 3-22](#) shows the filtered list with the element o (oxygen), selected.

There are a few things to keep in mind when using autofiltering:

- Excel detects your dataset automatically and assumes it ends when it encounters a blank row. Therefore, be sure not to include any blank rows in your data, so as not to confuse Excel.
- Filtered lists are shown with row numbers in blue.
- Filtering does not delete any data. Rows of data that don't meet the filter criteria are merely hidden.
- You can add rows of data to the list simply by inserting or appending them to your existing data range.
- You can perform calculations on your filtered data just as you normally would. For example, you could include a column of calculated fields in column D.
- You can turn autofiltering off at any time by selecting **Data** → **Filter** → **AutoFilter** again.

Figure 3-21. Isotope filter

The screenshot shows a Microsoft Excel spreadsheet titled "Atomic Weights and Isotopic Compositions for All Elements.xls". The spreadsheet contains a table with the following data:

Isotope	Relative Atomic Mass
1	1.0078250321(4)
2	2.0141017780(4)
3	3.0160492675(11)
4	4.02783(12)
5	5.03954(102)
6	6.04494(28)
3	3.0160293097(9)
4	4.0026032497(10)
5	5.012220(50)
6	6.0188881(11)
7	7.028030(30)
8	8.033922(8)
9	9.043820(70)
10	10.052400(80)
4	4.02718(23)
5	5.012540(50)
6	6.0151223(5)
7	7.0160040(5)
8	8.0224867(5)
9	9.0267891(21)

The filter dropdown menu is open, showing a list of elements: Ac, Ag, Al, Am, Ar, As, At, Au, B, Ba, Be, Bh, Bi, Bk, Br, C, Ca. The status bar at the bottom indicates "Ready" and "NUM".

Figure 3-22. Filtered list

Microsoft Excel - Atomic Weights and Isotopic Compositions for All Elements.xls

File Edit View Insert Format Tools Data Window Help Adobe PDF Type a question for help

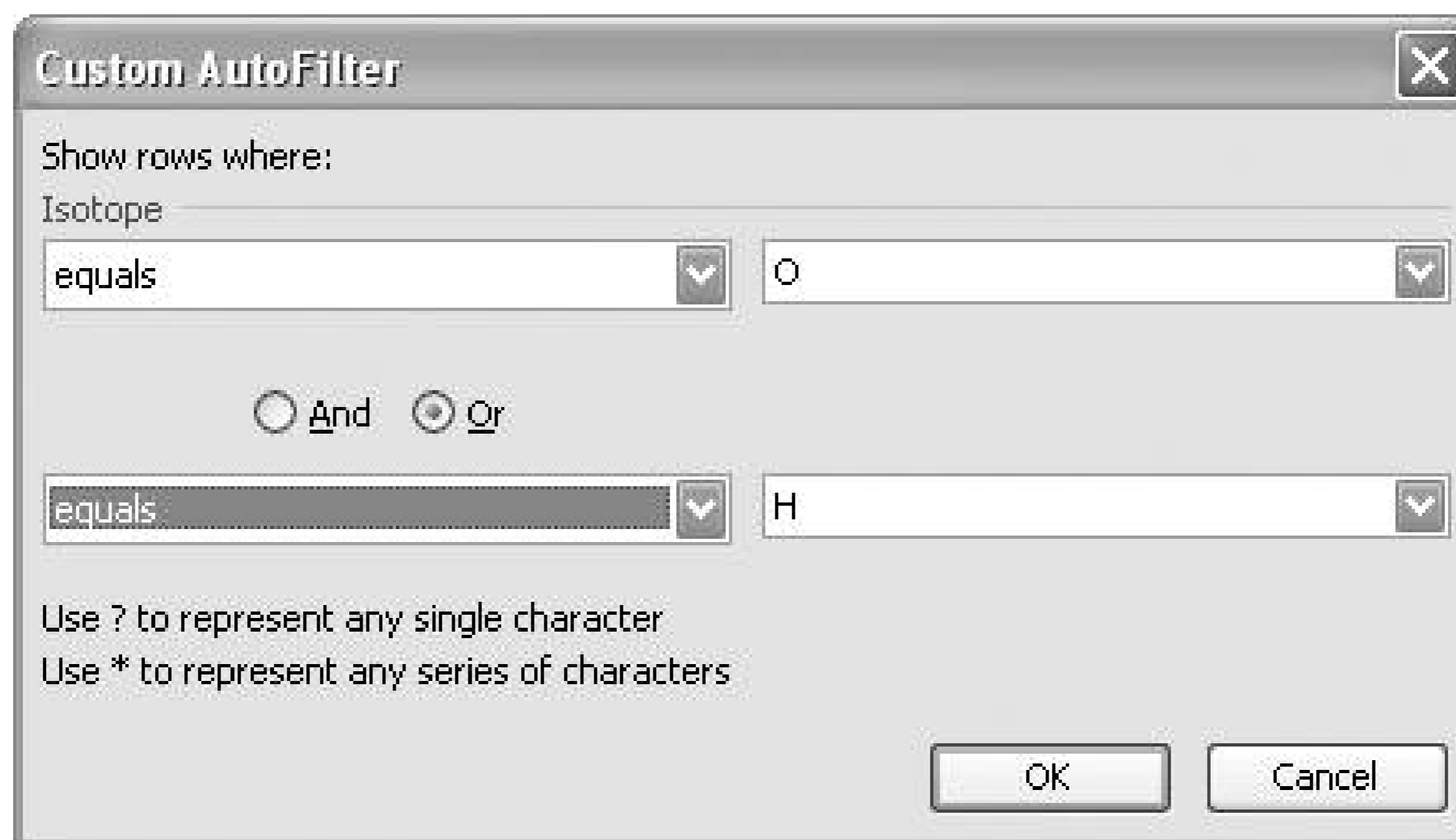
Security... Reply with Changes... End Review...

	A	B	C	D	E	F	G	H	I	J
1	Isotop		Relative Atomic Mass							
78	O	12	12.034405(20)							
79	O	13	13.024810(10)							
80	O	14	14.00859529(8)							
81	O	15	15.0030654(5)							
82	O	16	15.9949146221(15)							
83	O	17	16.99913150(22)							
84	O	18	17.9991604(9)							
85	O	19	19.003579(3)							
86	O	20	20.0040762(13)							
87	O	21	21.008655(13)							
88	O	22	22.009970(60)							
89	O	23	23.01569(11)							
90	O	24	24.02037(33)							
91	O	25	25.02914(40)#							
92	O	26	26.03775(46)#							
2937										
2938										
2939										
2940										

Atomic Weights and Isotopic Com / 15 of 2935 records found NUM

Occasionally, you may find you require a bit more flexibility when setting filter criteria. For example, say you wanted to see the isotopes for both hydrogen and oxygen at the same time. You can achieve this by setting up custom filter criteria. Take another look at [Figure 3-21](#) and notice the third item in the exposed filter list is "(Custom ...)." You can select this item to open the Custom AutoFilter dialog box (shown in [Figure 3-23](#)).

Figure 3-23. Custom AutoFilter dialog box



In this case I set up a custom filter to show all records whose **Isotope** column contains **O** or **H**. The drop-down listboxes showing "equals" in [Figure 3-21](#) allow you to select other criteria as well, such as "equals," "not equal," "greater than," "less than," "begins with," and "contains." This gives you quite a bit of flexibility. Pressing OK at this point will show a filtered list containing all records whose Isotope column contains **O** or **H**.

You should also be aware that you can specify criteria in more than one column at a time. Doing so will instruct Excel to show all records that meet the criteria specified in the first column *and* the next column and so on. Combine this capability with custom filters and you have a handy filtering mechanism at your disposal.

## See Also

Occasionally your data may be so complex that you require more advanced filtering capabilities. Excel facilitates this; see the help topic "Filter by using advanced criteria" in Excel's online help for more information.

## Recipe 3.11. Looking Up Values in Tables

### Problem

You're using tabular data in calculations and you need an easy way to look up values in the table for use in other calculations.

### Solution

Use the built-in functions [Lookup](#), [VLookup](#), [HLookup](#), [Match](#), or [Index](#).

### Discussion

Excel provides five functions to help you access data in tabular form. These functions come in handy when performing calculations that require looking up data from standard tables. For example, you can develop a spreadsheet to perform thermodynamic calculations and include standard steam tables in this spreadsheet. This way, instead of manually looking up fluid properties to use in your calculations, you can automatically include that data via the lookup functions discussed here.

### LOOKUP

The syntax for `LOOKUP` is `=LOOKUP(lookup value, lookup vector, result vector)`. `Lookup` finds the value in *result vector* corresponding to the position of the value in *lookup vector* that matches *lookup value*. The lookup vector must be sorted in ascending order (see [Recipe 3.9](#)). Also, the lookup and result vectors may be row or column ranges.

You can use text or a number as the lookup value; the function just compares the lookup value to the data stored in the lookup vector to find a match. This is an important caveat with this function (and the others discussed here), as no interpolation is performed. The function looks for an exact match and if it does not find one, it returns the value in the result vector corresponding to the value in the lookup vector that is closest to but less than the lookup value. (Take a look at the recipes in [Chapter 8](#) to learn how to interpolate data in Excel.)

[Figure 3-24](#) illustrates a typical application of `LOOKUP`. In this example, we have a simple table of data that presents design factors, say for ship structural design, as a function of length of the ship under consideration, the ship's hull material, and the units used. (Don't bother looking up this table of data in a marine design handbook it doesn't exist. I just made it up for illustration purposes.)

Let's say we have a design formula, or better yet a whole series of calculations that must be

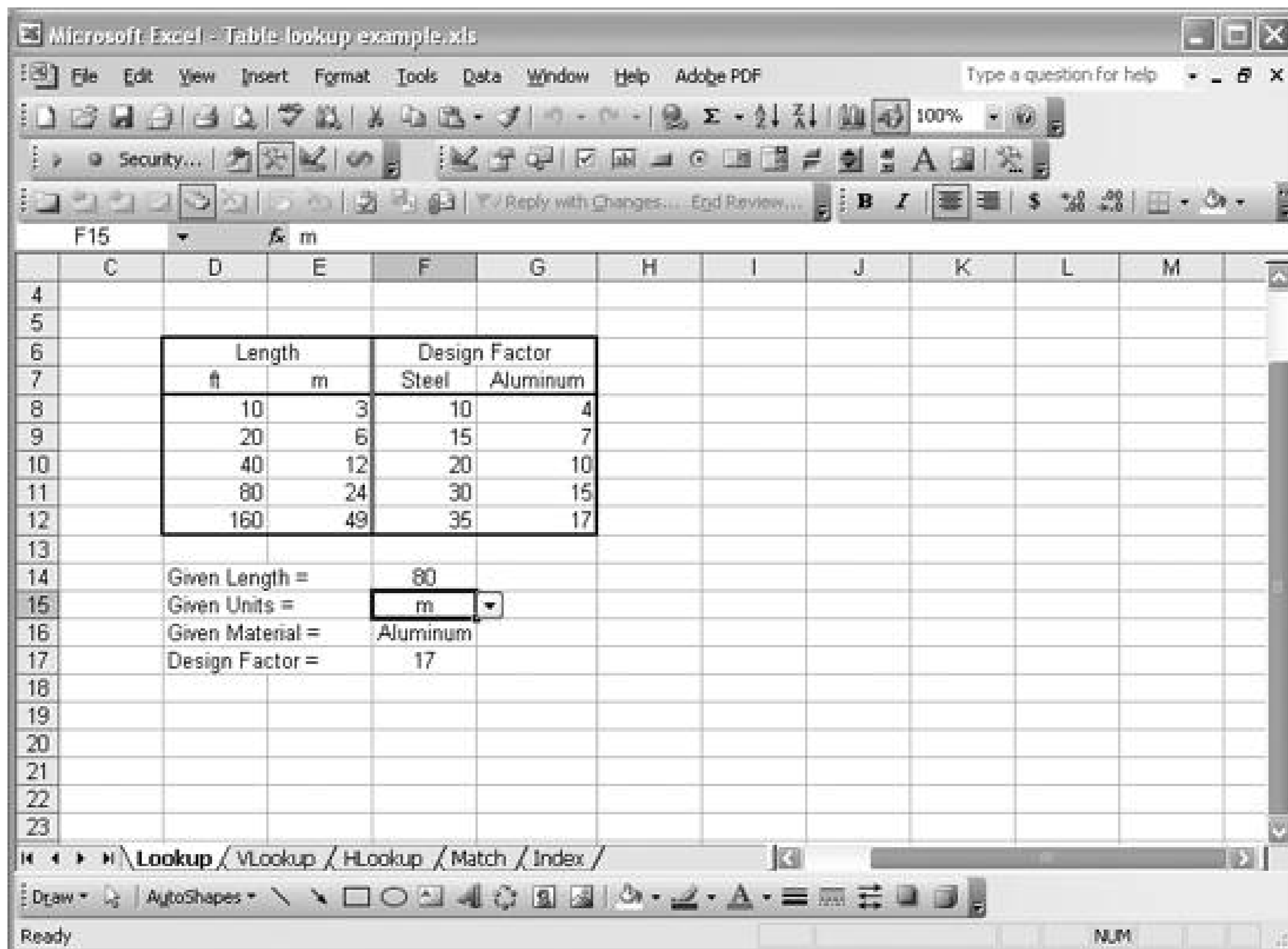
performed, that requires the design factor as input. Including the design factor table in the spreadsheet allows you to conveniently select the factor by specifying the length, material type, and units, rather than looking up the data in a handbook and entering it manually. This is especially handy if you distribute your spreadsheet to someone else.

The three values in cells F14 through F16 are user input values corresponding to the given length, units, and material, respectively. The design factor corresponding to this given data is contained in cell F17. The formula in cell F17 that actually performs the lookup is:

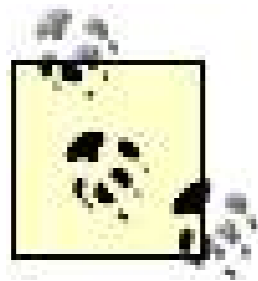
```
=LOOKUP(F14, IF(F15="ft",D8:D12,E8:E12), IF(F16="Steel", F8:F12,G8:G12))
```

This formula looks more complicated than it is, due to the nested **IF** functions.

Figure 3-24. LOOKUP







IF functions are like conditional statements in traditional programming languages. They are quite handy for use in spreadsheets. The syntax for the IF function is `=IF(logical expression, value if true, value if false)`. The logical expression can be any expression that evaluates to `true` or `false`. You can use cell references in a logical expression along with logical operators such as `>`, `<`, and `=`. There are also a few other logical functions (`AND`, `OR`, and `NOT`) that you can use to construct more elaborate logical statements like `=IF(AND(C5>F9, G9=2), ... )` or `=IF(AND(OR(C5>F9, G9=2), NOT(C11)), ...,)`. `value if true` is what will be used if the logical expression evaluates to `true`, whereas `value if false` will be used if the logical expression evaluates to `false`. Either of these values may be hardcoded numbers or text, cell references, or nested functions (for example, you can construct nested conditional statements by including another IF function as `value if true` or `value if false`, and so on).

Basically, the given length, cell F14, is used as the value to look up in the lookup vector, but the lookup vector itself depends on which units are selected. An IF function is used to test the given units and return the correct lookup vector. Another IF function is used to test the given material and return the correct result vector. The design factor is extracted from the result vector on the row corresponding to the row containing the given lookup value (the given length, in this case).

If you take a close look at [Figure 3-24](#), you will notice a little down arrow button next to selected cell, F15, corresponding to the given units. Pressing this button will display a listbox allowing you to select which units to enter (`ft` or `m`) instead of manually typing the units. I used the same technique for the given material input cell. Using drop-down selection lists in this way is a great way to control the input value of a cell, being sure the value entered is a valid value. This provides a more robust spreadsheet, especially when the spreadsheet is distributed to other users. To actually get the cell to display a list of valid data, use Excel's Data Validation feature (see [Recipe 1.15](#)). The Allow field shown in [Figure 1-21](#) includes a List option. When you select a List in the Allow field, a Source field will be displayed, allowing you to select a cell range containing the data to show in the list. For cell F15 in [Figure 3-24](#), I set the source range to D7:E7, which includes the two valid options, `ft` and `m`.

## VLOOKUP

The syntax for VLOOKUP is `=VLOOKUP(lookup value, table, column, match flag)`. VLOOKUP searches for the given `lookup value` in the first column of the range of cells given by `table` and returns the value on the corresponding row in the given `column`. `match flag` is optional. If it is set to `TRUE` or is omitted, then VLOOKUP will find an exact match or a close match. In this case, the first column in the given table should be sorted in ascending order (see [Recipe 3.9](#)). If an exact match is not found, the function returns the value in the given column corresponding to the value in the first column that is closest to but less than the lookup value. If `match flag` is `false`, VLOOKUP will look only for exact matches. If one is not found, the function returns `#N/A`, indicating the absence of an exact match.

[Figure 3-25](#) shows a spreadsheet containing a table of physical properties for water. The first column

of the table contains temperatures ranging from 10 to 90°C. Typical calculations would involve looking up physical properties in such a table for use in other calculations.

Physical properties of water for a given temperature can be retrieved from this table using **VLOOKUP**. In [Figure 3-25](#), cell E17 contains the given temperature for which properties are retrieved. In this case, density, internal energy, enthalpy, entropy, specific heats, sound speed, and viscosity corresponding to the given temperature are shown in cells E18 through E25. Each of these cells contains a formula that utilizes **VLOOKUP**. For example, cell E18 contains the formula `=VLOOKUP($E$17,$C$4:$K$12,2)`. The value to be looked up in the first column of the data table is stored in cell E17; this is the given temperature. The data table itself is stored in the cell range C4:K12. The last parameter in this formula indicates the column in the data table from which to retrieve data corresponding to the given temperature. Column 2 in this case is the density of water. The remaining cells contain similar formulas. The only differences are the last parameter in **VLOOKUP** (i.e., the column index from which to retrieve the desired value).

Figure 3-25. VLOOKUP

Temperature (C)	Density (kg/m <sup>3</sup> )	Internal Energy (kJ/kg)	Enthalpy (kJ/kg)	Entropy (J/g <sup>o</sup> K)	Cv (J/g <sup>o</sup> K)	Cp (J/g <sup>o</sup> K)	Sound Spd. (m/s)	Viscosity (Pa*s)
10	999.7	42.018	42.119	0.15108	4.1906	4.1962	1447.3	0.0013069
20	998.21	83.906	84.007	0.29646	4.1567	4.1641	1482.3	0.0010016
30	996.66	125.72	125.82	0.43673	4.1172	4.1798	1509.2	0.00079735
40	992.22	167.51	167.62	0.57237	4.0734	4.1794	1528.9	0.00065298
50	988.04	209.32	209.42	0.70377	4.0262	4.1813	1542.6	0.00054685
60	983.2	251.15	251.25	0.83125	3.9766	4.185	1551	0.0004664
70	977.76	293.02	293.12	0.95609	3.9251	4.1901	1554.7	0.00040389
80	971.79	334.95	335.05	1.0755	3.8728	4.1968	1554.4	0.00035435
90	965.31	376.96	377.06	1.1928	3.8203	4.2052	1550.5	0.00031441

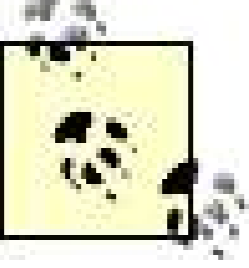
  

Temperature (C)	50
Density (kg/m <sup>3</sup> )	988.04
Internal Energy (kJ/kg)	209.32
Enthalpy (kJ/kg)	209.42
Entropy (J/g <sup>o</sup> K)	0.70377
Cv (J/g <sup>o</sup> K)	4.0262
Cp (J/g <sup>o</sup> K)	4.1813
Sound Spd. (m/s)	1542.6
Viscosity (Pa*s)	0.00054685

## HLOOKUP

**HLOOKUP** is very similar to **VLOOKUP**; however, **HLOOKUP** uses the first row of data in a given cell range as the lookup array. The syntax for **HLOOKUP** is `=HLOOKUP(lookup value, table, row, match flag)`. The only difference between this syntax and that of **VLOOKUP** is the third parameter; **HLOOKUP** requires a row number to be used to retrieve the result.

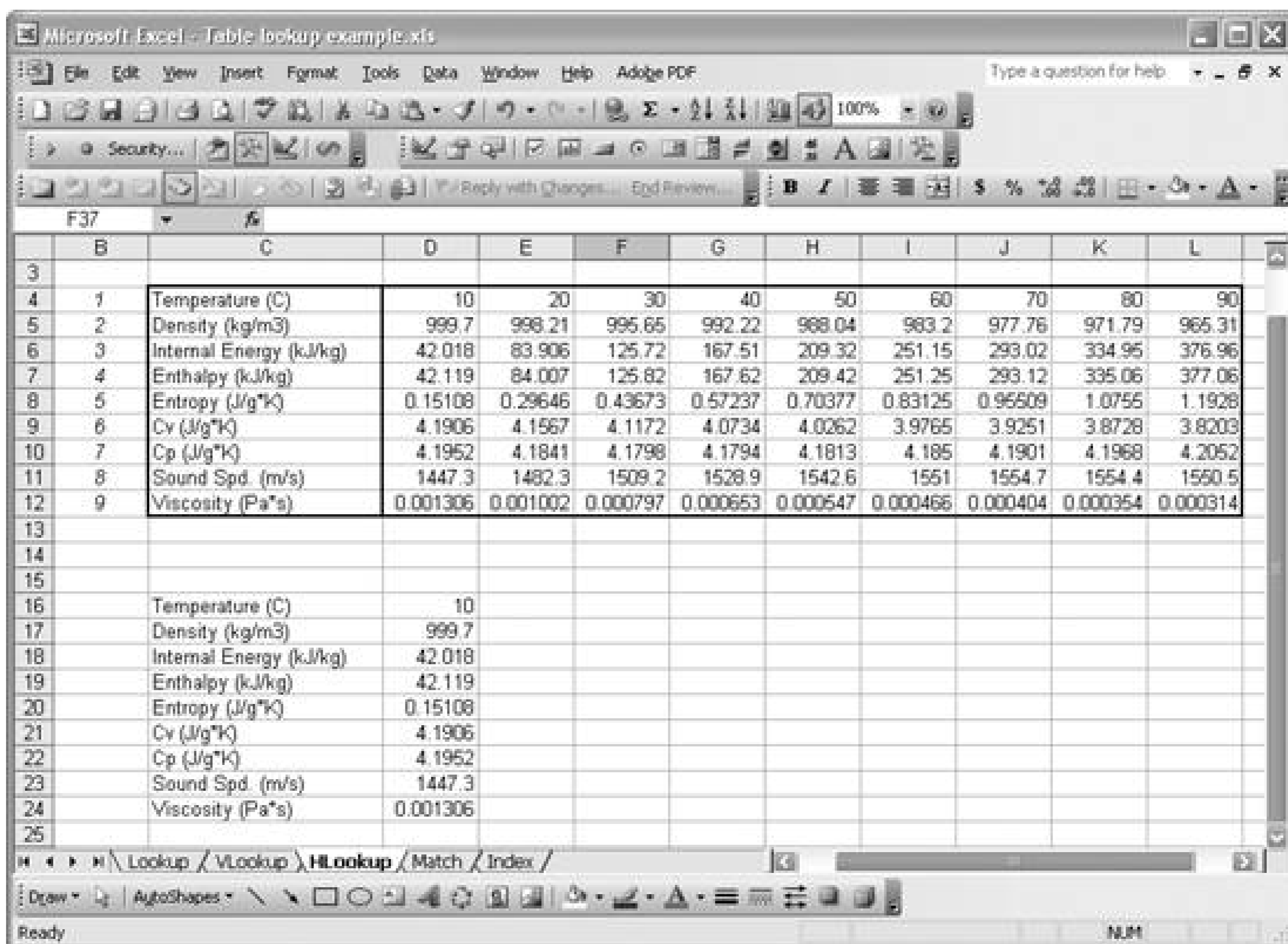
Figure 3-26 contains the same data shown in Figure 3-25; however, in this new example the data has been transposed. Now, temperatures are on the first row instead of in the first column.



It's easy to transpose data in Excel. Select the range of cells you want to transpose and copy it, using Ctrl-C or Edit Copy. Select a new cell where you want to paste the data and select Edit Paste Special ....In the Paste Special dialog box, check the Transpose box and press the OK button.

Physical properties of water for a given temperature can be retrieved from this table using **HLOOKUP**. For example, cell D17 extracts the density corresponding to the temperature given in cell D16. The formula in cell D17 is `=HLOOKUP($D$16,$D$4:$L$12,2)`. The formulas for the other properties in cells D18 through D24 are very similar; only the result row number changes.

Figure 3-26. HLOOKUP



	B	C	D	E	F	G	H	I	J	K	L
3											
4	1	Temperature (C)	10	20	30	40	50	60	70	80	90
5	2	Density (kg/m3)	999.7	998.21	995.65	992.22	988.04	983.2	977.76	971.79	965.31
6	3	Internal Energy (kJ/kg)	42.018	83.906	125.72	167.51	209.32	251.15	293.02	334.95	376.96
7	4	Enthalpy (kJ/kg)	42.119	84.007	125.82	167.62	209.42	251.25	293.12	335.06	377.06
8	5	Entropy (J/g*K)	0.15108	0.29646	0.43673	0.57237	0.70377	0.83125	0.95509	1.0755	1.1928
9	6	Cv (J/g*K)	4.1906	4.1567	4.1172	4.0734	4.0262	3.9765	3.9251	3.8728	3.8203
10	7	Cp (J/g*K)	4.1952	4.1841	4.1798	4.1794	4.1813	4.185	4.1901	4.1968	4.2052
11	8	Sound Spd. (m/s)	1447.3	1482.3	1509.2	1528.9	1542.6	1551	1554.7	1554.4	1550.5
12	9	Viscosity (Pa*s)	0.001306	0.001002	0.000797	0.000653	0.000547	0.000466	0.000404	0.000354	0.000314
13											
14											
15											
16		Temperature (C)	10								
17		Density (kg/m3)	999.7								
18		Internal Energy (kJ/kg)	42.018								
19		Enthalpy (kJ/kg)	42.119								
20		Entropy (J/g*K)	0.15108								
21		Cv (J/g*K)	4.1906								
22		Cp (J/g*K)	4.1952								
23		Sound Spd. (m/s)	1447.3								
24		Viscosity (Pa*s)	0.001306								
25											

## MATCH

**MATCH** returns the position of an item in an array. For example, in the spreadsheet shown in Figure 3-27 we could use **MATCH** to return the position of the cell containing the text **Sound Spd. (m/s)** in the

array of cells C4 to C12. In this case, the position returned would be 8. The syntax for **MATCH** is **=MATCH(lookup value, lookup array, match type)**. Here, *lookup value* is the value to be looked up in the array *lookup array*. The last parameter, *match type*, tells **MATCH** how you want to determine a match. If *match type* is 0, then **MATCH** looks for an exact match. If *match type* is -1, then **Match** returns the position of the smallest value greater than or equal to *lookup value*. If *match type* is 1, then **MATCH** returns the position of the largest value less than or equal to *lookup value*. Remember, **MATCH** returns the position of the matched value, not the value itself.

The spreadsheet in [Figure 3-27](#) shows how you might use **MATCH**. In this case, we have the same data table as in the **HLOOKUP** example. This time, however, in cell D18 we have this formula: **=HLOOKUP(D17,D4:L12,MATCH(C18,C5:C12,0)+1)**. This time I nested a call to **MATCH** as the third parameter in **HLOOKUP**. In the previous example, I specified a row in which to look up the result. In this case, I use **MATCH** to find the row corresponding to the description of the property selected in cell C18. Cell C18 contains a drop-down listbox that allows you to select any of the properties in the table except the temperature property. I used the same data validation technique described earlier in this recipe to create the selection list in this example.

Figure 3-27. MATCH

The screenshot shows a Microsoft Excel spreadsheet with a data table and a formula. The data table is located in cells C4:L12 and contains the following values:

			10	20	30	40	50	60	70	80	90
4	1	Temperature (C)									
5	2	Density (kg/m3)	999.7	998.21	995.65	992.22	988.04	983.2	977.76	971.79	965.31
6	3	Internal Energy (kJ/kg)	42.018	83.906	125.72	167.51	209.32	251.15	293.02	334.95	376.96
7	4	Enthalpy (kJ/kg)	42.119	84.007	125.82	167.62	209.42	251.25	293.12	335.06	377.06
8	5	Entropy (J/g*K)	0.15108	0.29646	0.43673	0.57237	0.70377	0.83125	0.95509	1.0755	1.1928
9	6	Cv (J/g*K)	4.1906	4.1567	4.1172	4.0734	4.0262	3.9765	3.9251	3.8728	3.8203
10	7	Cp (J/g*K)	4.1952	4.1841	4.1798	4.1794	4.1813	4.185	4.1901	4.1968	4.2052
11	8	Sound Spd. (m/s)	1447.3	1482.3	1509.2	1528.9	1542.6	1551	1554.7	1554.4	1550.5
12	9	Viscosity (Pa*s)	0.001306	0.001002	0.000797	0.000653	0.000547	0.000466	0.000404	0.000354	0.000314

The formula in cell D18 is **=HLOOKUP(D17,D4:L12,MATCH(C18,C5:C12,0)+1)**. The formula bar shows the formula being entered. The status bar at the bottom indicates 'Ready' and 'NUM'.

The call to **MATCH** in the call to **HLOOKUP** returns the position of the selected description in the list of descriptions contained in cells C5 to C12. To get the proper row number for the table in the cell range C4:L12, you have to add 1 to the value returned by **MATCH** since the number of rows in the description cell list is shifted by 1 from the data table.

## INDEX

**INDEX** is useful for extracting a value from a table (cell range) given the position of the value in the table. The syntax for **INDEX** is `=INDEX(cell reference, row, column)`. **INDEX** will return the value in the cell at position *row* and *column* in the range specified by *cell reference*. This is like extracting the value of an element in a matrix given its row and column position.

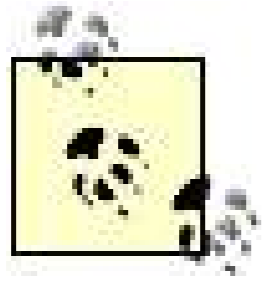
[Figure 3-28](#) shows a spreadsheet containing a set of data resembling a 4x3 matrix.

Cell D10 contains the value of element [2,2] extracted using **INDEX**. The formula in cell D10 is `=INDEX(D4:F7,2,2)`.

If you want to extract an entire row, set the column to 0 in the call to **INDEX**. Likewise, to extract an entire column, set the row to 0. To extract an entire row or column at a time, you need to use array formulas. These are nothing more than formulas that act over a range of cells (instead of over a single cell, like the formulas I've discussed so far).

Figure 3-28. INDEX

To extract the first row as shown in [Figure 3-28](#), I first selected the destination cell range D12:F12 by clicking and dragging the mouse. Then I pressed the equals sign to begin writing the formula: `=INDEX(D4:F7,1,0)`. Notice that I set the column to 0, indicating that I want an entire row. To enter this formula, which is now an array formula, press Ctrl-Shift-Enter instead of the usual Enter. This tells Excel you're entering an array formula, and it will fill all the selected cells with results of the formula you entered.



Array formulas are shown in the formula bar with braces around them, like `{=INDEX(D4:F7,1,0)}`. I show you how to use array formulas in other recipes throughout this book. Take a look at [Recipe 5.2](#) and [Recipe 7.11](#) for some examples.

To extract the entire second column as shown in [Figure 3-28](#), select the range D14:D17 and Ctrl-Shift-Enter the formula `=INDEX(D4:F7,0,2)`. In this case, the row is set to 0, indicating that you want to extract an entire column.

`INDEX` is handy when working with matrices or matrix-like data and calculations. See [Chapter 7](#) for more recipes dealing with matrix calculations.

## See Also

In this recipe, I used and briefly discussed Excel's `IF` function. While its use is fairly straightforward, you can read more about it and other logical functions in Excel's online help. Look for the topic "Logical Functions."

## Recipe 3.12. Retrieving Data from XML Files

### Problem

You need to analyze some data contained in an XML file and would like to do so in Excel.

### Solution

Import the XML file into Excel using File → Open ... and then manipulate the data as you normally would in Excel.

### Discussion

Extensible Markup Language (XML) files are text files that use the XML protocol for structuring data. XML was developed as a standard, cross-platform data exchange file format that would make transferring data from platform to platform and application to application easy. In this regard, XML is very similar to HTML used for web pages.

Programmers using a variety of languages can create XML datafiles of program output. Database applications can put their data in XML files. In fact, Microsoft Access allows you to import and export XML files. If you need to get at the data contained in an XML file, you can do so using any of a number of programs that read XML files, like Access. However, if you're also interested in manipulating or analyzing the data in Excel, then the more direct route is to open the XML file directly in Excel. Select File → Open ... from the main menu bar as you normally would to open a file, and select "XML Files (\*.xml)" from the "Files of Type" drop-down list instead of the default XLS file type.

This feature is currently available only in Microsoft Office Professional Edition 2003 and Microsoft Office Excel 2003.

[Example 3-2](#) shows the partial contents of a sample XML file as if opened in a text editor.

### Example 3-2. Sample XML file

```

- <Materials>
  <ID>1</ID>
  <Description>5083 - H111</Description>
  <UT>269</UT>
  <Sigma_x0020_YW>145</Sigma_x0020_YW>
  <Tau>83</Tau>
  <Sigma_x0020_Y>165.62</Sigma_x0020_Y>
  <Type>Aluminum</Type>
  <Units>Metric</Units>
</Materials>
- <Materials>
  <ID>2</ID>
  <Description>5083 - H116</Description>
  <UT>276</UT>
  <Sigma_x0020_YW>165</Sigma_x0020_YW>
  <Tau>96</Tau>
  <Sigma_x0020_Y>199.92</Sigma_x0020_Y>
  <Type>Aluminum</Type>
  <Units>Metric</Units>
</Materials>
- <Materials>
  <ID>3</ID>
  <Description>5083 - H321</Description>
  <UT>276</UT>
  <Sigma_x0020_YW>165</Sigma_x0020_YW>
  <Tau>96</Tau>
  <Sigma_x0020_Y>215.6</Sigma_x0020_Y>
  <Type>Aluminum</Type>
  <Units>Metric</Units>
</Materials>

```

Clearly the data contained in this file is difficult to decipher in this form. Once imported into Excel, the data appears as shown in [Figure 3-29](#).

Now the data is in a useful tabular form in Excel. At this point you can manipulate it as you would any other spreadsheet.

When Excel imports an XML file, it attempts to infer the structure of the data unless an *XMLschema* is available. A schema is like a template for an XML file. It provides further guidance on how to interpret the data in an XML file (e.g., how to interpret text versus numerical values and so on). Schemas can be embedded within an XML file or they can reside as separate XSD (XML Schema Definition) files. Excel calls these *XML Map* files, and they can be loaded in as well. See the help topic on XML Map files for your version of Excel for more information on loading map files.

Figure 3-29. XML data in Excel



	A	B	C	D	E	F	
1	Description	Sigma_x0020_Y	Sigma_x0020_Y	Sigma_x0020_YW	Sigma_x0020_YW	Tau	Ty
2	5083 - H111	165.62	165.62	145	145	83	Al
3	5083 - H116	199.92	199.92	165	165	96	Al
4	5083 - H321	215.6	215.6	165	165	96	Al
5	5083 - H323	234.22	234.22	165	165	96	Al
6	5083 - H343	268.52	268.52	165	165	96	Al
7	5086 - H111	142.1	142.1	124	124	69	Al
8	5086 - H112 6 mm (0.25 in)	124.46	124.46	117	117	65	Al
9	5086 - H112 12 mm (0.5 in)	109.76	109.76	110	110	62	Al
10	5086 - H112 25 mm (1.0 in)	96.04	96.04	96.5	96.5	55	Al
11	5086 - H116	193.06	193.06	131	131	76	Al
12	5086 - H32	193.06	193.06	131	131	76	Al
13	5086 - H34	234.22	234.22	131	131	76	Al
14	5383 - O	142.1	142.1	145	145	83	Al
15	5383 - H111	142.1	142.1	145	145	83	Al
16	5383 - H116	215.6	215.6	145	145	83	Al
17	5383 - H321	215.6	215.6	145	145	83	Al
18	5383 - H34	264.6	264.6	145	145	83	Al
19	5454 - H111	131.32	131.32	110	110	65	Al
20	5454 - H112	82.32	82.32	83	83	48	Al

## See Also

The O'Reilly book *Excel Hacks*, by David and Raina Hawley, provides a few useful hacks on dealing with XML data in Excel. For more in-depth coverage, you should check out the O'Reilly book *Office 2003 XML*, by Evan Lenz, Mary McRae, and Simon St.Laurent. [Chapter 6](#) is particularly interesting and even covers loading and saving XML files using VBA.

# Chapter 4. Charting

[Section 4.0. Introduction](#)

[Recipe 4.1. Creating Simple Charts](#)

[Recipe 4.2. Exploring Chart Styles](#)

[Recipe 4.3. Formatting Charts](#)

[Recipe 4.4. Customizing Chart Axes](#)

[Recipe 4.5. Setting Log or Semilog Scales](#)

[Recipe 4.6. Using Multiple Axes](#)

[Recipe 4.7. Changing the Type of an Existing Chart](#)

[Recipe 4.8. Combining Chart Types](#)

[Recipe 4.9. Building 3D Surface Plots](#)

[Recipe 4.10. Preparing Contour Plots](#)

[Recipe 4.11. Annotating Charts](#)

[Recipe 4.12. Saving Custom Chart Types](#)

[Recipe 4.13. Copying Charts to Word](#)

[Recipe 4-14. Displaying Error Bars](#)

[← PREV](#)

## 4.0. Introduction

Excel has built-in support for preparing a wide variety of charts, including custom charts, in your spreadsheet. You can even copy charts you create and embed them in Word documents. Charting is a handy data visualization technique that may provide valuable insight into the experimental or calculated data you may be trying to analyze. Excel makes charting quick and easy, and provides a great deal of flexibility in customizing your charts. Excel's charting capabilities are not as sophisticated as those of some specialized scientific data visualization applications, but in many cases Excel may be the fastest way to visualize data. This is especially true if you're generating your data in Excel via calculations or if you have already imported your data into Excel using techniques discussed in [Chapter 3](#).

[← PREV](#)

[← PREV](#)

## Recipe 4.1. Creating Simple Charts

### Problem

You've imported or generated data and you'd like to plot your data for further analysis.

### Solution

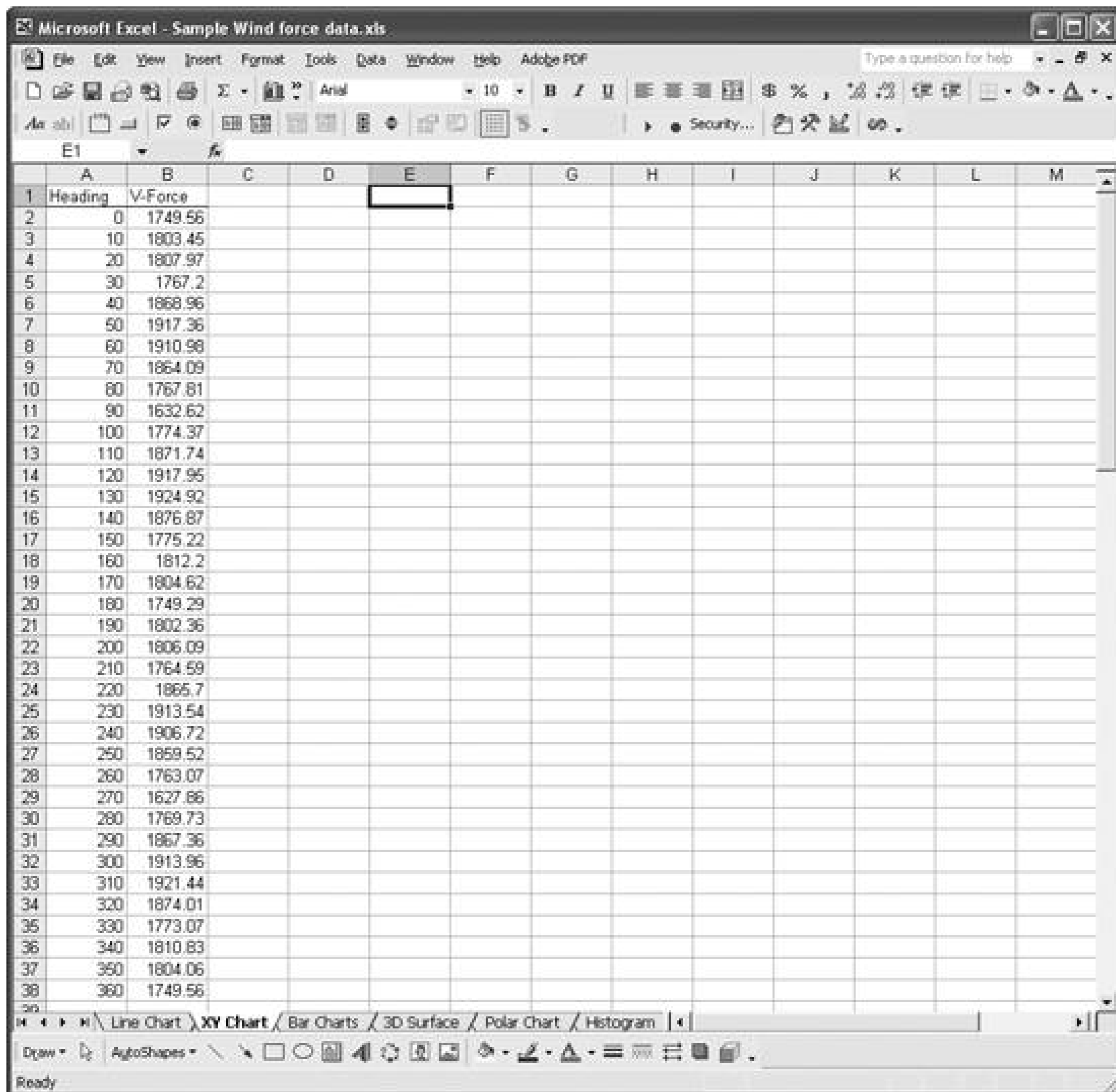
Use Excel's built-in charting features. Select the data you want to plot. Then select Insert Chart...from the main menu bar to launch the Chart Wizard. Now let the wizard walk you through the rest of the chart-creation process.

### Discussion

Let's walk through an example to illustrate the basic steps involved in creating a chart in Excel. [Figure 4-1](#) shows the data we'll use in this example.

The data consists of two columns. The first column contains heading angles in degrees and the second contains wind force in kilo-pounds (kips) corresponding to the heading angles. (In case you're curious, this data represents the wind forces exerted on an offshore drilling rig operating in the Gulf of Mexico.)

Figure 4-1. Sample XY data

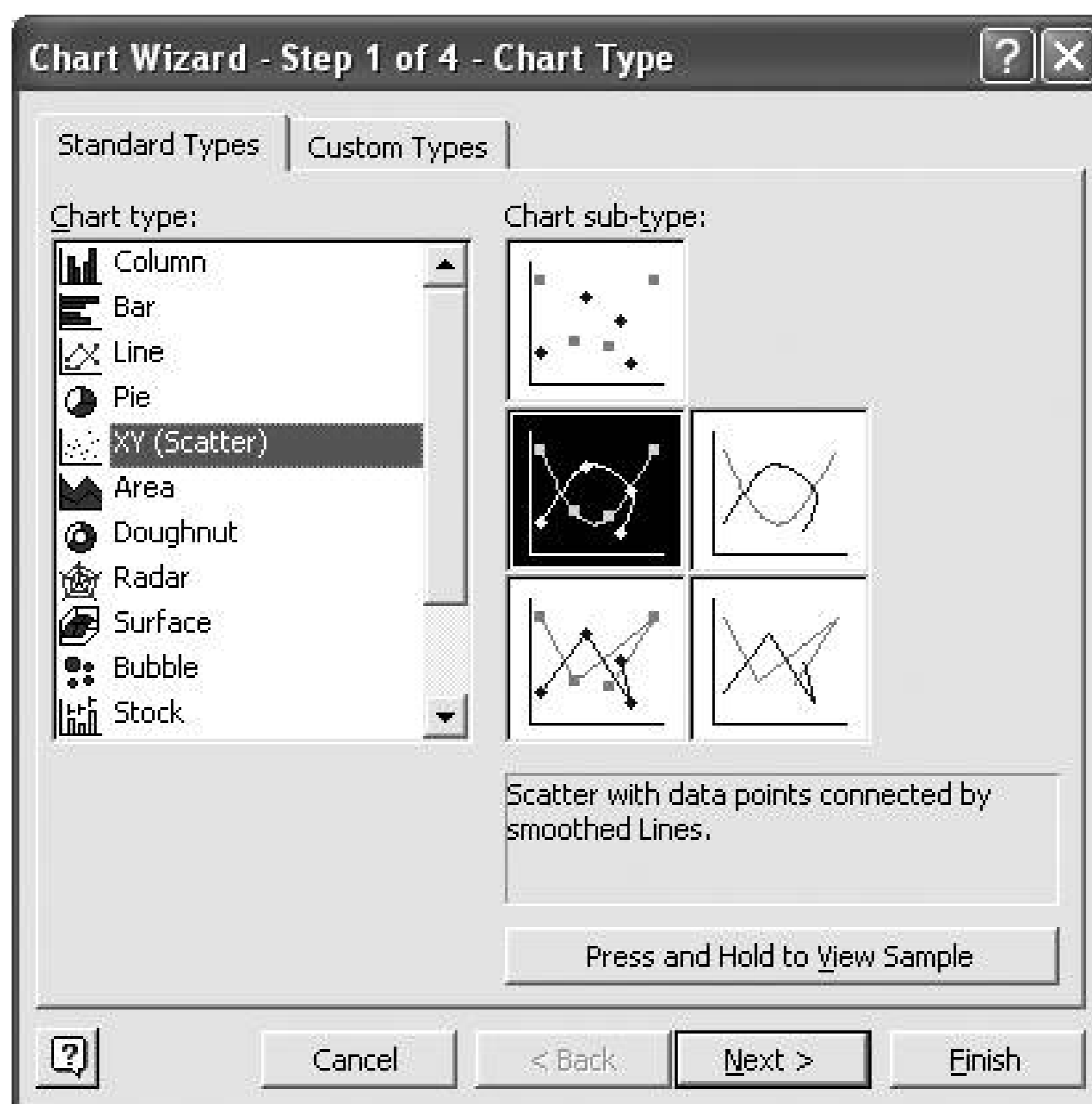


This is a good example of the sort of data that can be plotted on an XY chart. Basically, the heading data represents abscissa ( $x$ -axis) values, while force data represents ordinate ( $y$ -axis) data. Plotting the data should reveal how the wind force varies with heading.

To plot the data, click and drag the cell range containing the data. In this case, click and drag from cell A1 to B38. It's okay to include the column headings in the selected range, because Excel will use these as default series names.

After selecting the data, select Insert > Chart... from the main menu bar or press the Chart Wizard button on the toolbar (in [Figure 4-1](#), it's the eighth button from the left under the main menu bar and has a little bar chart image on it). This activates the Chart Wizard (see [Figure 4-2](#)).

Figure 4-2. Chart Wizard: Step 1



Excel offers a variety of built-in chart types, as shown in the "Chart type" list in [Figure 4-2](#). See [Recipe 4.2](#) for a list of other kinds of charts you can create in Excel. Here, we're working with an XY (or *scatter*) chart, so select XY (Scatter) in the "Chart type" list.

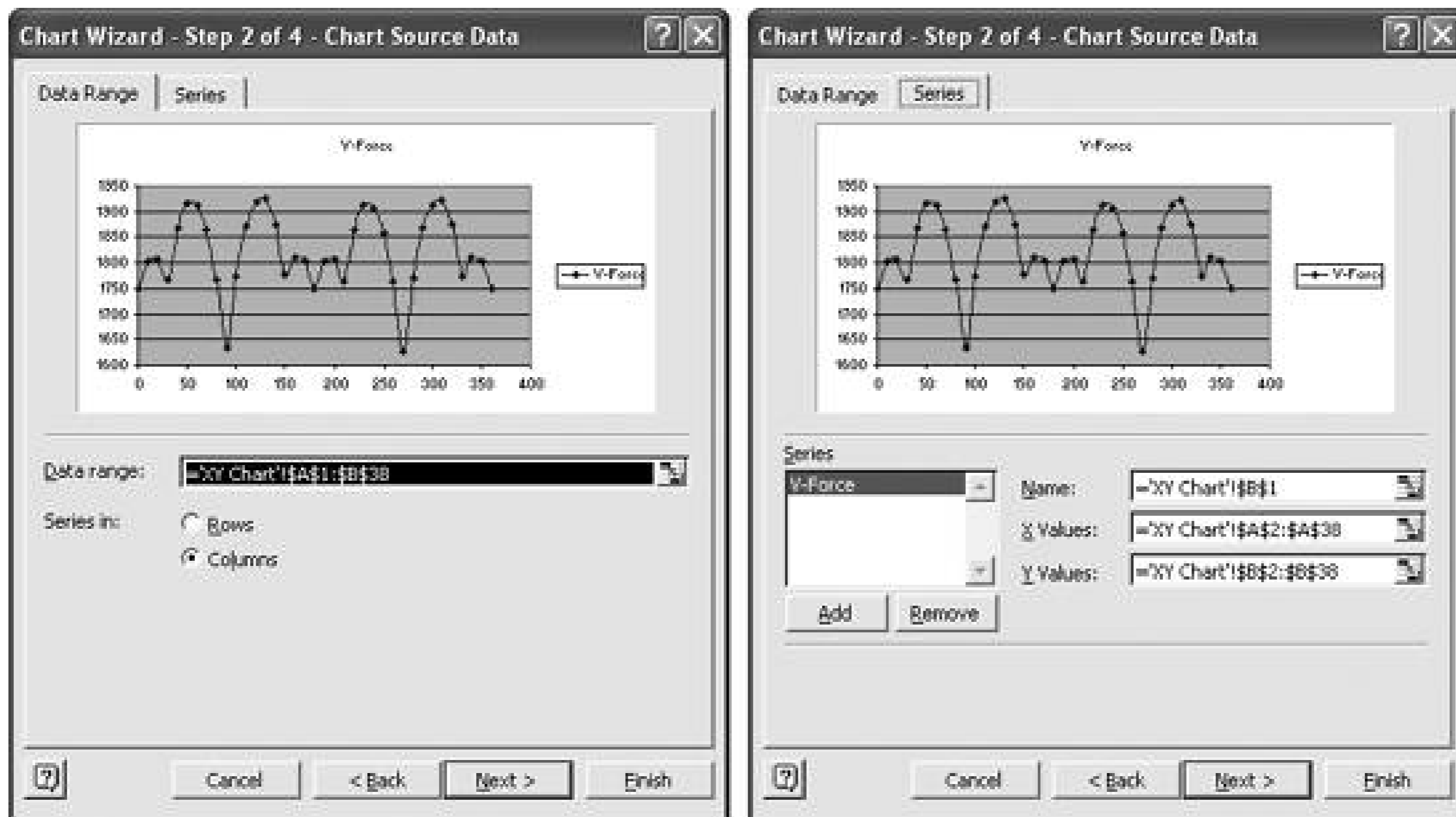
Both Line and XY scatter chart types allow you plot points or lines and curves through data. XY charts allow you to specify both the ordinate and abscissa of each point, which can be at irregular spaces. Line charts plot ordinates versus equally spaced abscissas corresponding to the data point number. XY scatter charts give you greater flexibility when plotting two-dimensional data.

The right side of the Chart Wizard contains images of available sub-types for each major type of chart. XY charts have the five sub-types shown in [Figure 4-2](#). The images are fairly representative of the sub-types. The choices here include simply plotting data points, plotting points with smoothed lines passing through them, plotting just the smooth lines, plotting points with straight line segments passing through them, and plotting just the straight line segments. I've selected smooth lines with point markers for this example.

If you press the button labeled "Press and Hold to View Sample," you'll see a little sample of your data plotted with the selected chart type and sub-type. Click the Next button to go to the second Chart Wizard step, or click Finish to stop here and create your chart, after which you can view and edit it manually.

[Figure 4-3](#) shows the second step in the Chart Wizard.

Figure 4-3. Chart Wizard: Step 2



I included two views of the step 2 dialog box in [Figure 4-3](#). The one on the left corresponds to the Data Range tab, while the one on the right corresponds to the Series tab.

On the Data Range tab, you can change the data range you selected for plotting. Since I selected the data before launching the Chart Wizard, the data range field is already populated with the correct data range. Excel makes an assumption as to whether or not your data is oriented in columns of data or rows of data. In this case, the data is contained in two columns and Excel correctly defaulted to "Series in Columns." Excel gets this right most of the time and usually goofs up only when you don't select headings in your data range and your data range is nearly square.

The Series tab shows the data series that will be plotted on the chart. In this example, the first column in the selected range represents  $x$ -values and the second column represents  $y$ -axis data. You can have more than one column of  $y$ -axis data corresponding to a single column of  $x$ -axis data. In such cases you could plot more than one data series. Here, you have the opportunity to add more data series to your chart. Further, you can always access the series dialog to add new series to an existing chart.

Once a chart has been created, you can access the Source Data dialog box to edit, add, or delete series. Right-click on the chart and then select Source Data from the pop-up menu to display the Source Data dialog box.

To add a new series, select each field on the right side of the dialog and fill in the appropriate cell

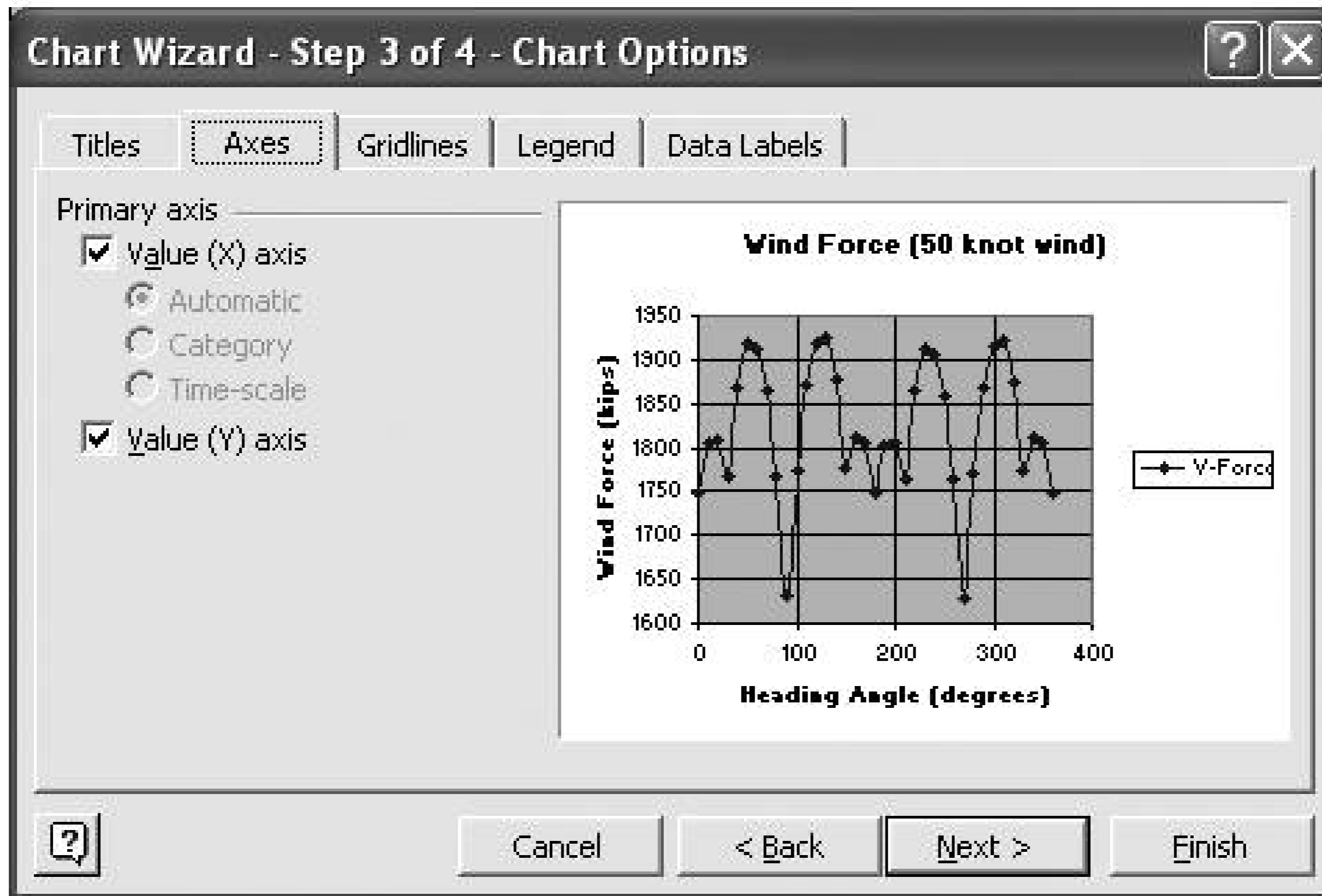
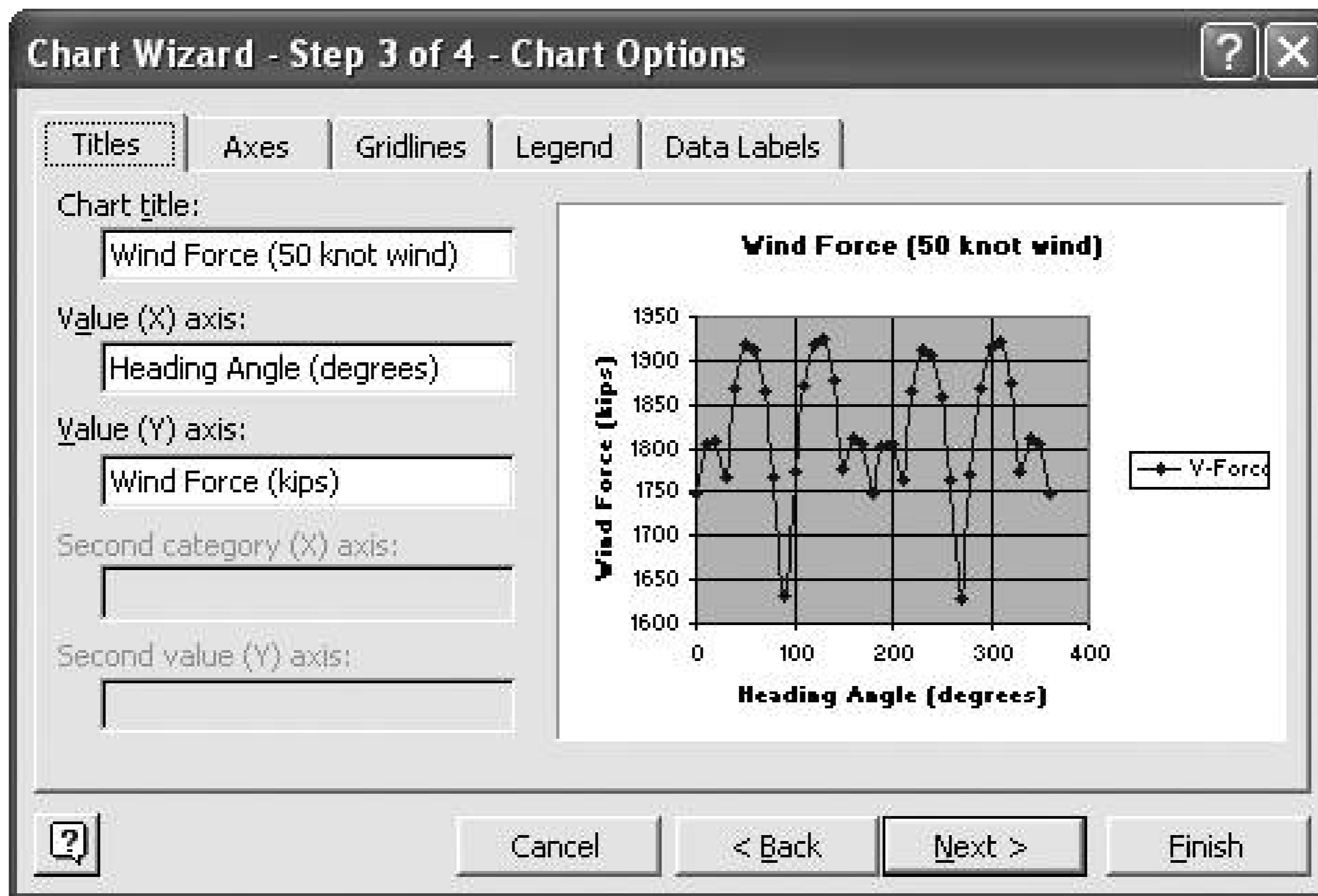
ranges corresponding to the range containing the series name, the series  $x$ -axis data (which could be the same for all series on a chart), and the series  $y$ -axis data. Pressing the little icon to the right of each range field allows you to temporarily switch back to your spreadsheet and select the ranges by using the mouse instead of trying to remember cell references and manually typing them in.

Again, at this point you can press the Next button to proceed to the third step or press the Finish button to quit the Chart Wizard, thus creating your chart.

The step 3 dialog contains five different tabs, as shown in [Figures 4-4](#), [4-5](#), and [4-6](#).

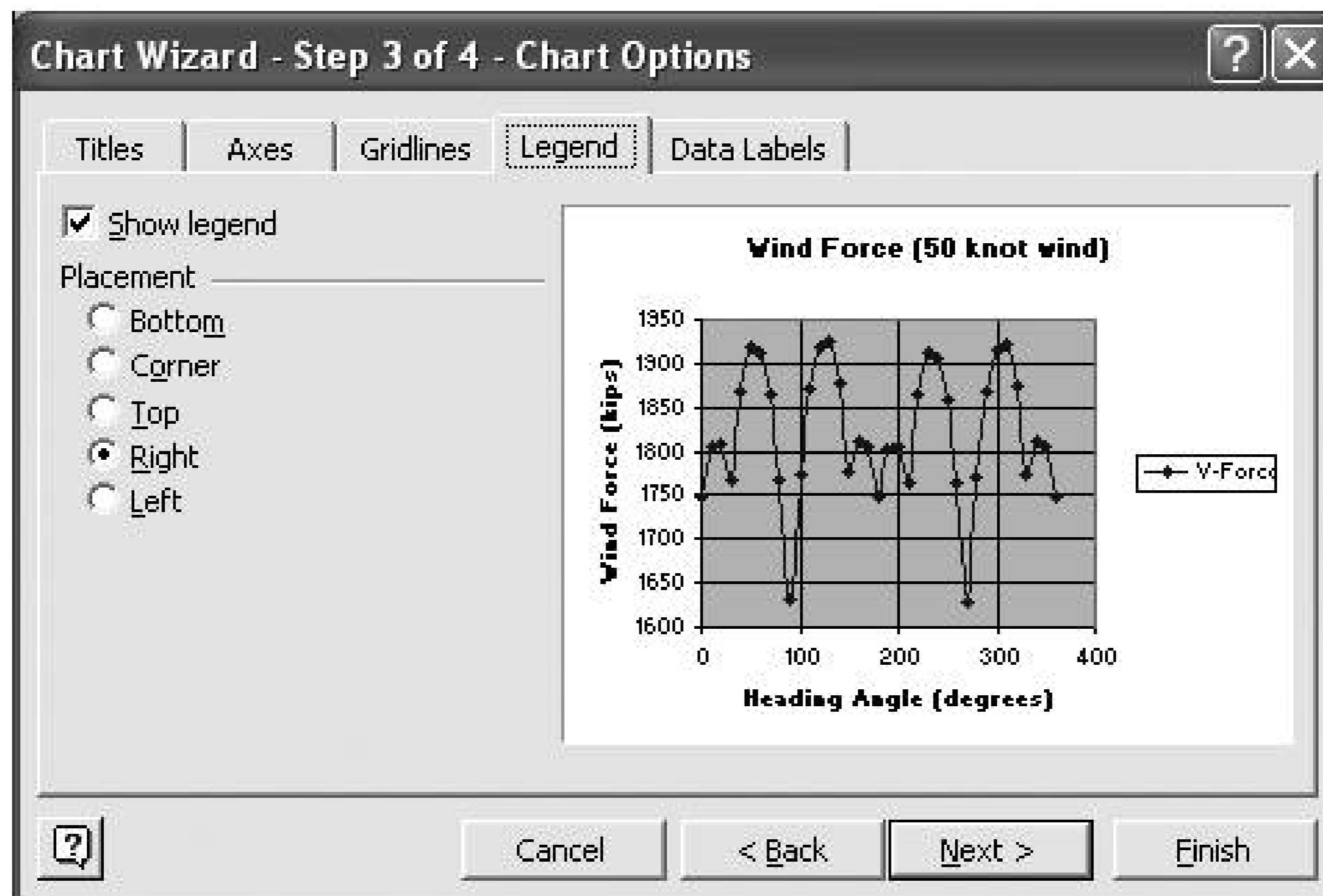
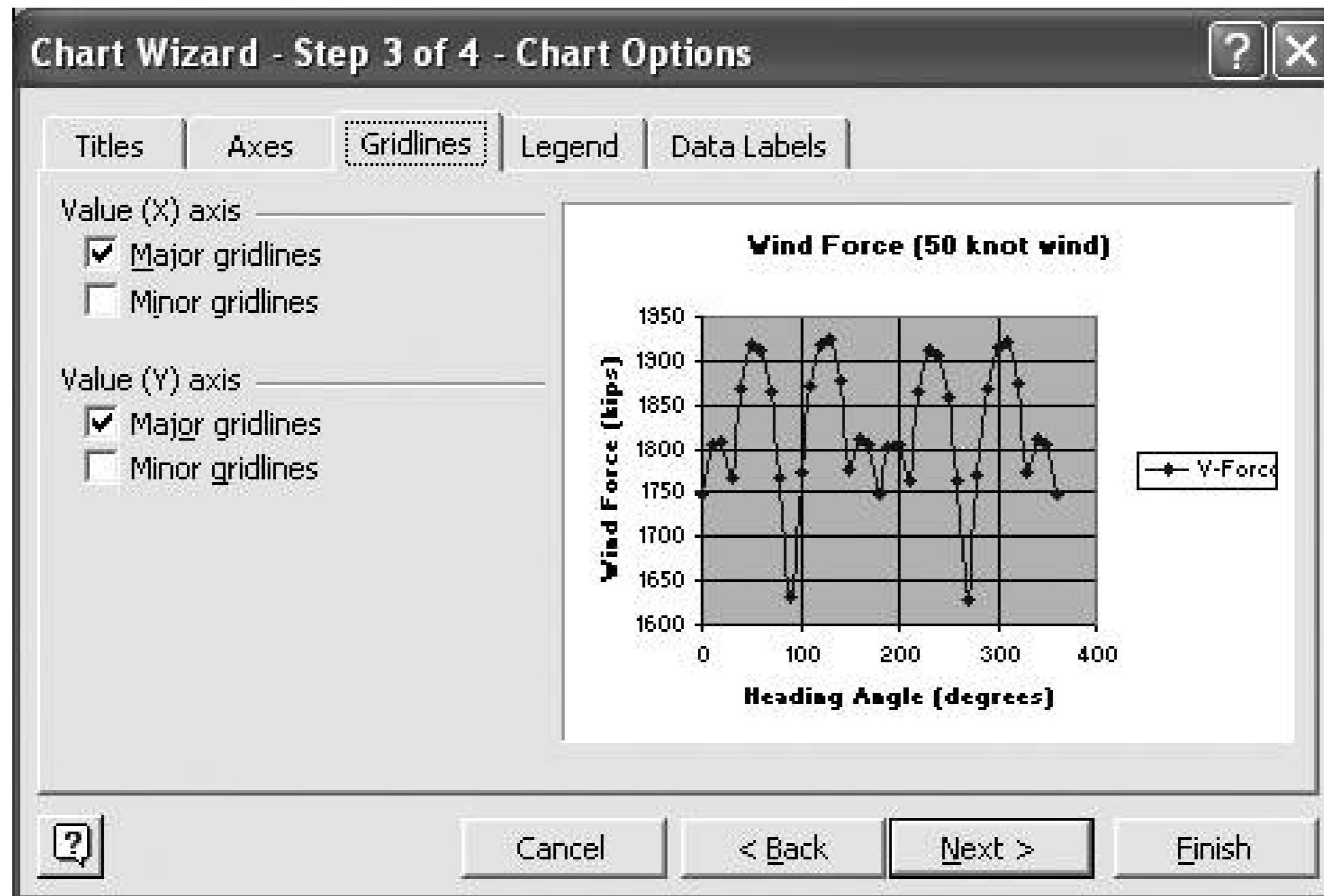
Figure 4-4. Chart Wizard: Step 3, Titles and Axes tabs





The Titles tab allows you to enter titles for major components of your chart, such as the chart title and the axes titles. If you have secondary axes (for plotting multiple series with different scales), you can also specify titles for the secondary axes. (See [Recipe 4.6](#) for more information.) The Axes tab allows you to specify which axes you'd like to show.

Figure 4-5. Chart Wizard: Step 3, Gridlines and Legend tabs



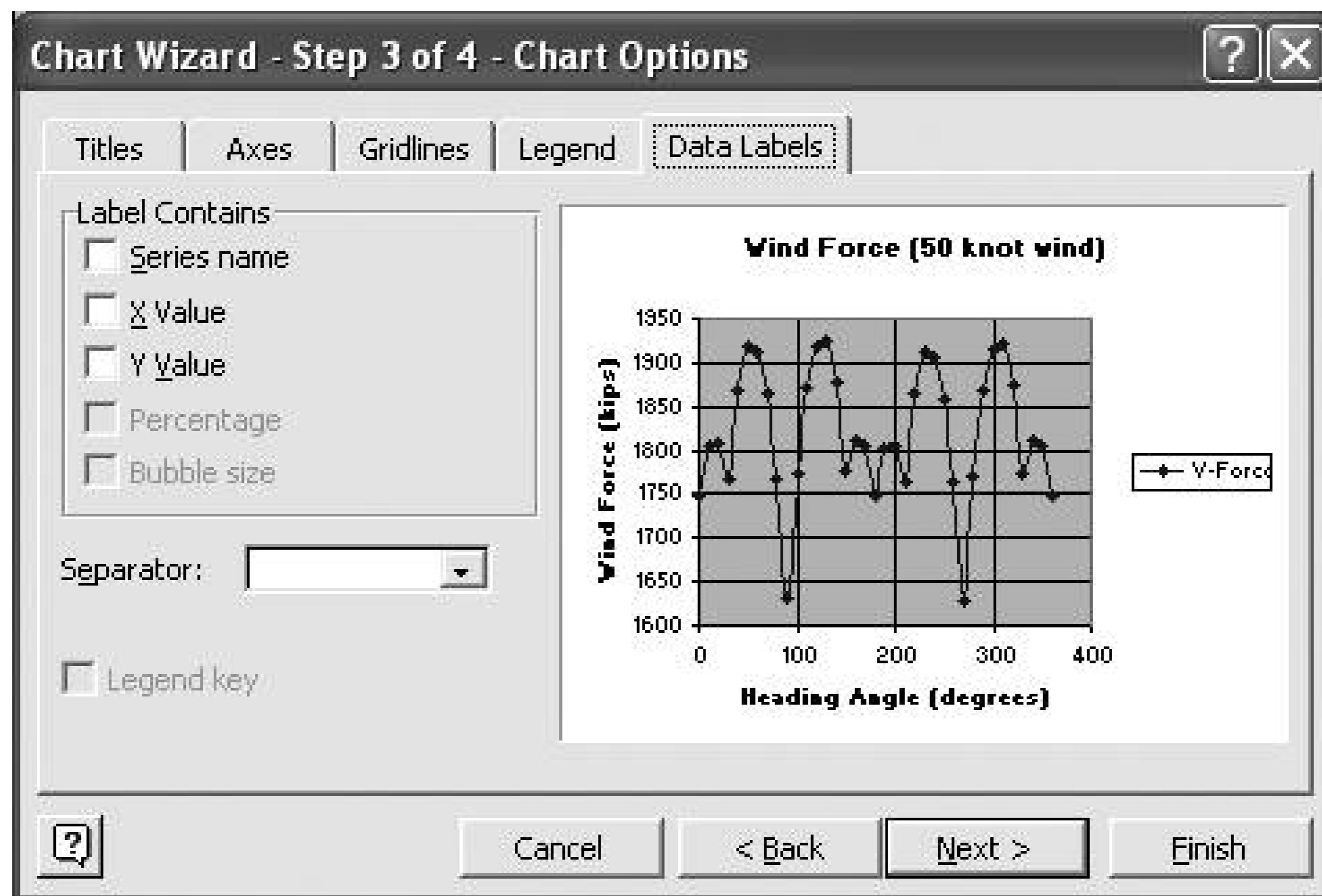
The Gridlines tab allows you to select which, if any, gridlines you'd like to show on the chart. Major gridlines correspond to the major scale divisions for each axis, while the minor gridlines correspond to the minor scale divisions. (See [Recipe 4.4](#) for more information on customizing your chart axes, including setting scale divisions.)

The Legend tab allows you to specify whether or not you want to include a legend on your chart and where you'd like it displayed. For this example, we really don't need a legend, since we're plotting a single data series. Legends are handy when you have multiple data series and would like a key that

lets the user know which is which.

The final tab, Data Labels, allows you to specify whether you want data labels plotted next to each data point on your series and which label you want shown. You can choose from the series name itself, the  $x$ -value of each point, or the  $y$ -value of each point. (As you can see, there are a few other options for some other specific chart types.)

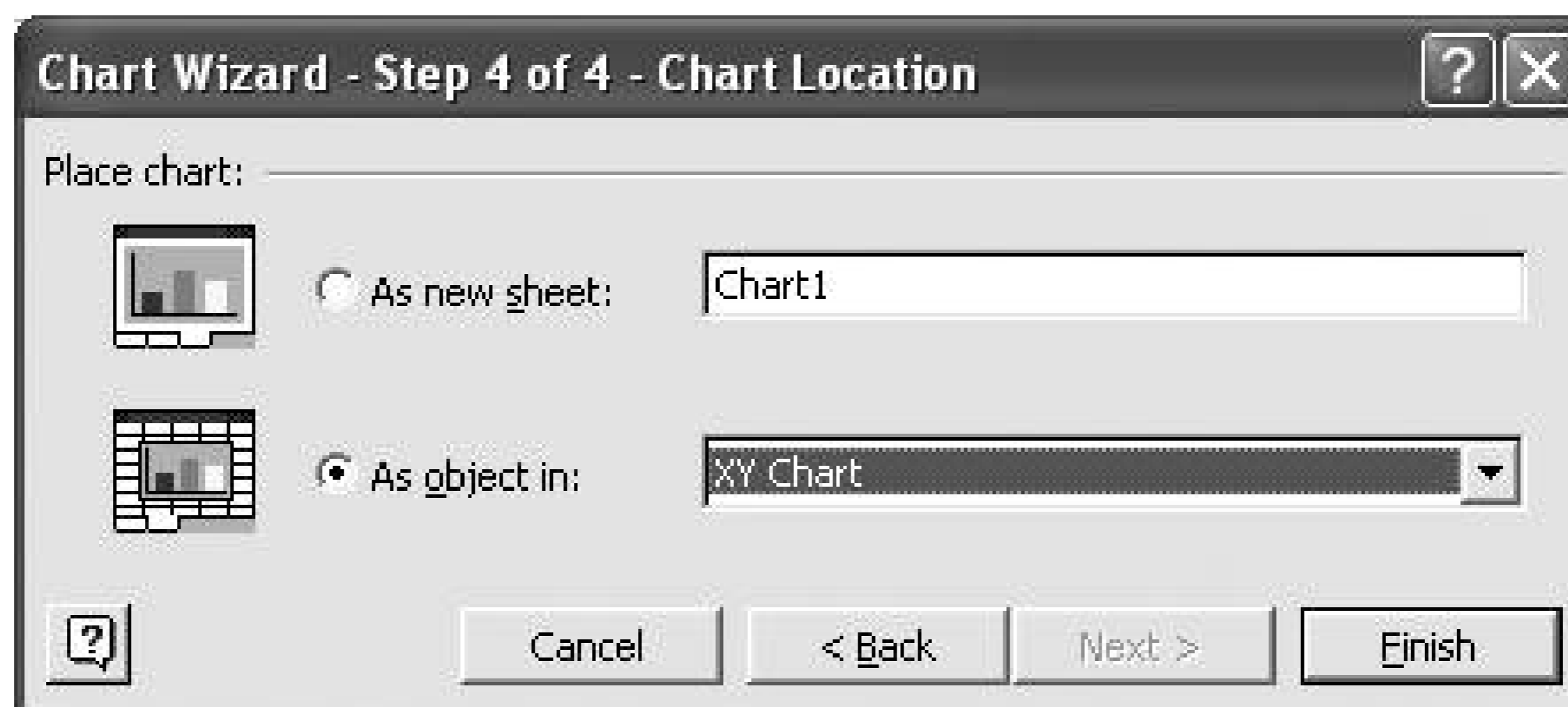
Figure 4-6. Chart Wizard: Step 3, Data Labels tab



At this point I usually just press the Finish button to quit the Chart Wizard and create my chart. However, let's take a look at the final Chart Wizard step by pressing the Next button.

[Figure 4-7](#) illustrates the final step in the Chart Wizard.

Figure 4-7. Chart Wizard: Step 4



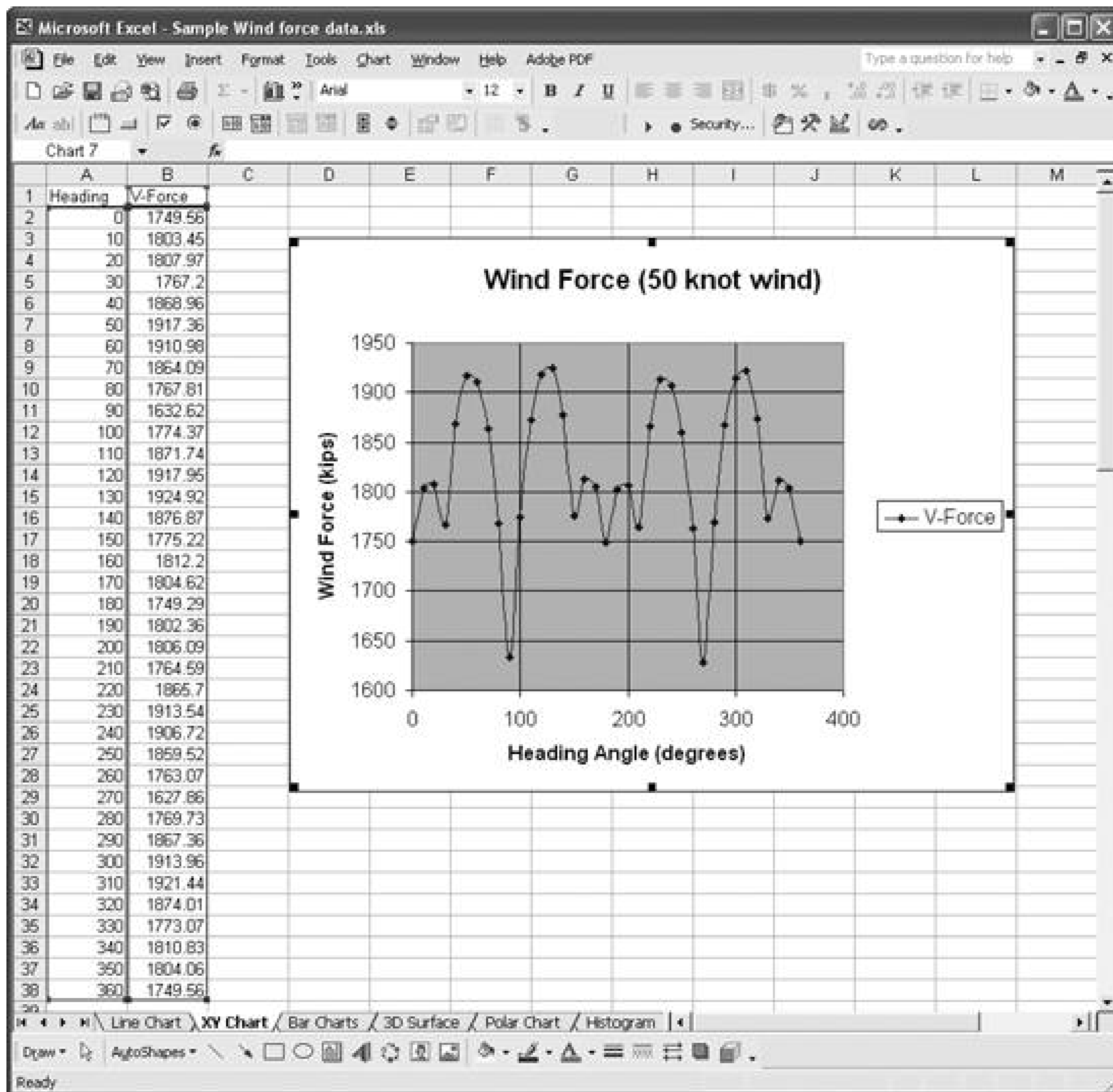
In this final step, you tell Excel where you want to create your chart. You can choose between having the chart created on a new chart sheet within your workbook or embedding it within a different worksheet.

Almost 99.9% of the time, I choose to have the chart embedded in the same sheet containing the data I'm plotting. The main reason I do this is that in many cases the data being plotted is generated or manipulated with custom calculations, and by placing the chart on the same pane as the data (and calculations), I can instantly view how changes in the underlying data (and calculations) affect the chart.

If you choose to place your chart on its own sheet, then you can specify the name of that sheet in this dialog box. The drop-down list next to the "As Object in" radio button allows you to select into which worksheet you want to embed your chart.

Now you can press the Finish button to exit the Chart Wizard and see the resulting chart. [Figure 4-8](#) shows the chart created for this example.

Figure 4-8. Newly created chart



You have some flexibility in where to locate your new chart in a worksheet. You can click and drag it anywhere to suit your needs.

You need not worry about skipping steps in the Chart Wizard or changing your mind. You can always go back and reformat your chart. See [Recipe 4.3](#) for more information.

## See Also

See [Recipe 4.2](#) to learn about other chart types available in Excel.

## Recipe 4.2. Exploring Chart Styles

### Problem

You've seen how to create handy XY charts in the previous recipe and would like to learn what other chart types are available in Excel.

### Solution

Excel has several built-in chart types to choose from. Excel's standard chart types are:

#### *Column*

The Column type includes the sub-types Clustered Column, Stacked Column, 100% Stacked Column, Clustered Column 3-D, Stacked Column 3-D, 100% Stacked Column 3-D, and 3-D Column.

#### *Bar*

The Bar type includes the sub-types Clustered Bar, Stacked Bar, 100% Stacked Bar, Clustered Bar 3-D, Stacked Bar 3-D, and 100% Stacked Bar 3-D.

#### *Line*

The Line type includes the sub-types Line, Stacked Line, 100% Stacked Line, Line with Markers, Stacked Line with Markers, 100% Stacked Line with Markers, and 3-D Line.

#### *Pie*

The Pie type includes the sub-types Pie, Pie 3-D, Pie of Pie, Exploded Pie, Exploded Pie 3-D, and Bar of Pie.

#### *XY (Scatter)*

The XY type includes the sub-types Scatter, Scatter with smoothed lines, Scatter with smoothed lines and markers, Scatter with lines, and Scatter with lines and markers.

### *Area*

The Area type includes the sub-types Area, Stacked Area, 100% Stacked Area, Area 3-D, Stacked Area 3-D, and 100% Stacked Area 3-D.

### *Doughnut*

The Doughnut type includes the sub-types Doughnut and Exploded Doughnut.

### *Radar*

The Radar type includes the sub-types Radar, Radar with Markers, and Filled Radar.

### *Surface*

The Surface type includes the sub-types 3-D Surface, Wireframe 3-D Surface, Contour, and Wireframe Contour.

### *Bubble*

The Bubble type includes the sub-types Bubble and Bubble 3-D.

### *Stock*

The Stock type includes the sub-types High-Low-Close, Open-High-Low-Close, Volume-High-Low-Close, and Volume-Open-High-Low-Close.

### *Cylinder*

The Cylinder type includes the sub-types Column, Stacked Column, 100% Stacked Column, Bar, Stacked Bar, 100% Stacked Bar, and 3-D Column.

### *Cone*

The Cone type includes the sub-types Column, Stacked Column, 100% Stacked Column, Bar, Stacked Bar, 100% Stacked Bar, and 3-D Column.

### *Pyramid*

The Pyramid type includes the sub-types Column, Stacked Column, 100% Stacked Column, Bar, Stacked Bar, 100% Stacked Bar, and 3-D Column.

In addition to these standard chart types, Excel has many other built-in, custom chart types to choose from. Moreover, you can combine many of the these chart types on a single chart to create your own custom chart type, which you can save in the Custom Types list for repeated use.

To different types of charts, follow the same procedure described in [Recipe 4.1](#). Just select a different type of chart during step 1 of the Chart Wizard process. The Chart Type dialog box (step 1 of the Chart Wizard) shows you all of the available built-in chart types. You can select any type and view a preview of the resulting chart by pressing the preview button, as discussed earlier. The Custom Type tab of the Chart Type dialog box also allows you to select or add custom chart types. (See [Recipe 4.12](#) for more information.)

## See Also

See [Recipe 4.8](#) to learn how to set different styles for each data series on a single chart.





## Recipe 4.3. Formatting Charts

### Problem

You've created a chart and would like to alter its appearance.

### Solution

Adjust the formats for your chart using Excel's chart formatting capabilities.

### Discussion

You can select any chart element, including the chart itself, by clicking on it with the mouse. Also, once a chart is selected, you can cycle through, selecting each element using the arrow keys. You can select and format the chart itself, the plot area, the axes, labels, data series, and the legend, as well as any other element that may appear in a chart.

The name of the currently selected chart element is displayed in the Name box to the left of the formula bar.

You can change the formatting of any chart element by selecting the element and pressing Ctrl-1 to open a format dialog box specific to the selected chart element. Or you can right-click on any element and select the Format option from the pop-up menu. Alternatively, you can use the various format tool buttons in Excel's main window to format various aspects of the chart element. Also, when a chart element is selected you can select Format from the main menu bar to access a context-specific format dialog box just as if you had pressed Ctrl-1.

Some of the most common format options available for chart elements include colors, line style and thickness, fill pattern (including gradients, patterns, textures, and pictures), and font styles. Specific chart elements, such as chart axes, have other options that allow you to format things such as axes scales, labels, position, and whether an axis is in linear or log scale. (See [Recipe 4.4](#) for more information.)

The bottom line is that pretty much everything on a chart can be customized. Simply select it and press Ctrl-1 to see what options are available. The recipes throughout this chapter provide specific examples and tips for formatting charts.

← PREV

## Recipe 4.4. Customizing Chart Axes

### Problem

While Excel automatically sets chart axis scales, they aren't always what you desire; therefore, you'd like to customize the axes to suit your needs.

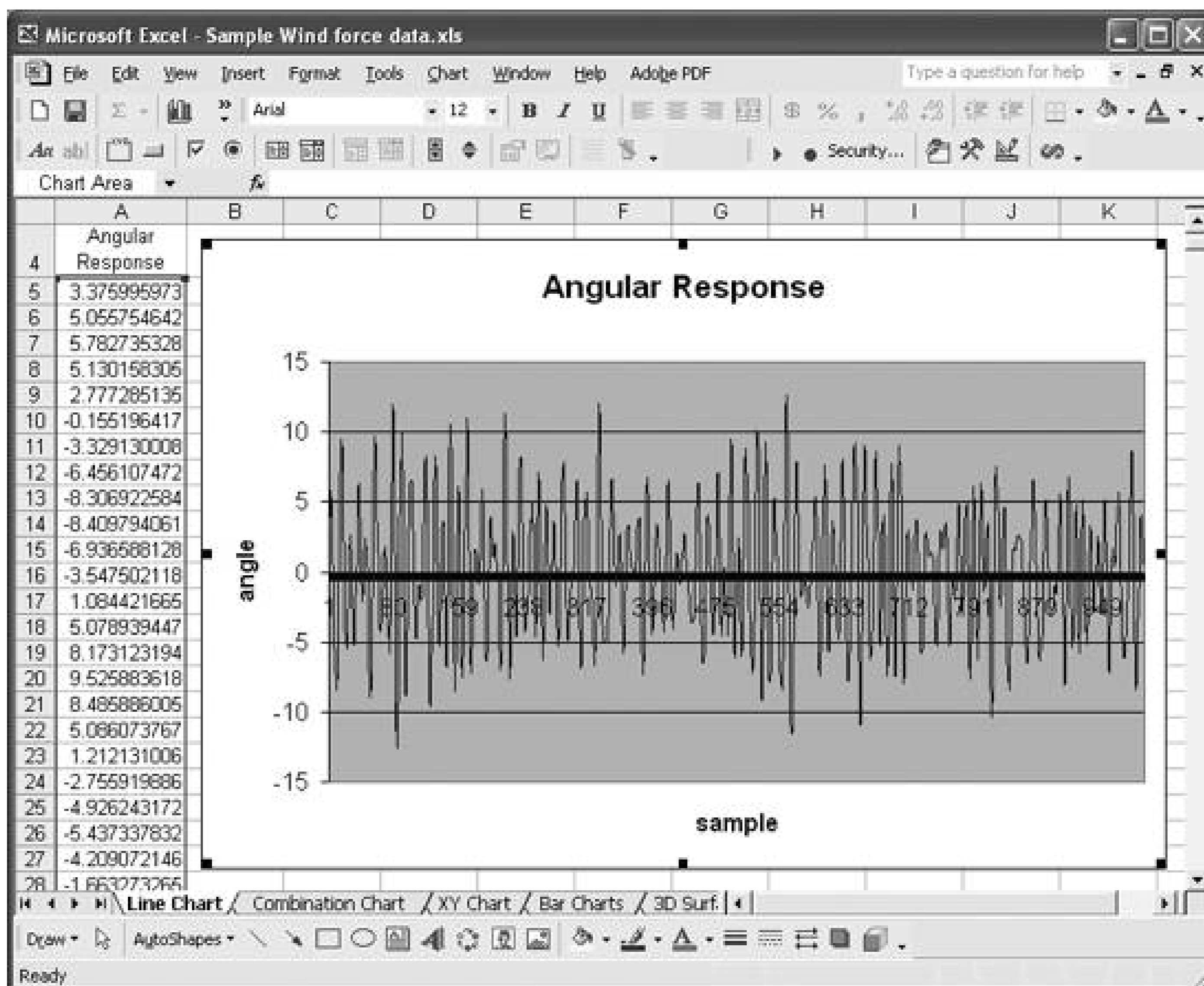
### Solution

To format an axis, right-click on it and select Format Axis from the pop-up menu. This launches the Format Axis dialog box, allowing you to format many aspects of the selected axis.

### Discussion

Right-clicking on an axis to select it can be difficult at times, especially when the chart is a little cluttered. Take a look at the chart in [Figure 4-9](#), which shows angular vibration samples taken during a laboratory experiment.

Figure 4-9. Dense chart

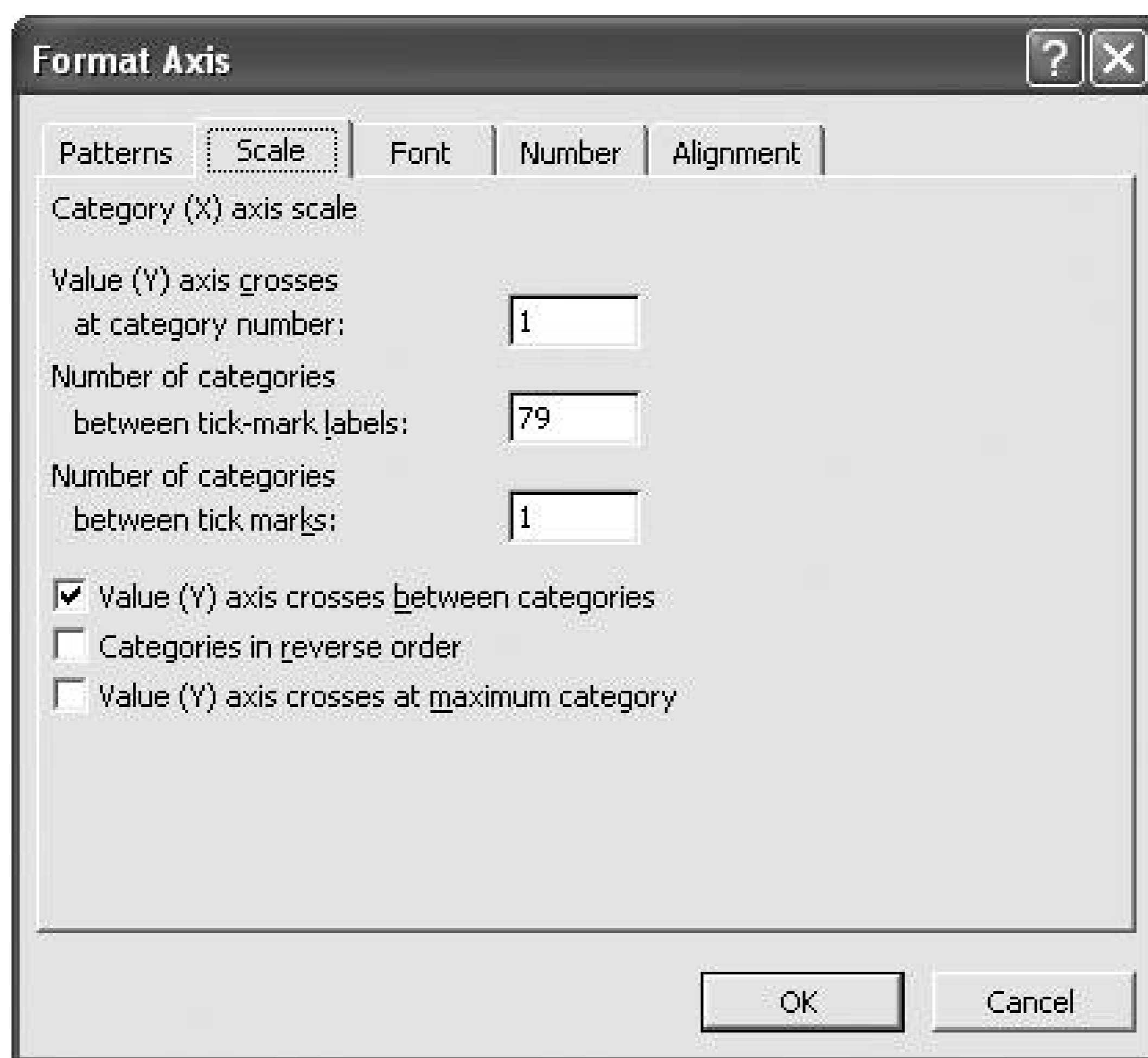


This is a standard Line chart with sample numbers (Excel calls them *categories*) shown on the horizontal axis. By default, Excel places tick marks along this axis, one at every other sample. With a thousand samples as shown here, the axis is quite cluttered. Indeed, the density of samples plotted makes selecting the axis with the mouse difficult.

An alternative way to select chart elements for formatting is to first select the chart itself by clicking anywhere on it, and then use the arrow keys to cycle through, selecting each chart element. The name of the currently selected element is displayed in the Name box to the left of the formula bar. In this case, we're looking for Category Axis. After selecting the chart axis in this manner, you can launch the Format Axis dialog by selecting Format Selected Axis... (Ctrl-1) from the main menu bar.

[Figure 4-10](#) shows the Format Axis dialog box with the Scale tab selected.

Figure 4-10. Format Axis dialog box



To make the horizontal axis a bit more readable you could set the number of categories between tick marks and tick-mark labels to, say, 100. You could also go to the Font tab and change the font size of the axis labels to make them larger and more readable. The Font tab also allows you to change the font type, style, and color, among other settings.

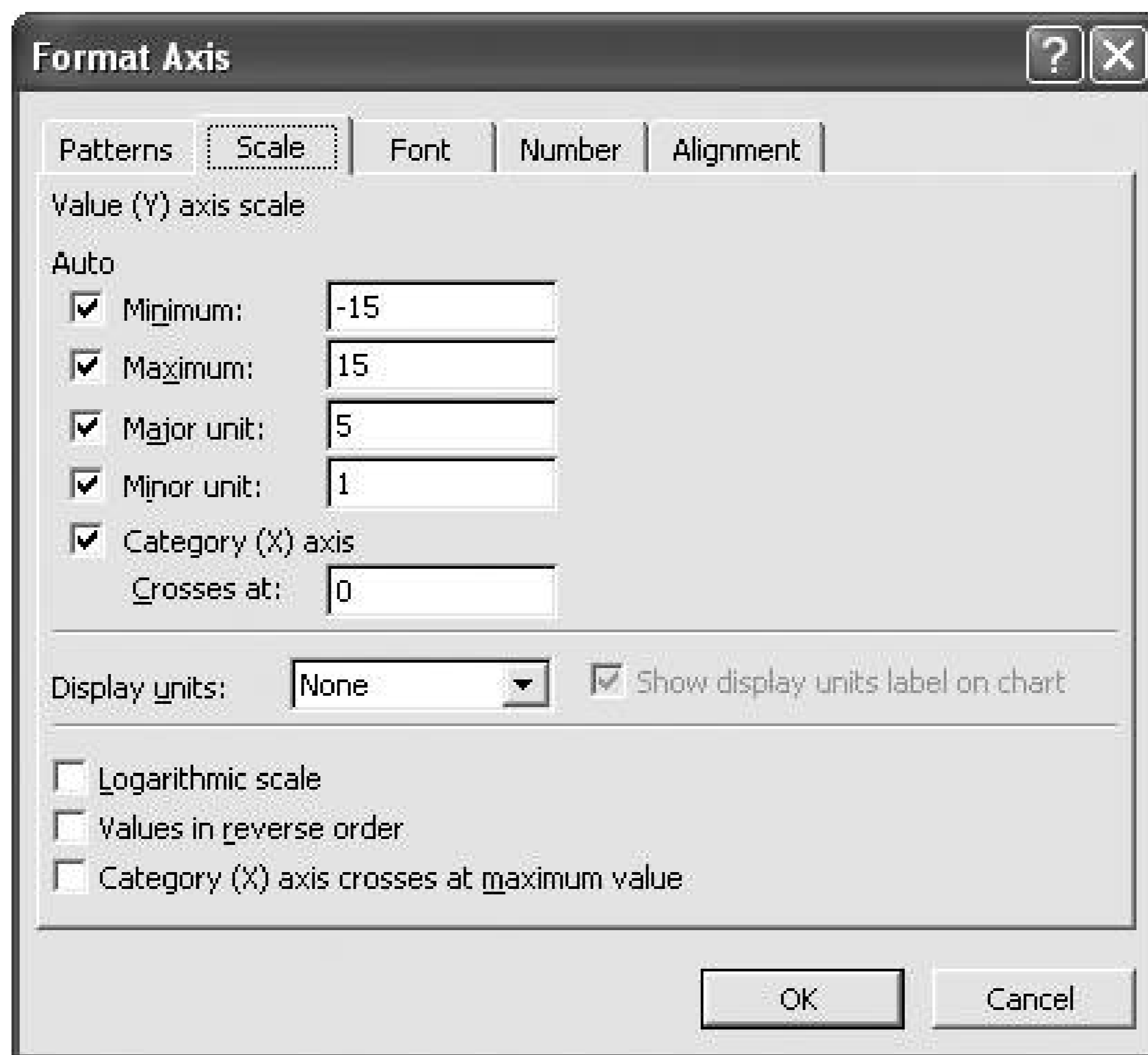
If you'd prefer to have your category labels oriented vertically, you can go to the Alignment tab and alter their orientation. The Number tab allows you to format the labels just as you might format numbers or text in a cell. For example, you could have numbers displayed in scientific notation, as percentages, or as fractions.

In addition to formatting axis labels, you can change the appearance of the axis line itself by using the format controls on the Patterns tab. For example, you could change the thickness, line style, and color of the axis line, as well the appearance of tick marks.

The Scale tab contains different controls depending on the type of data plotted along the axis. The horizontal axis in this case shows categories or numbers that are evenly spaced. An XY scatter chart's  $x$ -axis shows real numbers with arbitrary spacing. The vertical axes in both chart types also show arbitrary real numbers. [Figure 4-11](#) shows the scale controls available for axes that display arbitrary real numbers.

The controls on this tab allow you to specify the minimum and maximum values (i.e., the range of values shown along the axis). You can also specify the major and minor units, which control how often labels and gridlines are displayed. And you can specify where the  $x$ -axis intersects the  $y$ -axis, in case you want to shift the  $x$ -axis out of the way for clarity.

Figure 4-11. Format Axis dialog box with different scale controls



By default, all of these controls are set to Auto, which means Excel will attempt to determine suitable values for you. For the most part, Excel does an OK job at this. However, there are several reasons why you'd want to set these controls manually. One reason is that you may not want to view the entire range of data on the chart, but instead would rather focus in on a specific subrange. You can set the minimum and maximum values to a suitable range for this purpose. Another reason is that you may want specific scale divisions shown rather than those selected by Excel. For example, Excel may automatically set the major unit to 2 and the minor unit to 1; however, for your particular data you may prefer a major unit of 10 and a minor unit of 2.

As you can see from [Figure 4-11](#), Excel also allows you to specify a logarithmic scale (see [Recipe 4.5](#) for an example). You can even reverse the order of values along the axis or specify the  $x$ -axis crossing point at the maximum value, whatever that may be.



## Recipe 4.5. Setting Log or Semilog Scales

### Problem

You'd like to use log scales instead of linear scales.

### Solution

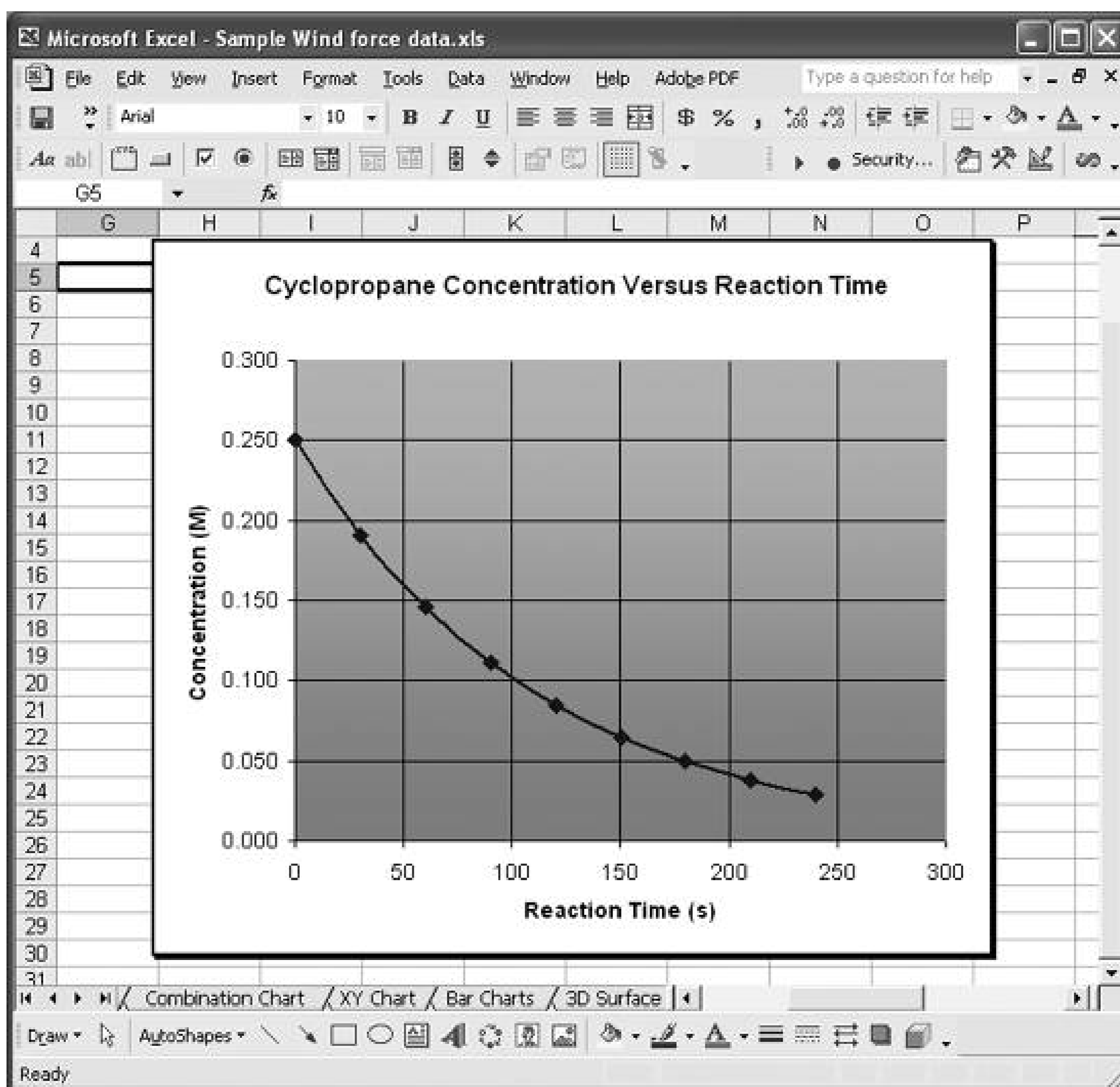
Select the axis you'd like to set to log scale and open the Format Axis dialog, as described in [Recipe 4.4](#). Go to the Scale tab and check the Logarithmic Scale option (see [Figure 4-11](#)).

### Discussion

You can create a log chart by setting both axes to logarithmic scale, or you can create a semilog chart by setting only one axis to logarithmic scale.

Consider the data shown in [Figure 4-12](#) as an example. This data represents the reduction in concentration of cyclopropane as a function of reaction time (the cyclopropane gets converted to propane gas).

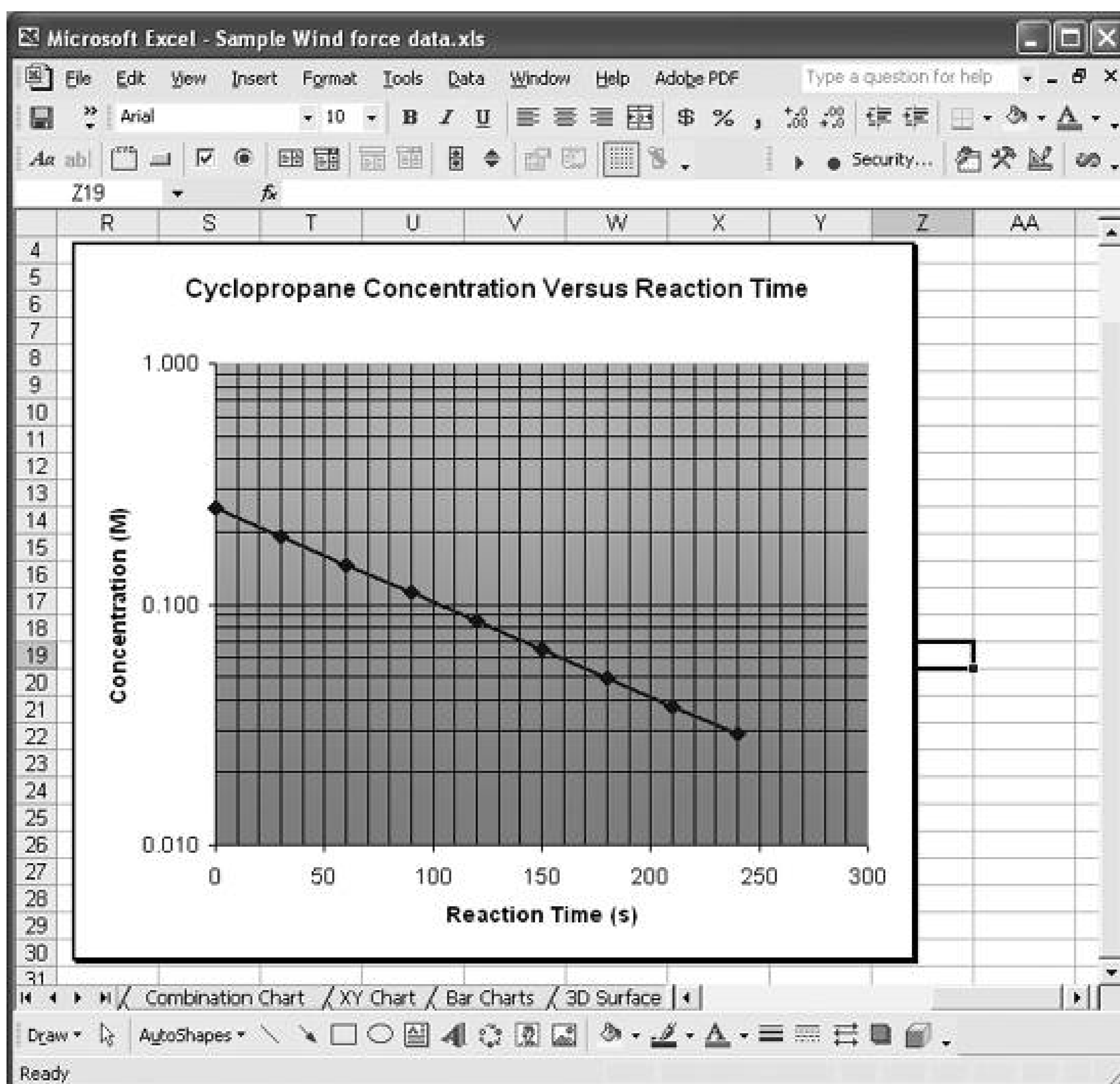
Figure 4-12. Reaction time plot



This plot of concentration versus reaction time shows the characteristic logarithmic form of so-called first-order reactions. Changing the vertical axis to logarithmic scale as described earlier results in the chart shown in [Figure 4-13](#).

Notice that the plotted line is now linear. Plotting the data in this form facilitates estimating the rate constant for such a reaction, since the slope of the line can be determined by using the simple equation for a straight line.

Figure 4-13. Reaction time plotted on semilog chart



## See Also

See [Chapter 8](#) to learn how to perform least-squares curve fitting in Excel.



## Recipe 4.6. Using Multiple Axes

### Problem

You've plotted several series on a chart, but one of them consists of values much larger than the others; when it is plotted on the same scale, you can barely discern the smaller values. Thus, you'd like to plot multiple data series on a single chart but with different scales.

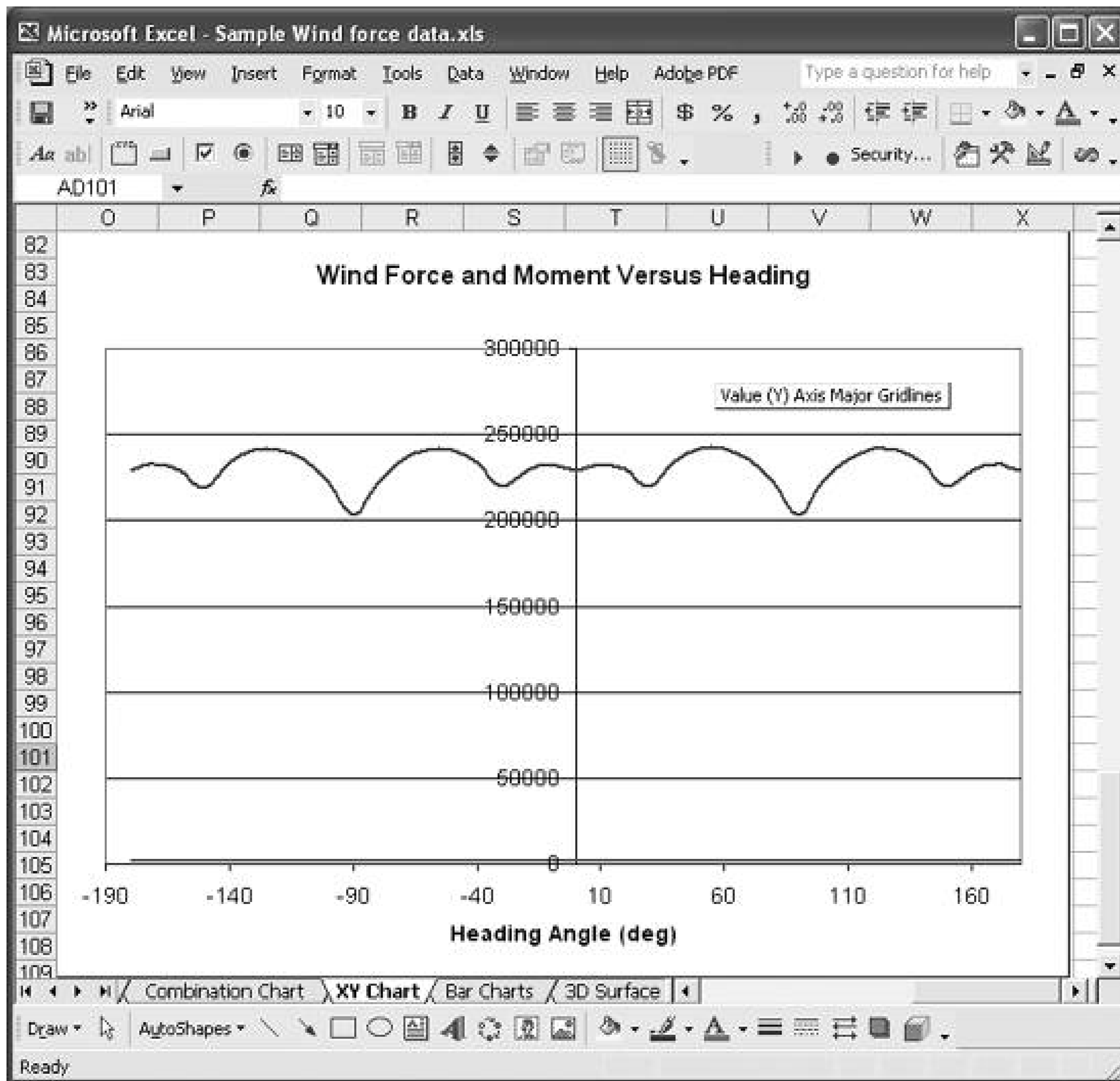
### Solution

You can use multiple scales. Excel has built-in support for secondary axes. This allows you to plot, for example, one data series using a primary  $y$ -axis and another data series using a secondary  $y$ -axis. To specify the axis for a data series, select the series and then select Format → Selected Data Series... from the main menu bar to open the Format Data Series dialog box. Click the Axis tab and then click either "Primary axis" or "Secondary axis" to specify the axis. Press OK when you're done, and Excel will automatically set up another axis and rescale your series.

### Discussion

[Figure 4-14](#) shows an example of two data series with very different scales plotted on the same chart.

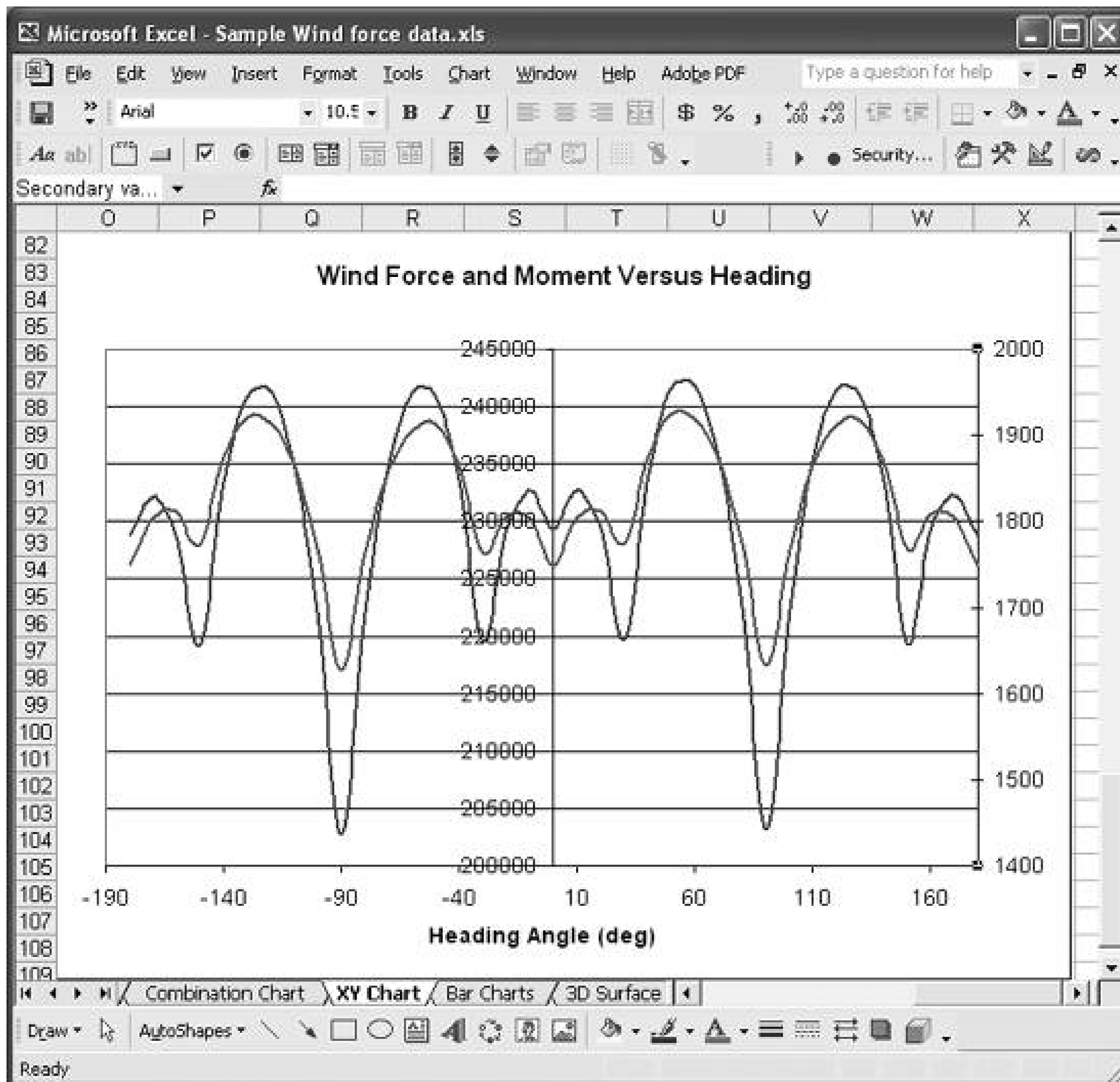
Figure 4-14. Two data series with very different scales



The scales of these two sets of data are so different that when they are plotted on the same chart, you can barely even see one of the data series; it's almost coincident with the  $x$ -axis.

Changing the axis of this series so that it's plotted on a secondary  $y$ -axis (as described earlier) results in the chart shown in [Figure 4-15](#).

Figure 4-15. Chart using a secondary  $y$ -axis



This version of the chart is far more readable than the original, and shows how useful secondary axes are. You can adjust the position of each series relative to the other by changing the minimum and maximum values of the primary or secondary axes.

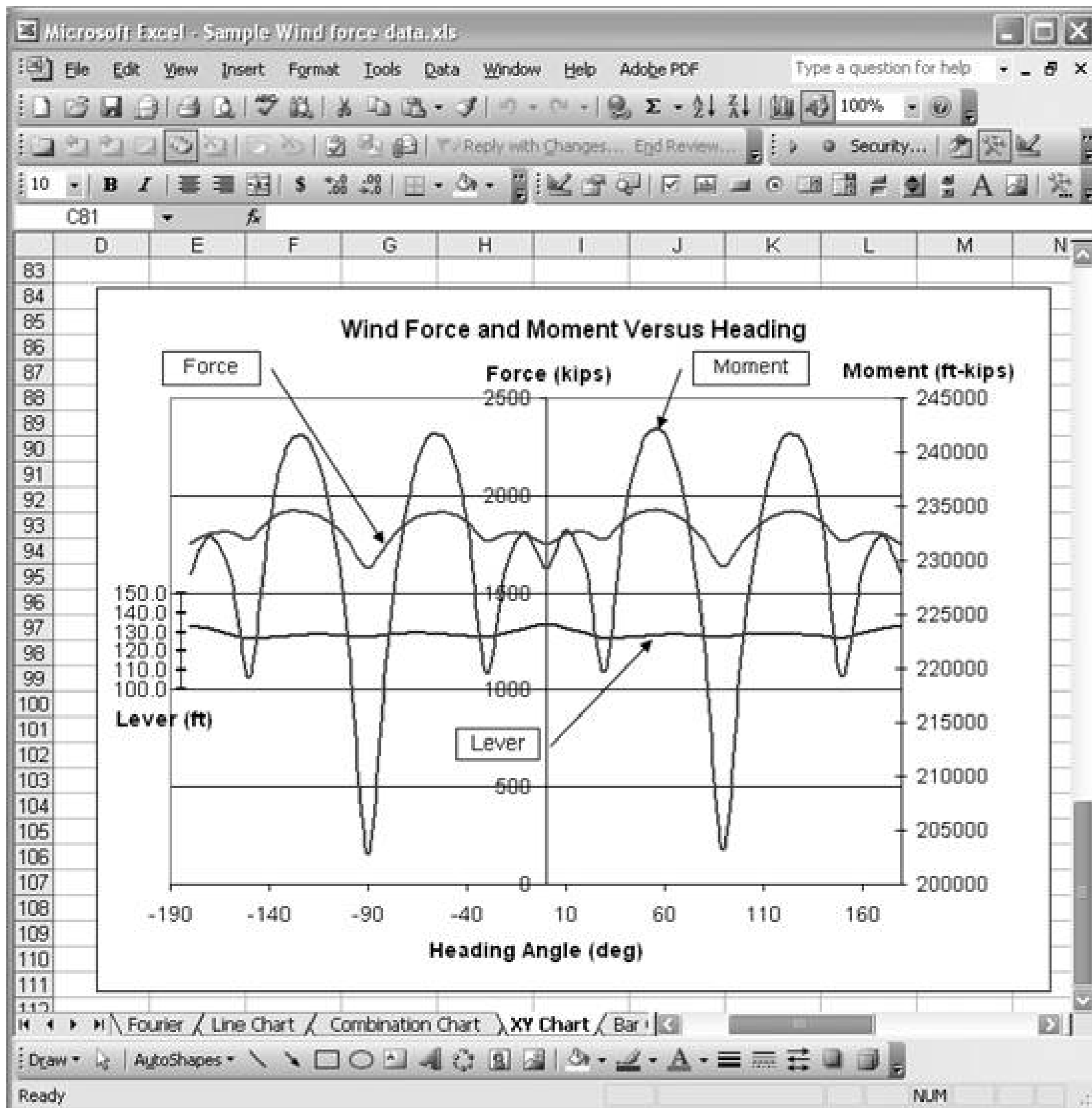
You can, of course, plot more than one series relative to either the primary or secondary axes. Unfortunately, Excel does not support more than one secondary axis. While this isn't the end of the world, there are times when more than one secondary axis would be useful. In these cases, there are a couple of workarounds you can try.

The easiest thing to try is to scale your data. For example, multiply (or divide) a data series by 10, 100, 1,000, or some other appropriate value and then plot it against the primary or secondary axis. Be sure to label the series properly so readers of your chart can clearly see that the data is scaled.

Another trick you might find useful is to fake another axis by using a specifically crafted dummy series. Let's say we wanted to plot another series on the chart shown in [Figure 4-15](#), but the values of this new series ranged from about 100 to 150. Clearly such a series would get buried by the scale of values for the other two series.

The solution is to multiply the values in this new series by an appropriate scale factor, say 10, and then plot it against the primary axis. This would make the values range from about 1,000 to 1,500, which would be clearly visible on the chart. Confusion may arise, however, if the scale is not annotated to indicate the new series has been scaled. To get around this, you can create a fake scale using a dummy series. [Figure 4-16](#) shows just such a fake scale.

Figure 4-16. Fake scale

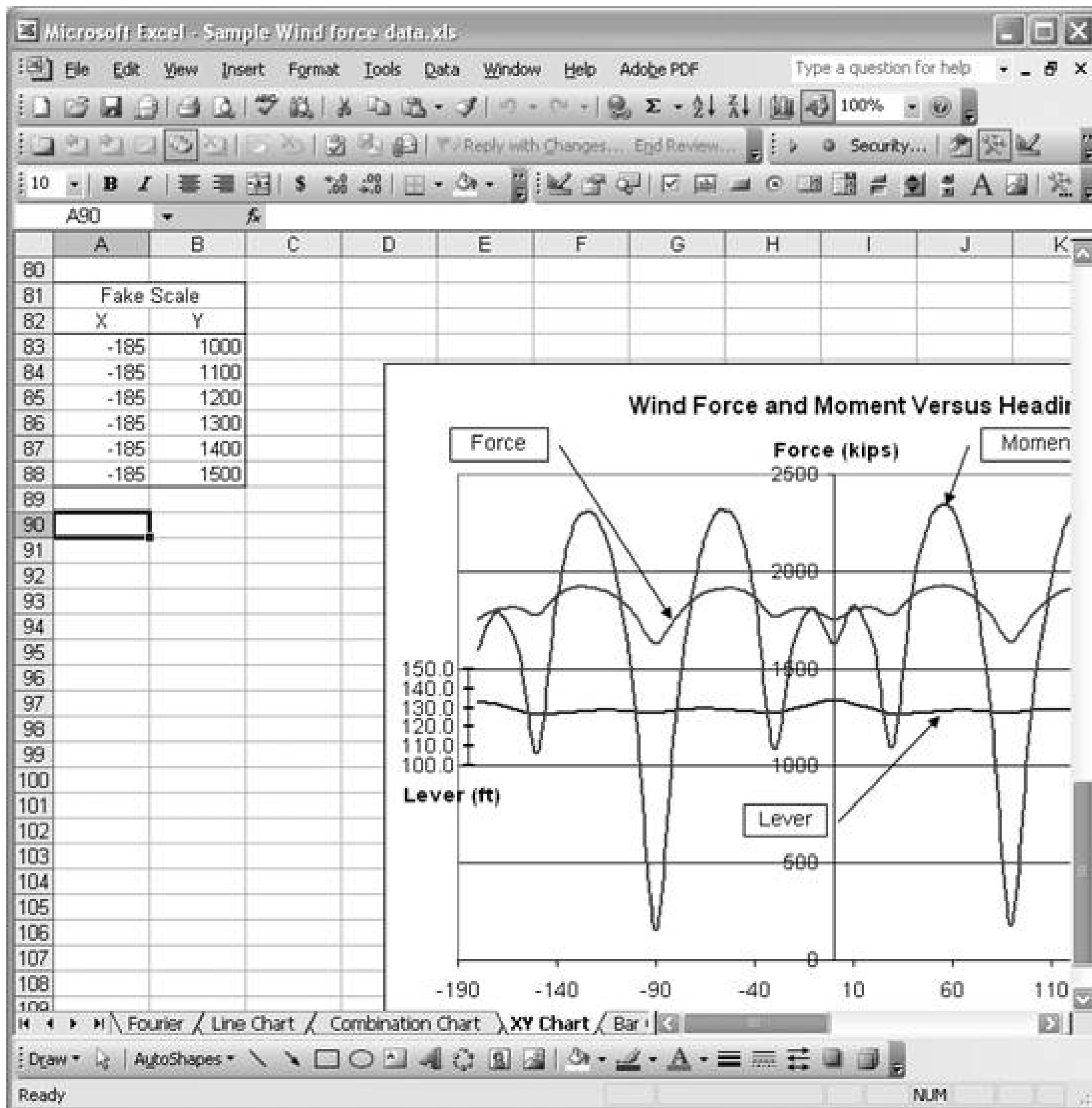


The series labeled Lever is the new, scaled series plotted against the primary y-axis (the center one). The fake axis is the short axis to the left, labeled Lever. This technique allows all of these series to be plotted on the same chart, even though the series values vary widely in magnitude.

To create the fake axis, you need to create and add a dummy series. The dummy series I created for

this example is shown in [Figure 4-17](#).

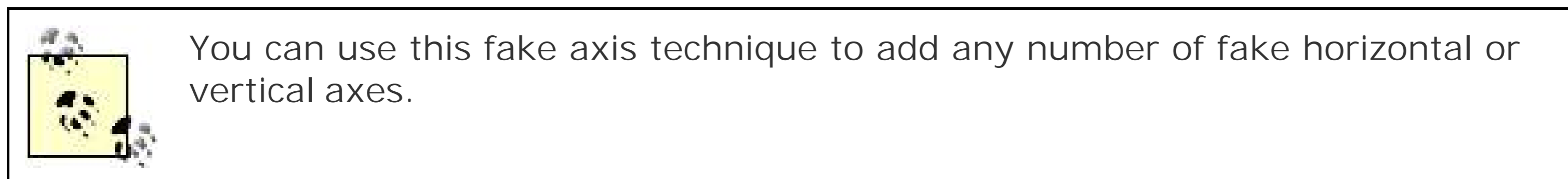
Figure 4-17. Fake series data



The key to using a series to represent an axis is to make one of the coordinates constant. In this case, since I want the series to represent a vertical axis, I set the  $x$ -value for the series to a constant. I used -185 to position the series all the way to the left of the chart. The  $y$ -values are simply selected to capture an appropriate range of values. Once the series data is prepared, you can add the series to the chart as discussed in [Recipe 4.1](#). You'll also want to turn on Y-labels for the dummy series. You can turn on Y-labels by opening the Format Data Series dialog box and selecting the Data Labels tab. Upon doing so, you'll notice a little hiccup that you have to correct.

The default Y-Labels for the dummy series displays the actual  $y$ -values. The problem is that these

values represent the scaled values for the series we wanted to use the fake axis for in the first place. This isn't what we want, so you must edit each data label manually to display the appropriate values. Click the series labels once to select all of the labels. Then click once again on the specific label you want to modify. This will put the label in edit mode, where you can change the label to whatever you desire. Make the changes to all of the labels and you'll end up with something like the chart shown in [Figure 4-16](#).



← PREY

## Recipe 4.7. Changing the Type of an Existing Chart

### Problem

You've created a chart but later decide a different chart type would be more effective. You'd like to change the chart type without having to re-create the chart from scratch.

### Solution

Select your chart and right-click on it with the mouse. On the pop-up menu that appears, select Chart Type to open the Chart Type dialog box, which is the same as that used during step 1 of the Chart Wizard (see [Figure 4-2](#)). Select the new chart type and sub-type and press the OK button when you're done.

### Discussion

As a simple example, reconsider the data used in the example for [Recipe 4.1](#). The force data is a function of heading in degrees from 0 to 360. In the previous example, we used an XY chart type that showed the data series as a function of heading angles along the  $x$ -axis. A better way to view this data would be to plot the force data radially as a function of heading in a polar plot fashion. In Excel this type of chart is called a Radar chart.

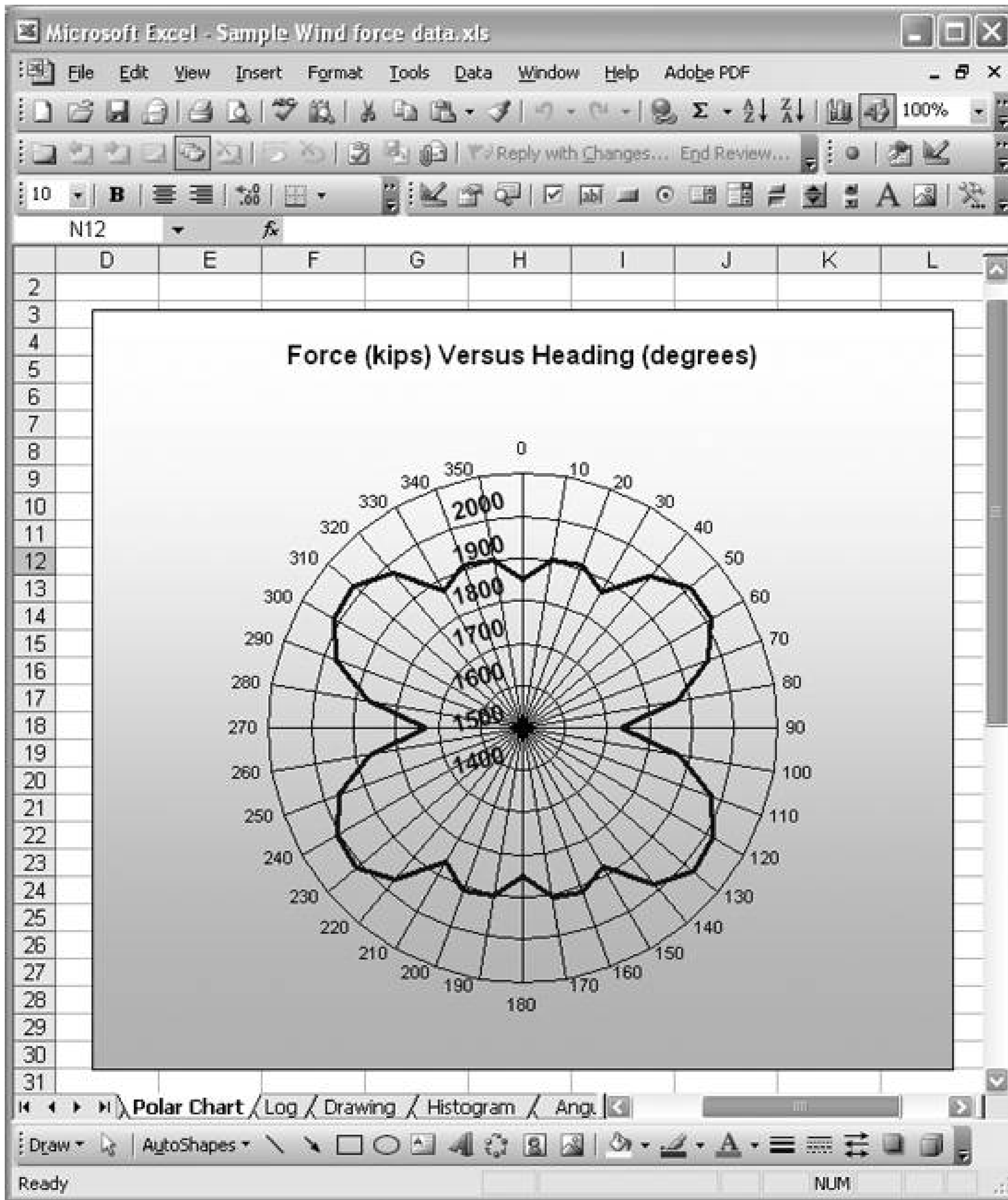
[Figure 4-18](#) shows what the data from [Figure 4-1](#) looks like as a Radar chart.

I should mention that Radar charts require the  $x$ -axis data to be evenly spaced. The heading data for this example is the  $x$ -axis data, but for this Radar chart the heading data is merely used for labeling the chart; it does not govern the structure of the polar axis. The spacing of radial gridlines on Radar charts is always uniform. The number of radial gridlines will equal the number of data points in the series being plotted.

[Figure 4-19](#) shows the Source Data dialog box for this example. To access this dialog box, select the chart and right-click to display a pop-up menu. Select Source Data from the pop-up menu.

The cell range in the Values field is the range of cells containing the  $y$ -data to be plotted, the force data in this example. The heading data happens to be uniform and ranges from 0 to 360 degrees. These facts make the chart appear as you would expect. To force the chart to display the heading angles on the radial axis, you have to supply the cell range containing the heading data in the "Category (X) axis labels" field. Since we converted this chart from an XY chart, Excel filled in the label field automatically.

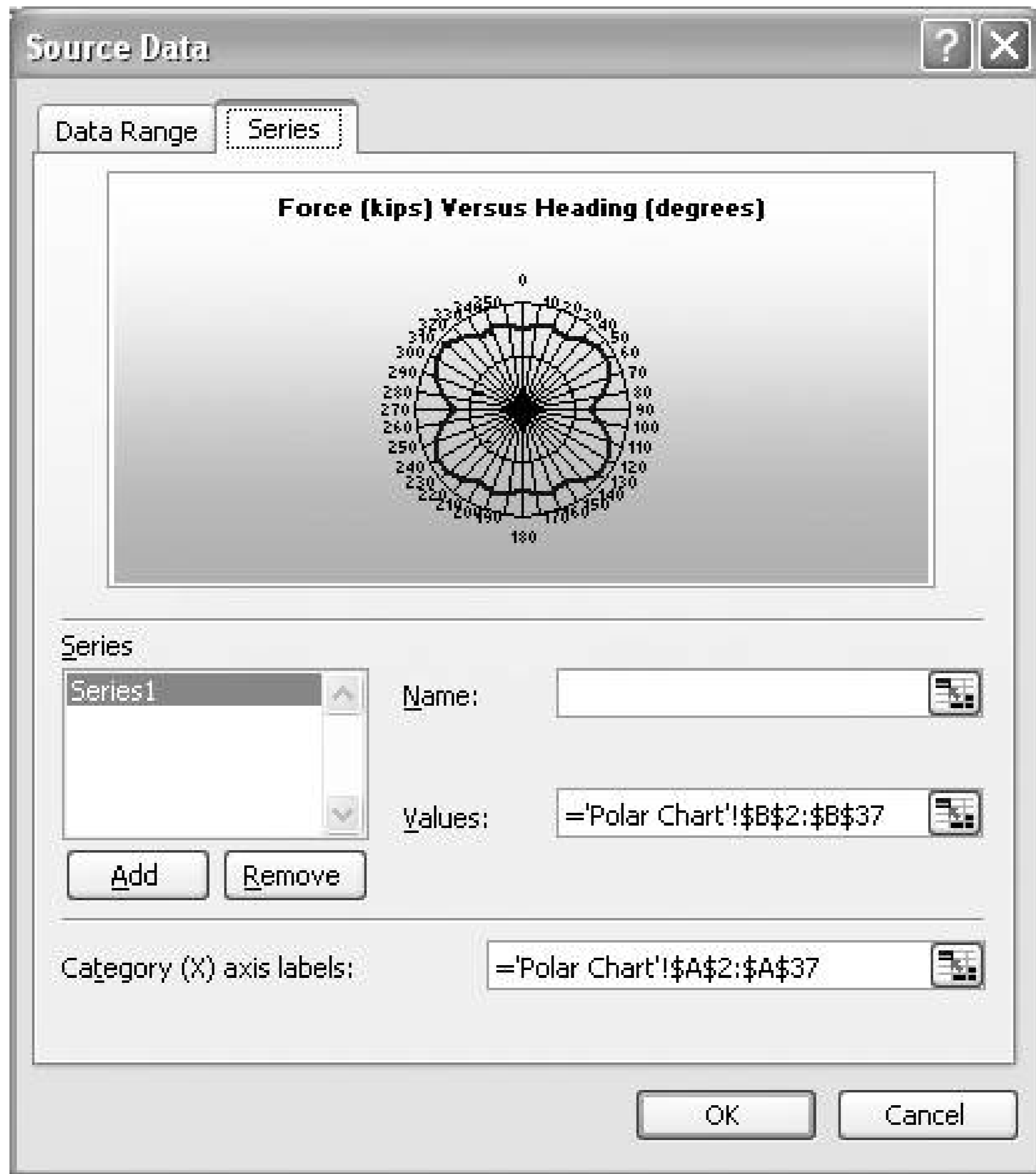
Figure 4-18. Radar chart



You'll notice that this chart looks a little fancier than some of the other examples discussed earlier. As I did was format the chart a little differently by including a background image as the fill pattern for the chart itself, and by tweaking the title and grid line colors using their format options. See [Recipe 4.3](#) for more information.



Figure 4-19. Source Data dialog box



## Recipe 4.8. Combining Chart Types

### Problem

You want to display different datasets on the same chart using different styles.

### Solution

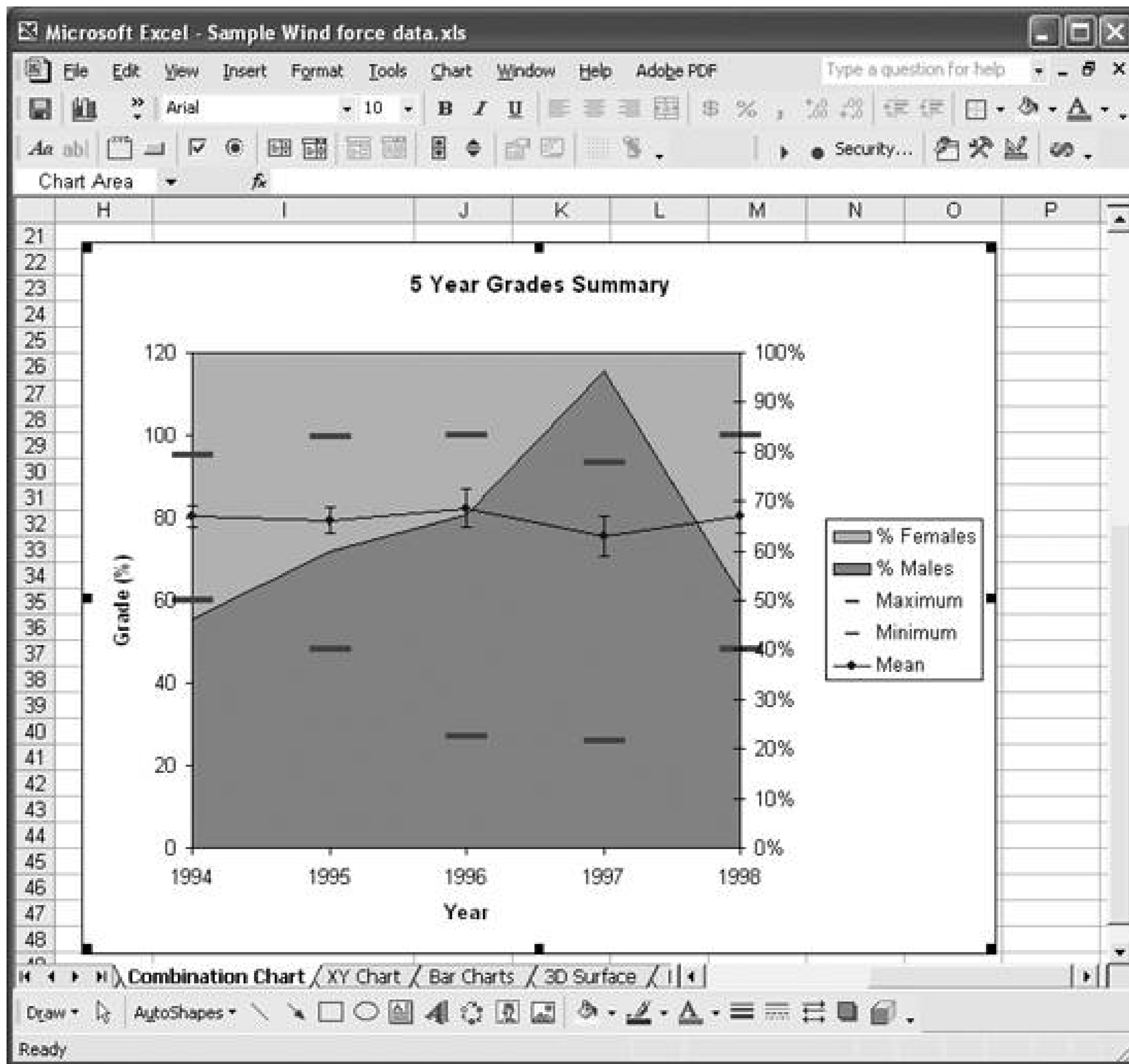
Set the chart style for each data series individually. First, right-click on the desired series and then select Chart Type from the pop-up menu to display the Chart Type dialog box (see [Figure 4-2](#) in [Recipe 4.1](#)). Now select the type of chart you'd like to apply to that series.

### Discussion

In [Recipe 4.7](#) you saw how easy it is to change the type of chart for an existing chart. Sometimes, instead of changing the chart type for the entire chart, it's more desirable to change the type for individual series on a chart, resulting in a combination chart.

Take a look at [Figure 4-20](#), for example. This provides a picture of student grade statistics for a college course over a five-year period. The series representing the mean of grades is a Line chart with error bars representing the standard error (see [Chapter 5](#) for recipes on generating descriptive statistics).

Figure 4-20. Combination chart



The series representing the minimum and maximum grades (the range for each year) are also Line charts.

The other two series, Females and Males, represent the relative population of males and females in each class for the given year. Both of these are Area charts. I could have made these Line charts like the others, but thought that making them Area charts as a sort of backdrop would be more effective.

This chart was created with all the data series initially in Line chart type. I then selected each of the Female and Male series and changed their type, as described earlier.

## Recipe 4.9. Building 3D Surface Plots

### Problem

You want to create a 3D surface plot in Excel (e.g., to present results on a multidimensional optimization study or perhaps to present topographic data).

### Solution

Use Excel's built-in Surface chart type.

### Discussion

Excel's Surface chart allows you to create plots that can be viewed from various angles.

In Excel, surface plots are like a collection of line plots that get connected to form a surface. Thus, they suffer from some of the limitations of line plots. In particular, the values to be plotted (we'll call these the  $z$ -data) must be uniformly spaced along the  $x$ - and  $y$ -axes. You can't have arbitrarily spaced data as you can in XY scatter plots.

[Figure 4-21](#) shows some sample data in the proper form for a 3D surface plot. The sample data shown here represents results from a 1D finite element simulation of nonlinear, non-Newtonian fluid flowing through a porous material.

Figure 4-21. 3D surface plot data

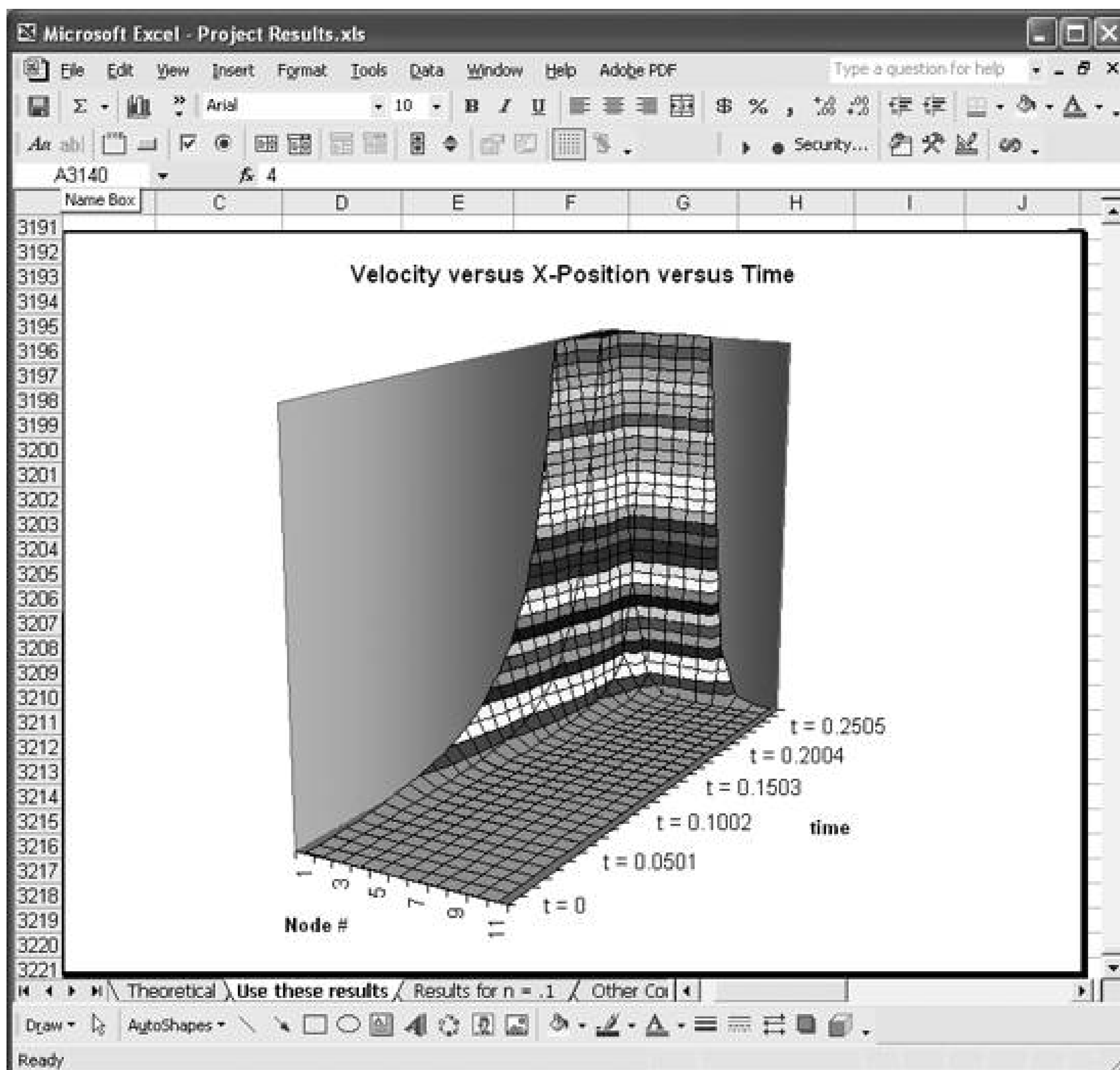
	A	B	C	D	E	F	G	H	I
3136	Node	t = 0	t = 0.01002	t = 0.02004	t = 0.03006	t = 0.04008	t = 0.0501	t = 0.06012	t = 0.07014
3137	1	2.53E-09	0.00255291	0.010279325	0.02341712	0.04237806	0.06777236	0.10044654	0.14153832
3138	2	3.69E-10	0.000373115	0.001510216	0.00347029	0.00635609	0.01032136	0.01558185	0.02243149
3139	3	5.39E-11	5.44005E-05	0.000219749	0.00050326	0.00091738	0.00148047	0.00221787	0.00316327
3140	4	7.86E-12	7.93897E-06	3.20937E-05	7.3693E-05	0.00013439	0.00021739	0.00032659	0.00046738
3141	5	1.15E-12	1.15819E-06	4.68066E-06	1.0728E-05	1.9577E-05	3.1639E-05	4.748E-05	6.7859E-05
3142	6	1.67E-13	1.69002E-07	6.83078E-07	1.5659E-06	2.8583E-06	4.6209E-06	6.9374E-06	9.9201E-06
3143	7	2.44E-14	2.46759E-08	9.97312E-08	2.2861E-07	4.1725E-07	6.7446E-07	1.0124E-06	1.4474E-06
3144	8	3.58E-15	3.61929E-09	1.46282E-08	3.3532E-08	6.1204E-08	9.8938E-08	1.4852E-07	2.1235E-07
3145	9	5.42E-16	5.47361E-10	2.21227E-09	5.0711E-09	9.2559E-09	1.4962E-08	2.246E-08	3.2112E-08
3146	10	9.9E-17	1.00536E-10	4.06336E-10	9.3143E-10	1.7001E-09	2.7482E-09	4.1255E-09	5.8983E-09
3147	11	4.4E-17	4.46826E-11	1.80593E-10	4.1397E-10	7.5559E-10	1.2214E-09	1.8335E-09	2.6214E-09
3148									
3149									
3150									
3151									

The **Node** column represents the finite element node at which data was calculated. This column also represents values (or categories) that will be displayed on one of the chart axes. The row of text showing  $t=0$ ,  $t=0.01002$ , and so on represents the other axis; more precisely, these are labels to be displayed on the other axis. Excel will display these labels (for both axes) uniformly spaced on the chart, so I purposely extracted data at regular time intervals so the chart would make sense.

The values in the cell range starting at B3137 and ending in the lower-right corner represent the  $z$ -values to be plotted as functions of node and time. These values represent the velocity of the fluid flowing through the porous material in this example.

To create the plot, select the entire range of cell data, including the label column and row (i.e., the **Node** column and the row containing the elapsed time labels). In this example, the complete range is A3136:AD3147. After selecting the data, click the Chart Wizard button and go through the Chart Wizard as discussed in [Recipe 4.1](#); however, this time select Surface for the chart type. The results are shown in [Figure 4-22](#).

Figure 4-22. 3-D Surface chart



Here you can see that one axis on the horizontal plane contains node number labels, while the other contains the elapsed time labels. The surface represents fluid flow velocity at each node over time. The surface clearly shows a lot of information in a concise fashion, facilitating interpretation of the results.

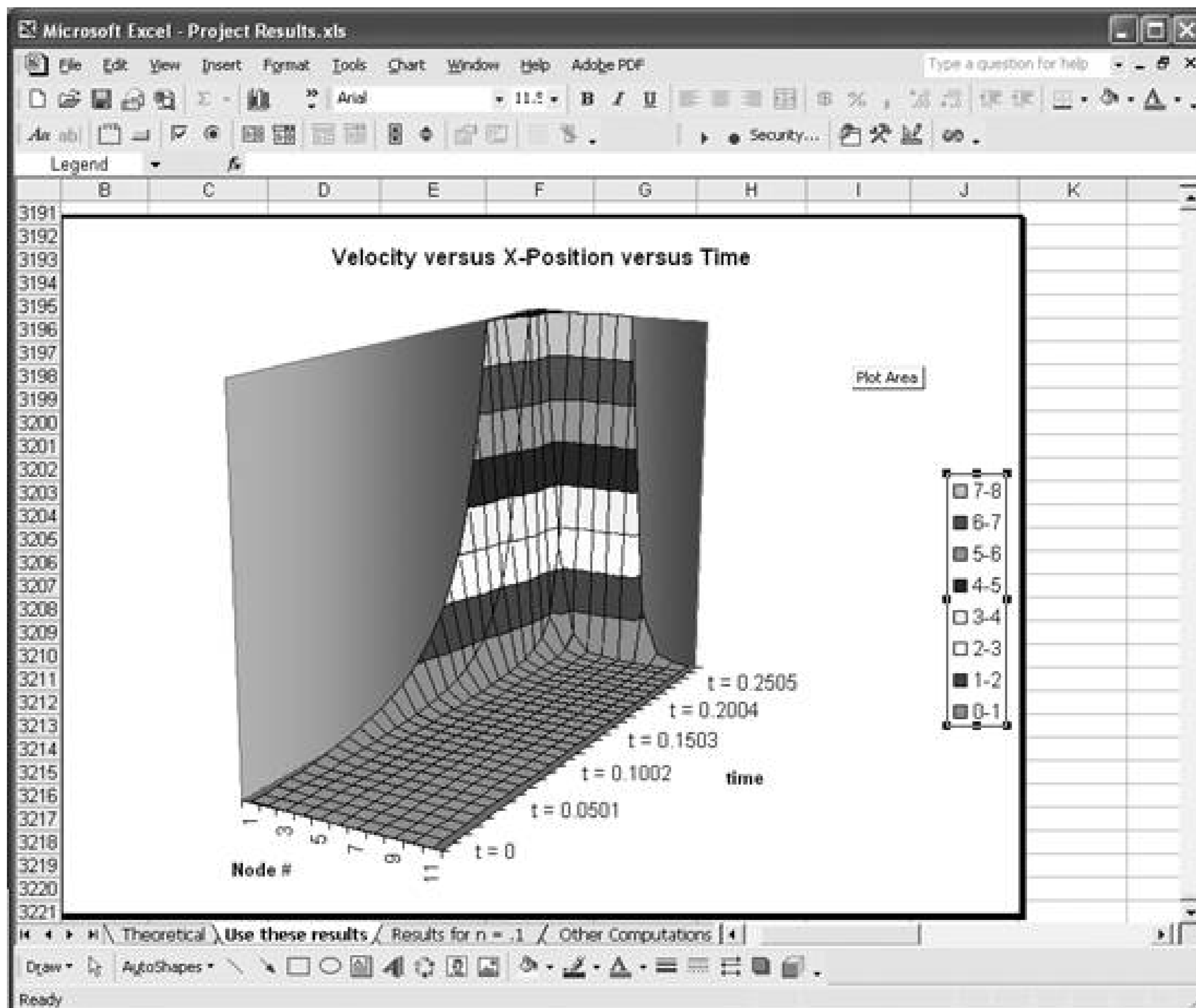
The colors shown on the surface represent values within small ranges; you can set these to control the number of different colors used on the plot. To do this, you need to format the legend's scale in much the same way as you would adjust the scale of an axis as discussed in [Recipe 4.4](#). Specifically, you can adjust the major and minor units (see [Figure 4-11](#)) to control the number of colors used to represent small ranges of values in the plot.

To format the legend's scale, you'll first have to display it. Right-click on the chart and select Chart Options from the menu to display the Chart Options dialog box. Click the Legend tab and then check the Show Legend control.

After closing the Chart Options dialog box, select the legend and press Ctrl-1 to open the Format Legend dialog box. Click the Scale tab to reveal scale controls like those shown in [Figure 4-11](#). Adjust the scale to suit your needs.

Figure 4-23 shows a new version of the chart from Figure 4-22, with a reduced number of colors.

Figure 4-23. 3-D Surface chart with adjusted legend scale



If you set the major and minor units too small, you could end up with a large number of colors and items in the legend. This may make the legend too big to display on your chart. If you want to keep such a large number of colors, then you'll probably want to hide the legend.

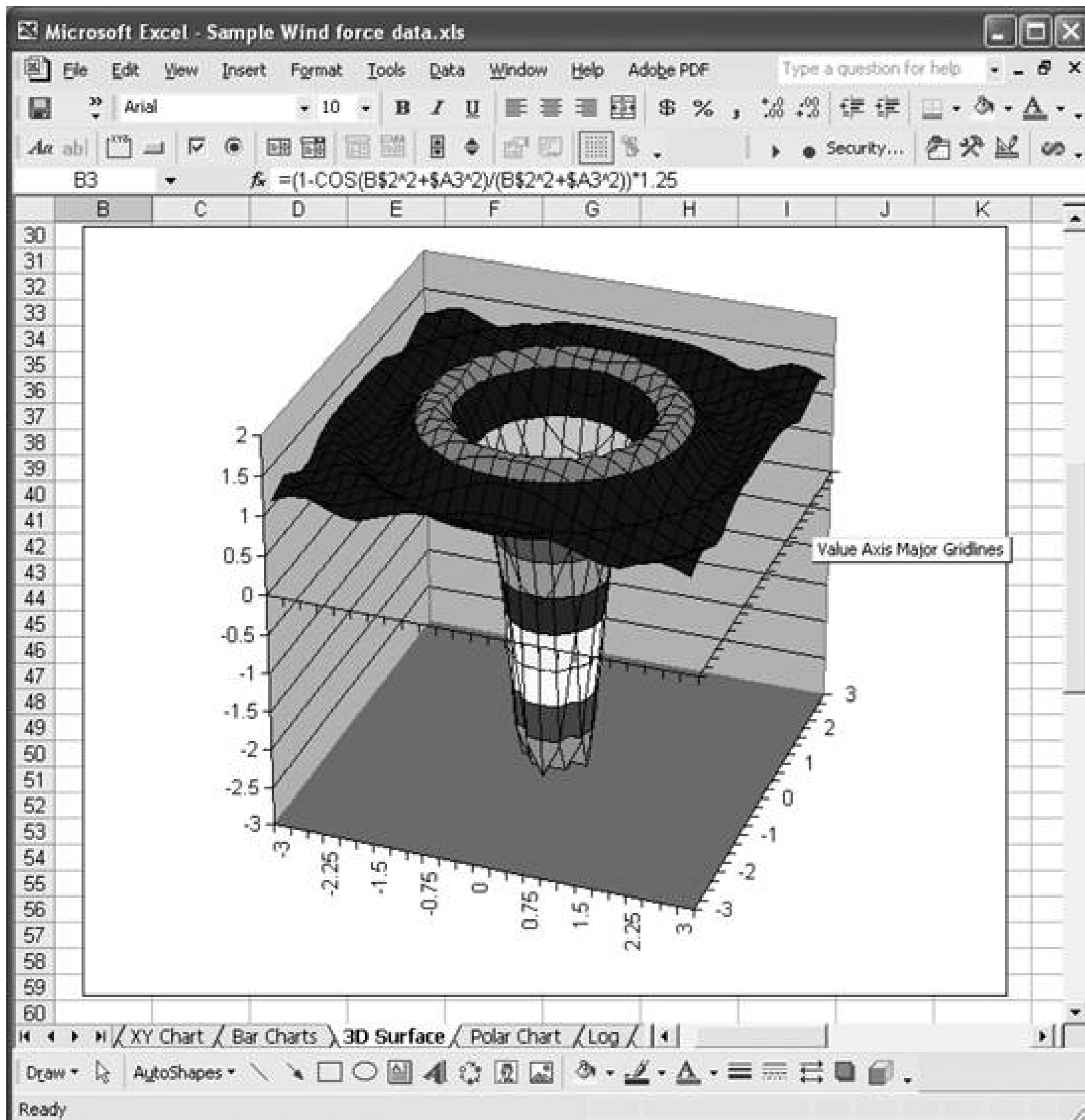
You can format the legend and data series even further. After selecting the legend, click on any legend entry to select it (you'll see selection handles around it). Right-click on it and select Format Legend Entry from the pop-up menu. You'll then be able to change the formatting of the legend entry font. With a legend entry selected, click on its legend key (the little colored square next to the legend entry text). Now right-click on the key and select Format Legend Key from the pop-up menu. You'll then be able to change the formatting of the legend key. For example, you can change its color or fill pattern, which also affects the chart. This allows you to set specific colors for the values displayed in the chart. In the Format Legend Key dialog box, you'll also find an Options tab. Click it to reveal additional formatting options that affect the entire chart. For example, you can change the depth of the chart or set a 3D shading effect, giving the chart more of a 3D look.

3-D Surface charts can also be useful for plotting 3D analytic functions to visualize their form. For example, the plot of the function

$$z = \left(1 - \frac{\cos(x^2 + y^2)}{x^2 + y^2}\right) \cdot k$$

looks like that shown in [Figure 4-24](#).

Figure 4-24. 3-D Surface chart of sample analytic function



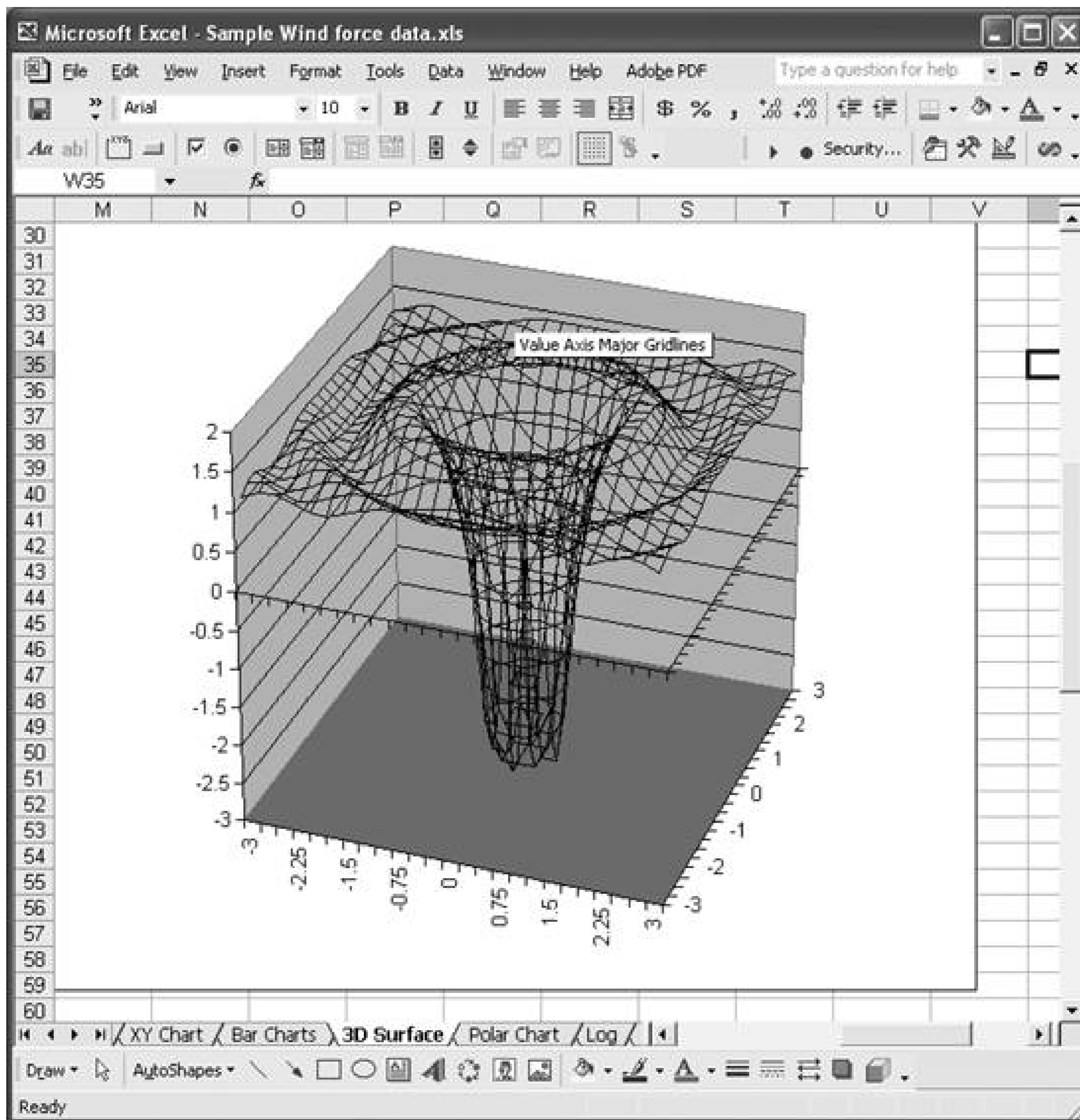
Notice that the  $x$ - and  $y$ -axes (in the horizontal plane) show real numbers. These resemble axes in



an XY scatter plot, but they are still just category labels here. The trick to pulling this sort of plot off is to make sure you perform your calculations at uniformly spaced  $x$ - and  $y$ -values.

Excel's Surface chart type also gives you the option of displaying wireframe representations of surfaces such as that shown in [Figure 4-25](#).

Figure 4-25. Wireframe 3-D Surface chart of sample analytic function



Sometimes, wireframe views are more effective in revealing the structure of such a surface without the distraction of all the colors.



You're not restricted to the point of view I used in these examples, by the way. Excel allows you to change the point of view by rotating, translating, or even changing the level of perspective of your 3D plot. Right-click on your chart to reveal a pop-up menu and then select 3-D View.... This will open the 3-D View dialog box, which contains several controls allowing you to adjust the point of the view for your chart.

← PREV

← PREV

## Recipe 4.10. Preparing Contour Plots

### Problem

You'd like to prepare a contour plot (e.g., to illustrate an elevation map, a map of pressure readings, or any other value distributed over a uniformly spaced grid).

### Solution

Use Excel's Surface chart type. Open the Chart Type dialog as discussed in [Recipes 4.1](#) and [4.9](#). Select the Surface chart type and then the Contour chart sub-type.

### Discussion

[Figure 4-26](#) shows a set of data representing measured height readings taken over a uniform 10 x 10 grid.

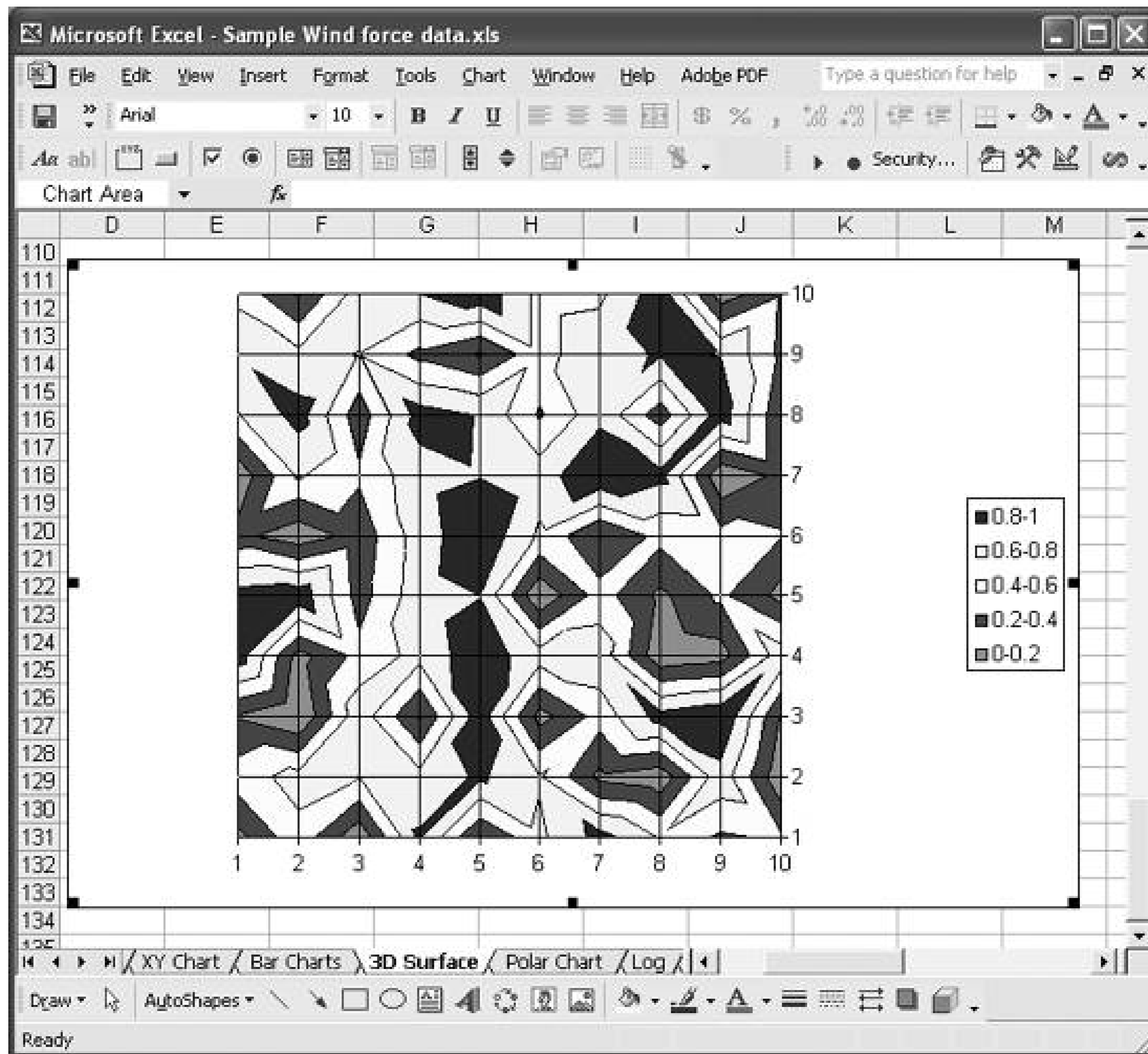
Figure 4-26. Contour plot data

	B	C	D	E	F	G	H	I	J	K	L
63		1	2	3	4	5	6	7	8	9	10
64	1	0.140676	0.534399	0.050435	0.828107	0.181307	0.55829	0.873775	0.579031	0.365988	0.430929
65	2	0.502589	0.677143	0.605485	0.665349	0.830919	0.619246	0.148804	0.067502	0.726693	0.039967
66	3	0.179645	0.000186	0.730047	0.177122	0.906014	0.117065	0.479982	0.882422	0.98174	0.182952
67	4	0.914466	0.118111	0.461858	0.667982	0.916043	0.686153	0.730117	0.043735	0.109748	0.724152
68	5	0.892751	0.931118	0.311447	0.712056	0.796304	0.031014	0.486694	0.151947	0.444144	0.127098
69	6	0.256561	0.067398	0.27623	0.694233	0.972964	0.562342	0.191462	0.460236	0.462198	0.414749
70	7	0.069957	0.622564	0.432992	0.722556	0.786428	0.704569	0.98773	0.851825	0.033259	0.247105
71	8	0.59015	0.876595	0.291382	0.876395	0.790822	0.371813	0.749635	0.285769	0.907145	0.227644
72	9	0.7744	0.645964	0.593161	0.346847	0.180342	0.544341	0.71269	0.820843	0.793769	0.358577
73	10	0.540328	0.167921	0.702979	0.802209	0.940901	0.539632	0.563927	0.910737	0.073485	0.376628
74											
75											
76											
77											
78											
79											
80											

You could use a 3-D Surface chart to display this data as discussed in [Recipe 4.9](#), or you may instead want to present this data as a contour plot.

Excel's Surface chart type allows you to plot contour plots in addition to 3D surface plots. Contour and Wireframe Contour plots are sub-types of the Surface type. If you create a Surface-Contour chart (see [Recipe 4.1](#) for the basic chart-creation steps) with the data shown in [Figure 4-26](#), you'll end up with the contour plot shown in [Figure 4-27](#).

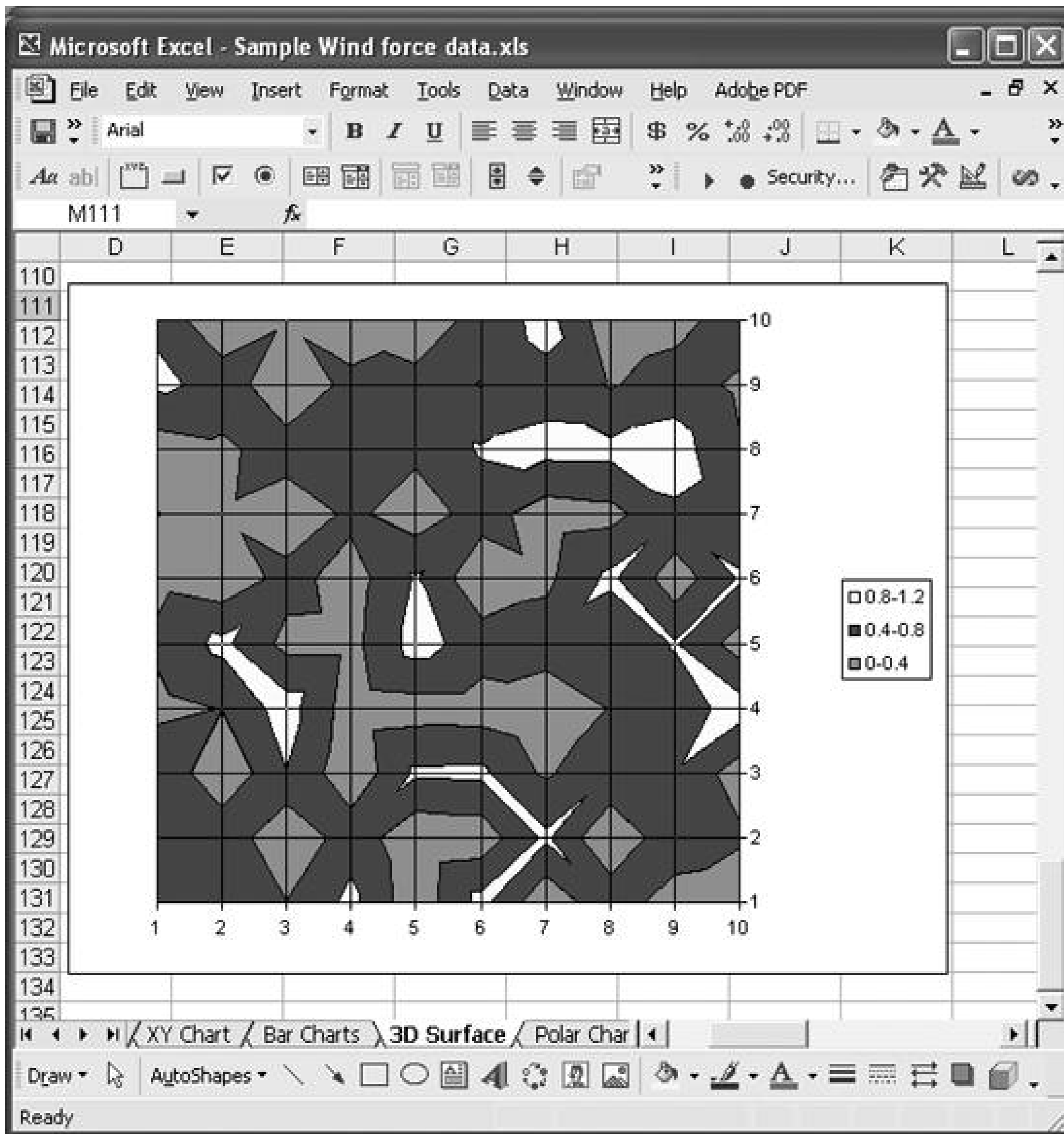
Figure 4-27. Contour chart



Note the legend displayed on this chart. You can format this legend as discussed in [Recipe 4.9](#) to either increase or decrease the number of colors (and represented data ranges) displayed on the chart. [Figure 4-28](#) shows a new version of the plot from [Figure 4-27](#), with a slightly reduced resolution, so to speak.

This chart uses only two fewer colors, but results in a very different lookit's less cluttered and perhaps more clearly highlights the data of interest. You can use the legend scale in this way to really draw attention to the data you want the viewer to focus on.

Figure 4-28. Contour chart with reduced number of colors



## Recipe 4.11. Annotating Charts

### Problem

You'd like to embellish your charts with annotations (for example, to add emphasis or provide notes for viewers of your charts).

### Solution

Use Excel's drawing tools to draw directly on your charts.

### Discussion

The callouts (text boxes with arrows) on the chart shown in [Figure 4-16](#) in [Recipe 4.8](#) were added using Excel's drawing tools. You need to display the Drawing toolbar if it isn't already visible. Select **View** → **Toolbars** → **Drawing** from the main menu bar. [Figure 4-16](#) shows the Drawing toolbar docked in the lower left of Excel's window.

The drawing toolbar contains all sorts of tools, allowing you to add text notes, callouts, arrows, shapes, and many other elements to your charts. There are also tools that allow you to format your annotations. For example, you can edit line style and thickness, color, and fill pattern, among other attributes. The 3D style buttons even allow you to create 3D effects to really make your annotations stand out. There are also tools that allow you to change the depth order, spacing, and alignment of your annotations.

To draw on your chart, simply select your chart and then click on the desired element from the drawing toolbar. Now start drawing on your chart. For example, to add a line to your chart, click the chart, then click the line button on the drawing toolbar, then click and drag on your chart to add the line. You can change the position of the line after it has been drawn by clicking and dragging it. You can also edit the line endpoints by clicking and dragging the endpoint handles (little circles).

By the way, you can use these same drawing tools directly on your spreadsheets; they aren't limited to drawing on charts. Follow the same procedure described in this recipe for drawing on charts, but instead just draw directly on your spreadsheet.

[Figure 4-29](#) shows an example chart that I embellished using Excel's drawing tools.

This chart illustrates results of a standard beam analysis. The results include load, shear, and bending moment curves, as well as calculated reactions at each support. To create this chart, I

prepared a table of data including the load, shear, and bending moment values at various locations along the beam. I plotted this data on a standard XY scatter chart using the procedure outlined in [Recipe 4.1](#). I formatted the primary and secondary axes to display specific ranges with minimum and maximum units, as described in [Recipe 4.4](#). For the primary  $x$ -axis, I specified the crossing point for the  $x$ -axis as -800. I did this to force the  $x$ -axis toward the bottom of the chart.

After setting up the curves and axes, I labeled the bending moment and shear curves using Callout text boxes with leader arrows, which are accessible from the Drawing toolbar's Autoshapecs Callouts menu. To represent the beam, I added a basic rectangle shape and manually positioned it where the  $y$ -axis values show zero. I formatted the rectangle using a cross-hatched fill pattern.

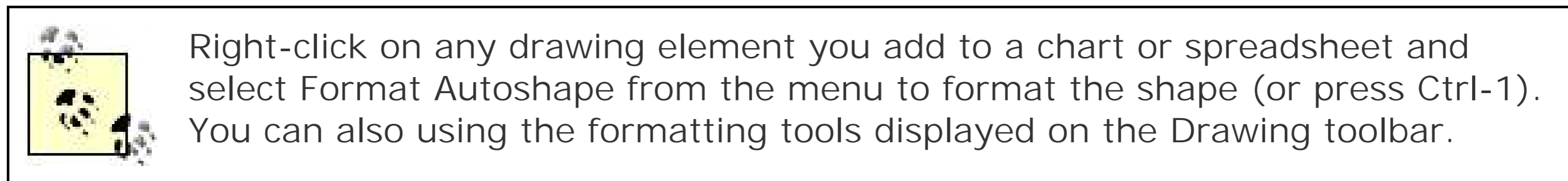
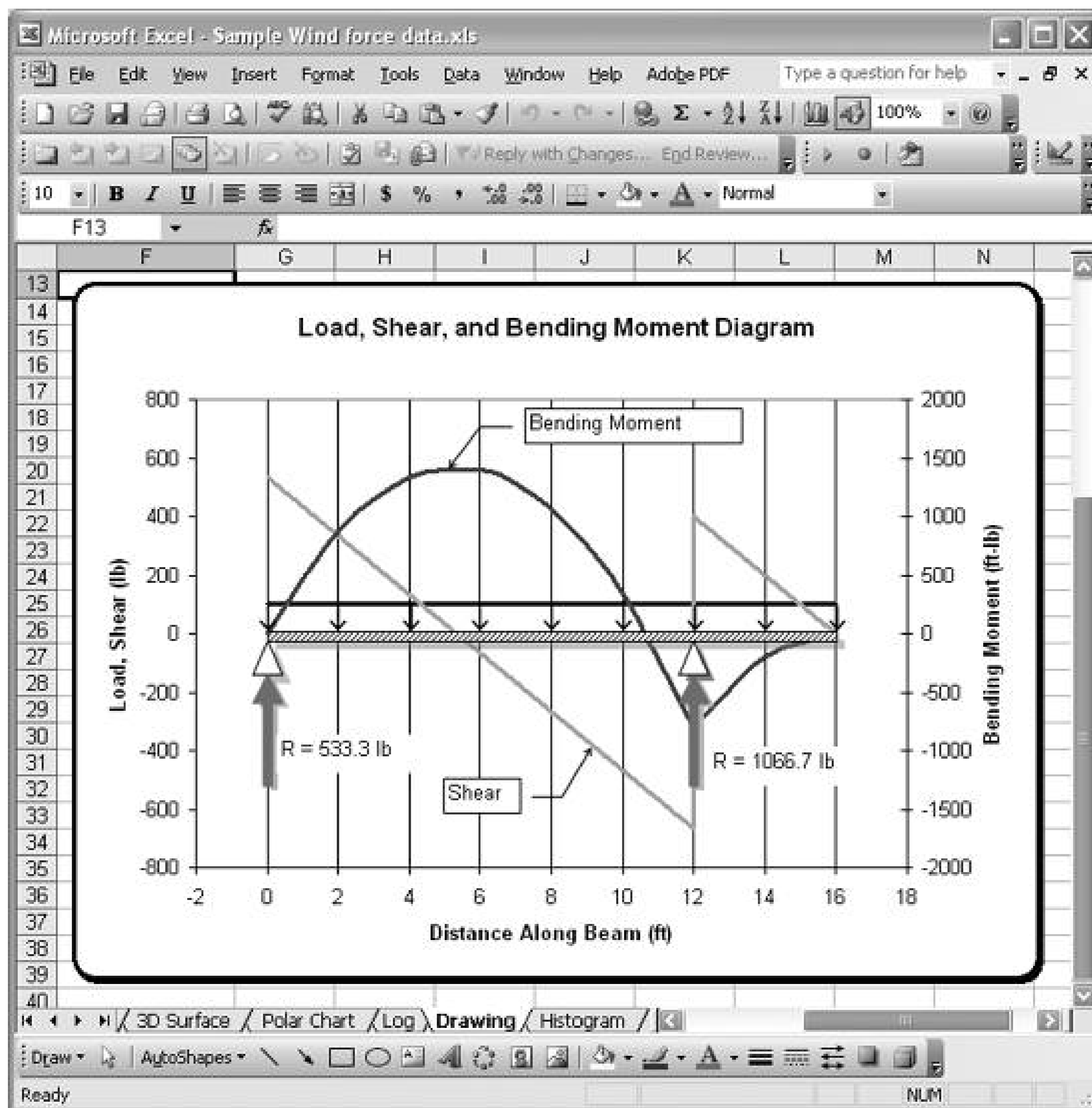


Figure 4-29. Annotated chart





I used small triangle shapes to represent standard, simple supports for the beam. I added these by accessing the AutoShapes Basic Shapes menu from the Drawing toolbar. I manually positioned them at the proper  $x$ -location using the mouse. For the reaction forces, I added a couple of arrows from the AutoShapes Block Arrows menu on the Drawing toolbar. I formatted these with a red fill color. Further, I added a couple of text boxes (also accessible from the Drawing toolbar) to indicate the calculated reaction forces. I formatted the beam, support, and reaction shapes with a faint drop shadow for a little depth effect. I used the Shadow Style tool (the button with a rectangle and drop shadow, second from the right) from the Drawing toolbar to format the drop shadow.

These are just a few examples of the sort of embellishments you can add to your technical charts to make them more meaningful, readable, and attractive. Excel's drawing tools are far more extensive than has been demonstrated here. You can add 3D text and shapes, draw smooth lines and edit their points, and prepare flow charts, among many other drawing tasks. I encourage you to explore the drawing tools more.

## See Also

Do a search in Excel's help using the keywords "drawing" or "autoshapes" to find help topics specifically related to drawing in Excel.



[← PREVIOUS](#)

## Recipe 4.12. Saving Custom Chart Types

### Problem

You find yourself repeatedly creating the same type of chart, with the same custom formats, titles, colors, and so on, and you'd like to be able to save your customizations so you don't have to redo them each time you create a new chart.

### Solution

Add your customized chart to the Custom Types list to make it available in the Custom Type dialog box and Chart Wizard. After creating your customized chart, right-click on it and select Chart Type from the pop-up menu. In the Chart Type dialog box, select the Custom Types tab. Select the "User-defined" radio button toward the bottom of the tab. Press the Add... button to bring up the Add Custom Chart Type dialog box. Type a unique name in the Name field and type a short description in the Description field. Press OK when you're done; you should then see your custom chart's name added to the Custom Types chart list. Now you can select this type when creating new charts, instead of having to recustomize one of Excel's built-in charts each time.

## Recipe 4.13. Copying Charts to Word

### Problem

You've created a chart in Excel and would like to include it in a report written in Word.

### Solution

Simply copy and paste the chart from Excel to Word. Select your chart and press Ctrl-C, or select Edit → Copy from the main menu bar. Open your Word document and select the line where you'd like to paste the chart and press Ctrl-V or select Edit → Paste from the main menu bar.

### Discussion

This seems simple enough, and it is. But you do have some options to consider when pasting an Excel chart into a Word document. By default, a picture of the chart gets pasted into Word. This picture is static: you can't edit the chart and it isn't linked to the original spreadsheet.

When you paste a chart into Word, you'll notice a little clipboard icon near the lower-right corner of the pasted chart. If you select that icon, you'll see three paste options: Picture of Chart; Excel Chart; Link to Excel Chart. The first option is just a picture of the chart, as explained a moment ago.

The second option is really quite powerful. The entire spreadsheet gets embedded into your Word document. This means that you can transmit the entire spreadsheet along with the Word document to other readers of your document. Within Word you can right-click on the chart and select Chart Object → Edit from the pop-up menu to edit the chart right there, as though you were in Excel. You can even edit the spreadsheet data used to create the chart. This may prove handy in some cases. Be careful, however, not to transmit underlying data and formulas to people who shouldn't see that data.

The third option embeds just the chart (not the entire spreadsheet), but not as a static picture. Instead it's a link to the original Excel spreadsheet. When the spreadsheet gets updated, the embedded chart will be updated too. You can access the linked spreadsheet by right-clicking on the chart in Word and selecting Linked Worksheet Object → Open Link from the pop-up menu. This will open the linked spreadsheet in Excel, where you can make changes if you'd like. Any changes will be reflected in the Word document containing the linked chart.

When you open a Word document containing a linked chart, Word will ask you if you want to update the linked chart so that recent changes to the Excel chart are reflected in the Word document. If for some reason the link between the Word document and Excel file gets broken (for example, if you move or rename the Excel file), then Word will inform you that it could not update the linked chart.





## Recipe 4-14. Displaying Error Bars

### Problem

You want to display error bars on data series in your charts.

### Solution

You can display error bars for a series by opening the Format Data Series dialog box. Select the series and press Ctrl-1. Click the Y Error Bars tab to specify how you'd like the error bars displayed. You can choose to display plus or minus bars or both. Further, you can specify the error as a fixed value, a percentage of the value being plotted, or as custom values in a cell range you provide. [Figure 4-20](#) (in [Recipe 4.9](#)) shows an example data series with error bars.

[← PREV](#)

# Chapter 5. Statistical Analysis

[Section 5.0. Introduction](#)

[Recipe 5.1. Computing Summary Statistics](#)

[Recipe 5.2. Plotting Frequency Distributions](#)

[Recipe 5.3. Calculating Confidence Intervals](#)

[Recipe 5.4. Correlating Data](#)

[Recipe 5.5. Ranking and Percentiles](#)

[Recipe 5.6. Performing Statistical Tests](#)

[Recipe 5.7. Conducting ANOVA](#)

[Recipe 5.8. Generating Random Numbers](#)

[Recipe 5.9. Sampling Data](#)

## 5.0. Introduction

As you've seen in [Chapter 3](#), it's fairly easy to import data into Excel for analysis. Certainly, performing statistical analyses on such data would be very convenient to do in Excel. Indeed, Excel has dozens of built-in spreadsheet functions that allow you to perform all sorts of statistics calculations. The Analysis ToolPak add-in also contains several other statistical tools. Moreover, being able to visualize your data as discussed in [Chapter 4](#) makes Excel especially convenient for certain analyses.

Excel's help documents contain descriptions of and syntax for all of its statistical functions; there are dozens of them, covering all sorts of calculations from descriptive statistics to significance tests. The help topic "Statistical Functions" lists all such functions, providing links to more details on the syntax and use of each function, along with examples in many cases.

The help topic "About Statistical Analysis Tools" also provides information on the use of the analysis tools available in the Analysis ToolPak add-in. To make sure you have the Analysis ToolPak add-in available in your version of Excel, select Tools from the main menu bar and see if the Data Analysis menu option appears toward the bottom of the Tools menu. If not, select Tools → Add-Ins from the main menu bar and select the Analysis ToolPak option from the list.

In this chapter I'm going to show you how to use Excel to perform standard statistical calculations. I don't go into the theory behind all of these analyses, so you should consult your favorite statistics book for relevant background material.

Also, I don't include any recipes on regression and curve fitting in this chapter. That material deserves a chapter of its own; thus, [Chapter 8](#) is devoted to curve fitting and regression.



## Recipe 5.1. Computing Summary Statistics

### Problem

You need to compute summary statistics (measures of central tendency and dispersion) but are not sure what support Excel offers for these calculations.

### Solution

Excel offers many built-in spreadsheet functions allowing you to quickly compute summary statistics. Further, the Analysis ToolPak offers a Descriptive Statistics tool. You can perform the same statistical calculations offered in the Descriptive Statistics tool using spreadsheet functions. There are however, caveats that you need to know about. Read the following discussion for more information.

### Discussion

I'll show you how to perform basic statistical calculations in Excel on an example set of data. The dataset I'll use was downloaded from the National Institute of Standards and Technology statistical reference dataset page at <http://www.itl.nist.gov/div898/strd/general/dataarchive.html>. The dataset consists of speed of light measurements from a study conducted by Michelson in 1879. The data consists of 100 values representing the speed of light in units of millions of meters per second. [Figure 5-1](#) illustrates a portion of this data along with several computed statistics for the dataset.

The dataset is in column B, labeled *Speed of Light (millions of m/s)*. There are 100 values in this column, but only the first dozen and a half or so are shown in the figure.

Column F contains statistics for this dataset computed using the Analysis ToolPak's Descriptive Statistics tool. The corresponding labels contained in column E indicate what each value represents. Column H contains statistics computed using Excel's built-in spreadsheet formulas. I'll show you how to use both methods.

First, to use the Descriptive Statistics tool, you have to open the Analysis ToolPak by selecting Tools Data Analysis from the main menu bar. This action opens the Data Analysis dialog shown in [Figure 5-2](#).

Select Descriptive Statistics from the list of available tools and press OK to open the Descriptive Statistics dialog box shown in [Figure 5-3](#).

In the Input Range field, type (or select directly from your spreadsheet) the cell range containing the data you want to analyze. Since the example data is in a column, I selected the Grouped By: Columns option.

For this example, I decided to place the output on the same spreadsheet, next to the dataset. Therefore, I specified cell E6 for the Output Range. This specified cell represents the upper-left cell of the output table. You may choose to place the output on a new worksheet, in which case you need to specify a worksheet name.

Figure 5-1. Summary statistics of speed of light measurements

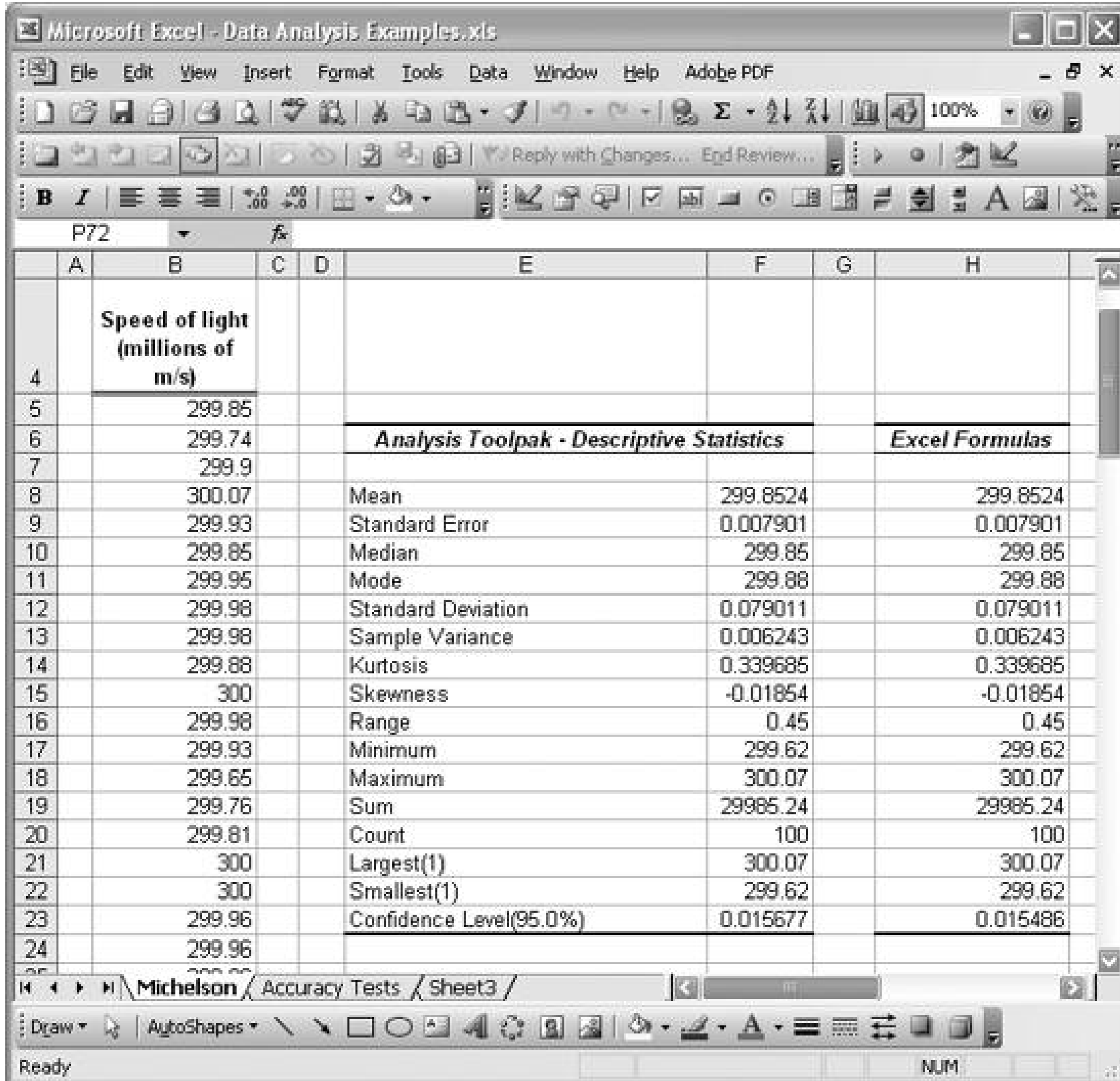
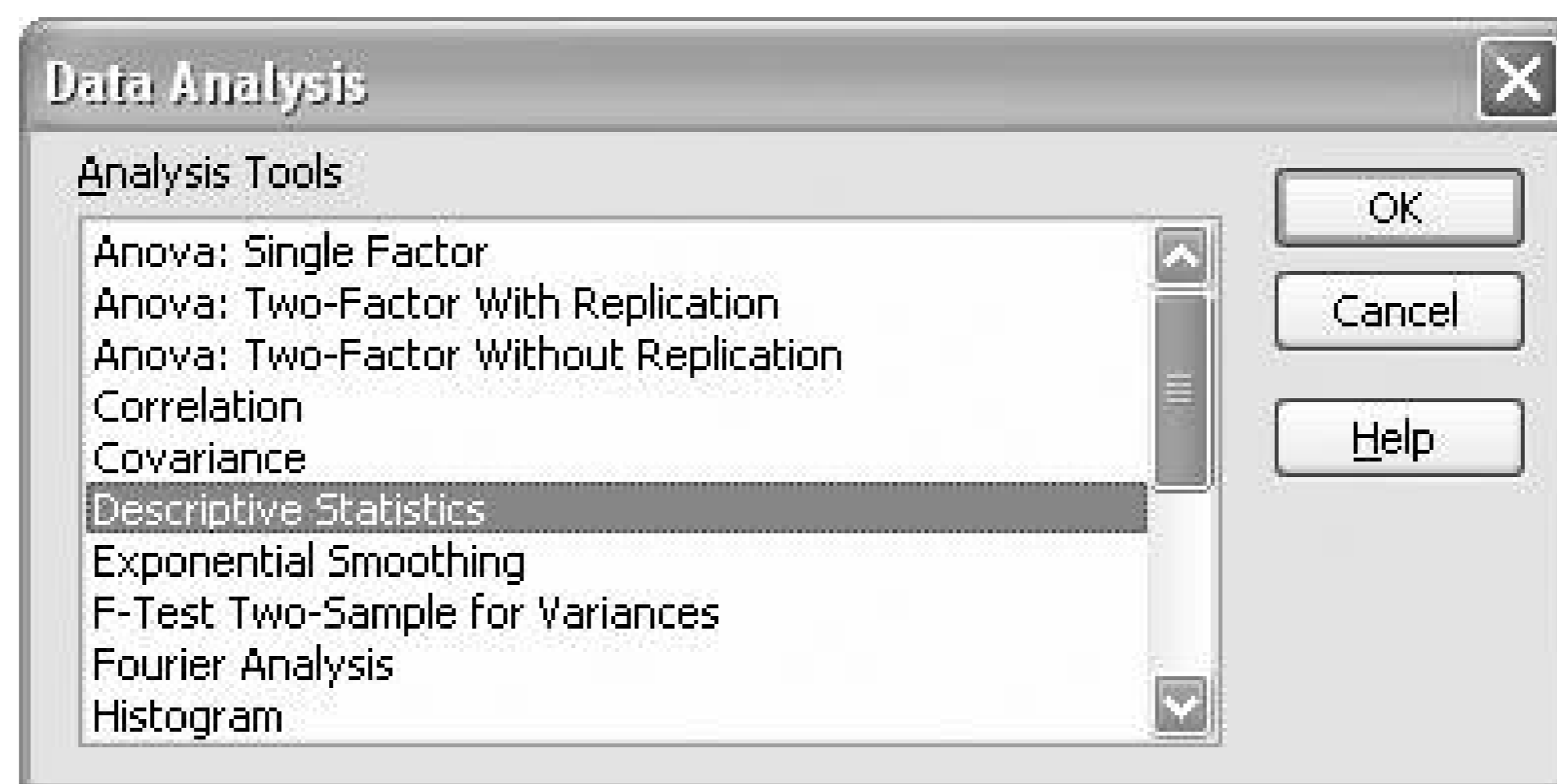


Figure 5-2. Data Analysis dialog box



I checked all of the options shown in the lower part of the dialog box in [Figure 5-3](#) in order to generate all the available statistics. Pressing OK generates the results shown earlier in [Figure 5-1](#).

Figure 5-3. Descriptive Statistics dialog box

You can compute the same statistics using only Excel's built-in functions. Cells H8 through H23 in [Figure 5-1](#) contain the same set of statistics, computed using Excel's functions. [Table 5-1](#) shows the cell formula I used to compute each statistic.

Table 5-1. Summary statistics cell formulas

Statistic	Cell formula
Mean	=AVERAGE(Data)
Standard Error	=H12/SQRT(H20)
Median	=MEDIAN(Data)
Mode	=MODE(Data)
Standard Deviation	=STDEV(Data)
Sample Variance	=VAR(Data)
Kurtosis	=KURT(Data)
Skewness	=SKEW(Data)
Range	=h18-H17
Minimum	=MIN(Data)
Maximum	=MAX(Data)
Sum	=SUM(Data)
Count	=COUNT(Data)
Largest(1)	=LARGE(Data,1)
Smallest(1)	=SMALL(Data,1)
Confidence Level(95.0%)	=CONFIDENCE(0.05,H12,100)

Most of these formulas require the cell range containing the input data as an argument. For convenience, I named the cell range B5 to B104 (containing the input data) `Data`. (See [Recipe 1.14](#) to learn how to name cells and cell ranges.)

[Table 5-1](#) illustrates that Excel has built-in formulas for all of these basic statistics with two exceptions. There's no formula for computing the Standard Error (standard error of the mean in this case). Nor is there a formula for computing the Range. However these two statistics are easily computed using the results of other formulas. For example, to compute the range you simply take the difference between the Maximum and the Minimum. To compute the Standard Error of the mean, you simply divide the Standard Deviation by the square root of Count.

If you take a close look at [Figure 5-1](#), you'll see that the statistics computed using Excel's formulas exactly match those computed with the Analysis ToolPak with one exception: the values for the 95% confidence interval are different. This difference is due to the fact that the `CONFIDENCE` function uses the inverse of a standard normal distribution to compute the confidence interval, whereas the Analysis ToolPak uses the inverse of Student's t-distribution. Excel does provide other functions that allow you to compute confidence intervals using Student's t-distribution. See [Recipes 5.3](#) and [8.8](#) for more information.

I should also mention that some of Excel's statistics functions (including those in the Analysis ToolPak) have historically dealt poorly with some troublesome datasets. For example, if your dataset contained very large numbers with very small differences between them, the Descriptive Statistics

tool would compute the standard deviation and variance incorrectly, and this would then affect results of other statistics that used those values. The workaround in this case is to center your data by subtracting the mean from each value (or by shifting the data by a sufficiently large value), and then compute the standard deviation and variance based on the centered data.

In Excel 2003, it appears that the statistics functions have been fixed, so this error no longer occurs. To see if this error exists on your version of Excel, enter the following data in a spreadsheet and compute the standard deviation: 10000000001, 10000000002, 10000000003, 10000000004, and 10000000005. The average, standard deviation, and variance for this dataset are 10000000003, 1.58113883, and 2.5, respectively. In older versions of Excel, the standard deviation and variance would come out to 0.

If you center your data by subtracting the mean from each value, your dataset should look like this: 2, -1, 0, 1, 2. The standard deviation and variance should be the same as for the uncentered dataset. In older versions of Excel, the standard deviation and variance for this centered dataset should come out correctly.

If you shift the original dataset by subtracting, say,  $1e10$  from each value, you end up with a dataset that looks like this: 1, 2, 3, 4, 5. The standard deviation and variance should be the same as in the previous two cases. Again, older versions of Excel deal with this shifted dataset just fine and compute the correct statistics.

This sort of troublesome data can also adversely affect the results of statistical tests covered later in this chapter. Here again, you should center the data before conducting such tests.

## See Also

Excel's help guide includes a complete list of all its built-in statistical functions. You can find this list by searching Excel's help for the topic "Statistical Functions."

Excel also has several built-in summation functions that are very handy for statistical calculations that involve sums, sums of squares, sums of squared differences, and so on. These functions are discussed in [Recipe 7.1](#).

Moreover, Excel has built-in functions to support regression and curve fitting. These, along with other regression techniques, are discussed in [Chapter 8](#).

[← PREVIOUS](#)

## Recipe 5.2. Plotting Frequency Distributions

### Problem

You're analyzing a dataset and would like to create a histogram plot of the data showing its frequency distribution.

### Solution

Use Excel's `FREQUENCY` function to compute the frequency of occurrence of each data point and use Excel's charting features (see [Chapter 4](#)) to create the histogram plot. Or use the Histogram tool in the Data Analysis add-in to compute the frequencies of occurrence and then use Excel's plotting features.

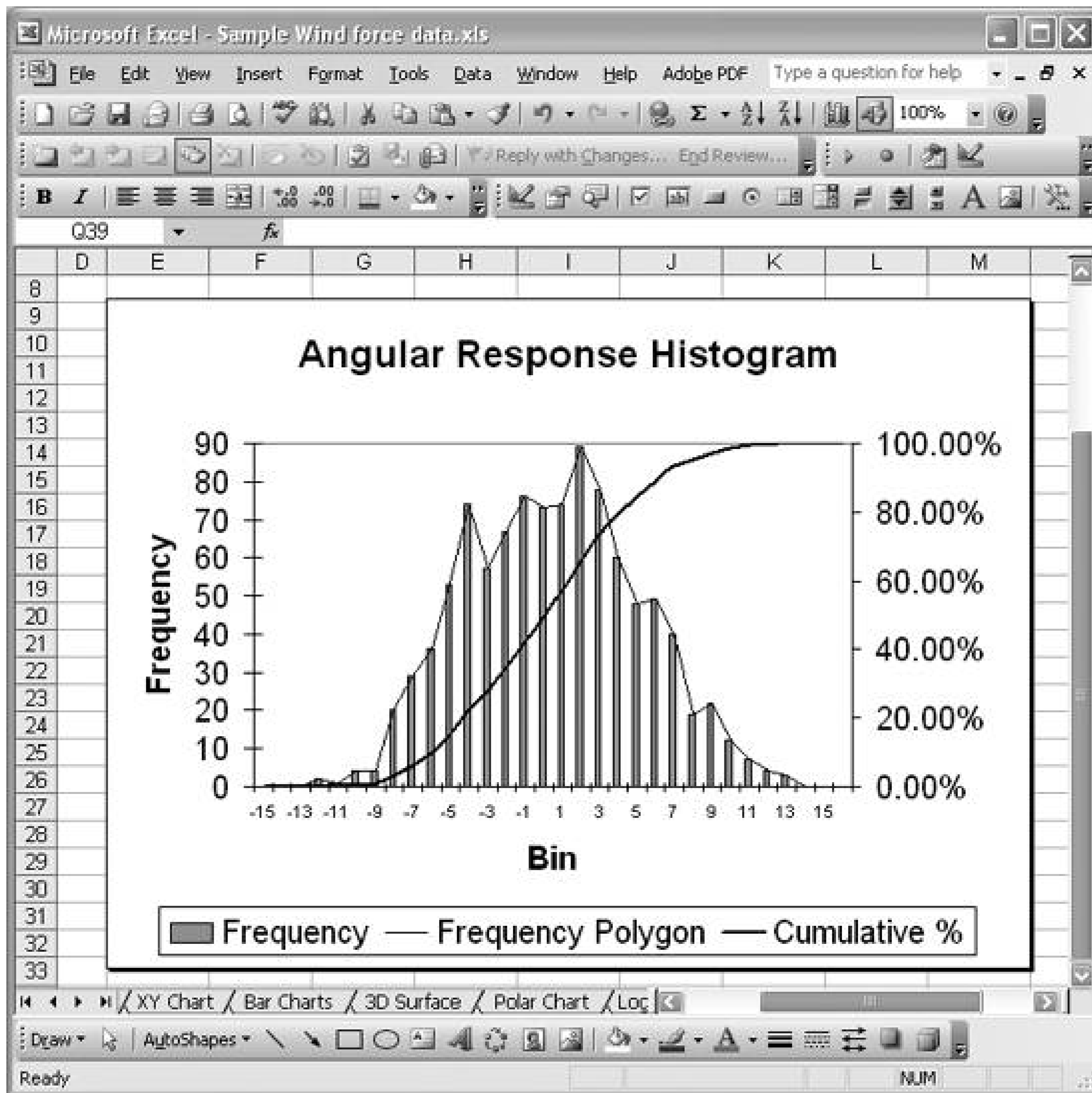
### Discussion

[Figure 5-4](#) shows a histogram plot (bar graph) along with a frequency polygon (the thin line) and cumulative frequency plot (the bold line) for a set of data. The dataset for this example plot represents the angular response of a torsional spring.

You can quickly create this sort of chart using the techniques discussed throughout [Chapter 4](#). You'll first have to compute the actual frequency distribution for your dataset, and to that end you have two choices in Excel: the `FREQUENCY` function or the Histogram tool included in the Data Analysis add-in.

[Figure 5-5](#) shows a portion of a spreadsheet containing the data used for the histogram plot shown in [Figure 5-4](#).

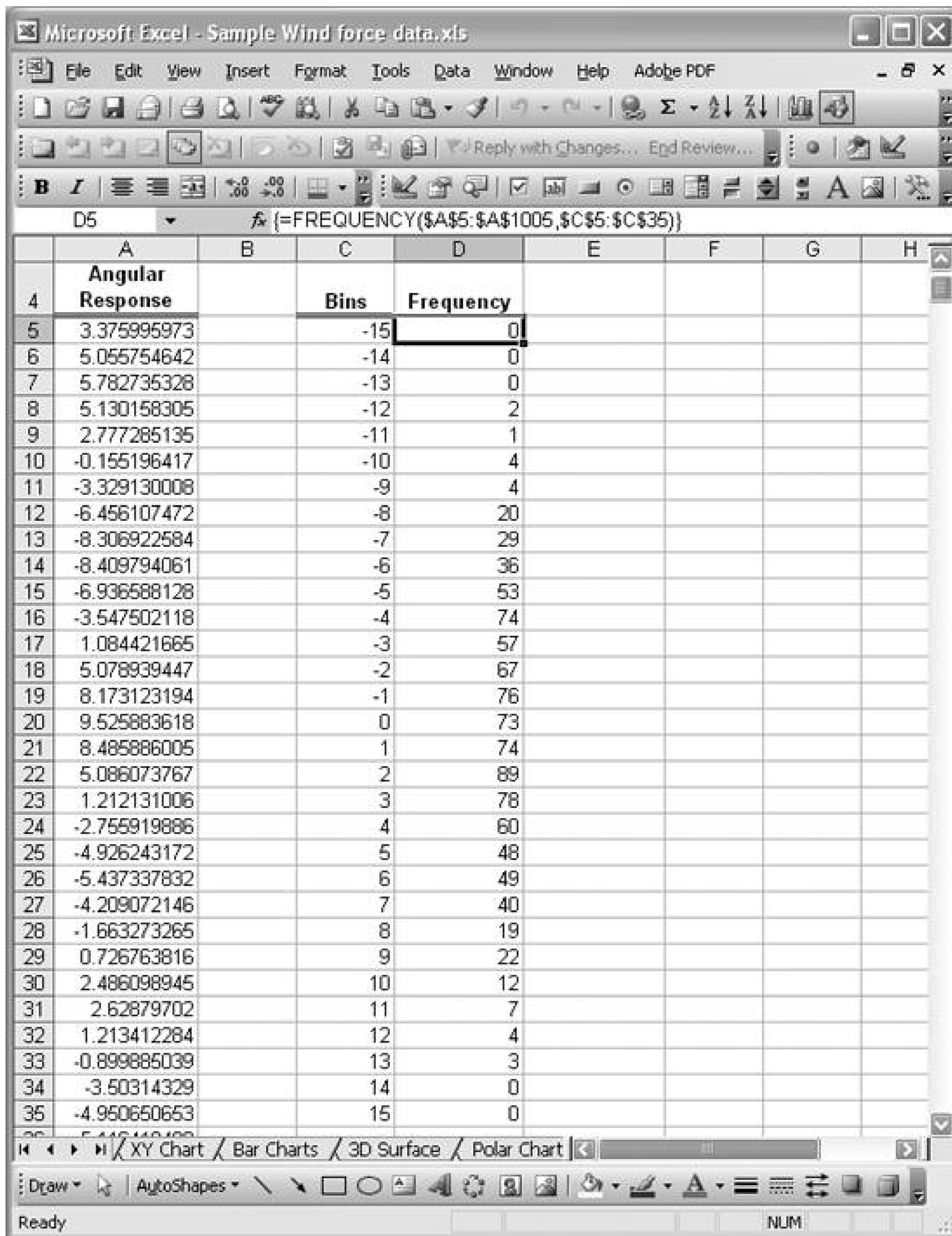
Figure 5-4. Histogram plot



The angular response data contained in column A is the original dataset for which frequencies are computed. Before applying the **FREQUENCY** function, you must set up a range of discrete bin values, called *bins*. The bins form bands, or ranges of values, within which values in the original dataset will be lumped when compiling frequencies of occurrence. In this example, I know the angular response values fall within approximately -15 to 15 degrees; therefore, I set up a range of bin values from -15 to 15 degrees in increments of 1 degree. These are contained in column C in [Figure 5-5](#).

To compute the frequencies for each bin, I applied the **FREQUENCY** formula, passing the dataset cell range as the first parameter and the bin cell range as the second parameter. For example, the formula in cell D5 is `{=FREQUENCY($A$5:$A$1005,$C$5:$C$35)}`. Notice that this is an array formula and to enter it you must first select the entire output cell range, cells D5 to D35 in this case, and then enter the formula by pressing Ctrl-Shift-Enter.

Figure 5-5. Histogram example data



You can compute such a frequency distribution using the Histogram tool instead of the `FREQUENCY` function if you'd like. To do so, select **Tools** → **Data Analysis** from the main menu bar to open the Data Analysis dialog box shown in [Figure 5-6](#).



Then select the Histogram tool from the list of available tools and press OK to open the Histogram dialog box shown in [Figure 5-7](#).

Figure 5-6. Data Analysis dialog box

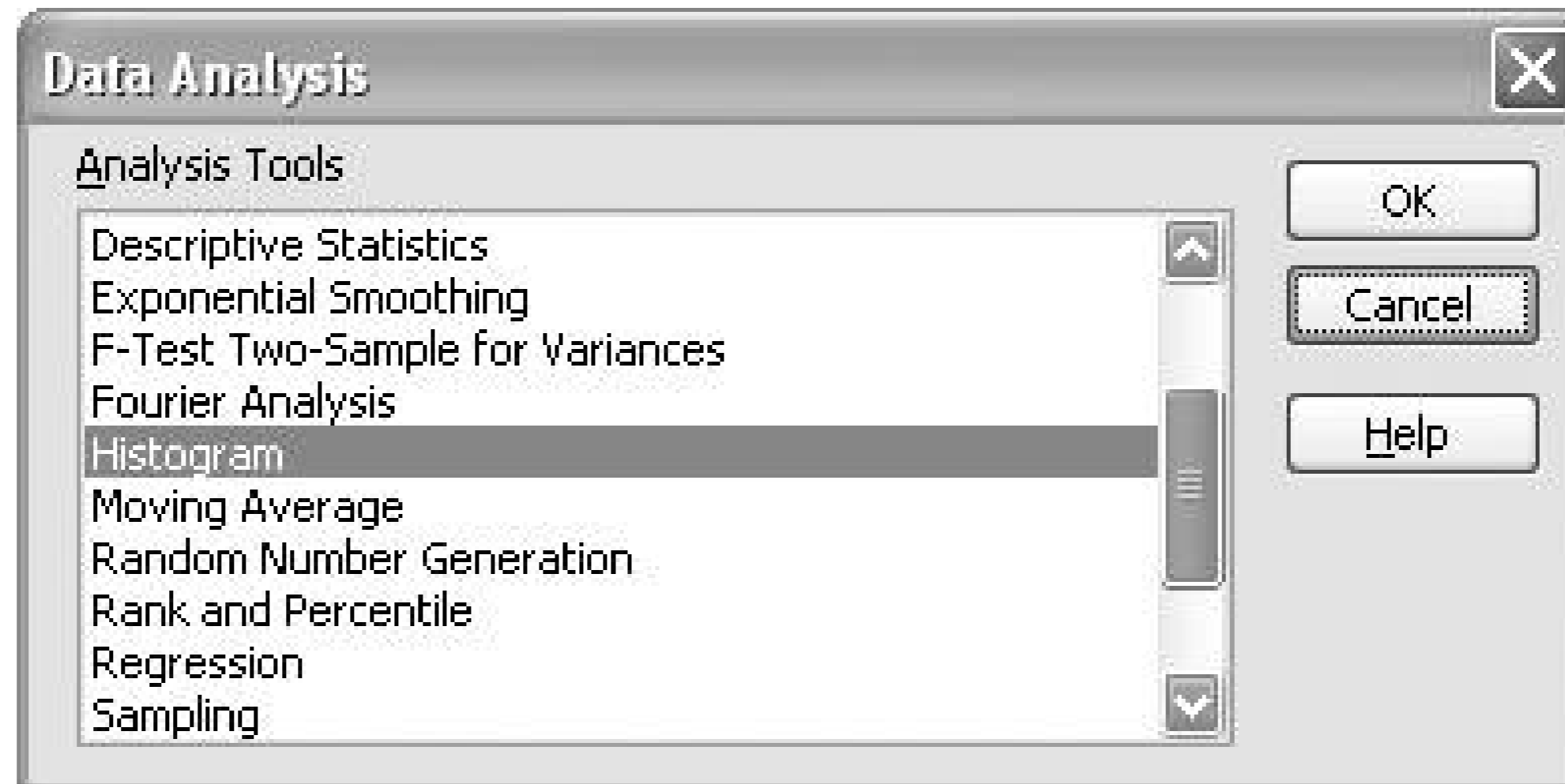


Figure 5-7. Histogram tool dialog box

In the Input Range field, enter or select from your spreadsheet the range of cells containing the input dataset. In the Bin Range field, enter or select the range of cells containing the bin values.

Under "Output options," select your preferred location for the output. In [Figure 5-7](#), I set the output range to start at cell C39 in the same worksheet that contains the original dataset.

There are a few other options you can choose, and in this example I chose the Cumulative

Percentage option to generate values for the cumulative plot shown earlier in [Figure 5-4](#). Upon pressing OK, you should see results similar to those shown in [Figure 5-5](#).

## See Also

Check out [Chapter 4](#) to learn more about creating charts in Excel.



## Recipe 5.3. Calculating Confidence Intervals

### Problem

You need to compute confidence intervals for certain estimates and you're not sure what support Excel provides for these calculations.

### Solution

Excel offers a built-in function called `CONFIDENCE` that allows you to compute confidence intervals based on a normal distribution. Moreover, Excel provides functions such as `NORMSINV` and `TINV` that allow you to compute inverses for normal and Student's t-distributions, respectively. You can then use these results to compute confidence intervals.

### Discussion

In [Recipe 5.1](#), I showed you how to use Excel's built-in functions to compute summary statistics. I also showed you how to compute summary statistics using the Descriptive Statistics tool available from the Analysis ToolPak. In both cases, confidence intervals for the mean were computed and I pointed out that the confidence interval returned by Excel's `CONFIDENCE` function is different from that returned by the Descriptive Statistics tool. [Figure 5-1](#) shows the different values.

The difference in the computed confidence interval is due to the fact that the `CONFIDENCE` function uses the inverse of a standard *normal distribution* to compute the confidence interval, whereas the Analysis ToolPak uses the inverse of *Student's t-distribution*. Instead of using the `CONFIDENCE` function, you can use the functions `NORMSINV` or `TINV` to compute confidence intervals; this gives you the option of using a normal distribution or Student's t-distribution, which is more suitable for small samples.

`NORMSINV` computes the inverse of a standard normal distribution with a mean of 0 and standard deviation equal to 1. In other words, `NORMSINV` returns the *z-score* corresponding to a given probability. The syntax for `NORMSINV` is `=NORMSINV(probability)`. For a 95% confidence interval, you need to enter a probability of  $1-0.05/2$  or `0.975`. Referring to the example shown in [Figure 5-1](#), you can compute the confidence interval for the mean using the formula `=H9*NORMSINV(0.975)`, where cell H9's value contains the standard error of the mean. This formula returns a confidence value of 0.015486, which agrees with the results returned using the `CONFIDENCE` function.

You can use `TINV` to compute the confidence interval based on Student's t-distribution. `TINV` computes the inverse of Student's t-distribution given a probability and the degrees of freedom characterizing the distribution. Again, referring back to the example in [Figure 5-1](#), you can compute the confidence interval using the formula `=H9*TINV(0.05,H20-1)`. In this case, the probability for 95%

confidence is entered as  $1-0.95$  or  $0.05$ . As before, H9 refers to the cell containing the standard error of the mean. Cell H20 contains the value returned by `COUNT`, and the number of degrees of freedom is computed by  $h20-1$ . The resulting confidence value is 0.015677, which agrees with the result returned by the Descriptive Statistics tool.

## See Also

You can use the formulas discussed in this recipe to compute confidence intervals for other estimates as well. In [Recipe 8.8](#), I show you how to compute confidence intervals for regression curves.



[← PREV](#)

## Recipe 5.4. Correlating Data

### Problem

You're working with a multivariable problem and want to compute the correlation coefficient between variables.

### Solution

Use Excel's built-in functions `CORREL` or `PEARSON`, or use the Correlation tool available in the Analysis ToolPak.

### Discussion

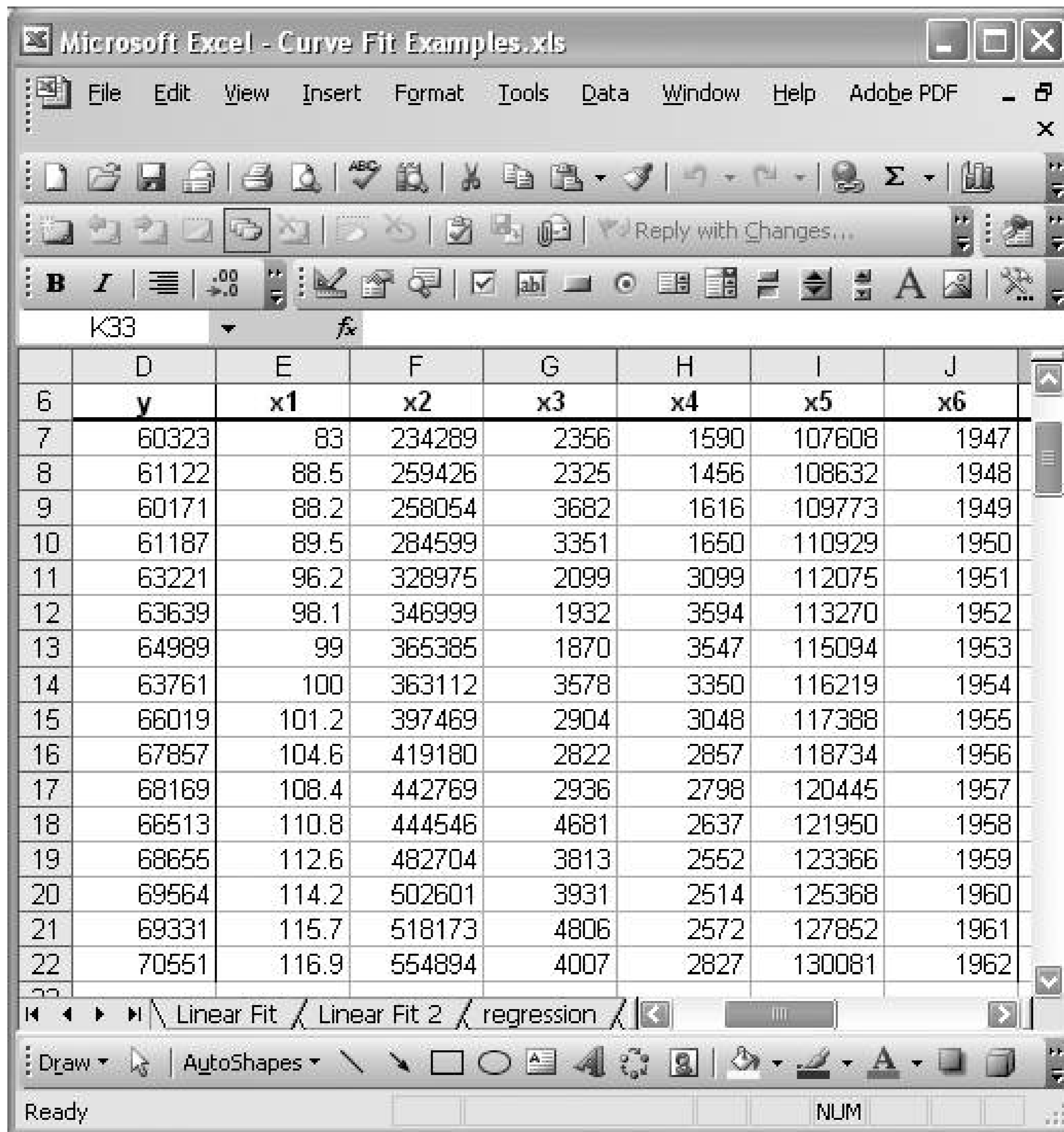
In [Recipe 8.4](#), I show you how to perform linear regression where the dependent variable depends on more than one independent variable. The example I discuss in that recipe involves six independent variables. This is a typical example where it's prudent to assess the correlation between independent variables before conducting the regression analysis. This way you can avoid using highly correlated (or colinear) independent variables, which can give you trouble.

Excel provides two built-in functions that allow you to compute the Pearson product-moment correlation coefficients between variables. These functions are `CORREL` and `PEARSON` and, according to Excel's help documents, they perform exactly the same calculation. (I've no idea why there are two functions to do the same thing.)

[Figure 5-8](#) shows the data from the multiple linear regression example I discuss in [Chapter 8](#). For now, I'll show you how to examine the correlation between independent variables using Excel's correlation functions as well as the Analysis ToolPak.

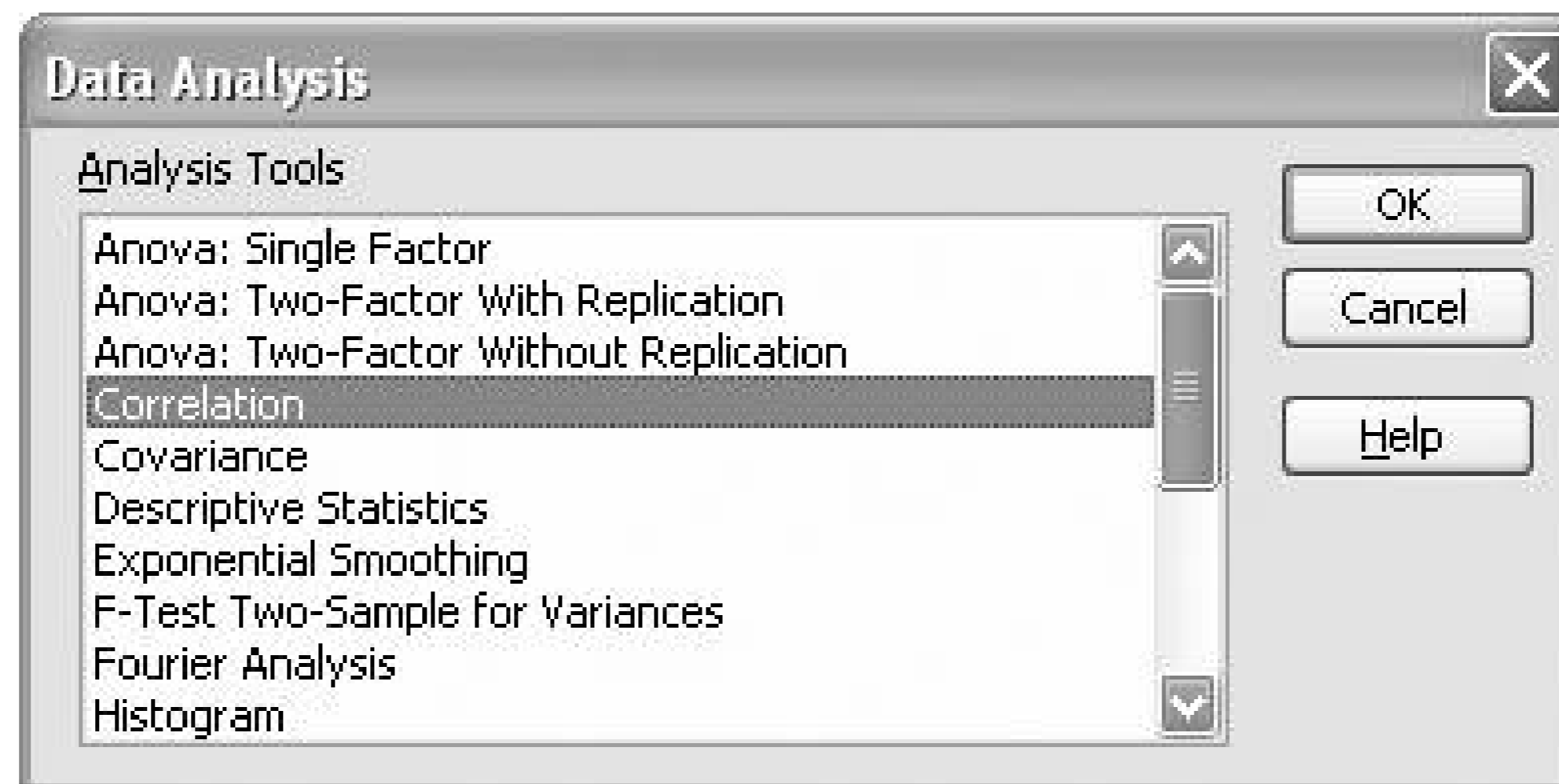
To compute the correlation coefficient between variables `x1` and `x2`, you can use the cell formula `=CORREL(E7:E22,F7:F22)` or `=PEARSON(E7:E22,F7:F22)`. They both produce the same results, which is 0.991589. Clearly, these two variables are positively correlated and in a regression analysis, for example, you would probably exclude one of these variables from the curve fit. You can compute correlation coefficients between the other variables in a similar manner.

Figure 5-8. Multiple linear regression data



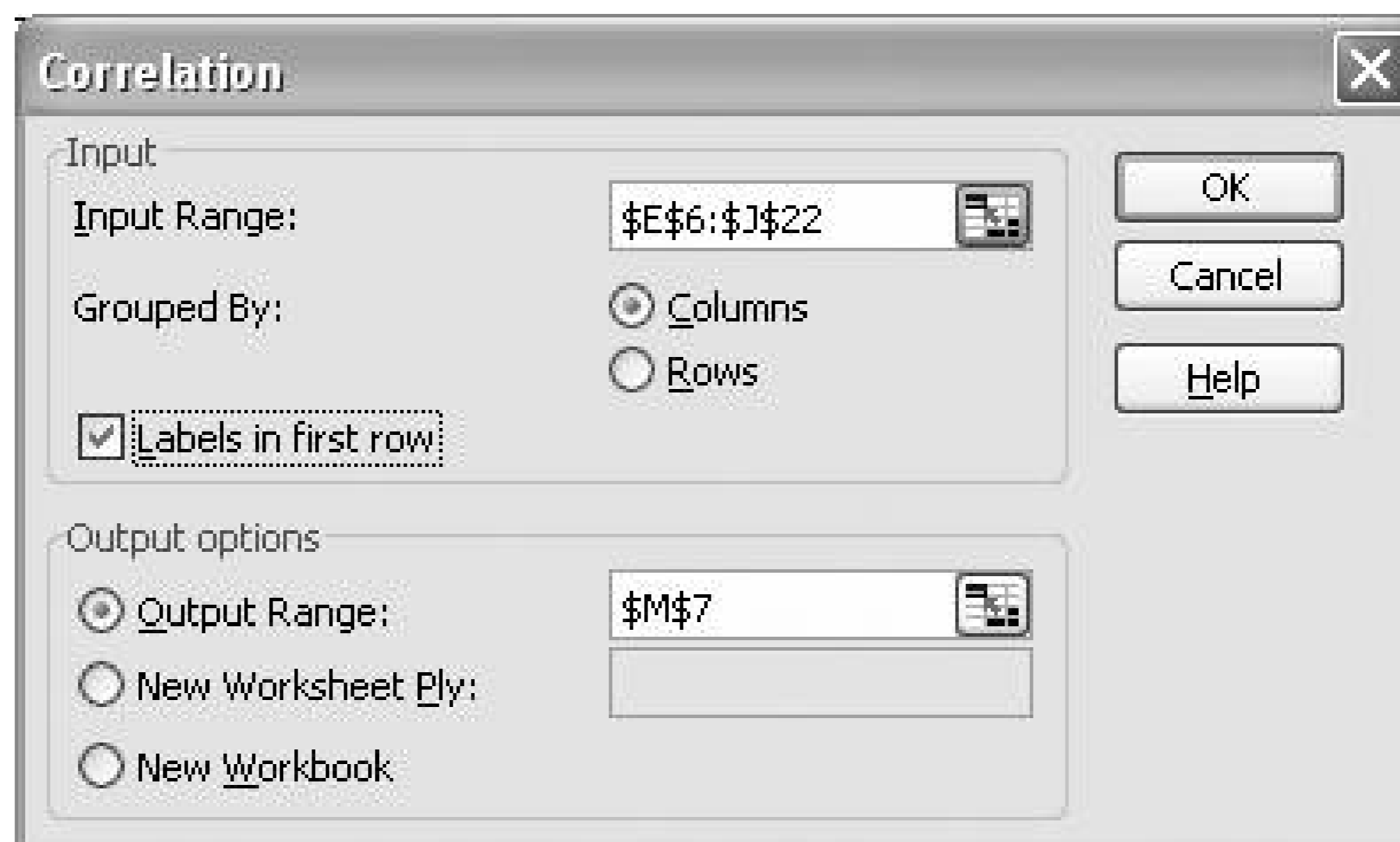
To save some time when dealing with numerous variables, you might consider using the Correlation tool available in the Analysis ToolPak. Select Tools > Data Analysis from the main menu bar to open the Data Analysis dialog box shown in [Figure 5-9](#).

Figure 5-9. Data Analysis Correlation tool



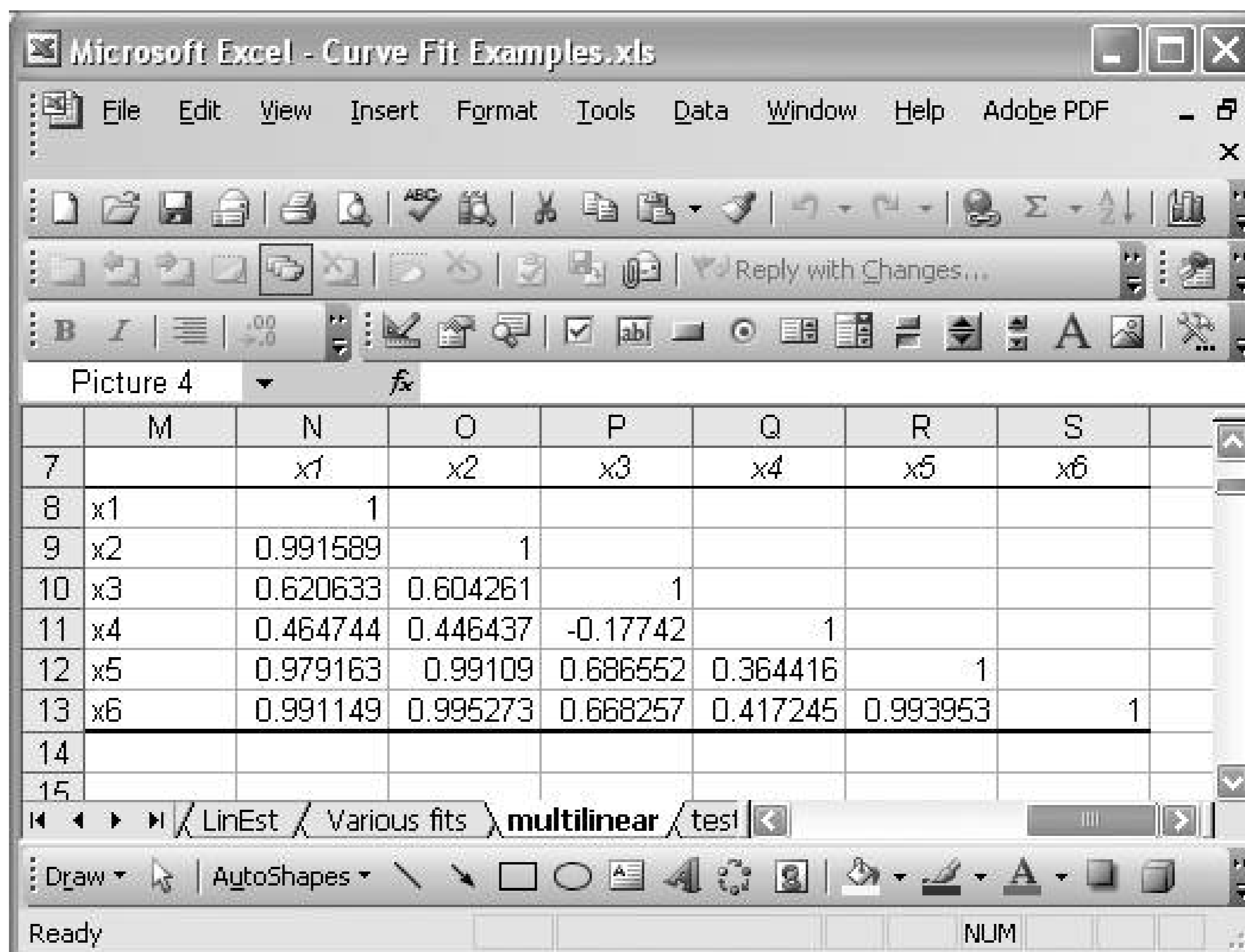
Select the Correlation tool and press OK to open the Correlation tool dialog box shown in [Figure 5-10](#).

Figure 5-10. Correlation tool dialog box



The Correlation tool allows you to select multiple datasets at once; it will compute the correlation coefficients between all the datasets. For example, in [Figure 5-10](#) I entered cells E6 through J22 in the Input Range. This cell range includes all of the  $x$  variables shown in [Figure 5-8](#). I included the header labels in the input range, so I checked the "Labels in first row" option. Further, I specified cell M7 (on the same worksheet) as the starting cell where I'd like the output displayed. [Figure 5-11](#) shows the results.

Figure 5-11. Correlation results



	M	N	O	P	Q	R	S
7		x1	x2	x3	x4	x5	x6
8	x1	1					
9	x2	0.991589	1				
10	x3	0.620633	0.604261	1			
11	x4	0.464744	0.446437	-0.17742	1		
12	x5	0.979163	0.99109	0.686552	0.364416	1	
13	x6	0.991149	0.995273	0.668257	0.417245	0.993953	1
14							
15							

This table presents the correlation coefficient between each given variable and every other variable. It also correlates each variable with itself, which of course results in a correlation coefficient of 1 (as you can see in [Figure 5-11](#)).

Sometimes it's useful to compute *coefficients of determination* instead of correlation coefficients alone. To compute a coefficient of determination, simply square the correlation coefficient. For example, if cell N9 contains the correlation coefficient between variables **x1** and **x2**, then the formula `=N9^2` would yield the coefficient of determination.

There are other types of correlation coefficients common in statistical analysis. For example, *Phi*, *rank biserial*, *point biserial*, and *Spearman rank* coefficients are common, in addition to the Pearson coefficient. Excel does not have built-in support for these other correlation coefficients; however, you can certainly set up a spreadsheet to calculate these coefficients.

For example, you can readily prepare a spreadsheet to compute the Spearman rank correlation coefficient between variables **x1** and **x2** from the previous example. The Spearman coefficient requires you to convert the raw data to ranks first. You can achieve this conversion using Excel's built-in `RANK` function (see [Recipe 5.5](#)). [Figure 5-12](#) shows a simple spreadsheet I set up to compute the Spearman rank correlation coefficient between the variables **x1** and **x2** from the previous example.

The first two columns in the table contain the raw data. The third and fourth columns contain the rank of each data point. I used the `RANK` function for these calculations. For example, cell E41 contains the cell formula `=RANK(C41,$C$41:$C$56,1)`, which computes the rank for the first **x1** value, assuming the data are sorted in ascending order (see [Recipe 5.5](#)).



The formula for computing the Pearson rank correlation coefficient is:

$$r = 1 - \frac{6 \sum d^2}{N(N^2 - 1)}$$

Here,  $d$  is the difference in ranks between corresponding values for both  $x_1$  and  $x_2$ , and  $N$  is the number of data points. You can compute  $N$  using the `COUNT` function. Cell D58 contains the formula `=COUNT(D41:D56)` which returns a value of 16 for the number of points.

To compute  $d^2$ , I set up another column containing formulas like `=(E41-F41)^2`. The results are shown in the last column of the table shown in [Figure 5-12](#).

Cell G57 computes the sum of  $d^2$  values using the formula `=SUM(G41:G56)`. Cell G59 computes the  $N(N^2 - 1)$  term using the formula `=(D58*(D58^2-1))`. And finally, the correlation coefficient is computed in cell G60 using the formula `=1-6*G57/G59`. The resulting correlation coefficient is 0.997.

Figure 5-12. Spearman rank correlation coefficient calculation

Microsoft Excel - Data Analysis Examples.xls

File Edit View Insert Format Tools Data Window Help Adobe PDF

10 B  $f_x = 1-6*G57/G59$

Spearman Correlation Coefficient:					
Raw Data		Rank		d <sup>2</sup>	
x1	x2	x1	x2		
83	234289	1	1	0.0	
88.5	259426	3	3	0.0	
88.2	258054	2	2	0.0	
89.5	284599	4	4	0.0	
96.2	328975	5	5	0.0	
98.1	346999	6	6	0.0	
99	365385	7	8	1.0	
100	363112	8	7	1.0	
101.2	397469	9	9	0.0	
104.6	419180	10	10	0.0	
108.4	442769	11	11	0.0	
110.8	444546	12	12	0.0	
112.6	482704	13	13	0.0	
114.2	502601	14	14	0.0	
115.7	518173	15	15	0.0	
116.9	554894	16	16	0.0	
				2.0	
	N =	16			
			N(N <sup>2</sup> -1) =	4080	
			r =	<b>0.99706</b>	

Ready NUM

## See Also

Take a look at [Recipe 8.4](#) to see the resulting regression model for this example dataset. [Chapter 8](#) also discusses coefficients of determination in the context of least-squares curve fitting.

Read [Recipe 5.5](#) to learn more about **RANK** and other ranking functions.



## Recipe 5.5. Ranking and Percentiles

### Problem

You need to compute certain percentiles of a set of data and you'd also like to compute the rank of certain values in the dataset.

### Solution

Use the built-in functions `PERCENTILE`, `RANK`, and `PERCENTRANK`. Or you can use the Rank and Percentile tool available in the Analysis ToolPak.

### Discussion

By way of example, let's assume your dataset consists of the values 1, 3, 5, 7, 9, 2, 4, 6, 8, 10, and 0. Let's also assume this dataset resides on a spreadsheet in the cell range C18:C28. (This is a rather simple dataset for example purposes; in practice your dataset can be anything and it need not be sorted.)

To compute the 25% percentile, use the formula `=PERCENTILE(C18:C28,0.25)`, which returns a value of 2.5. To compute the 95% percentile, use `=PERCENTILE(C18:C28,0.95)`, which returns a value of 9.5 as expected.

To compute the rank of a given value, use the formula `=RANK(2,C18:C28,1)`. The rank of the value 2 in this dataset is 3, assuming the dataset is sorted in ascending order. If you want to compute the rank of a value as though the dataset were in descending order, then use a value of 0 for the third argument in the call to `RANK`.

If you'd like to compute the rank of a value in percentage terms, then use the `PERCENTRANK` function. For example, `=PERCENTRANK(C18:C28,2,2)` returns the rank of the value 2 as 20%. The second argument in this function call represents the value whose rank is sought. The third argument is the number of significant digits for the result (in this case, two significant digits).

Instead of using these built-in functions, you could use the Analysis ToolPak's Rank and Percentile tool. Select Tools → Data Analysis from the main menu bar to open the Data Analysis dialog box shown in [Figure 5-13](#).

Select "Rank and Percentile" from the list and press OK to open the "Rank and Percentile" dialog box shown in [Figure 5-14](#).

In the Input Range field, type (or select from your spreadsheet) the cell range containing the input

dataset you'd like to rank. You can also specify the output location, as shown in [Figure 5-14](#).

Figure 5-13. Data Analysis dialog box

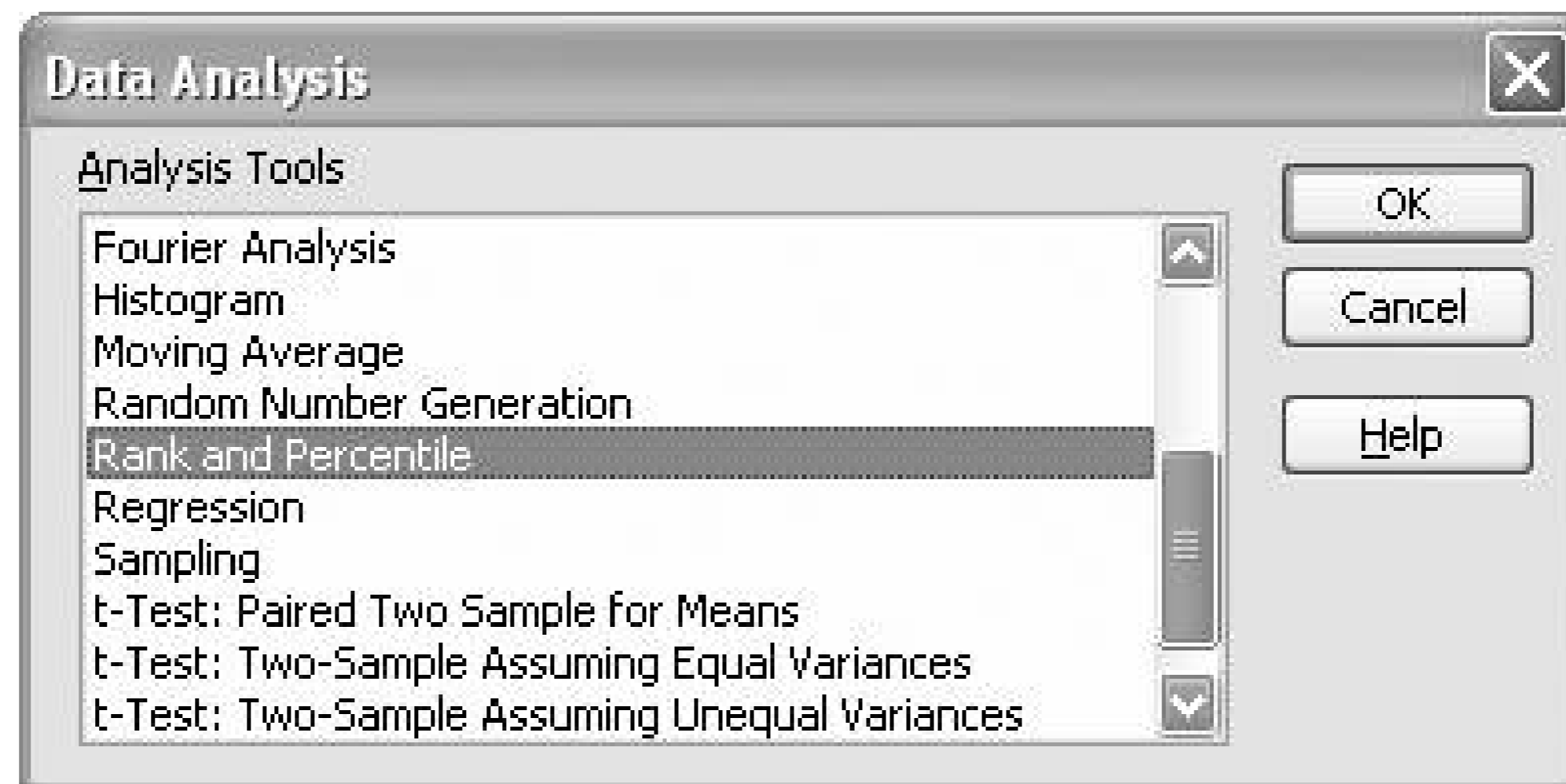


Figure 5-14. Rank and Percentile tool dialog box

For the simple example dataset discussed earlier, the Rank and Percentile tool returns the results shown in [Figure 5-15](#).

The ranks are computed assuming the data is sorted in descending order. The first column in the resulting table contains an index for each data point. The second column contains the raw value of the data point. The third column contains the corresponding rank. The last column contains the percentage rank.

In the event of ties (that is, where two or more values in the dataset are the same), you may want to use the correction recommended in Excel's help file for computing ranks. The formula to use is  $=[\text{COUNT}(\text{data})+1-\text{RANK}(\text{value}, \text{data}, 0)-\text{RANK}(\text{value}, \text{data}, 1)]/2$ . In this formula, *data* is a cell

reference containing the input dataset and *value* is the value for which you want to find a rank. See the Excel help topic for the **RANK** function for more information. To the best of my knowledge, the Rank and Percentile tool in the Analysis ToolPak does not handle ties. So if your data contains ties and you need to correct for this, you should use the **RANK** function instead.

Figure 5-15. Rank and Percentile results

The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Data Analysis Examples.xls". The spreadsheet displays the following data:

	H	I	J	K	L	M
3						
4		<i>Point</i>	<i>Column1</i>	<i>Rank</i>	<i>Percent</i>	
5		11	10	1	100.00%	
6		10	9	2	90.00%	
7		9	8	3	80.00%	
8		8	7	4	70.00%	
9		7	6	5	60.00%	
10		6	5	6	50.00%	
11		5	4	7	40.00%	
12		4	3	8	30.00%	
13		3	2	9	20.00%	
14		2	1	10	10.00%	
15		1	0	11	0.00%	
16						

## Recipe 5.6. Performing Statistical Tests

### Problem

You're engaged in hypothesis testing and you'd like to learn about Excel's support for conducting statistical tests .

### Solution

Excel provides support for conducting standard *z-tests*, *t-Tests*, and *F-tests*. Read the following discussion to learn more.

### Discussion

The built-in function `ZTEST` computes the one-tailed z-test using the normal distribution. The syntax for this function is `=ZTEST(cell ref,  $\mu$ ,  $\sigma$ )` where *cell ref* is a cell reference containing an input dataset,  $\mu$  is the value being tested, and  $\sigma$  is the population standard deviation. If  $\sigma$  is not known, it can be omitted, in which case the sample's standard deviation will be computed and used in the z-test calculation.

You can also compute two-tailed z-tests as follows: `=2 * MIN(ZTEST(cell ref,  $\mu$ ,  $\sigma$ ), 1 - ZTEST(cell ref,  $\mu$ ,  $\sigma$ ))`.

The function `TTEST` computes the one- or two-tailed t-test using Student's t-distribution. Further, `TTEST` allows you the option of performing paired, two sample with equal variance, or two sample with unequal variance tests. The syntax for `TTEST` is `=TTEST(cell ref 1, cell ref 2, tails, type)`, where *cell ref 1* and *cell ref 2* are cell references to the two datasets under consideration and *tails* is the number of tails to consider for the test (1 or 2). *type* is a value of 1, 2, or 3, indicating the test type: paired, two sample equal variance, or two sample unequal variance, respectively.

`FTEST` computes the two-tailed F-test for two datasets. The syntax for `FTEST` is `=FTEST(cell ref 1, cell ref 2)`, where *cell ref 1* and *cell ref 2* are cell references to the two datasets under consideration.

All three of these test functions have corresponding inverse functions (namely, `NORMSINV`, `TINV`, and `FINV`) that are useful when computing critical values for comparison. `NORMSINV` and `TINV` were discussed in [Recipe 5.3](#). [Recipe 8.7](#) makes use of `FTEST` and `FINV` while performing an F-test in the context of regression analysis.

The Analysis ToolPak also provides several tools for conducting statistical tests. These tools include:

- F-Test Two-Sample for Variances
- t-Test Paired Two-Sample for Means
- t-Test Two-Sample Assuming Equal Variances
- t-Test Two-Sample Assuming Unequal Variances
- z-Test Two-Sample for Means

To access these tools, select Tools → Data Analysis from the main menu bar to open the Data Analysis dialog box. This is the same Data Analysis dialog box shown earlier in [Figure 5-13](#). You'll find each of the statistical test tools listed in this dialog box. They all work pretty much the same way: select a tool to open the tool's dialog box, select input cell ranges, specify your hypothesis, and select your output options.

I'll show an example using the t-Test Two-Sample Assuming Unequal Variances tool. [Figure 5-16](#) shows a portion of a spreadsheet containing measured weights of North American bullfrogs sampled from two southern states.

Each state's sample contains 60 data points. An example hypothesis to test here (*a null hypothesis*) is that there's no difference between the mean weights of bullfrogs from Louisiana and Alabama.

To conduct this test, select t-Test: Two-Sample Assuming Unequal Variances from the Data Analysis dialog box discussed earlier to open the t-Test dialog box shown in [Figure 5-17](#).

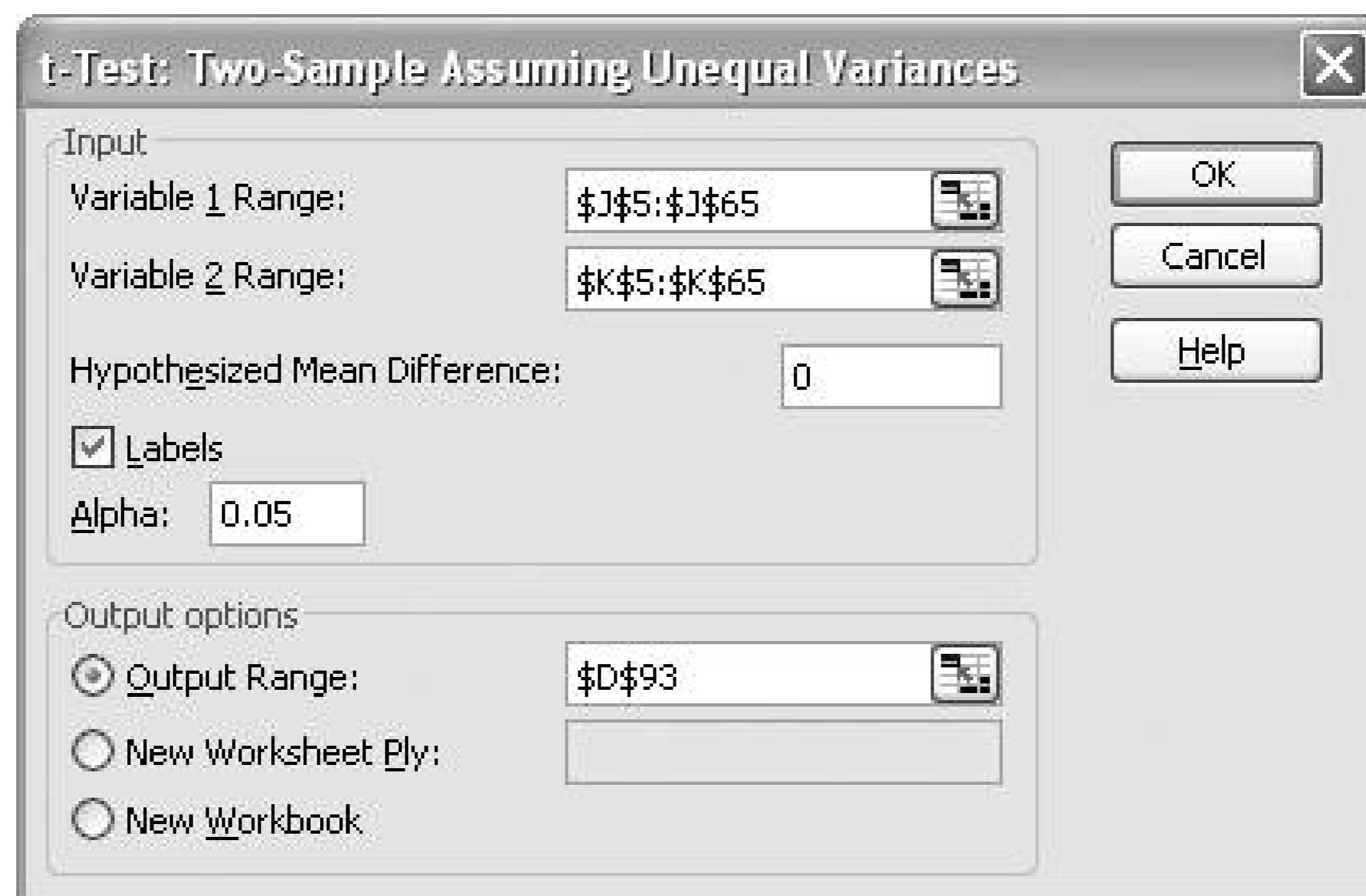
Figure 5-16. Bullfrog weight data



The screenshot shows a Microsoft Excel window titled "Data Analysis Examples.xls". The spreadsheet contains a table of bullfrog weights. A t-test dialog box is open, centered over the data. The dialog box has a title bar "t-Test" and a "Data" tab selected. The "Input Range" is set to \$J\$4:\$K\$18, and the "Output Range" is set to \$L\$4:\$M\$18. The "Labels in the first row" checkbox is checked. The "Variable names" list contains "Louisiana" and "Alabama". The "Variable labels" list contains "Louisiana" and "Alabama". The "Data labels" list contains "Louisiana" and "Alabama". The "Data in groups" list contains "Louisiana" and "Alabama". The "Data in groups" list is set to "Columns in the input range". The "Output options" section has "Output Range" checked and "Output to new worksheet" unchecked. The "Output range" is \$L\$4:\$M\$18. The "Output options" section also has "Output to new worksheet" unchecked. The "Output range" is \$L\$4:\$M\$18. The "Output options" section also has "Output to new worksheet" unchecked. The "Output range" is \$L\$4:\$M\$18.

	I	J	K	L	M	N
3						
4		<b>Weight (g) of American Adult Male Bullfrogs</b>				
5		<b>Louisiana</b>	<b>Alabama</b>			
6		535.2	505.1			
7		520.1	503.9			
8		537.4	497.0			
9		535.6	504.6			
10		515.2	495.2			
11		520.4	497.5			
12		507.4	505.5			
13		510.7	496.1			
14		532.3	502.4			
15		533.4	506.8			
16		509.3	503.5			
17		506.7	501.9			
18		538.5	504.2			

Figure 5-17. t-Test dialog box



In the Variable 1 Range and Variable 2 Range fields, type (or select from the spreadsheet) the cell ranges containing the dataset corresponding to Louisiana and Alabama, respectively. In this example I included the column labels in these input ranges, so I checked the Labels checkbox.

Alpha represents the level of significance related to the probability of encountering a *type I error*. 0.05 represents a 5% significant level. 0.01 represents a 1% significant level, and so on. See any standard text on statistics for more information on this value and hypothesis testing.

The hypothesis is that there is no difference between the means; therefore, enter a value of 0 in the Hypothesized Mean Difference field. In this example, I left the Alpha field (confidence field) at the default 5%; but you can change that as well. Finally, choose your output options. In this case, I chose to have the output on the same spreadsheet, starting at cell D93. Press OK; the results are shown in [Figure 5-18](#).

Figure 5-18. t-Test results

The screenshot shows a Microsoft Excel window titled "Data Analysis Examples.xls". The spreadsheet displays the results of a t-Test: Two-Sample Assuming Unequal Variances. The data is organized as follows:

		Louisiana	Alabama
Mean		501.527205	494.510846
Variance		595.416698	74.572942
Observations		60	60
Hypothesized Mean Difference		0	
df		74	
t Stat		2.09968131	
P(T<=t) one-tail		0.01958324	
t Critical one-tail		1.66570689	
P(T<=t) two-tail		0.03916648	
t Critical two-tail		1.99254347	

These results include computed means and variances for the two input datasets, along with other information such as the number of observations, the hypothesized mean difference, and the degrees of freedom. The t-statistic is also reported, along with both the one-tailed and two-tailed probabilities and critical values. In this case, the t-statistic is 2.1, which is greater than the two-tailed t-critical value, thus indicating there is a statistically significant difference in means between the two datasets. You can also see that the P-value is less than the significance level (Alpha value of 5% in this case), which also indicates rejection of the null hypothesis.

The other statistical test tools available in the Analysis ToolPak generate results that look very much like the ones shown in [Figure 5-18](#).

## See Also

To learn more about the Analysis ToolPak statistical tests, read the Excel help topic entitled "About statistical analysis tools." Refer to any standard text that covers statistical tests, or tests of

significance, to learn more about these and other statistical tests.



## Recipe 5.7. Conducting ANOVA

### Problem

You'd like to conduct an *analysis of variance* (ANOVA).

### Solution

The Analysis ToolPak offers three flavors of ANOVA: single factor, two-factor with replication, and two-factor without replication. Select Tools → Data Analysis from the main menu bar to access these tools.

### Discussion

By way of example, consider the hypothetical drug effectiveness data shown in [Figure 5-19](#).

The data values represent some measure of merit, or effectiveness, for drugs A, B, and C, along with a control group. The Anova: Single Factor tool allows you to perform an analysis of variance for two or more groups, so we'll use that tool to test the hypothesis that there's no difference between the means of these groups.

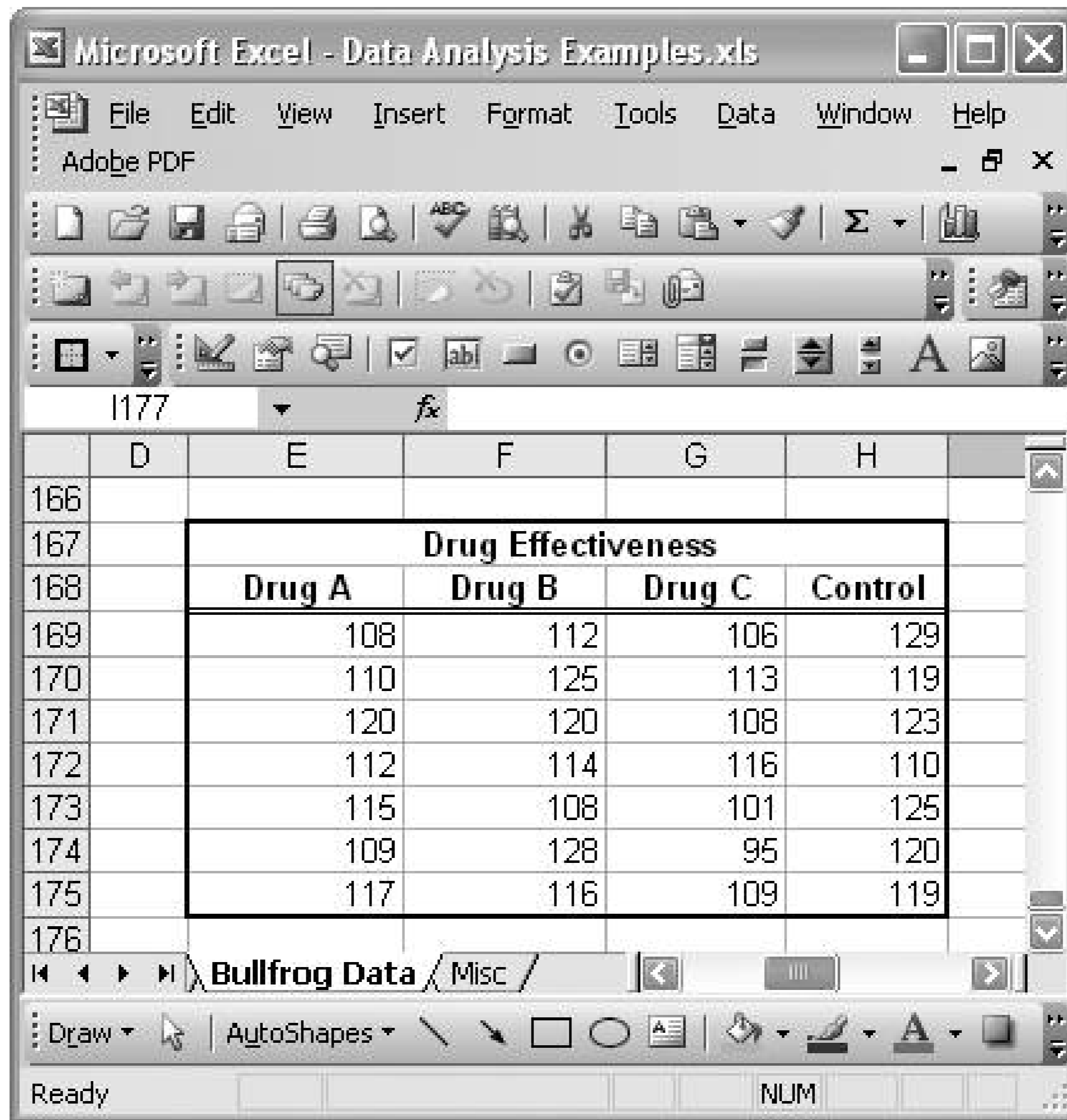
After selecting the Anova: Single Factor tool from the Data Analysis dialog box, you should see the Anova: Single Factor dialog box like the one shown in [Figure 5-20](#).

In the Input Range field, type or select the cell range containing all the data under consideration. Your dataset must be in contiguous columns for this tool. If you include the column labels as I did in this example, then be sure to check the "Labels in first row" option. You can set the Alpha parameter to whatever you'd like. In this case I left it at 5%.

Finally, choose your output options and press OK. Here, I chose to generate the output in the same worksheet, starting in cell E178. The results of this analysis are shown in [Figure 5-21](#).

These results show descriptive statistics for each group, along with standard ANOVA information. ANOVA results are displayed for *between groups* and *within groups*. **SS** represents the sum of squared deviations from the mean. **df** represents degrees of freedom. **MS** is the mean square value. **F** is the F-Ratio. And finally, **F crit** is the critical F-value based on the F-distribution. In this example, the **P-value** is less than the supplied Alpha value, and the obtained F-value is greater than the critical F-value. These results imply that we should reject the null hypothesis that there's no significant difference between the means of these groups and conclude that there's some statistically significant difference.

Figure 5-19. Drug effectiveness



The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Data Analysis Examples.xls". The spreadsheet contains a table titled "Drug Effectiveness" with the following data:

	D	E	F	G	H
166					
167		<b>Drug Effectiveness</b>			
168		<b>Drug A</b>	<b>Drug B</b>	<b>Drug C</b>	<b>Control</b>
169		108	112	106	129
170		110	125	113	119
171		120	120	108	123
172		112	114	116	110
173		115	108	101	125
174		109	128	95	120
175		117	116	109	119
176					

Figure 5-20. Anova: Single Factor dialog box

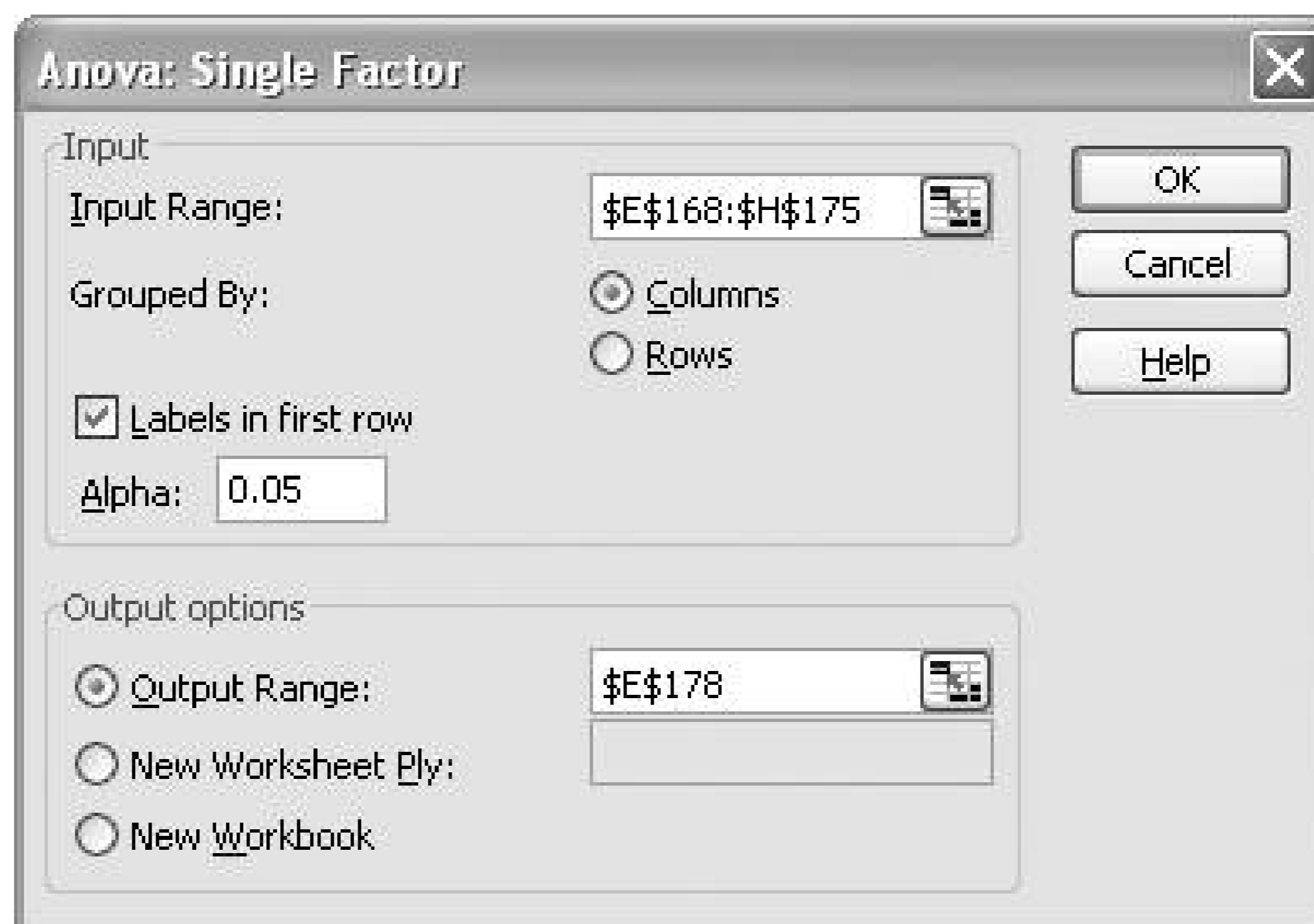


Figure 5-21. ANOVA results

See Also

To learn more about the Analysis ToolPak ANOVA tools, read the Excel help topic "About statistical analysis tools." Refer to any standard text that covers statistical tests, or tests of significance, to learn more about ANOVA and other statistical tests.





## Recipe 5.8. Generating Random Numbers

### Problem

You'd like to compute random numbers between specific upper and lower bounds and you're not sure how to do so in Excel.

### Solution

Use Excel's built-in spreadsheet function `RAND` or VBA's `Rnd` function.

### Discussion

`RAND` generates an evenly distributed random number between 0 and 1. Its syntax is `=RAND( )` with no arguments. When this function is used in a spreadsheet, its value changes every time the spreadsheet is recalculated. For a random number that won't be recalculated, enter `=RAND( )` in the formula bar and then press F9. This will enter a random number between 0 and 1 in the selected cell instead of entering the formula `=RAND( )`. Therefore, the random value will remain unchanged when the spreadsheet is recalculated.

To compute a random number between two specific values, use the formula `=RAND( )*(MaxVal-MinVal)+MinVal`, where `MinVal` and `MaxVal` are the two values between which you want the random number to be generated. The result will be a real number. To generate a random integer, use `=ROUND(RAND( )*(MaxVal-MinVal)+MinVal, 0)`.

You can use the VBA function `Rnd` in a similar manner. For example, the code `x = (MaxX-MinX+1)*Rnd+MinX` generates a random number between `MinX` and `MaxX`. In VBA, you should seed the random number generator by making a call to the `Randomize` function prior to using the `Rnd` function.

### See Also

See [Recipe 13.4](#) for an example that uses random numbers.

Generating random numbers can also be useful when selecting random samples from some population. [Recipe 13.7](#) shows how to select random samples from a population where the probability of selecting any given sample is not uniform.

In cases where you need to draw samples from a finite population of discrete values, you should consider the Sampling tool. This tool is discussed in [Recipe 5.9](#).



← PREV

## Recipe 5.9. Sampling Data

### Problem

You need to draw random samples from a finite population of discrete values.

### Solution

Use the Analysis ToolPak Sampling tool. Select Tools → Data Analysis from the main menu bar and then select Sampling from the list of available tools.

### Discussion

[Figure 5-22](#) shows a spreadsheet containing a set of discrete values from which three were selected at random. The Sampling tool discussed a moment ago was used to draw these three random samples.

Upon selecting the Sampling tool from the Data Analysis tool list, you should see the Sampling dialog box shown in [Figure 5-23](#).

Figure 5-22. Sampling discrete values

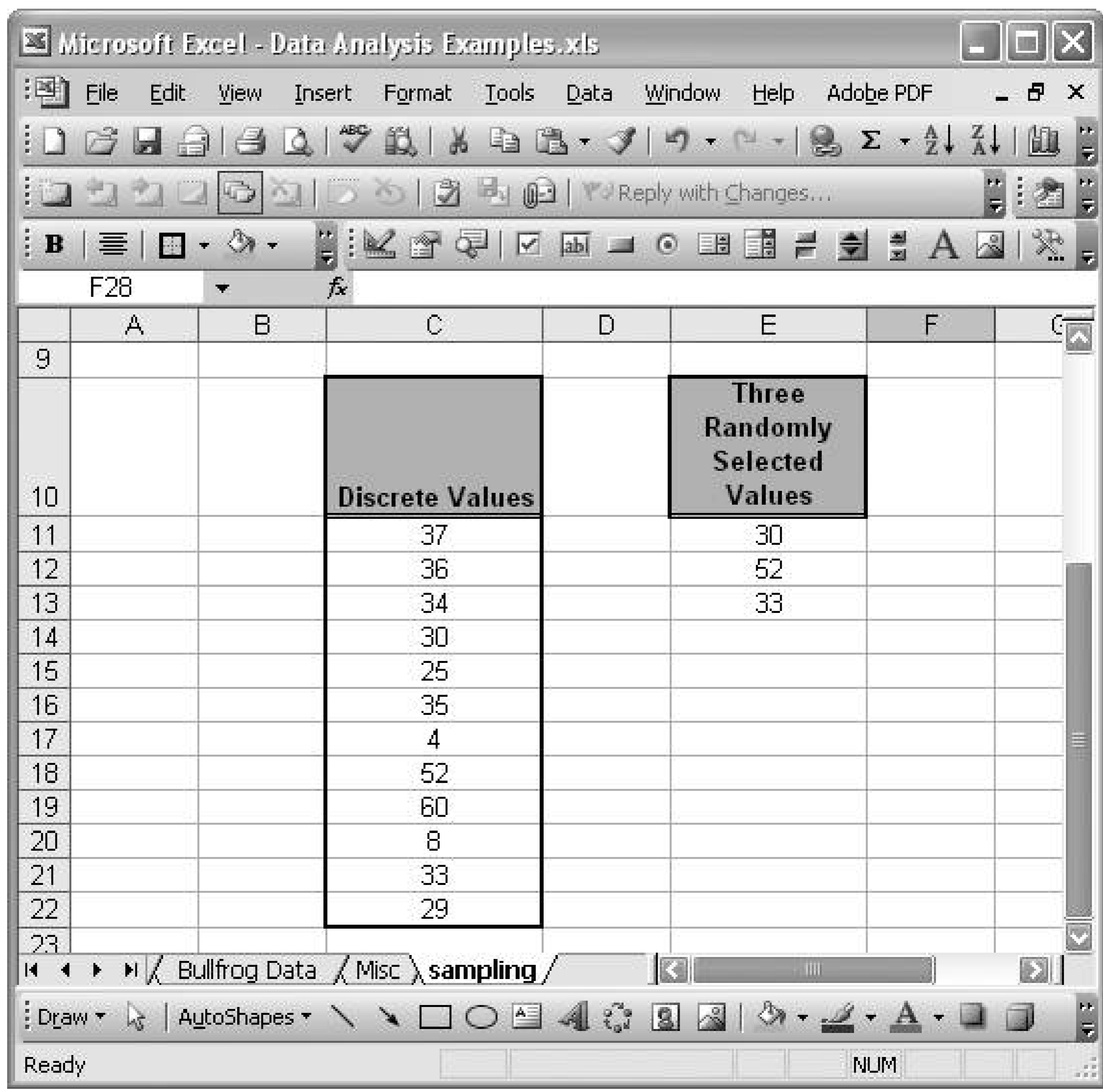
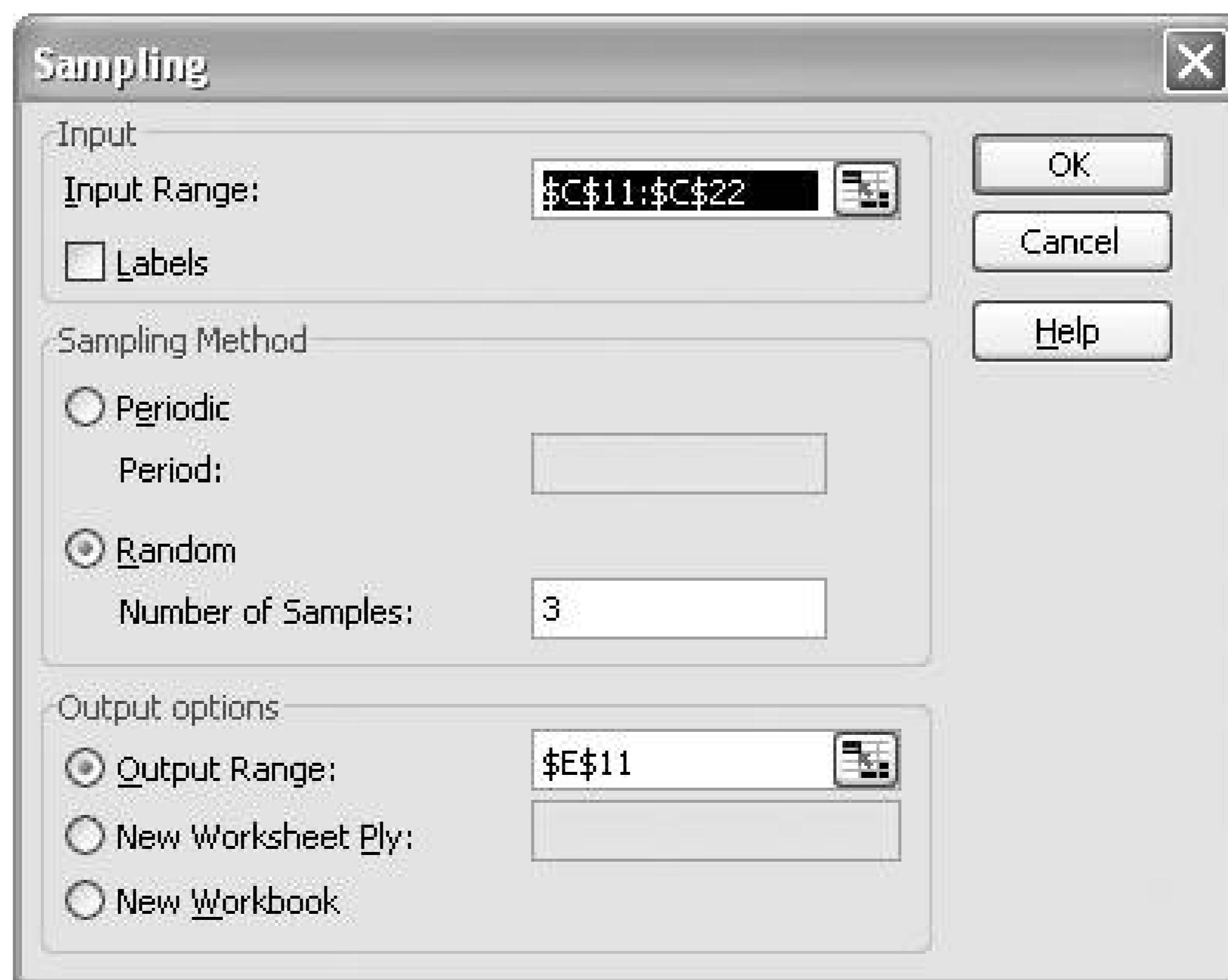


Figure 5-23. Sampling dialog box



In the Input Range, enter or select the cell range containing the discrete values from which to draw random samples. Next, select the Sampling Method, which can be either Periodic or Random. If you choose Periodic you also have to specify a periodic interval where sampling will take place. At every  $k$ th period, a random sample will be drawn. For example, if you choose Periodic with a period of 2 for the example data shown in [Figure 5-22](#), then six samples will be drawn, corresponding to every other value in the input range starting from the second value.

If you choose Random, then you have to specify how many random samples to draw. In this example, I chose to draw three random samples. Finally, select your output options and press OK. In this example I chose to display the results in the same worksheet, starting at cell E11. The results are shown in [Figure 5-22](#).

## See Also

See [Recipe 5.8](#) to learn how to generate random numbers in Excel.

# Chapter 6. Time Series Analysis

[Section 6.0. Introduction](#)

[Recipe 6.1. Plotting Time Series Data](#)

[Recipe 6.2. Adding Trendlines](#)

[Recipe 6.3. Computing Moving Averages](#)

[Recipe 6.4. Smoothing Data Using Weighted Averages](#)

[Recipe 6.5. Centering Data](#)

[Recipe 6.6. Detrending a Time Series](#)

[Recipe 6.7. Estimating Seasonal Indices](#)

[Recipe 6.8. Deseasonalization of a Time Series](#)

[Recipe 6.9. Forecasting](#)

[Recipe 6.10. Applying Discrete Fourier Transforms](#)

## 6.0. Introduction

Time series analysis is a broad field of study with applications ranging from engineering analysis to socioeconomic forecasting to stock market prediction, among many other applications. You can use Excel to perform many tasks required in time series analysis, such as pre- and post-processing of data, visualizing results, and making forecasts. Excel also offers a Fourier Analysis tool that allows you to transform time series data to the frequency domain (and back) for further analysis. This chapter aims to show you some specific examples of how you can effectively use Excel when analyzing time series data.

There are many time series analysis techniques available, many of which are tailored to specific kinds of time series, so I'm not going to attempt to cover them all. Instead, the examples discussed here serve to show you how you can leverage Excel's standard functionality to perform some important analysis tasks. Building on the techniques I show you here, you have the ability and power within Excel to devise all sorts of other analysis techniques.

[← PREV](#)

## Recipe 6.1. Plotting Time Series Data

### Problem

You need to plot a set of time series data for further analysis.

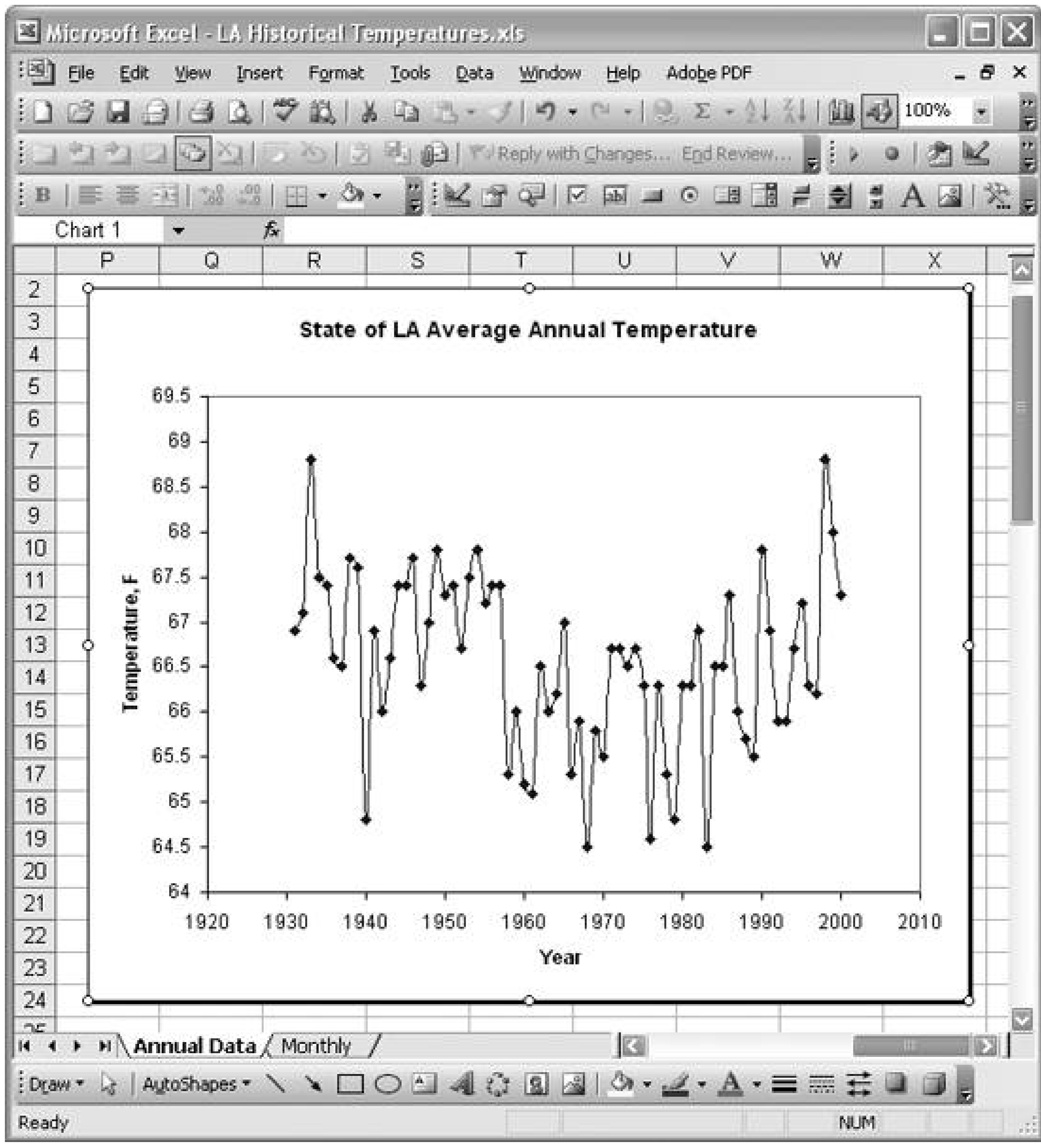
### Solution

Time series charts, such as the one shown in [Figure 6-1](#), are easy to create in Excel.

Select the data series (both  $x$  and  $y$  values) you want to chart and then press the Chart Wizard button on the Standard toolbar, or select Insert → Chart... from the main menu bar to open the Chart Wizard. The Chart Wizard allows you to select the chart style and customize charts in various ways. Read [Chapter 4](#) to learn more about creating charts in Excel.

Figure 6-1. Time series chart







## Recipe 6.2. Adding Trendlines

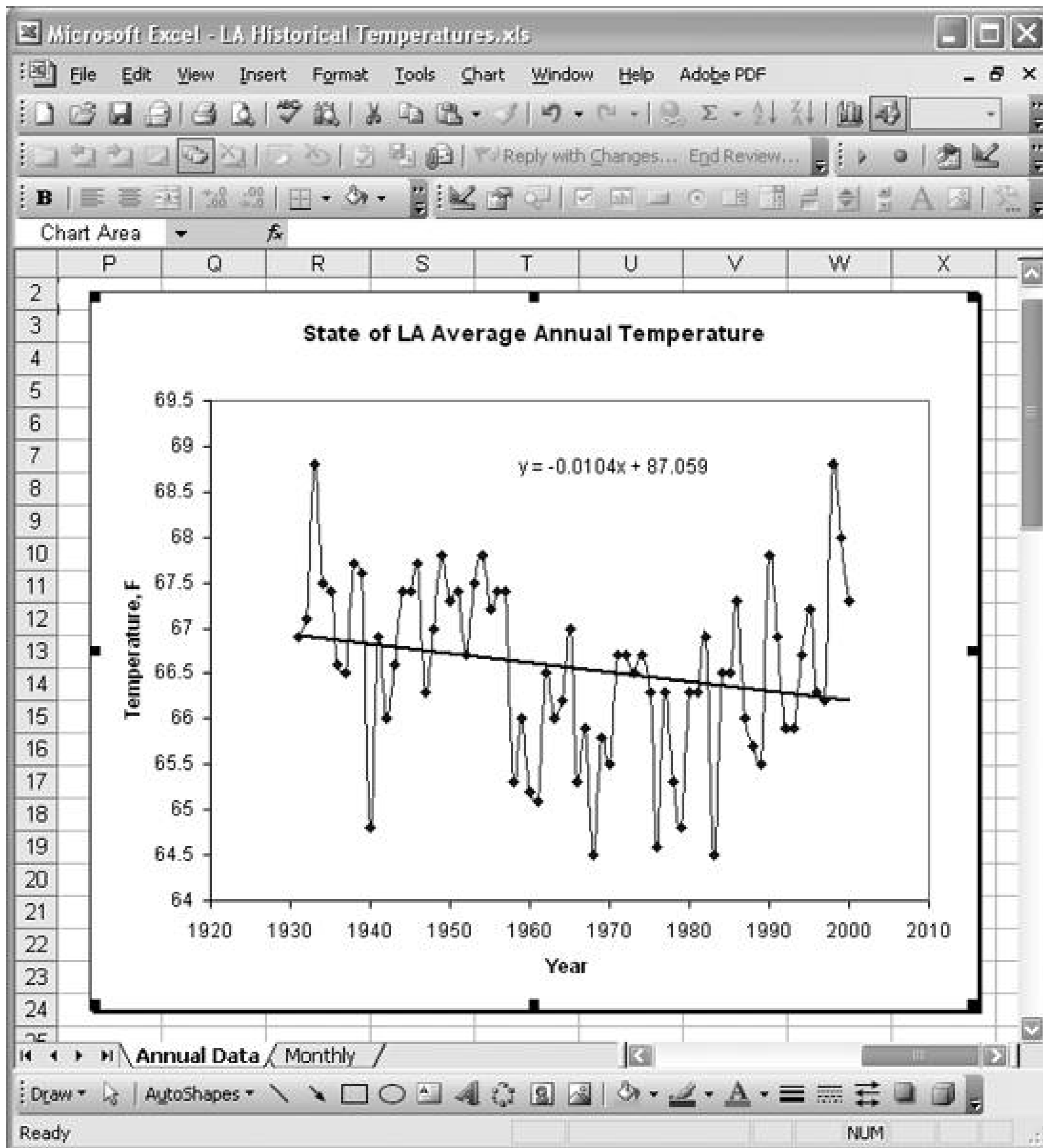
### Problem

You've plotted your time series using Excel's charting feature and now you'd like to add a trendline.

### Solution

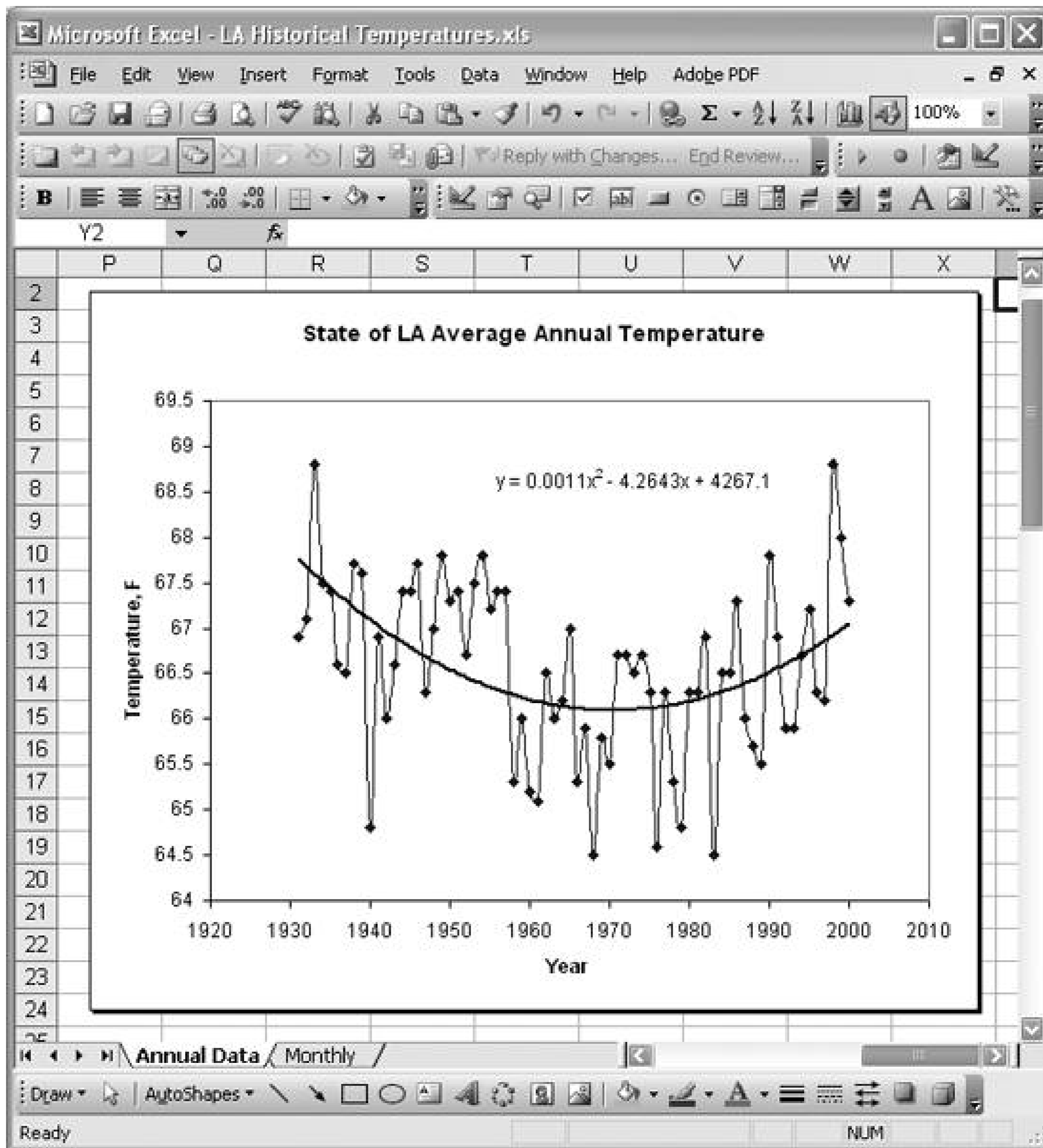
[Figure 6-2](#) shows an example linear trendline you can create on Excel charts. Notice that you also have the option of displaying the trendline equation on the chart, as shown in the figure.

Figure 6-2. Linear trendline



Excel also supports nonlinear trendlines. [Figure 6-3](#) shows the same time series with a second-order polynomial trendline instead of a linear trendline.

Figure 6-3. Polynomial trendline



To create a trendline for a given data series, select the series by clicking on it and then right-click to reveal a pop-up menu. Select Add Trendline from the pop-up menu to open the Add Trendline dialog box, where you can select the type of trendline and other options. Refer to [Chapter 8](#) to learn more about adding these and other types of trendlines to Excel charts.

[← PREV](#)

## Recipe 6.3. Computing Moving Averages

### Problem

You'd like to smooth a set of time series data by computing moving averages.

### Solution

You can compute moving averages in several ways in Excel. You can use the built-in Moving Average trendline charting feature, the Analysis ToolPak Moving Average feature, or spreadsheet functions to compute moving averages.

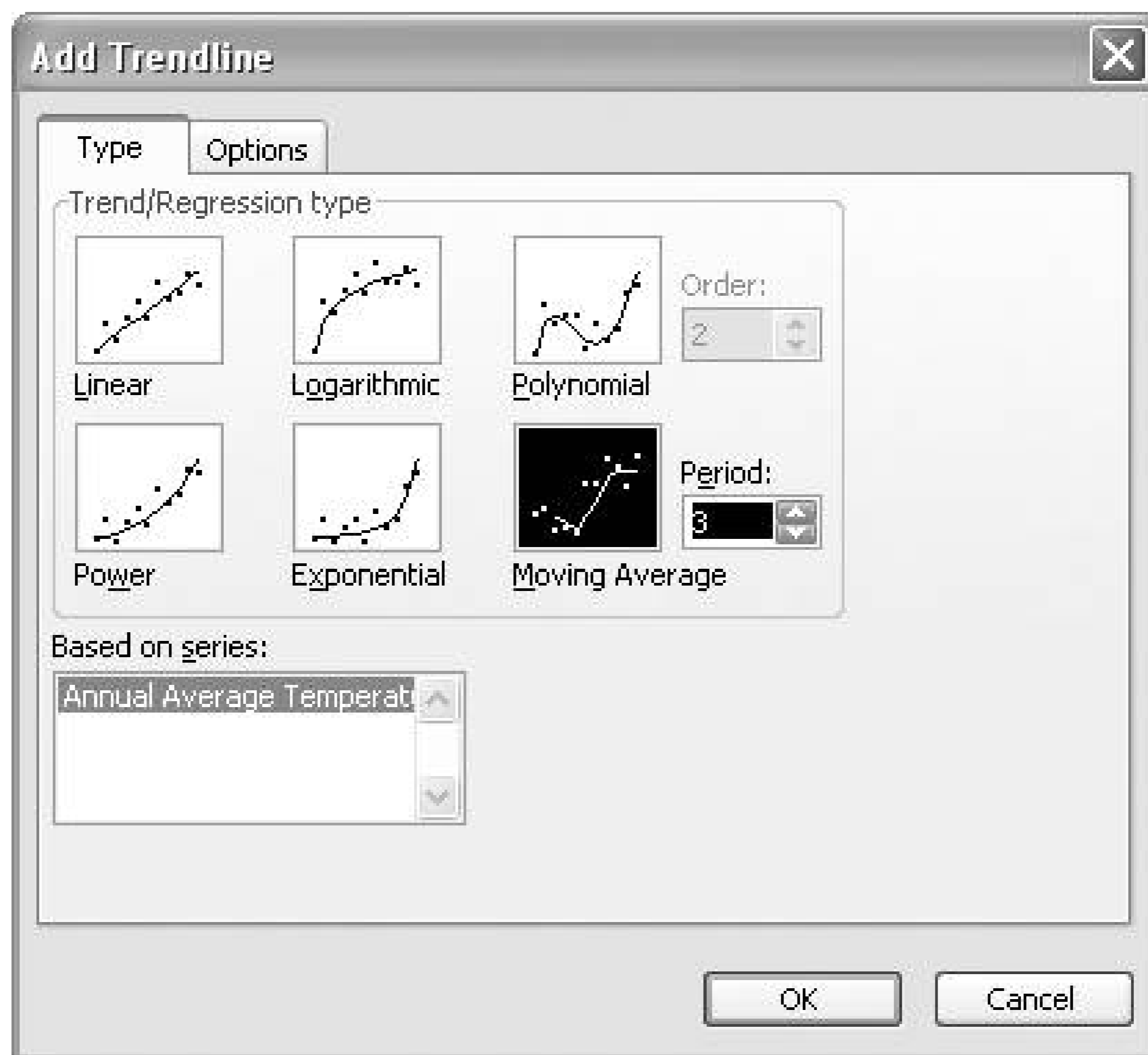
### Discussion

Excel allows you to compute moving averages in several ways. I'll show you three ways: using chart trendlines, using the Analysis ToolPak, and using spreadsheet formulas.

#### Moving Average trendline

As mentioned in [Recipe 6.2](#), you can add a trendline to a chart by selecting the data series and then right-clicking to reveal a pop-up menu. You can then select Add Trendline to open the Add Trendline dialog box shown in [Figure 6-4](#).

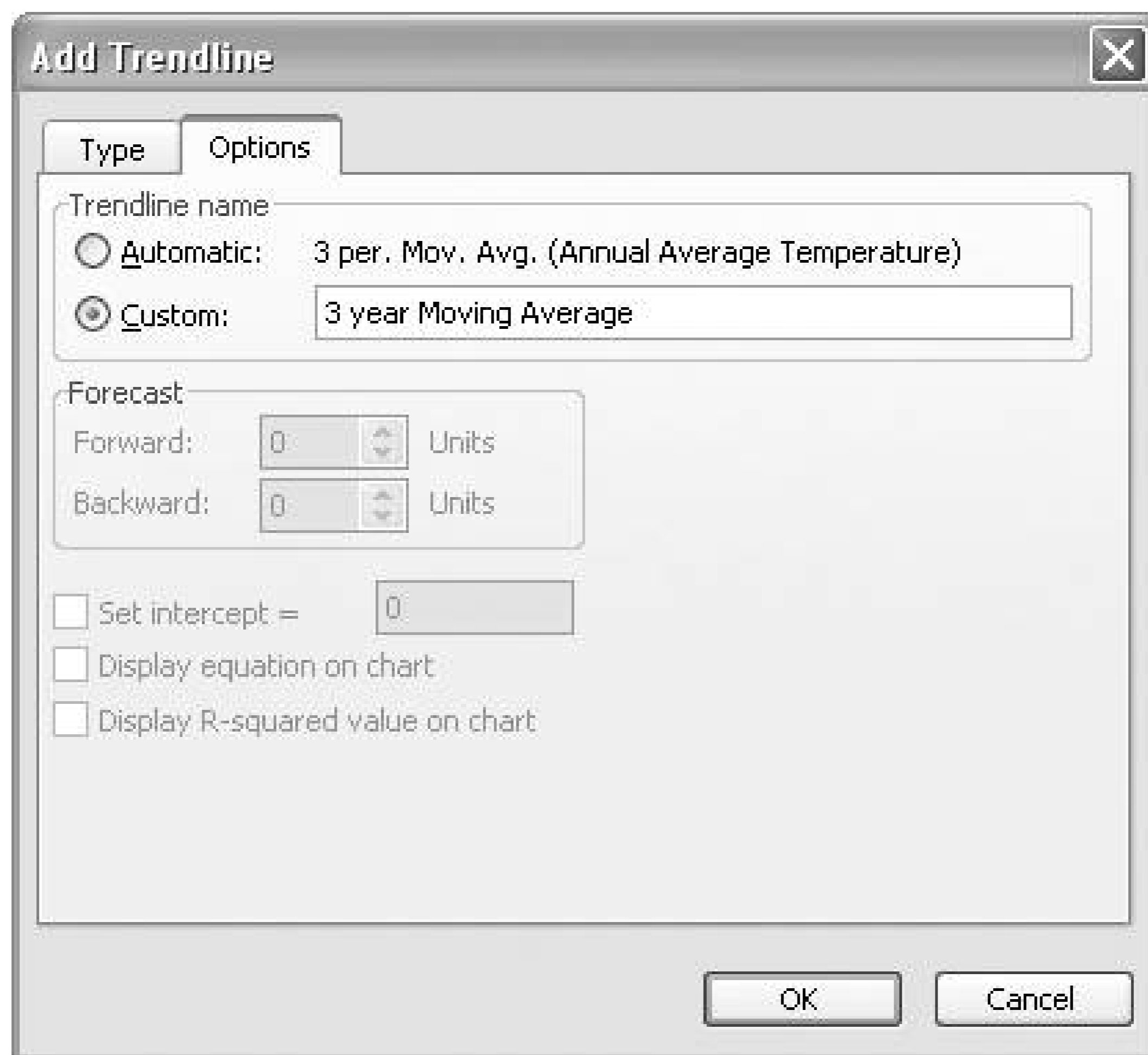
Figure 6-4. Add Trendline dialog box



You can select from six different types of trendlines. In this case, select the Moving Average type. To the right of the Moving Average icon, you should see a Period spinner control (as shown in the figure). Use the spinner arrows to select the period over which averages will be computed. You can also type in a number. In this example, I entered 3 so as to create a three-year moving average trendline.

On the Options tab you can type in a name for the new trendline. [Figure 6-5](#) shows the Options tab.

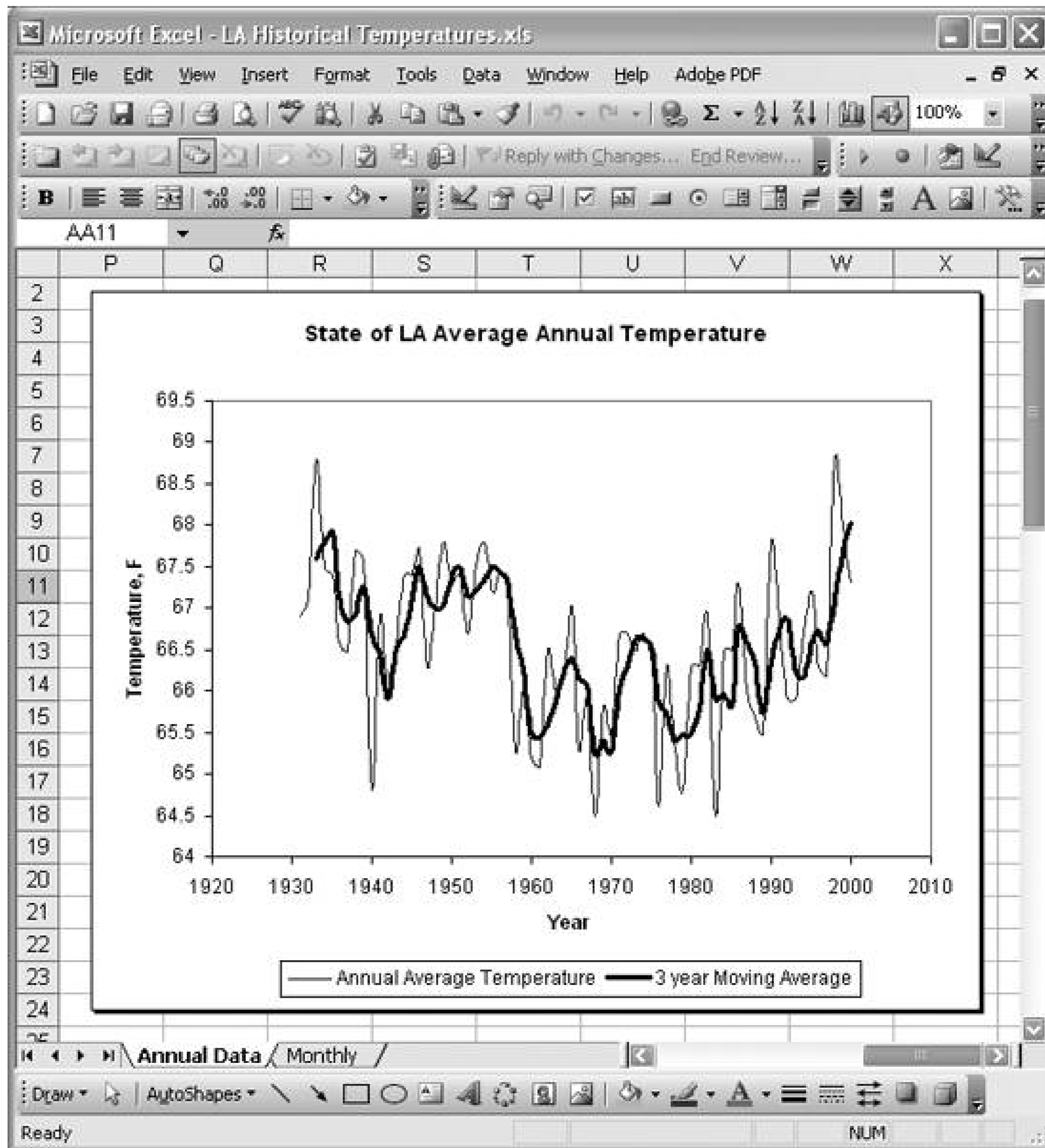
Figure 6-5. Add Trendline: Options tab



Moving Average trendlines have available only the "Trendline name" options. The other options apply to the other trendline types shown in [Figure 6-4](#). (See [Chapter 8](#) for examples using the other trendline types.)

Pressing OK closes the Add Trendline dialog box and adds the new trendline to your chart. [Figure 6-6](#) shows the time series from [Figure 6-2](#) and [Figure 6-3](#) with a three-year moving average trendline.

Figure 6-6. Three-year moving average trendline



If you decide to modify the resulting moving average trendline, perhaps changing the averaging period, simply select the trendline on the chart and then right-click to reveal a pop-up menu. Next, select Format Trendline from the pop-up menu to open the Format Trendline dialog box.

You can also open the Formation Trendline dialog box by selecting the trendline and pressing Ctrl-1.

The Format Trendline dialog box includes the same the Add Trendline dialog box, where you can change the trendline type and any of its options. Further, the Format Trendline dialog box has an



additional tab called Patterns that allows you to format the actual line drawn for the trendline. For example, you can change the line weight, color, and style.

Chart trendlines offer an easy way of creating moving average trendlines for time series data. However, this method does not allow you to extract the actual numerical values representing the moving average trendline. Also, this method requires that you first create a chart of your data. If you need numerical values for the trendline or you don't want to create a chart, then you can use one of the two following approaches.

## Analysis ToolPak Moving Average

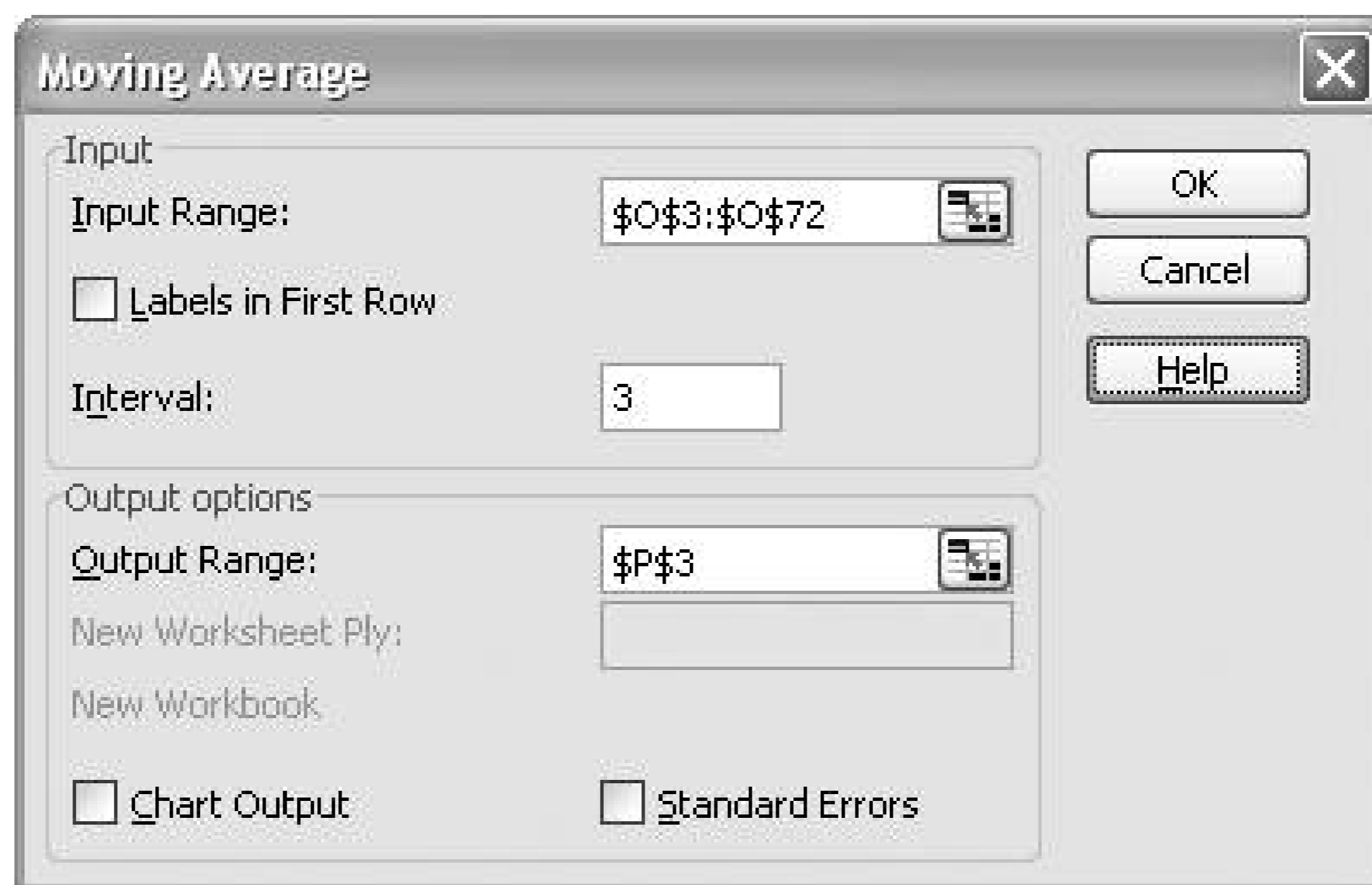
The Analysis ToolPak includes a Moving Average feature that allows you to compute moving averages in a manner similar to that of the Excel chart Moving Average trendline. The benefit of using the Analysis ToolPak is that you get the actual numerical data for the moving average series and not just a chart of the data.

To compute moving averages for a data series, select Tools Data Analysis... from the main menu bar to open the Data Analysis dialog box. [Figure 6-7](#) shows the Data Analysis dialog box, where you're presented with a list of analysis tools to choose from.

Figure 6-7. Data Analysis dialog box

Select Moving Average from this list and press OK to open the Moving Average dialog box shown in [Figure 6-8](#).

Figure 6-8. Moving Average dialog box



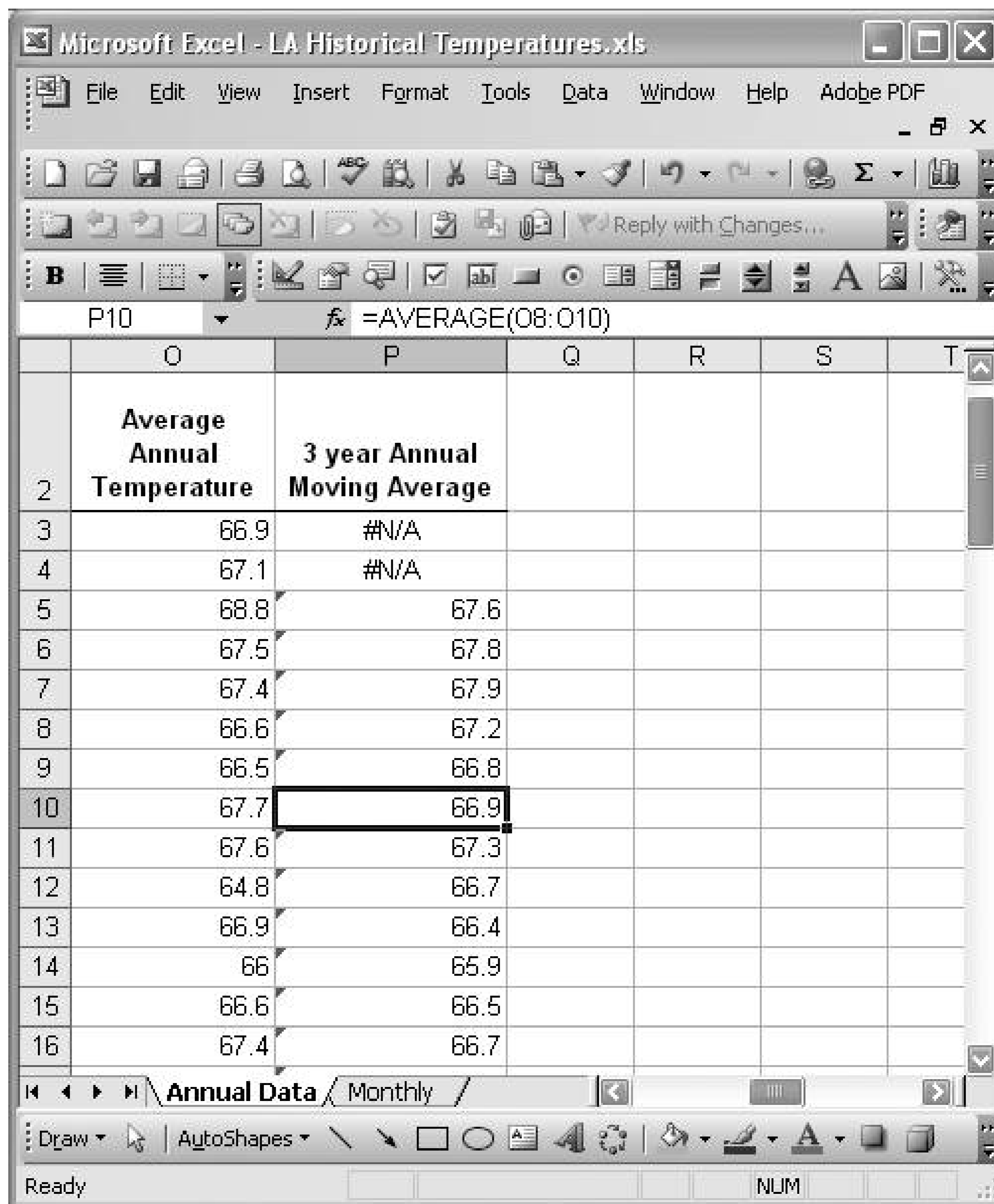
In the Input Range field, enter (or select directly on your spreadsheet) the cell range containing the values of the data series for which you'd like to compute moving averages. For my example, cells O3 to O72 contained the relevant data.

Next, enter the Interval over which you'd like to compute averages. This is the same as the Period field used in the Trendline dialog box, as discussed earlier.

Finally, enter the topmost cell in the range where you'd like the output created. In this case I selected cell P3, to place the moving average data adjacent to the original time series data.

You may also elect to have a chart created automatically or have standard errors computed. Press OK to close the Moving Average dialog box and create the data. Your spreadsheet will now contain the new moving average data. [Figure 6-9](#) shows a portion of the average annual temperature data series shown in [Figure 6-6](#), along with the computed moving averages.

Figure 6-9. Moving Average results



Notice that the first two entries in the moving average series are #N/A. This is because the Moving Average tool computes moving averages based on the previous data points, and there are not enough points to compute the three-year average at the beginning of the series.

In [Figure 6-9](#), I intentionally selected one of the computed moving averages to show you that these results are not static numerical values. Instead, the Moving Average tool entered formulas to compute the moving average. The formula used is the standard **AVERAGE** spreadsheet formula, as shown in the formula bar in [Figure 6-9](#). Using the spreadsheet formula in this way means that if your original data changes, the moving average will update accordingly.

At this point, you can perform other spreadsheet operations on the moving average data or plot the data as you wish. This approach gives you greater flexibility, as opposed to the trendline moving

average method discussed earlier. However, what if you want to compute the moving averages over a period centered about each data point in the original series? You can use spreadsheet formulas to accomplish this, as discussed in the next section.

## Spreadsheet formulas

Aside from the two previous approaches, you can always use simple spreadsheet formulas to compute moving averages. Specifically, you can use the `AVERAGE` formula, as shown in [Figure 6-10](#).

Figure 6-10. Centered moving averages

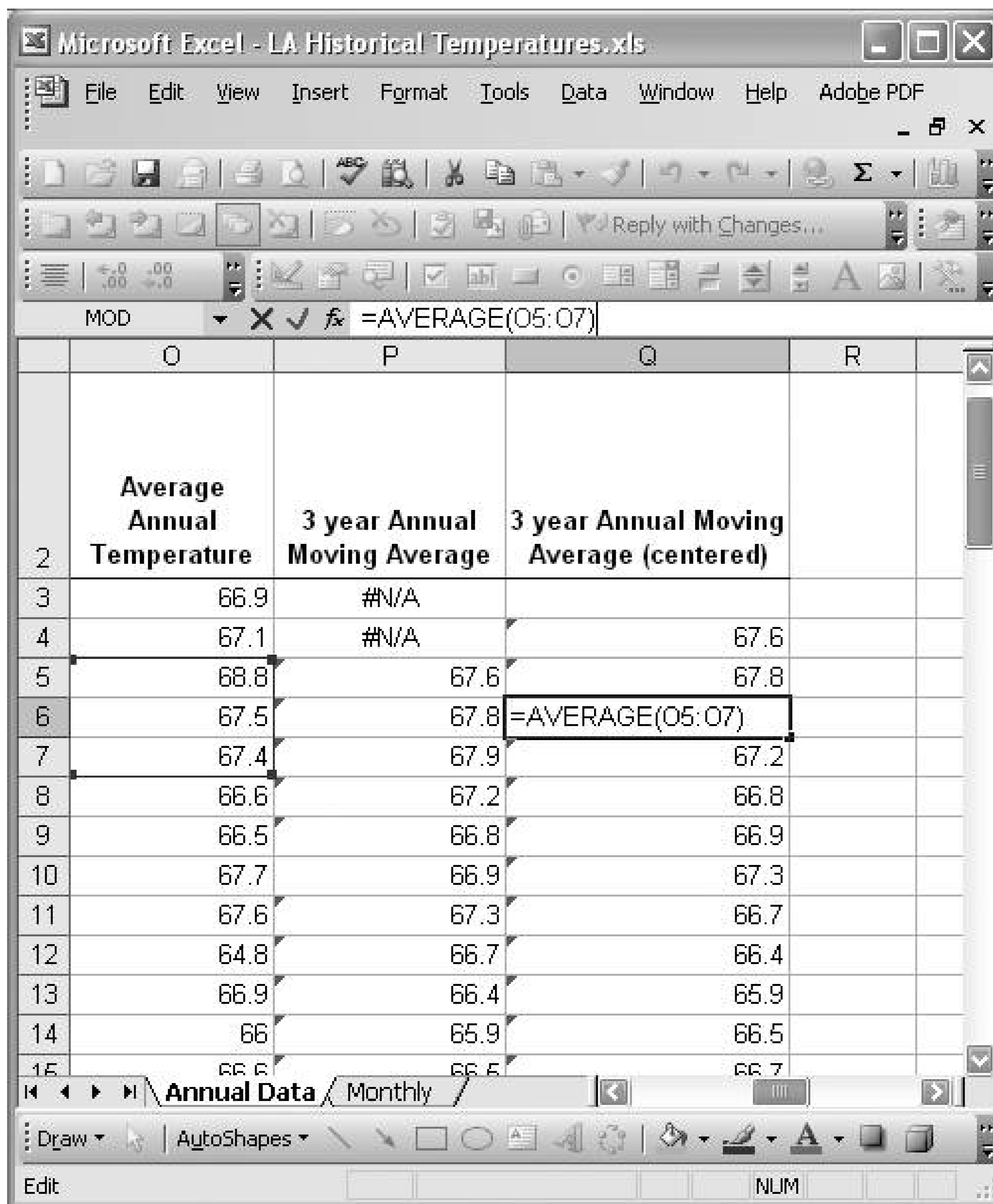


Figure 6-10 shows the same original data and three-year moving average data computed using the Analysis Toolpak from the previous section. Further, Figure 6-10 shows a new column of three-year moving averages computed using the **AVERAGE** formula manually. To generate the new column, I entered the **AVERAGE** formula in Q4 with the cell range O3:O5 passed as the argument. The formula is **=AVERAGE(O3:O5)**. This formula computes the average of the three original series data points centered over the second data point. Once I had that formula entered, I simply copied and pasted the same formula in all the remaining cells in column Q, corresponding to the original data. Using relative cell references meant that Excel automatically adjusted the cell range argument passed into the **AVERAGE** formula. For example, Figure 6-10 shows the resulting formula contained in cell Q6, along with the corresponding cell range argument.

The result of these simple steps yields a new data series containing three-year centered moving

averages. This approach is direct in that you don't need to use charts first if you don't want to, nor do you need to use the Analysis ToolPak. Further, this approach is flexible. For example, you can change the cell range passed into the `AVERAGE` formula to compute averages over any period of your choosing.

## See Also

You can also compute weighted moving averages in Excel. See [Recipe 6.4](#) to learn how.



## Recipe 6.4. Smoothing Data Using Weighted Averages

### Problem

You'd like to smooth data in a time series.

### Solution

Use Excel's exponential smoothing feature in the Analysis ToolPak or construct a smoothed data series yourself using spreadsheet functions and VBA.

### Discussion

Smoothing data is often desirable to remove unwanted noise in a data series. Computing moving averages as discussed in [Recipe 6.3](#) is actually a smoothing process. Besides the moving averages methods discussed earlier, there are other ways to smooth data. Excel offers an exponential smoothing feature as part of the Analysis ToolPak. Further, you can construct any smoothing operations you desire using standard spreadsheet functions (and/or VBA). You already saw this approach in the previous recipe; however, this time I'll show you how to use a weighted moving average that makes use of a cubic spline interpolation kernel.

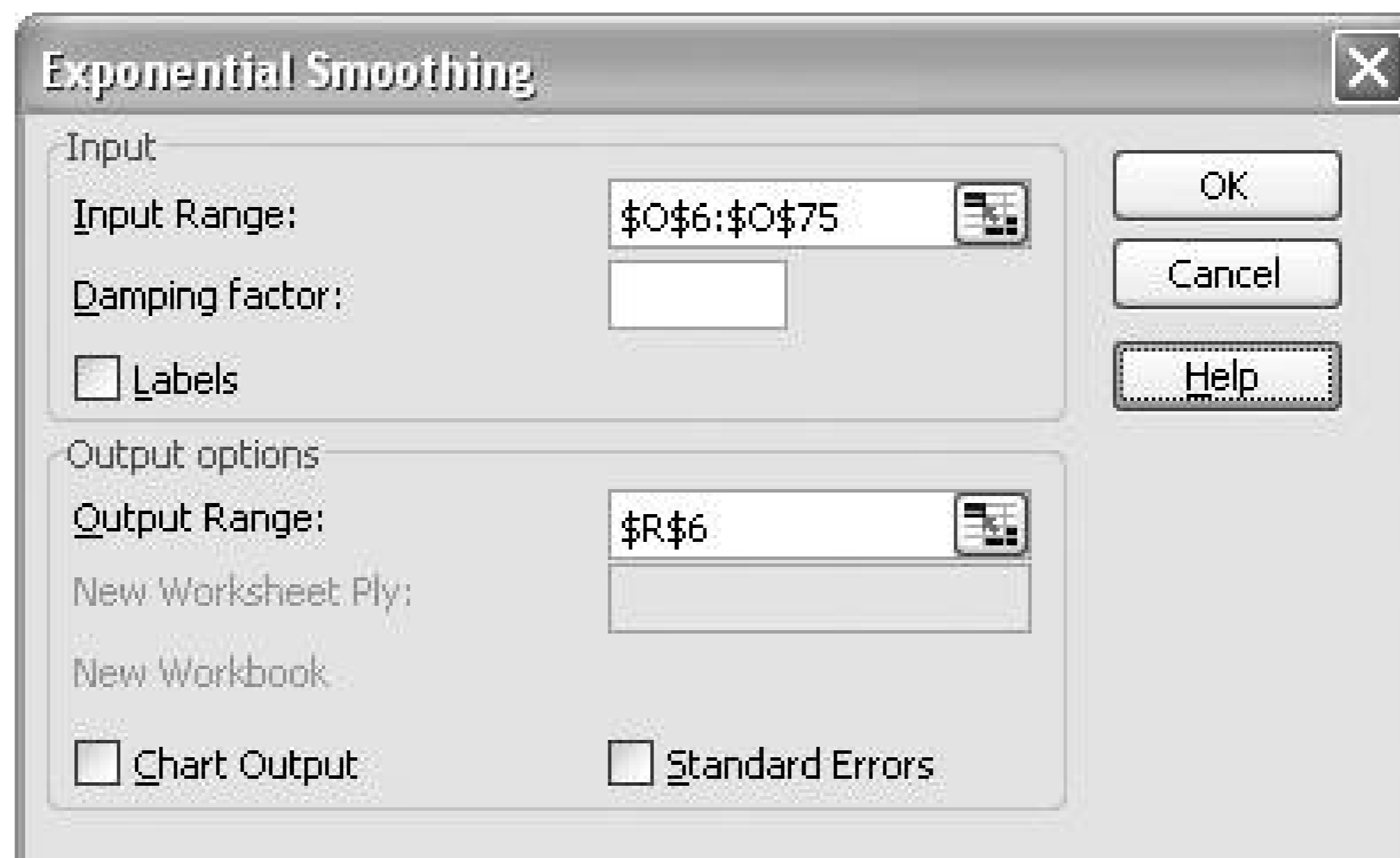
### Exponential smoothing

Exponential smoothing is also a weighted-averaging technique. The idea behind weighted averaging is to give data values closest to the value being forecast or estimated greater importance, or influence, as opposed to values further away. Exponential smoothing uses the following formula:

$F_{n+1}$  is the value being estimated at time interval  $n + 1$ .  $\alpha$  is a weighting factor, which is called a *damping factor* in Excel.  $F_n$  is the previous estimated value, and  $Y_n$  is the previous actual value in the original data series.

To use exponential smoothing, select Tools → Data Analysis... from the main menu bar to open the Data Analysis dialog box (shown earlier in [Figure 6-7](#)). Select Exponential Smoothing from the list of available analyses. Doing so opens the Exponential Smoothing dialog box shown in [Figure 6-11](#).

Figure 6-11. Exponential Smoothing dialog box

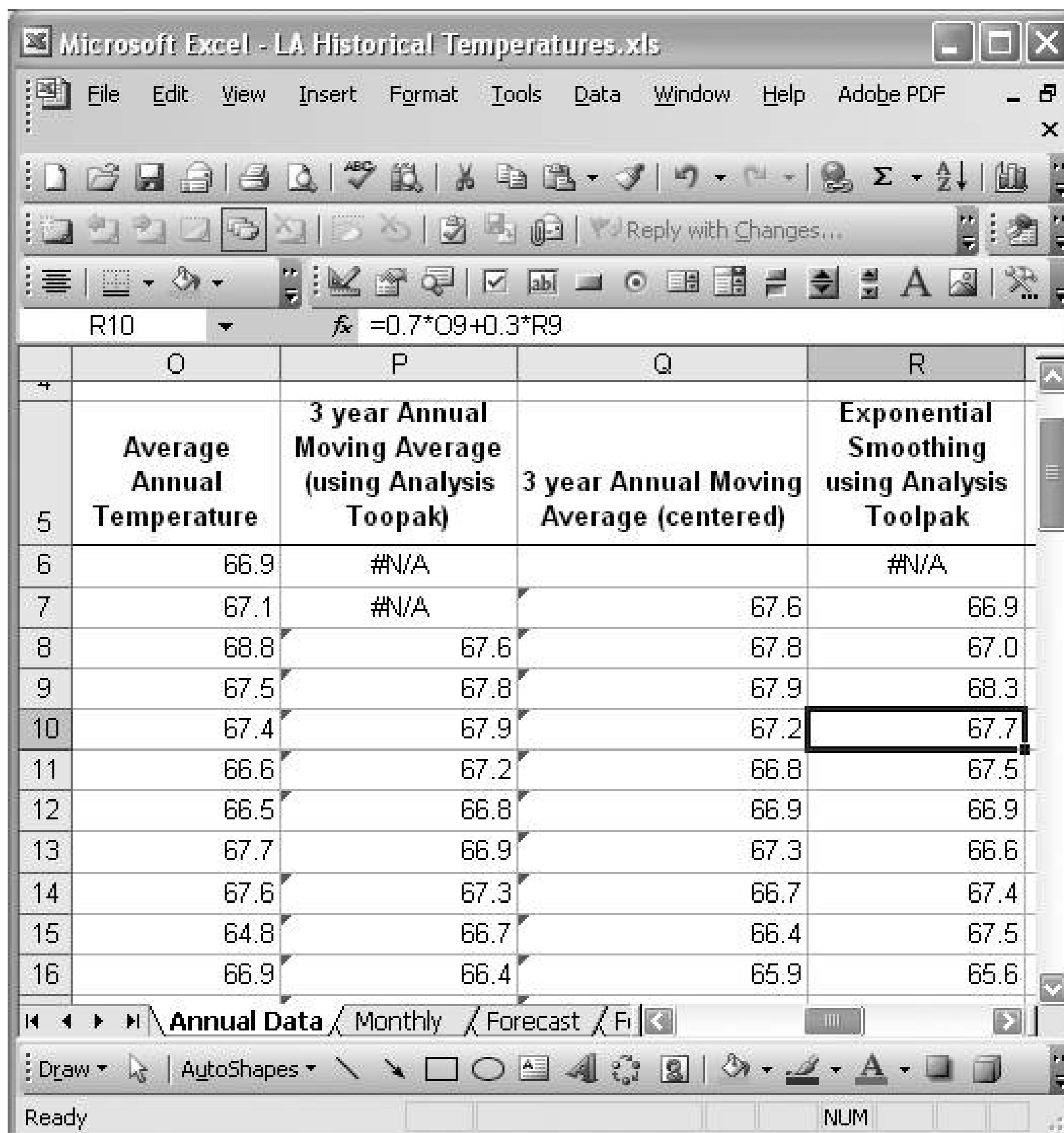


In the Input Range field, type (or select on the spreadsheet) the cell range containing the input data you wish to smooth. Enter a damping factor in the "Damping factor" field or leave it blank to use the default value of 0.3. Enter a reference to the topmost cell in the desired output cell range in the Output Range field.

Press OK when you're done and you should see the results on your spreadsheet. [Figure 6-12](#) shows an example for the same annual temperature data used in the previous recipe. The resulting data series after smoothing is contained in column R.

Figure 6-12. Exponential smoothing results

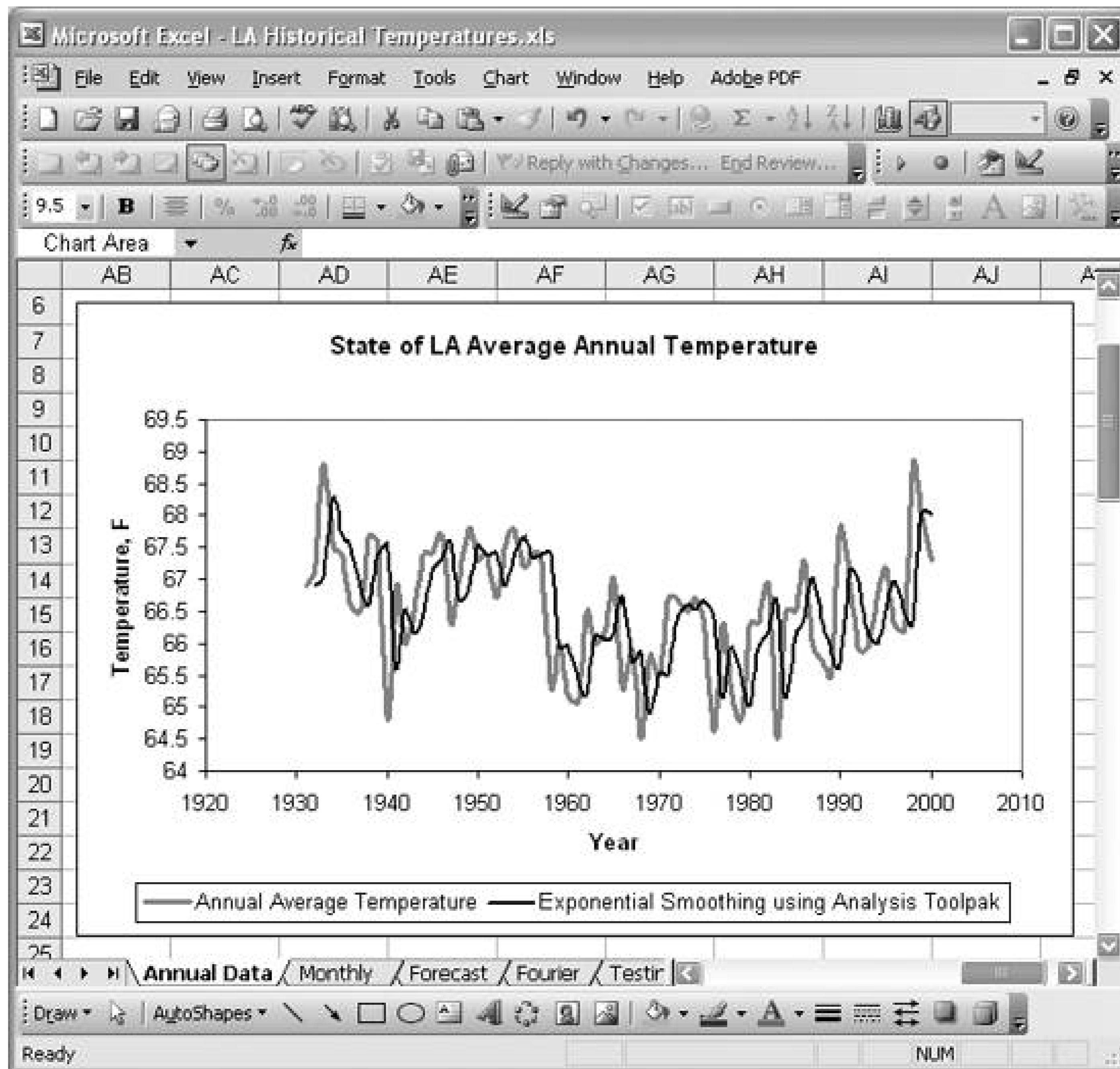




Just like the Moving Average tool discussed earlier, the Exponential Smoothing tool places a cell formula rather than a value in each cell of the resulting data series. This way, if your original data changes, the smoothed data will update automatically. Cell R10 is selected in [Figure 6-12](#) so you can see what the formulas look like. The formula is shown in the formula bar and is of the form  $=0.7*O9+0.3*R9$ . As you can see, this formula corresponds to the exponential smoothing equation I showed you earlier. Of course, you can enter these formulas yourself, bypassing the need to use the Analysis ToolPak, if you desire.

[Figure 6-13](#) shows a plot of the smoothed data series superimposed over the original series so you can see the difference between the two.

Figure 6-13. Exponentially smoothed temperature data



Other weighted averaging techniques are commonly used in time series analysis. In the next section I'll show you one based on a cubic spline interpolation kernel.

## Kernel smoothing

In this example, I'll show you a weighted averaging technique that uses a cubic spline interpolation kernel designed to approximate a Gaussian kernel. You can use all sorts of different kernels for smoothing with only slight modifications to the technique I'll present. The kernel I'll use is:

$r$  represents the distance between data points and  $h$  represents the effective smoothing radius divided by 2. I'll use this smoothing function to compute weights when computing weighted averages of the time series. To make this task easier, I added a VBA function called `wcs` that you call from the spreadsheet. [Example 6-1](#) shows my VBA implementation for this smoothing kernel.

### Example 6-1. Cubic spline kernel

```
Public Function Wcs(r As Double, h As Double) As Double
    Dim s As Double
    Dim w As Double

    s = r / h

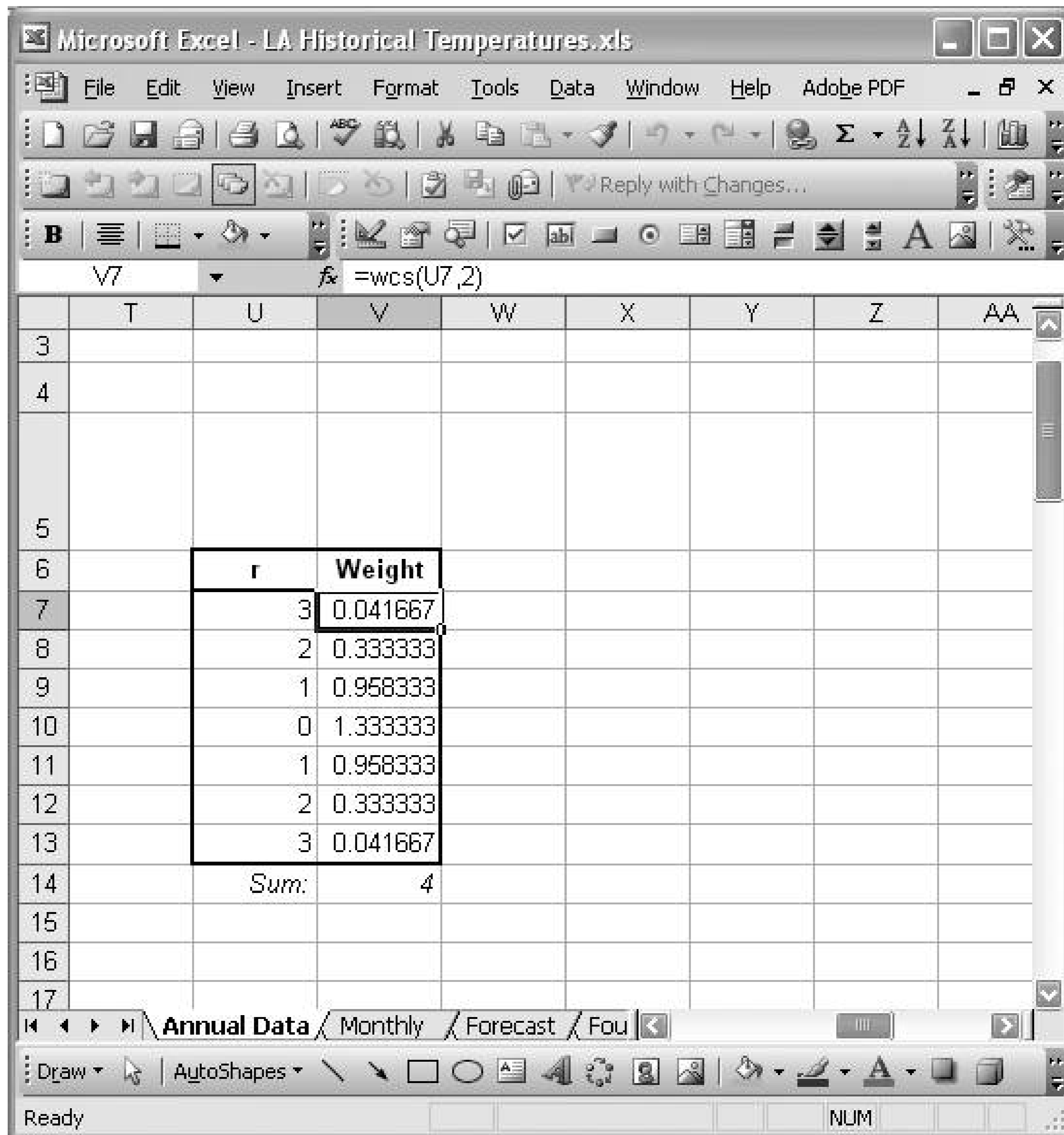
    If s > 2 Then
        w = 0
    ElseIf (s <= 2) And (s > 1) Then
        w = (1# / 3#) * (2# - s) ^ 3
    ElseIf (s <= 1) And (s >= 0) Then
        w = (4# / 3#) * (1 - (3# / 2#) * s ^ 2 + (3# / 4#) * s ^ 3)
    End If

    Wcs = w
End Function
```

This is a fairly straightforward implementation of the equation I showed earlier, so I won't go over each line of code. Refer to [Chapter 2](#) if you're not already up to speed on using VBA and adding custom functions and subroutines.

I'll use this function to generate weights in order to smooth the same temperature data discussed earlier. [Figure 6-14](#) shows a weight table I computed using this function.

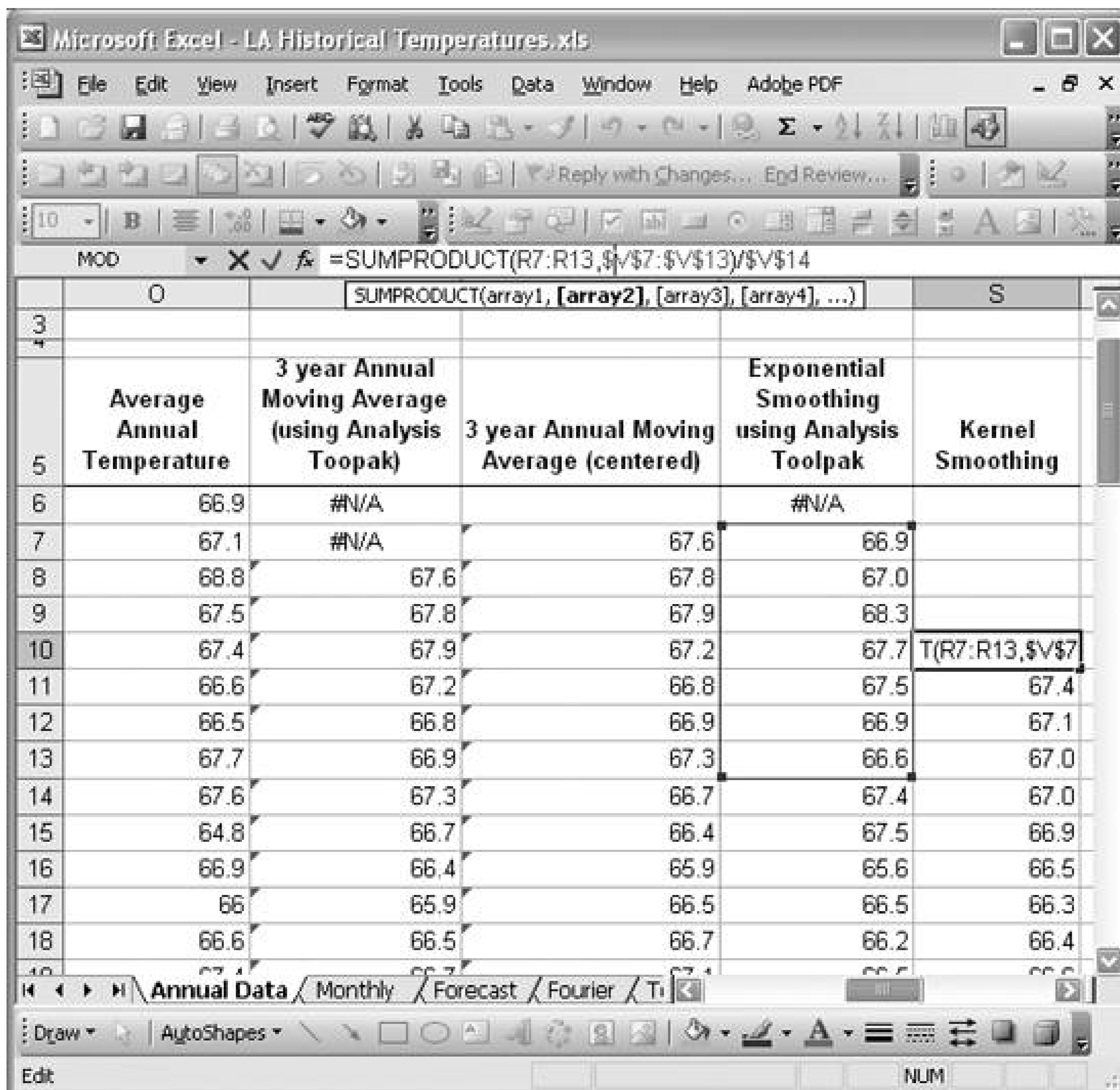
Figure 6-14. Weight table



The first column in the table, labeled  $r$ , contains relative indices around the fourth index, labeled 0. We can do this here to compute  $r$  because the time series data is sampled at evenly spaced intervals. The second column contains formulas like  $=wcs(U10,2)$ . This is the formula contained in cell V10. All the other cells contain similar formulas. The sum of the weights (computed using the  $SUM$  formula shown in cell V14) is required to normalize the weighted average computation. You can avoid this step if your kernel integrates to unity over its range of support.

Now, to compute the smoothed data series, I set up another column in my spreadsheet, as shown in [Figure 6-15](#).

Figure 6-15. Kernel-smoothing results



The smoothed data is contained in column S under the heading **Kernel Smoothing**. I selected the first smoothed data value in cell S10 in the figure so you can see the cell formula. The formula is `=SUMPRODUCT(R7:R13,$V$7:$V$13)/$V$14`. As you can see, it uses the `SUMPRODUCT` formula to compute the sum of the products of each corresponding term in the two given cell ranges (see [Chapter 7](#) for more information on this and other handy functions). In this case, the first cell range corresponds to the six data items around the given data item, including the given data item. This means the weighted average in this case is centered over the *n*th data item and includes the weighted influence of the three previous items and the three following items.

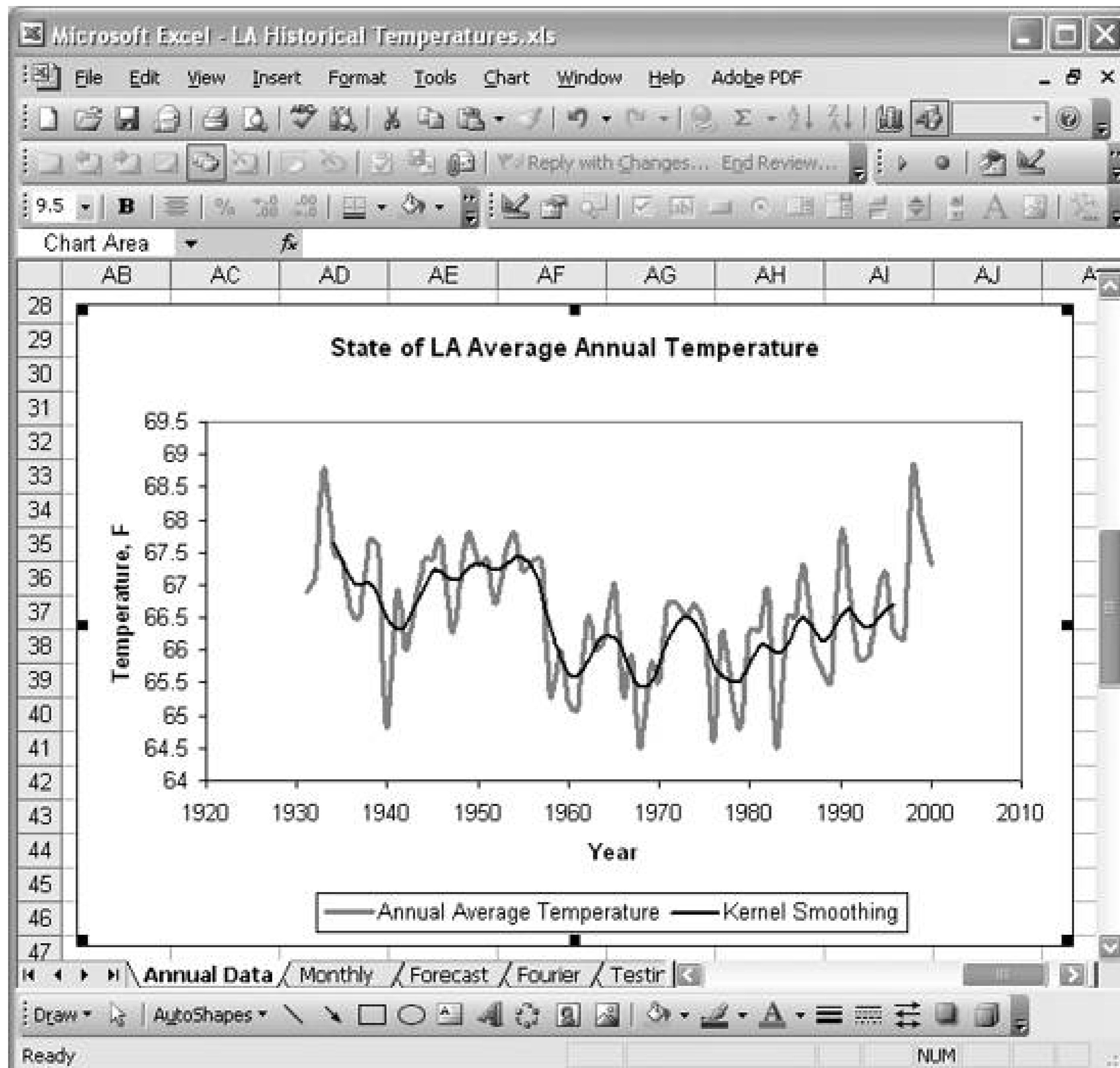
The second cell range contains the weights shown in [Figure 6-14](#). Further, dividing by the sum of weights, contained in cell V14, normalizes the result. Once the first formula was set up, I simply copied and pasted it into the remaining cells in the data series.

Notice that we do lose some data. Specifically, we lose the first and last three data items in the range because of the range of influence of the weighting function I used. You can increase the range of influence by increasing the value for the `h` parameter passed into `Wcs`, or you can decrease it by

decreasing this value. The greater the range of influence (hence the greater the number of data items in the series averaged for each smoothed result), the smoother the resulting data series.

[Figure 6-16](#) shows a plot of the smoothed data series compared to the original data series; the effect of smoothing is clearly evident.

Figure 6-16. Kernel-smoothed temperature data



## See Also

See [Recipe 6.3](#) to learn how to compute non-weighted moving averages in Excel. Smoothing data using the techniques in this and the previous recipe is actually a form of filtering. You can also filter data in the frequency domain in Excel using Fourier transforms; read [Recipe 6.10](#) to learn more.



[← PREV](#)

## Recipe 6.5. Centering Data

### Problem

You need to center a data series before further analysis.

### Solution

Use basic spreadsheet techniques and Excel's built-in formulas to center the data.

### Discussion

*Centering* in this context refers to the process of removing a bias or offset in a series of data so that the resulting series values fluctuate approximately about 0. Centering is often required before processing data using forecasting techniques, and can be accomplished quite easily in Excel. There are a number of ways to center a data series, and here we'll use the standard approach of subtracting the average of the data in the series from each data item in the series.

[Figure 6-17](#) shows an example spreadsheet I set up.

Figure 6-17. Centering example



The screenshot shows a Microsoft Excel spreadsheet titled "LA Historical Temperatures.xls". The spreadsheet has three columns: "Month" (C), "Average Temp." (D), and "Centered Data" (E). The data spans from row 3 to row 26. Row 27 contains a summary row with the text "Average =" in column C and the value "66.2375" in column D. The formula bar at the top shows the formula `=AVERAGE(D3:D26)` for cell D27. The spreadsheet also includes a "List Box" and a "Draw" toolbar at the bottom.

	C	D	E	F
2	<b>Month</b>	<b>Average Temp.</b>	<b>Centered Data</b>	
3	1	49.10	-17.14	
4	2	53.00	-13.24	
5	3	54.70	-11.54	
6	4	64.50	-1.74	
7	5	76.70	10.46	
8	6	79.10	12.86	
9	7	82.20	15.96	
10	8	80.20	13.96	
11	9	75.80	9.56	
12	10	66.90	0.66	
13	11	58.90	-7.34	
14	12	54.20	-12.04	
15	13	49.20	-17.04	
16	14	53.90	-12.34	
17	15	63.80	-2.44	
18	16	62.20	-4.04	
19	17	72.40	6.16	
20	18	78.70	12.46	
21	19	83.10	16.86	
22	20	81.30	15.06	
23	21	78.40	12.16	
24	22	67.10	0.86	
25	23	55.10	-11.14	
26	24	49.20	-17.04	
27	Average =	66.2375		

The first column contains the month label, while the second column contains the original data series before centering. This series represents average monthly temperatures.

To center the data, I first computed the average of all the temperature values in the original data series. Cell D27 contains the formula `=AVERAGE(D3:D26)` that computes the average of all temperatures.

The column adjacent to the original data series contains the centered data (column E). The formulas in this column are of the form  $=D3-\$D\$27$ . All this does is subtract the average temperature contained in cell D27 from every temperature value in the original series. The result is a centered data series with the offset removed.

I set up a few plots so you can see the difference. [Figure 6-18](#) shows the original data series and [Figure 6-19](#) shows the centered data series.

Figure 6-18. Original data series

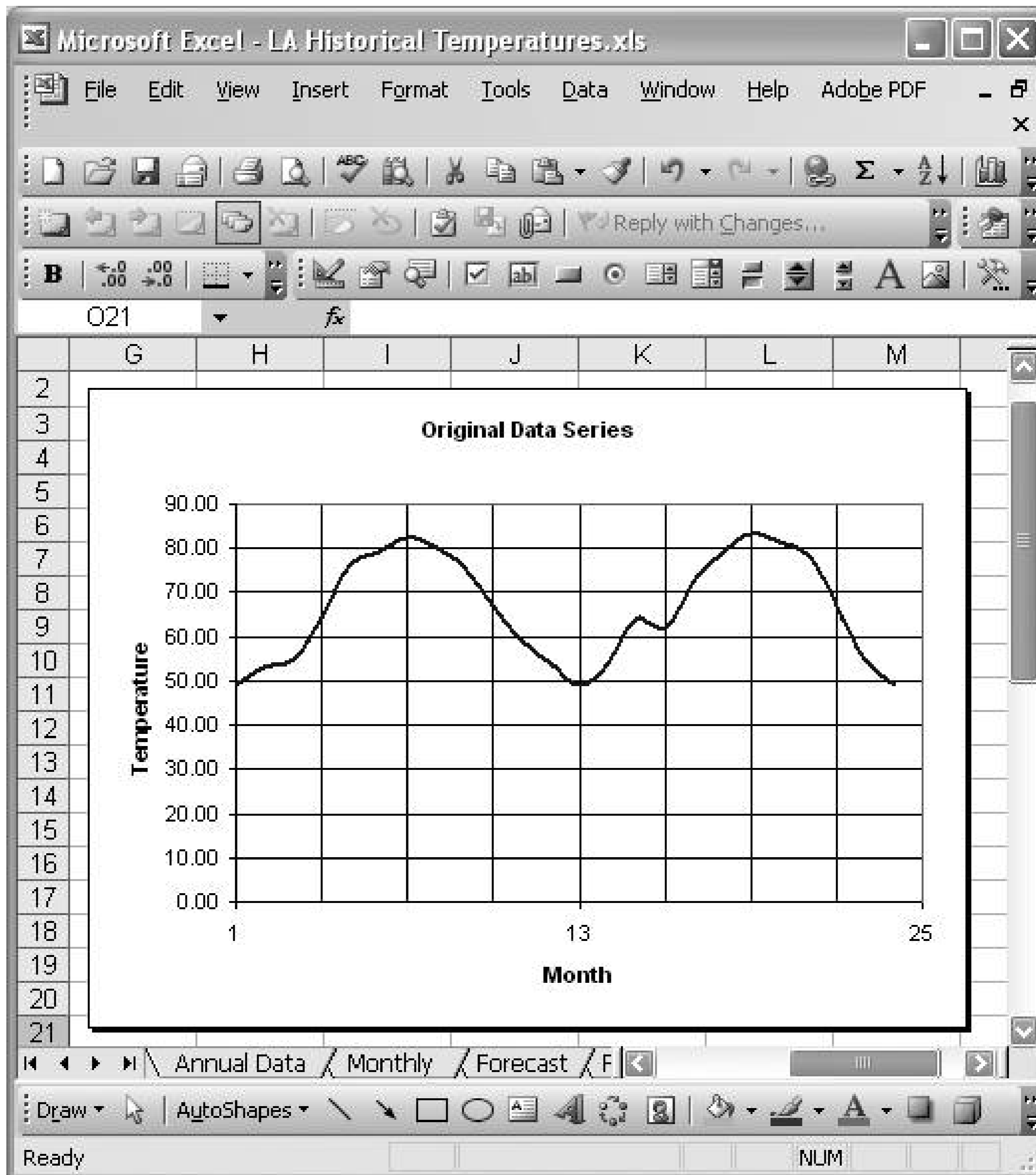
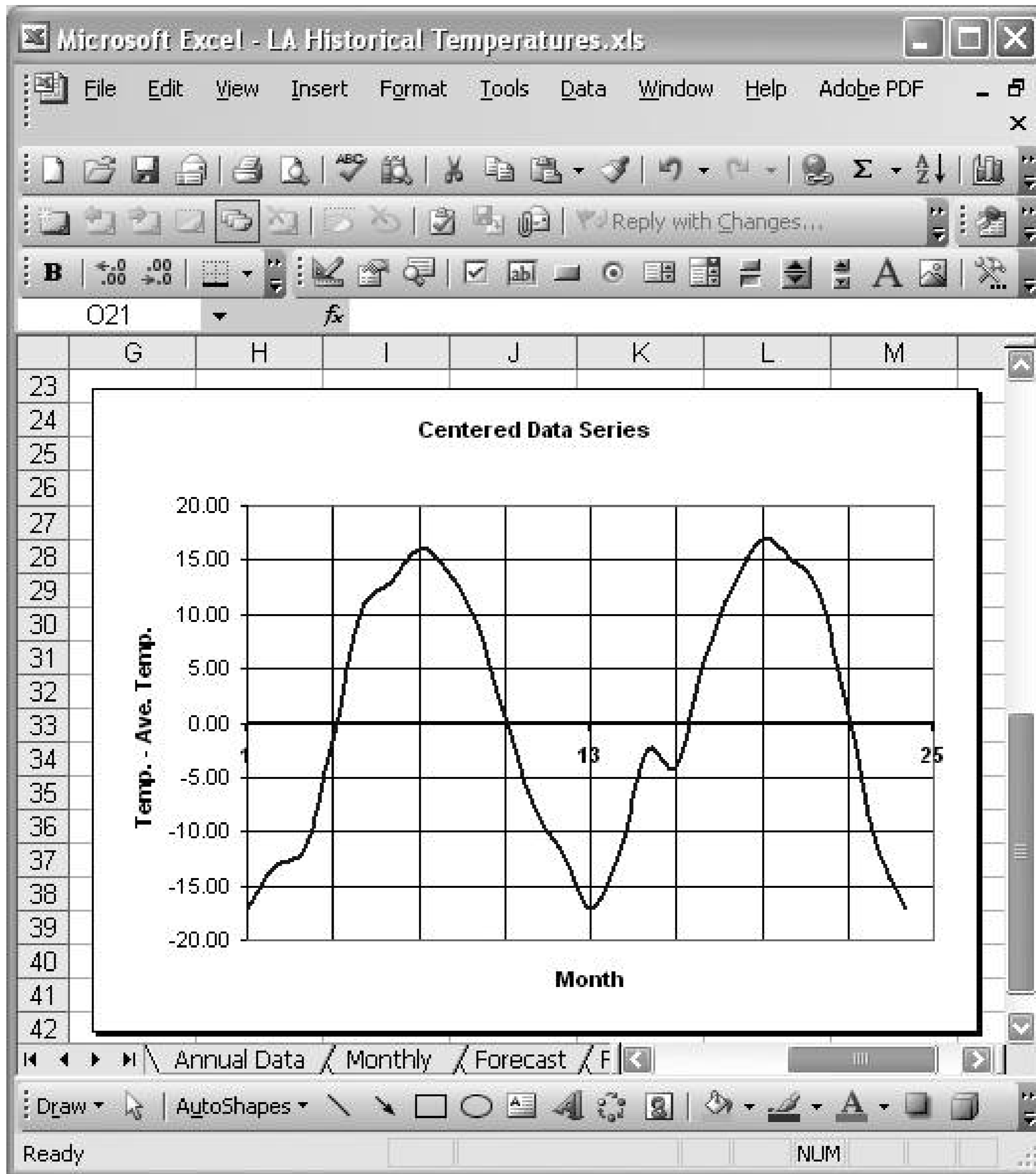


Figure 6-19. Centered data series



The original data series oscillates about the average temperature up in the 60s, while the centered data series oscillates about the 0 axis.

This isn't the only way to center data. Some techniques involve computing a weighted average of the data and using the result to center the series.

## Recipe 6.6. Detrending a Time Series

### Problem

You're working with a data series that exhibits a clear trend and before processing the data further you need to remove the trend from the data. This is called *detrending*.

### Solution

Construct a trendline in Excel using one of the techniques discussed in [Chapter 8](#) (see [Recipe 6.2](#) for an introduction to trendlines in Excel). Use the resulting trendline to detrend the original data as discussed in this recipe.

### Discussion

Time series data is often thought of as being comprised of several components: *long-term trend*, *seasonal variation*, and *irregular variations*. (Some models assume a fourth, *cyclic*, component.) When analyzing time series data (e.g., when making forecasts based on historical data), it's often desirable to decompose the time series into its various components. To this end, additive or multiplicative models are often used.

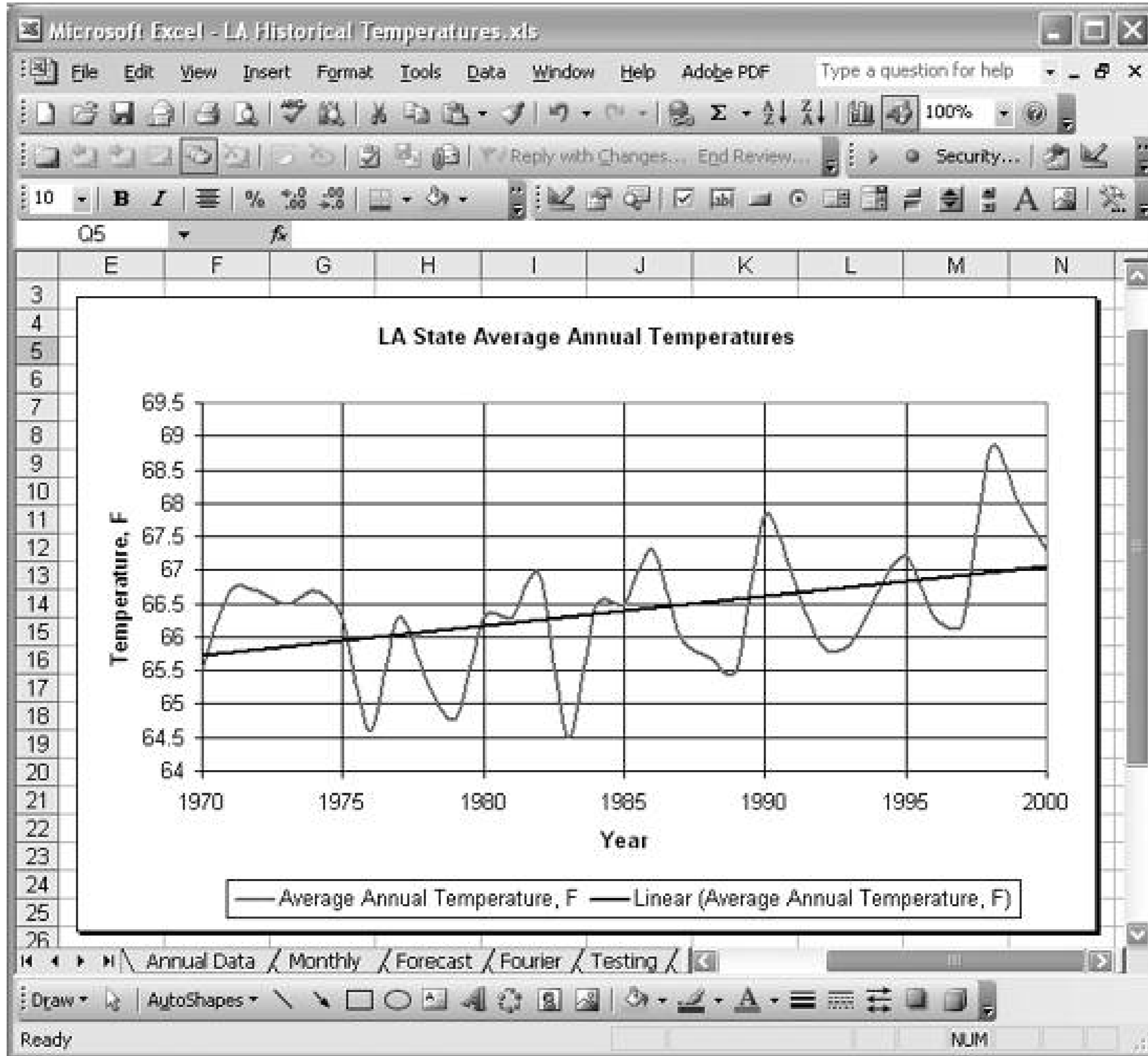
Let  $Y$  represent the ordinates of a time series such that  $Y = f(t)$ , where  $f$  is some function of time. Then, we can decompose  $Y$  as follows:

or

In these equations,  $T$  represents the long-term trend component,  $S$  the seasonal component, and  $I$  the irregular variation component of the total time series,  $Y$ . The first equation is a multiplicative model, while the second is an additive model. Which model you decide to use largely depends on the nature of your data and which model yields the best results. For example, when forecasting you might try both models and use whichever model yields the most accurate predictions. In this and the remaining recipes in this chapter, I'm going to use the multiplicative model; however, you can apply the additive model using very similar Excel techniques.

[Figure 6-20](#) shows historical average annual temperatures for the state of Louisiana from 1970 to 2000. This series exhibits a clear upward trend, highlighted by the linear trendline superimposed over the original data.

Figure 6-20. Annual temperatures from 1970 to 2000



If you were going to make a forecast using this historical data, one of the first steps you'd take would be to detrend the original series to remove the long-term trend component.

Using the multiplicative model, divide both sides of the equation  $Y = TS$  by  $T$  to yield  $Y/T = S$ . This means the detrended series,  $Y/T$ , consists only of the seasonal and irregular variation components. To actually compute  $Y/T$ , you must first compute a trendline as shown in [Figure 6-20](#) (see [Recipe 6.2](#) or [Chapter 8](#)). Then compute the trend value for each year in the series. Next, divide the original series ordinate ( $Y$ ) by the computed trend value ( $T$ ) to yield the detrended series ( $S$ ). [Figure 6-21](#) shows a portion of the spreadsheet I set up to perform these calculations for the example temperature series.

Figure 6-21. Detrending example spreadsheet

The screenshot shows a Microsoft Excel spreadsheet titled "LA Historical Temperatures.xls". The spreadsheet has columns labeled A through H. Column B is labeled "Year", column C is labeled "Y", column D is labeled "T", and column E is labeled "Y/T = SI". The data rows are numbered 39 to 56. The values in column C (Y) range from 64.5 to 67.3. The values in column D (T) range from 66.1 to 66.5. The values in column E (Y/T = SI) range from 0.97 to 1.01.

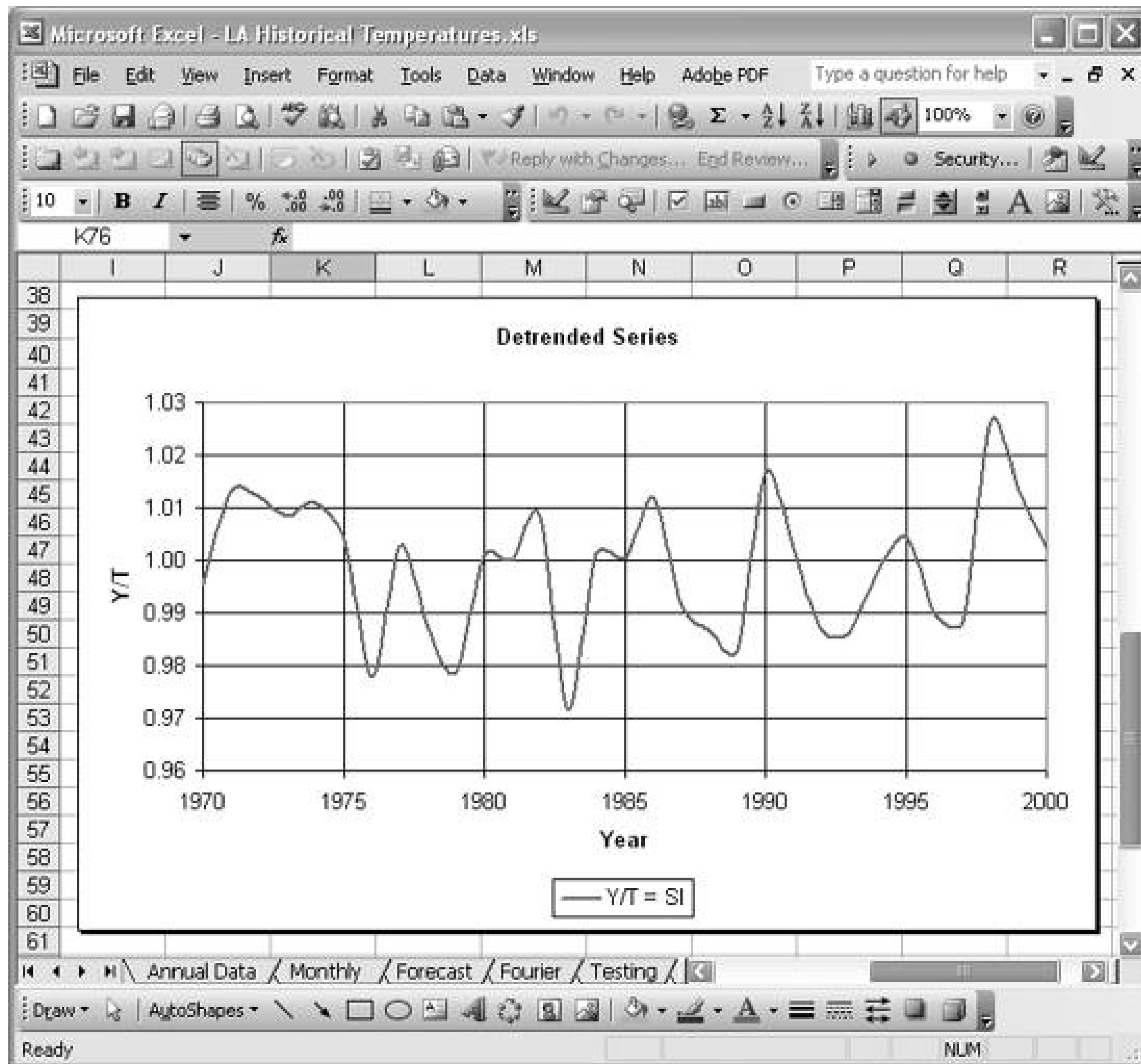
	A	B	C	D	E	F	G	H
38								
39		Year	Y	T	Y/T = SI			
40		1970	65.5	65.8	1.00			
41		1971	66.7	65.8	1.01			
42		1972	66.7	65.9	1.01			
43		1973	66.5	65.9	1.01			
44		1974	66.7	66.0	1.01			
45		1975	66.3	66.0	1.00			
46		1976	64.6	66.1	0.98			
47		1977	66.3	66.1	1.00			
48		1978	65.3	66.2	0.99			
49		1979	64.8	66.2	0.98			
50		1980	66.3	66.2	1.00			
51		1981	66.3	66.3	1.00			
52		1982	66.9	66.3	1.01			
53		1983	64.5	66.4	0.97			
54		1984	66.5	66.4	1.00			
55		1985	66.5	66.5	1.00			
56		1986	67.3	66.5	1.01			

Column B contains the year while column C (under the heading **Y**), contains the original temperature series.

The formula for the trendline shown in [Figure 6-20](#) is  $T=0.0446x-22.061$ , where  $x$  is the year. (This trendline equation was determined using Excel's chart trendline feature; see [Recipe 6.2](#).) Column D (under the heading **T**), contains this formula for each year. The cell formulas in column D are of the form  $=0.0446*B40-22.061$ . This series represents the long-term trend component for the original time series.

Finally, column E (under the heading **Y/T = SI**), contains the detrended series. You simply divide each value in the **Y** column by the corresponding value in the **T** column to yield  $Y/T$ . The cell formulas in the **Y/T = SI** column are of the form  $=C40/D40$ . [Figure 6-22](#) shows the resulting detrended series.

Figure 6-22. Detrended series



If you were to construct a linear trendline for this series, it would simply consist of a horizontal line. The variations shown in [Figure 6-22](#) are around the long-term trend, and they consist of both seasonal components (if present) and irregular variations. Further analysis (e.g., when forecasting), would probably require you to decompose this series even further to remove the seasonal component. This process is called *deseasonalization* and is covered in the next two recipes.

## See Also

There are other methods of detrending a time series besides using the least squares linear trendline used in this example. Sometimes higher-order trendlines are used, while at other times linear trendlines are computed using only the two series values at each end of the time series. Consult a standard text on time series analysis for more detailed information on these and other detrending methods.





← PREV

## Recipe 6.7. Estimating Seasonal Indices

### Problem

You're working with a time series that shows some seasonal variation and you'd like to compute the seasonal indices prior to deseasonalizing the data.

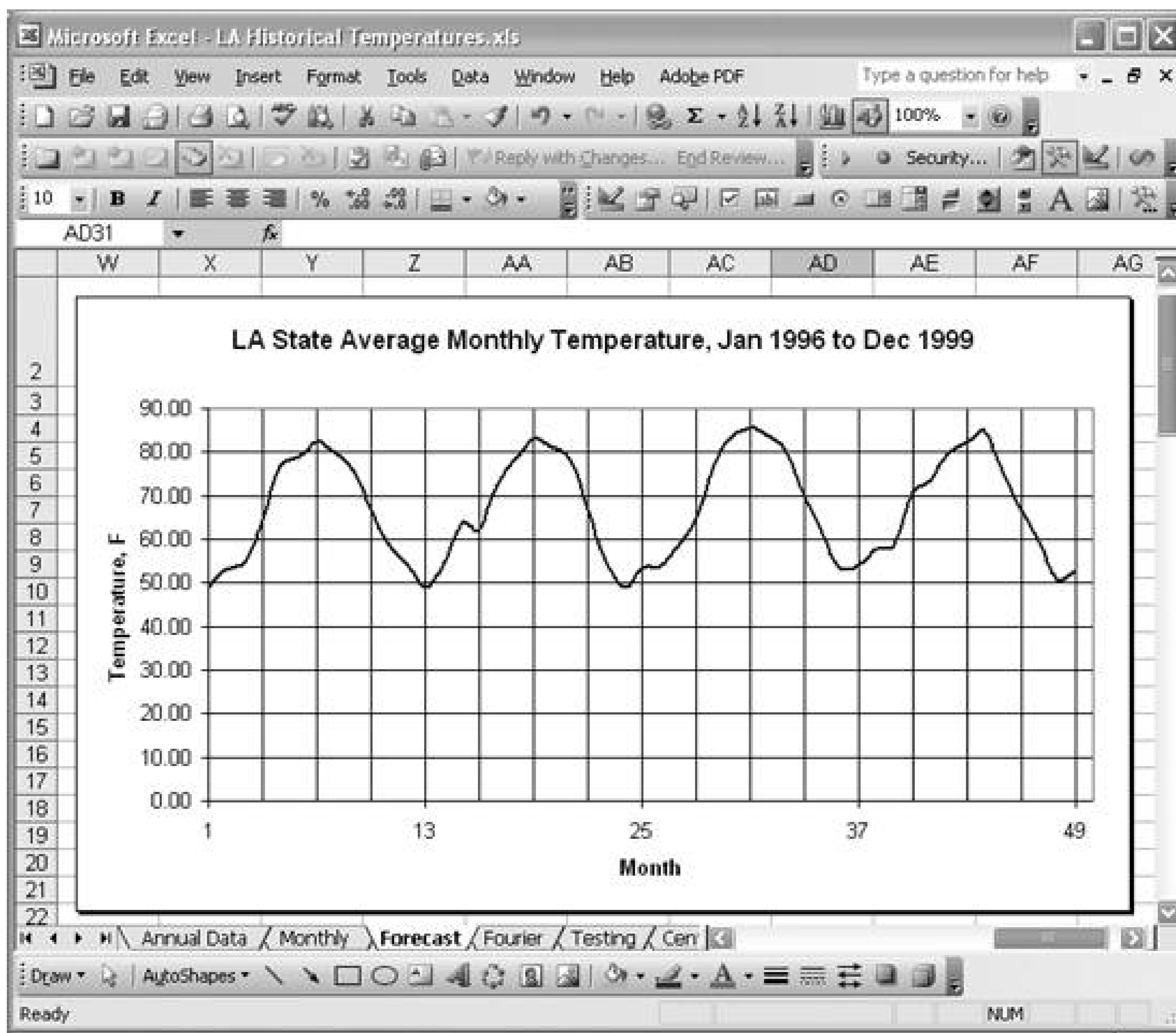
### Solution

You can compute seasonal indices using any of a number of methods. I'll show you how easy it is to compute such indices in Excel using the *average-percentage* method.

### Discussion

Take a look at the time series shown in [Figure 6-23](#).

Figure 6-23. Average monthly temperatures



This time series consists of average monthly temperatures for the state of Louisiana from 1996 to 1999. The vertical gridlines correspond to three-month quarters, and it's clear there's an obvious seasonal variation in average temperatures. In [Recipe 6.9](#), I'll show you how to forecast the average monthly temperatures for the year 2000 given this historical data. Part of that forecast analysis requires you to isolate the seasonal variation in temperatures. To do so, you must first compute the seasonal indices.

There are many standard methods for computing seasonal indices. The method I'll show you to illustrate how to use Excel for such calculations is the *average-percentage* method.

I'll use the series shown in [Figure 6-23](#) as an example. To apply the average-percentage method, compute the annual average temperature for each year and then express each monthly temperature as a percentage of the average annual temperature for the corresponding year. Then you average these percentages for corresponding months over all years to arrive at the seasonal index for each month.

The data shown in [Figure 6-23](#) was given as a series of temperature values over a range of months from 1 to 49, covering the years 1996 to 1999. To compute the seasonal indices, it's convenient to reorganize the data as shown in [Figure 6-24](#).

Figure 6-24. Original data and averages

	D	E	F	G	H	I	J
66							
67		<b>Original Data</b>					
68			<b>1996</b>	<b>1997</b>	<b>1998</b>	<b>1999</b>	
69		<b>Jan</b>	49.1	49.2	53.5	54.0	
70		<b>Feb</b>	53.0	53.9	53.6	57.9	
71		<b>Mar</b>	54.7	63.8	58.1	58.6	
72		<b>Apr</b>	64.5	62.2	64.7	70.9	
73		<b>May</b>	76.7	72.4	77.0	73.7	
74		<b>Jun</b>	79.1	78.7	83.5	79.9	
75		<b>Jul</b>	82.2	83.1	85.5	82.2	
76		<b>Aug</b>	80.2	81.3	83.8	85.0	
77		<b>Sep</b>	75.8	78.4	80.5	75.8	
78		<b>Oct</b>	66.9	67.1	70.0	66.7	
79		<b>Nov</b>	58.9	55.1	61.5	59.7	
80		<b>Dec</b>	54.2	49.2	53.5	51.0	
81		<i>Average:</i>	66.3	66.2	68.8	68.0	
82							

The original data is now in a table with rows representing monthly values over years represented by columns. Now it's a simple matter of applying the **AVERAGE** worksheet formula to compute annual averages. The last row of the table in [Figure 6-24](#) computes these averages using formulas of the form `=AVERAGE(F69:F80)`.

The next step is to divide each monthly temperature value by the annual average for the corresponding year. [Figure 6-25](#) shows the table I set up for this calculation.

Figure 6-25. Seasonal indices

		1996	1997	1998	1999	Index
82						
83	<b>Seasonal Index</b>					
84						
85	Jan	0.74	0.74	0.78	0.79	0.76
86	Feb	0.80	0.81	0.78	0.85	0.81
87	Mar	0.83	0.96	0.84	0.86	0.87
88	Apr	0.97	0.94	0.94	1.04	0.97
89	May	1.16	1.09	1.12	1.08	1.11
90	Jun	1.19	1.19	1.21	1.18	1.19
91	Jul	1.24	1.26	1.24	1.21	1.24
92	Aug	1.21	1.23	1.22	1.25	1.23
93	Sep	1.14	1.18	1.17	1.12	1.15
94	Oct	1.01	1.01	1.02	0.98	1.01
95	Nov	0.89	0.83	0.89	0.88	0.87
96	Dec	0.82	0.74	0.78	0.75	0.77
97					Sum:	12.00
98					Average:	1.00
99						

This table is similar in format to the one shown in [Figure 6-24](#); however, each value in the table represents the ratio of monthly temperature to annual average for corresponding years. For example the cell formula in cell F85 corresponding to January 1996 is  $=F69/F\$81$ , which is the temperature value for January 1996 from the table shown in [Figure 6-24](#) divided by the annual average for the year 1996, from the last row of the table in [Figure 6-24](#). The cell formulas are similar for the other months and years in the table in [Figure 6-25](#).

The last column in [Figure 6-25](#) contains the seasonal index for each month. The seasonal index is simply the average of the ratios for the corresponding month over all years. For example, the January seasonal index in cell J85 is computed using the formula  $=AVERAGE(F85:I85)$ . The remaining indices are computed similarly.

The average of the seasonal indices for all months should come out to a value of 1. If it does not, then a suitable factor should be applied to each index so that the average does indeed work out to a value of 1. Cell J98 computes the average seasonal index as a check. As you can see, it comes out to a value of 1.00, so no adjustments are required.

The seasonal indices computed here confirm the observation made earlier that the temperatures are seasonally higher in the middle months of the year (over the second and third quarters of each year). You can see this by observing that the seasonal indices for the months of May through October are above the average index of 1, while the remaining indices are below this average.

## See Also

You can decompose a time series such as the one discussed here to isolate the seasonal variation in a manner similar to the way in which we isolated the long-term trend in [Recipe 6.6](#). Further, as you can also deseasonalize a time series. The next recipe shows you how.

 PREVIOUS

## Recipe 6.8. Deseasonalization of a Time Series

### Problem

You're working with a time series with clear seasonal components, which you'd like to isolate and remove from the original data.

### Solution

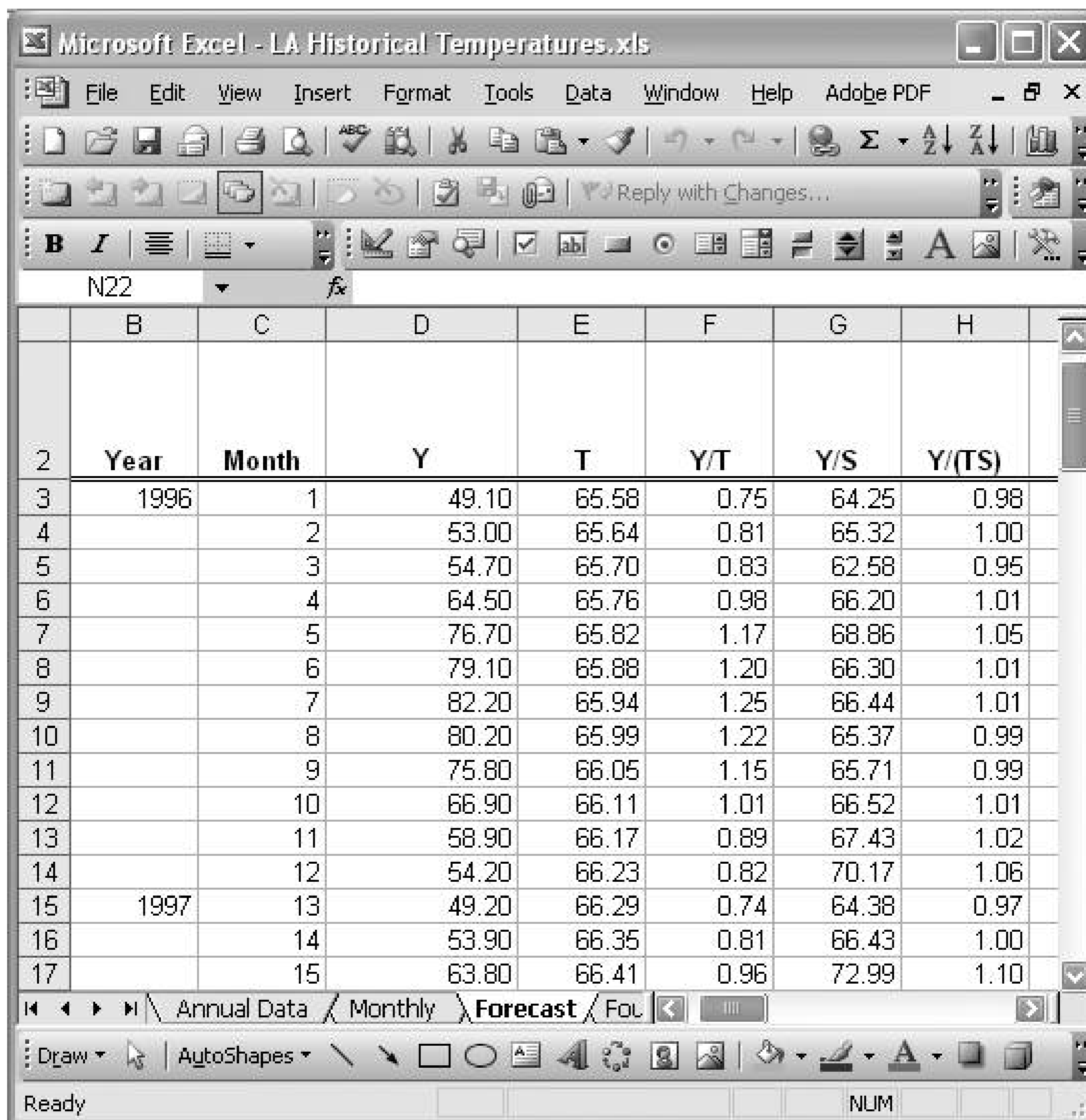
Deseasonalize your data in Excel using standard spreadsheet techniques.

### Discussion

In the previous recipe, I showed you how to compute seasonal indices for a time series using the average-percentage method. That recipe required nothing more than standard spreadsheet techniques to compute the indices. Further, the earlier [Recipe 6.6](#) showed you how to use standard spreadsheet techniques to isolate and remove the long-term trend in a time series. The model used there was the multiplicative model in which the original time series,  $Y$ , is assumed to be composed of a long-term trend, seasonal variation, and irregular variation, which is represented in the formula  $Y = TS I$ . Deseasonalizing a time series amounts to estimating the  $S$  contribution and removing it by dividing  $Y$  by  $S$  (that is, by computing  $Y/S$ ). You can extend these ideas further to isolate the irregular variation as well by computing  $Y/(TS)$  to yield  $I$ .

Let's reconsider the average monthly temperature series shown earlier in [Figure 6-23](#). That data series has a slight upward trend and a very clear seasonal variation over each year. Using the seasonal indices computed in the previous recipe for this time series, we can easily decompose and deseasonalize the series. [Figure 6-26](#) shows a spreadsheet I set up to decompose the time series.

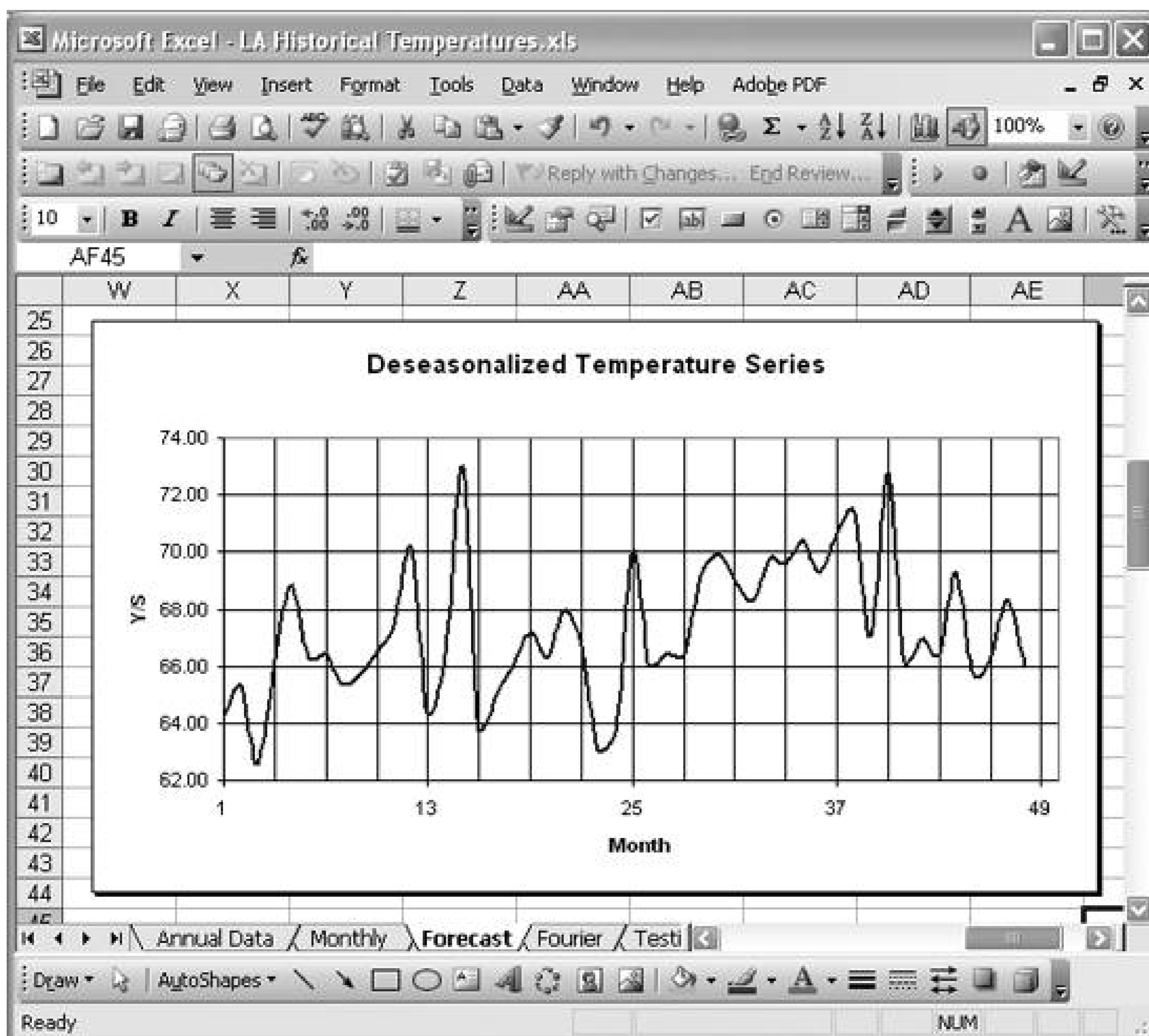
Figure 6-26. Average monthly temperature decomposition



Column **Y** represents the original time series data. These are the average monthly temperatures plotted in [Figure 6-23](#). This data series exhibits a slight upward trend, which can be modeled with the linear trendline equation  $T = 0.0592x + 65.521$ . Column **T** in [Figure 6-26](#) computes this trend. Detrending this series is carried out as before (in [Recipe 6.6](#)), by dividing the original data **Y** by the trend **T**. Column **Y/T** contains the result.

Deseasonalization is carried out in a similar manner. However, here we divide the original series, **Y**, by the seasonal index for corresponding months. For example, the first cell in column **Y/S**, cell G3, contains the formula `=D3/Jan`. `Jan` is a name I defined for the January seasonal index computed in the previous recipe. Likewise, the formula in cell G4 for the month of February 1996 is `=D4/Feb`. I named all the seasonal indices in this manner and each monthly **Y** in each year is divided by the corresponding seasonal index. The results are contained in column **Y/S** and represent the deseasonalized series. [Figure 6-27](#) shows the resulting deseasonalized temperature series.

Figure 6-27. Deseasonalized temperature series



You can now detrend the deseasonalized series by dividing every value in the  $Y/S$  column by the trend value in the  $T$  column. These calculations are carried out in the  $Y/(TS)$  column in [Figure 6-26](#), where the cell formulas are of the form  $=G3/E3$ .

In some cases (e.g., if your data exhibits a significant long-term trend), you might prefer to detrend the data series first and then compute seasonal indices on the detrended series using the same techniques discussed in the previous recipe. Then seasonal components can be removed from the detrended series using the techniques discussed here. In either case, you've effectively isolated the irregular variation component,  $I$ , of the original series.

## See Also

In [Recipe 6.9](#), I'll show you how to extend the material presented in this and earlier recipes to predict the average monthly temperatures for the year 2000 based on the 1996 through 1999 historical data



shown in [Figure 6-23](#).



## Recipe 6.9. Forecasting

### Problem

You need to make a forecast of future values given historical values for a time series.

### Solution

Excel does not provide built-in functionality for sophisticated forecasting. It does have a `FORECAST` spreadsheet function that sounds enticing; however, that merely uses a linear trendline and is unsuitable for time series that exhibit significant seasonal variability.

As an alternative, you could use the various moving average techniques to make near-term forecasts. Or you could use some higher-order regression equation (see [Chapter 8](#)) to make near-term predictions. Keep in mind that extrapolating too far beyond the extents of the original time series could result in significant errors.

If you have the resources, you could purchase one of the many Excel add-ins that offer sophisticated autoregressive moving average (ARMA) or neural network techniques. If you're so inclined, you can even program such techniques yourself in VBA.

In lieu of all these approaches, you can use standard spreadsheet techniques to make forecasts. I'll show you how by extending the techniques discussed in the previous several recipes.

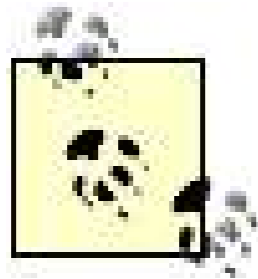
### Discussion

I'll again use the time series shown in [Figure 6-23](#). Recall that this series consists of average monthly temperatures in Louisiana from January 1996 to December 1999. The task now is to make a forecast for the average monthly temperatures in the year 2000.

The brunt of this forecasting effort has already been covered in the previous recipes, in which I showed you how to decompose the time series into its various components using the model  $Y = TS$ . (If you have not read the previous three recipes, you should do so now for relevant background discussion and calculations.)

In [Recipe 6.8](#) I showed you how to compute the long-term trend component,  $T$ , for this example time series. Also, in [Recipe 6.7](#) I showed you how to compute the seasonal indices for this example time series.

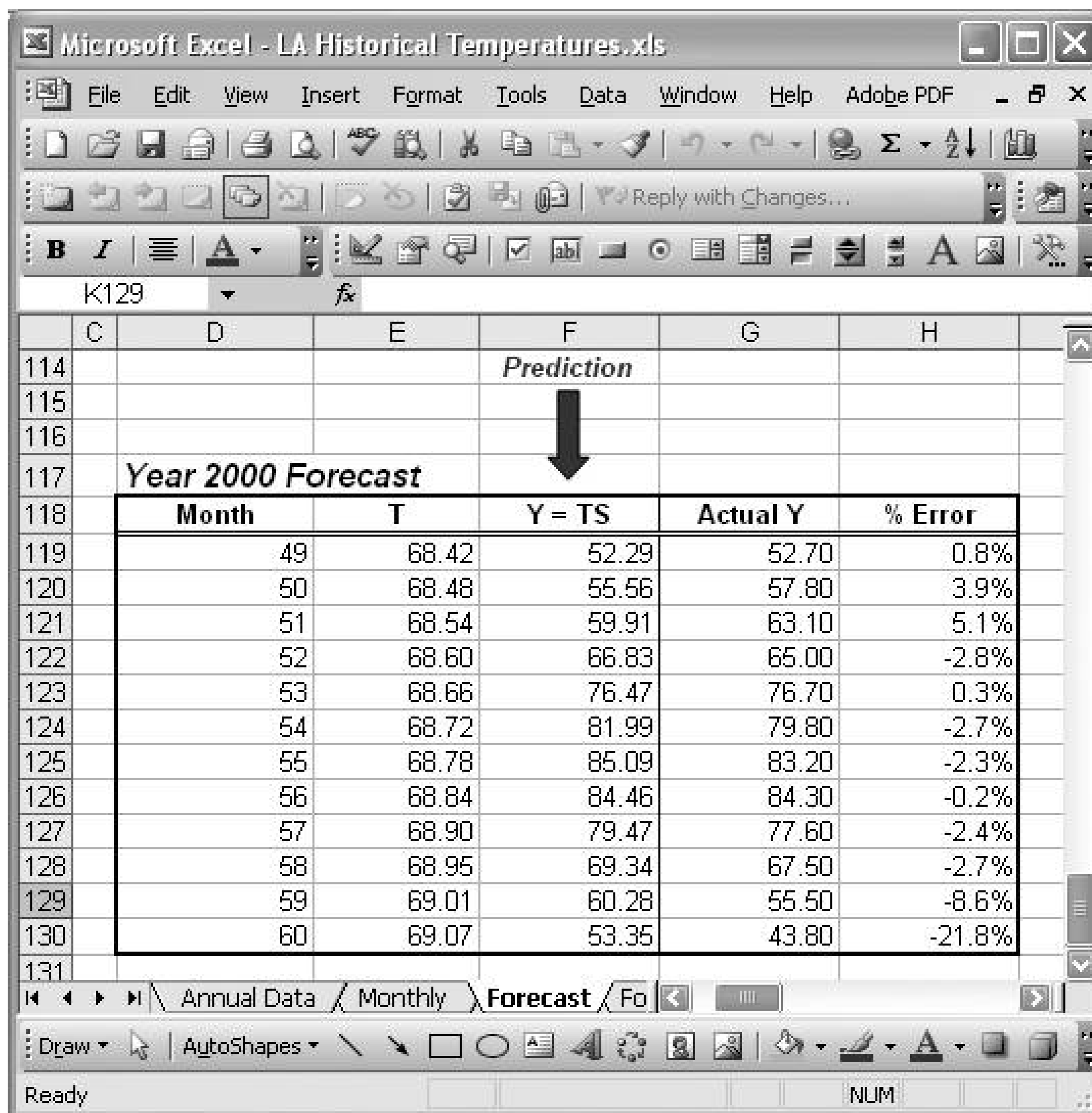
Now we'll put those results together to make a forecast for the year 2000. We can do so by computing  $Y_f = TS$ . Here  $Y_f$  represents the forecast temperature values.  $T$  represents the long-term trend, this time extended out over the year 2000. And  $S$  represents the seasonal indices as before.



I've ignored the irregular variability component here. This component requires a more sophisticated modeling approach such as an ARMA or neural network approach. As you'll see shortly, we still get fairly good predictions without this component. Further, you could take this irregular variability into account when computing confidence intervals on your predictions by using techniques discussed in [Chapter 5](#).

So all you really need to do now is compute the long-term trend values for the year 2000 using the trend equation from the previous recipe. Then multiply these trend values by the seasonal index for each corresponding month to arrive at a prediction for each month in the year 2000. [Figure 6-28](#) shows a spreadsheet I set up to perform these calculations.

Figure 6-28. Year 2000 forecast



The column labeled **Month** contains the months representing the year 2000 as an extension of the original series, in which December 1999 was month number 48. You need to number the months for the year 2000 this way because the trendline equation computed in the previous recipe is based on the original monthly series from months 1 to 48 and now we're extending that trend out over another 12 months.

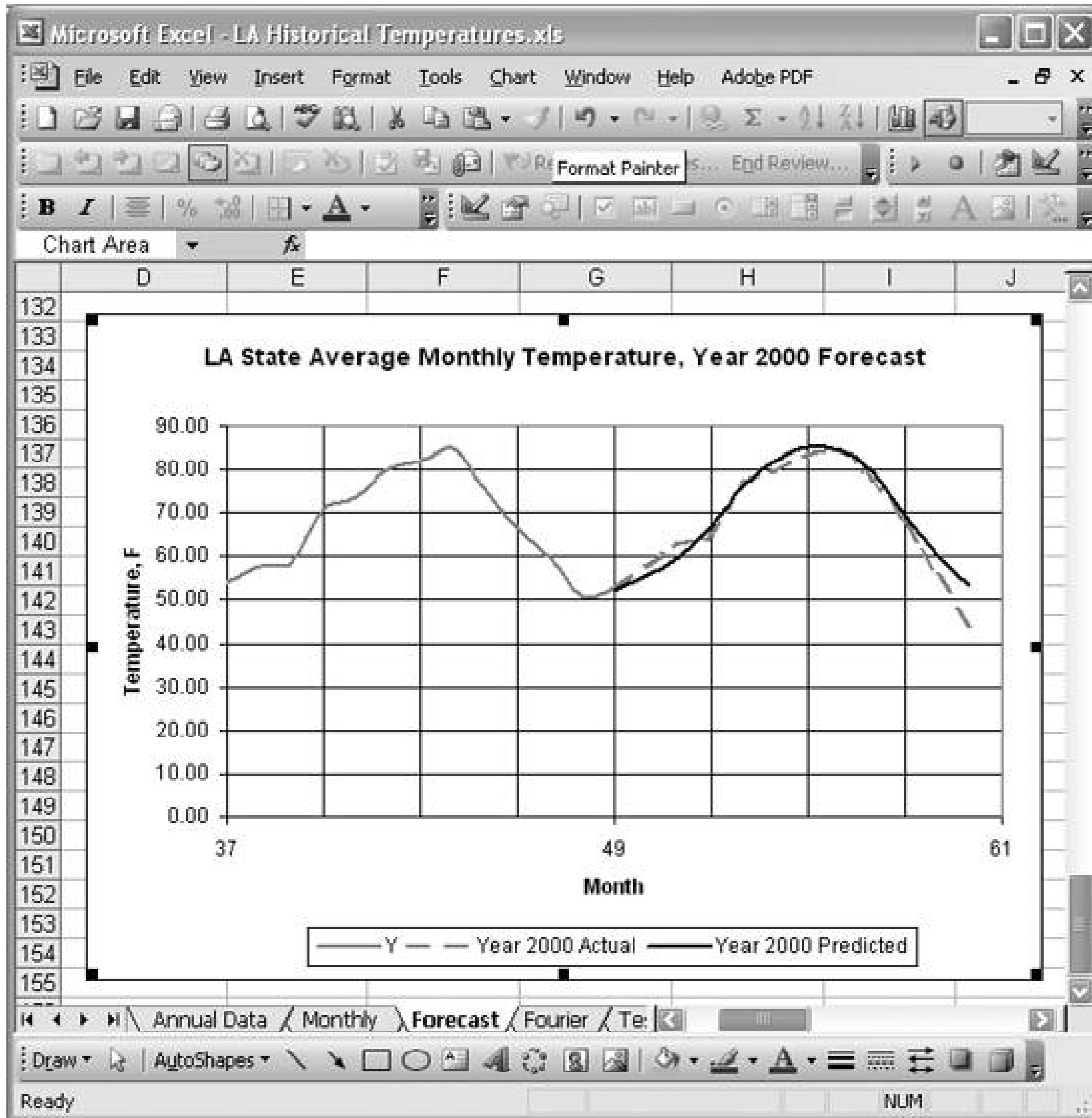
Column **T** in [Figure 6-28](#) shows the results of the forecast trend in the year 2000. The formulas in this column are of the form  $=0.0592 * D119 + 65.521$ .

Column **Y = TS** contains the predicted average monthly temperatures for the year 2000. These are computed by multiplying each trend value for the year 2000 by the corresponding seasonal index for each corresponding month. For example, the predicted value for January 2000, cell F119, is computed using the formula  $=E119 * Jan$ . Here again, I'm making use of the cell names I set up corresponding to each seasonal index computed in [Recipe 6.7](#).

[Figure 6-29](#) shows the predicted results compared to the actual average monthly temperatures on

record for the year 2000.

Figure 6-29. Year 2000 forecast chart



As you can see, these predictions are fairly good. They won't get me a starring role as the local weatherman, but they're not bad for such a simple forecast model. The table in [Figure 6-28](#) also contains computed %Error values, and you can see that errors are on the order of a few percent except for the months of November and December.

The forecasting technique shown in this recipe is by no means the only technique available to you. Time series forecasting is a huge field of study and there are many techniques and models available, many of which are tailored to specific kinds of time series. The examples discussed here serve to show how you can leverage Excel's standard functionality to perform predictions.



← PREV

## Recipe 6.10. Applying Discrete Fourier Transforms

### Problem

You'd like to use a discrete Fourier transform to analyze a dataset but aren't sure how to do so in Excel.

### Solution

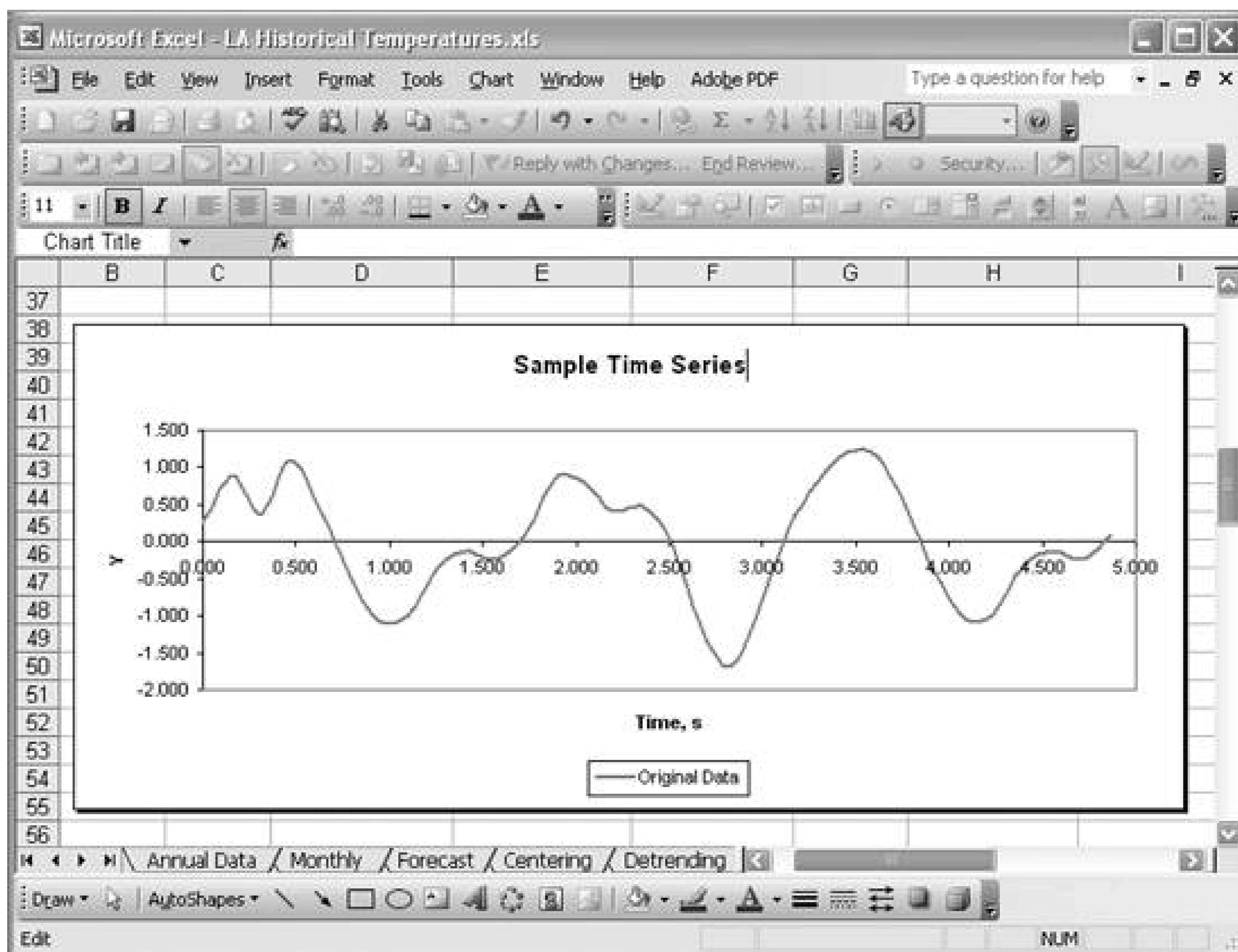
Use the Fourier Analysis tool in the Analysis ToolPak.

### Discussion

Excel provides a Fourier Analysis tool as part of the Analysis ToolPak. This tool allows you to perform discrete Fourier transforms and inverse transforms directly in your spreadsheet. Once your data is transformed, you can manipulate it in either the frequency domain or time domain, as you see fit.

Consider the time series shown in [Figure 6-30](#).

Figure 6-30. Sample time series



I generated this time series by superimposing several cosine and sine waves of varying frequencies. Discrete samples were taken every 0.15708 ( $\pi/20$ ) seconds for a total of 32 samples. [Figure 6-31](#) shows a portion of a spreadsheet containing the data making up this time series.

Figure 6-31. Sample cosine and sine wave frequency data



	A	B	C	D	E	F
1	<b>Sample Interval =</b>		<b>0.15708</b>	<b>seconds</b>		
2	<b>Number of Samples</b>		<b>32</b>			
3						
4	i	Time (s)	Y(t)	Frequency (Hz)	Frequency (cycles/sample)	DFT(Y)
5	0	0.000	0.250	0.000	0.000	-0.531073000697
6	1	0.157	0.894	0.199	0.031	0.432411242541
7	2	0.314	0.363	0.398	0.063	1.372887000860
8	3	0.471	1.106	0.597	0.094	9.135892653439
9	4	0.628	0.436	0.796	0.125	-1.044784347558
10	5	0.785	-0.472	0.995	0.156	-5.104610531853
11	6	0.942	-1.066	1.194	0.188	-0.904678384234
12	7	1.100	-0.995	1.393	0.219	-0.327014506463
26	21	3.299	0.805	4.178	-0.344	7.534331657041
27	22	3.456	1.201	4.377	-0.313	1.218791831959
28	23	3.613	1.149	4.576	-0.281	-0.716098634640
29	24	3.770	0.464	4.775	-0.250	3.772240428453
30	25	3.927	-0.439	4.974	-0.219	-0.327014506463
31	26	4.084	-1.046	5.173	-0.188	-0.904678384234
32	27	4.241	-0.981	5.371	-0.156	-5.104610531853
33	28	4.398	-0.336	5.570	-0.125	-1.044784347558
34	29	4.555	-0.131	5.769	-0.094	9.135892653439
35	30	4.712	-0.235	5.968	-0.063	1.372887000860
36	31	4.869	0.105	6.167	-0.031	0.432411242541

The first column, labeled *i*, contains an index identifying the sample number. The second column, labeled *Time*, contains the time at which each sample was taken. And the third column, labeled *Y(t)*, contains the sampled ordinate. There are a total of 32 samples, numbered 0 to 31.

I hid several rows of data in this table so I could fit the first and last rows in one screenshot. You can hide and unhide rows (or columns) by selecting the appropriate menu option from the Format menu. For example, Format → Row → Hide hides the selected row(s).

Before transforming this time series to the frequency domain, I set up a few columns to contain frequency values. A Fourier transform produces the same number of frequency bins, or bands, as time series samples. Therefore, there are 32 frequency bins to label in this example. I used the standard formula  $f_j = i/(ns)$  to compute the frequencies in cycles per second (Hz), as shown in the **Frequency (Hz)** column. In this formula,  $i$  represents the sample number,  $n$  the number of samples, and  $s$  the sample interval. The cell formulas in the **Frequency (Hz)** column are of the form `=A5/(n*s)`. I used cell names (see [Recipe 1.14](#) to learn how to name cells) to name the cells containing the number of samples and the sample interval. Their names are **n** and **s**, respectively.

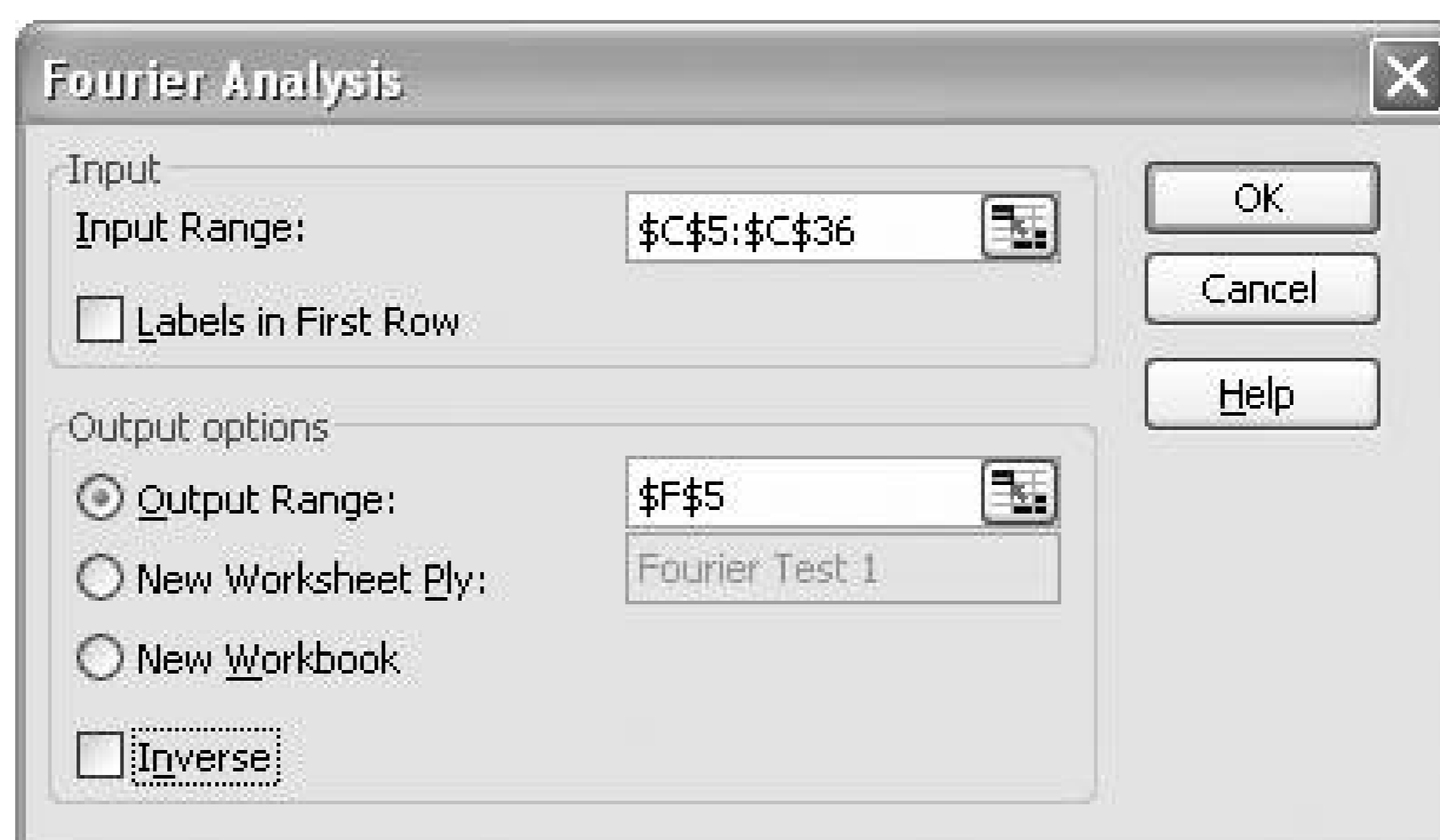
I also computed frequencies in standard units of cycles per sample using the formula  $f_j = i/n$ . These frequencies are contained in the **Frequency (cycles/sample)** column. The formulas in this column are of the form `=A5/n`.

The column **DFT(Y)** contains the discrete Fourier transform of the time series data. To generate these values you have to use the Fourier Analysis tool. Select **Tools** → **Data Analysis...** from the main menu bar to open the Data Analysis dialog box. Select **Fourier Analysis** from the list of available analysis options shown in [Figure 6-32](#).

Figure 6-32. Fourier Analysis tool selection

Press OK to open the Fourier Analysis dialog box shown in [Figure 6-33](#).

Figure 6-33. Fourier Analysis dialog box



At this point in the example, we're performing a time to frequency domain transform, so make sure the Inverse checkbox is not checked. In the Input Range field, type (or select from your spreadsheet) the cell range containing the time series ordinates to transform. The cell range for this example is C5 to C36. The size of this range must be a power of 2 (for example,  $2^1$ ,  $2^2$ ,  $2^3$ ,  $2^4$ , and so on). The maximum size is 4,096 (i.e.,  $2^{12}$ ). If your time series data is not a power of 2, then you should pad the end of the series to the next power of 2.

Time series data is often padded on one end before it is transformed, for a couple of reasons. One reason is to build a set of data of a size that's a power of 2. Another reason involves the so-called wraparound effect that can distort results when convolving data. In either case, the padded values of the time series should be 0 if your original time series is centered (see [Recipe 6.5](#)). If your time series is not centered, then you can use the average value of the time series ordinates as the pad value. It's also helpful to detrend your time series first if it exhibits a significant trend (see [Recipe 6.6](#)).

You have a choice as to where to place the output. You can have the tool place the transformed data on a new worksheet, in which case you supply the name of the new worksheet. Or you can specify a cell on the current worksheet in which to place the results. The results will actually be placed in a column of cells with the specified output cell as the topmost cell. In this example, I specified cell F5 on the current worksheet as the first cell for the output column.

Press OK to close the Fourier Analysis tool and view your results. Column  $DFT(Y)$  in [Figure 6-31](#) contains the results for this example. Note that these are complex numbers (you can't see them in their entirety, because I didn't make the cells wide enough). These cell entries look like this:  $0.432411242541142-4.7611558001505E-002i$ . These aren't actually numbers. In fact, they are text. Excel uses text in this form to represent complex numbers. You can still perform mathematical operations on these complex numbers using Excel's complex number worksheet functions. Refer to [Recipe 7.13](#) for a list, along with descriptions, of Excel's built-in complex number functions.

At this point, you have the frequency domain version of your original time series. You can manipulate these results in many different ways to suit the type of analysis you're conducting. For example, you can easily generate a power spectrum of these results to study the frequency content of the original series. All you need to do is add another column to the table shown in [Figure 6-31](#) and use standard

formulas for computing the power contained in each frequency band. Figure 6-34 shows a modified table containing a power spectrum calculation for this example.

Figure 6-34. Power spectrum calculation

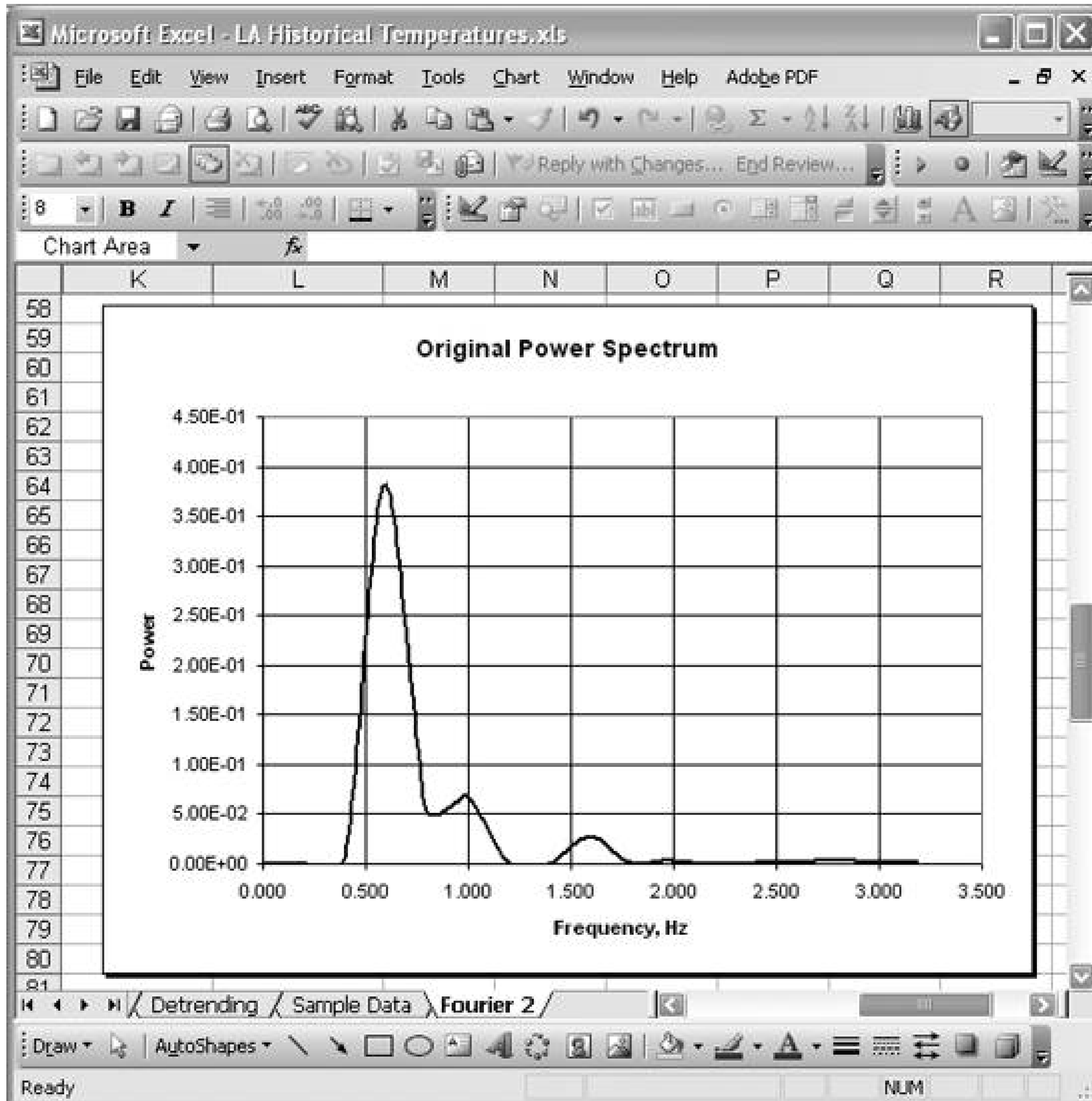
	A	B	C	D	E	F	K
1	Sample Interval =		0.15708	seconds			
2	Number of Samples		32				
3							
4	i	Time (s)	Y(t)	Frequency (Hz)	Frequency (cycles/sample)	DFT(Y)	P-DFT(Y)
5	0	0.000	0.250	0.000	0.000	-0.531073000697	2.75E-04
6	1	0.157	0.894	0.199	0.031	0.432411242541	3.70E-04
7	2	0.314	0.363	0.398	0.063	1.372887000860	4.60E-03
8	3	0.471	1.106	0.597	0.094	9.135892653439	3.81E-01
9	4	0.628	0.436	0.796	0.125	-1.044784347558	5.43E-02
10	5	0.785	-0.472	0.995	0.156	-5.104610531853	6.77E-02
11	6	0.942	-1.066	1.194	0.188	-0.904678384234	2.17E-03
12	7	1.100	-0.995	1.393	0.219	-0.327014506463	4.36E-04
13	8	1.257	-0.348	1.592	0.250	3.772240428453	2.78E-02
14	9	1.414	-0.127	1.790	0.281	-0.716098634640	1.98E-03
15	10	1.571	-0.228	1.989	0.313	1.218791831959	3.01E-03
16	11	1.728	0.107	2.188	0.344	7.534331657041	5.37E-04
17	12	1.885	0.844	2.387	0.375	-0.399108668090	1.18E-03
18	13	2.042	0.807	2.586	0.406	-0.604203729216	1.90E-03
19	14	2.199	0.411	2.785	0.438	-1.107004322121	3.24E-03
20	15	2.356	0.468	2.984	0.469	-0.980700414242	1.97E-03
21	16	2.513	0.011	3.183	0.500	-1.107652870110	1.20E-03

Column P-DFT(Y) is the new one containing the power calculations. These calculations use one of Excel's built-in complex number functions, IMABS, to compute the absolute value of a complex number. For example, the power for the second frequency band, 0.031 cycles/sample, is computed using the cell formula =2\*IMABS(F6)^2/n^2. Cell F6 contains the complex number corresponding to a frequency of 0.031 in the discrete Fourier transform data column.

I computed the power for each frequency band up to the highest frequency, the Nyquist frequency (0.5 cycles/sample). The formula used for the power of the 0 frequency band is=IMABS(F5)^2/n^2, while the formula used for the Nyquist frequency band is=IMABS(F21)^2/n^2. The factor of 2 for all intermediate frequency power calculations is due to the periodicity of the Fourier transform and the

fact that I'm only computing power from 0 to 0.5 cycles per sample. [Figure 6-35](#) shows a plot of the resulting power spectrum.

Figure 6-35. Power spectrum

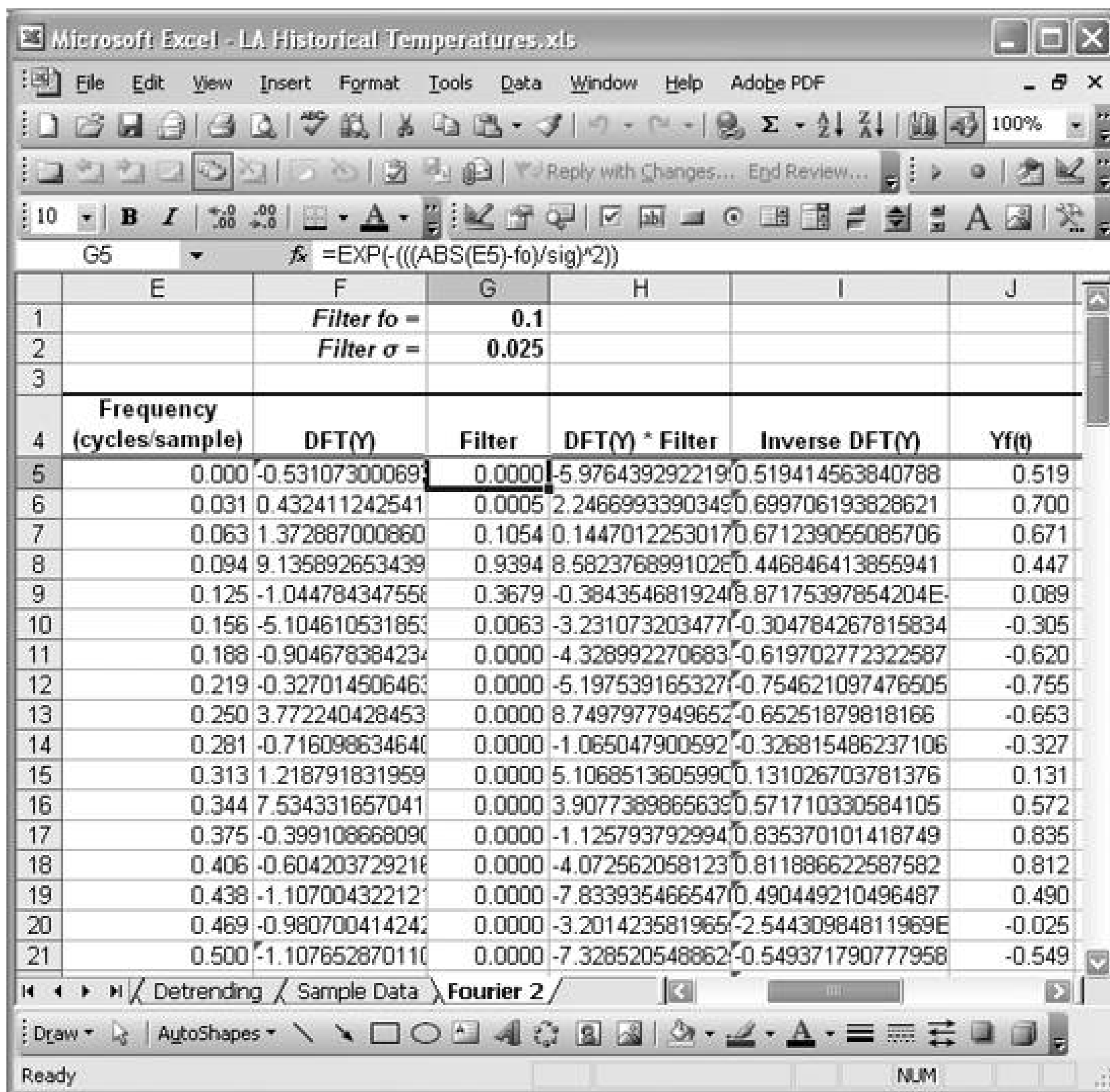


The frequencies on the horizontal axis are in Hz. As you can see, there's a strong component at a frequency of about 0.6 (0.637 to be more specific). Further, there are clear contributions at frequencies of 0.952 and 1.59, though these are not as strong as the one at 0.637.

You can take your analysis even further at this point. Let's say you want to filter the data in the frequency domain to isolate the 0.637 Hz component of the original series. It's relatively easy to construct and apply an appropriate filter in Excel and then perform an inverse Fourier transform to see the resulting filtered series in the time domain.

A suitable filter for this example would be a band pass filter centered at a frequency of 0.637 Hz or 0.1 cycles per sample. I used a standard Gaussian band pass filter to calculate the filter values shown in Figure 6-36.

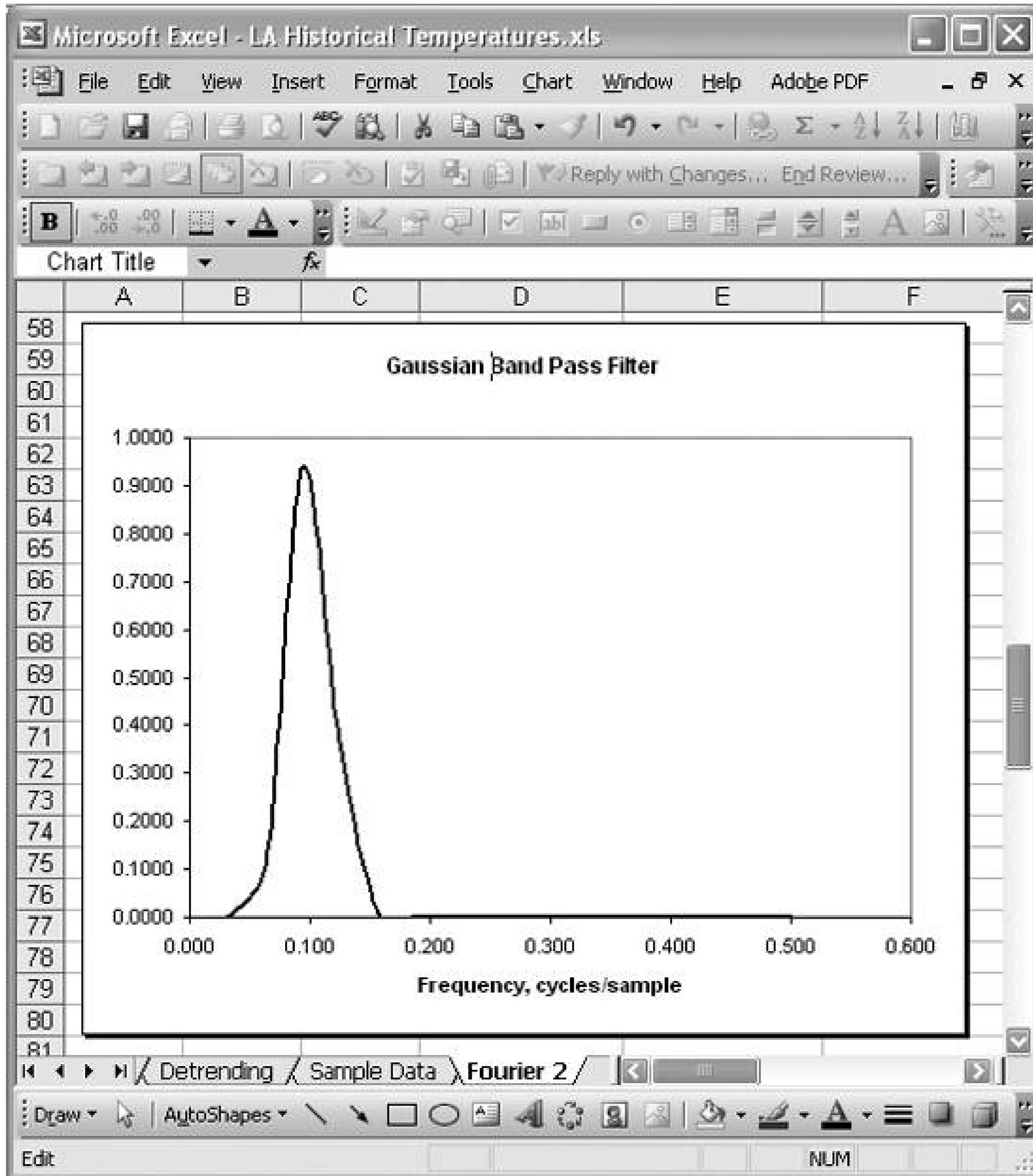
Figure 6-36. Band pass filter calculation



The Filter column contains filter values using formulas of the following form:  $=EXP(-((ABS(E5)-fo)/sig)^2)$ . This equation is a standard Gaussian filter equation.  $fo$  is the center frequency, which is contained in cell G1 (I defined the cell name  $fo$  to refer to this cell).  $sig$  is the spreading factor for the filter, which defines how wide it is in terms of how many frequency bands the filter covers.  $ABS(E5)$  is the absolute value of the frequency contained in the column labeled Frequency (cycles/sample). Notice that the center frequency in cell G1 is also expressed in units of cycles per sample.

Figure 6-37 shows a plot of the resulting filter over a frequency range from 0 to 0.5 cycles per sample.

Figure 6-37. Band pass filter

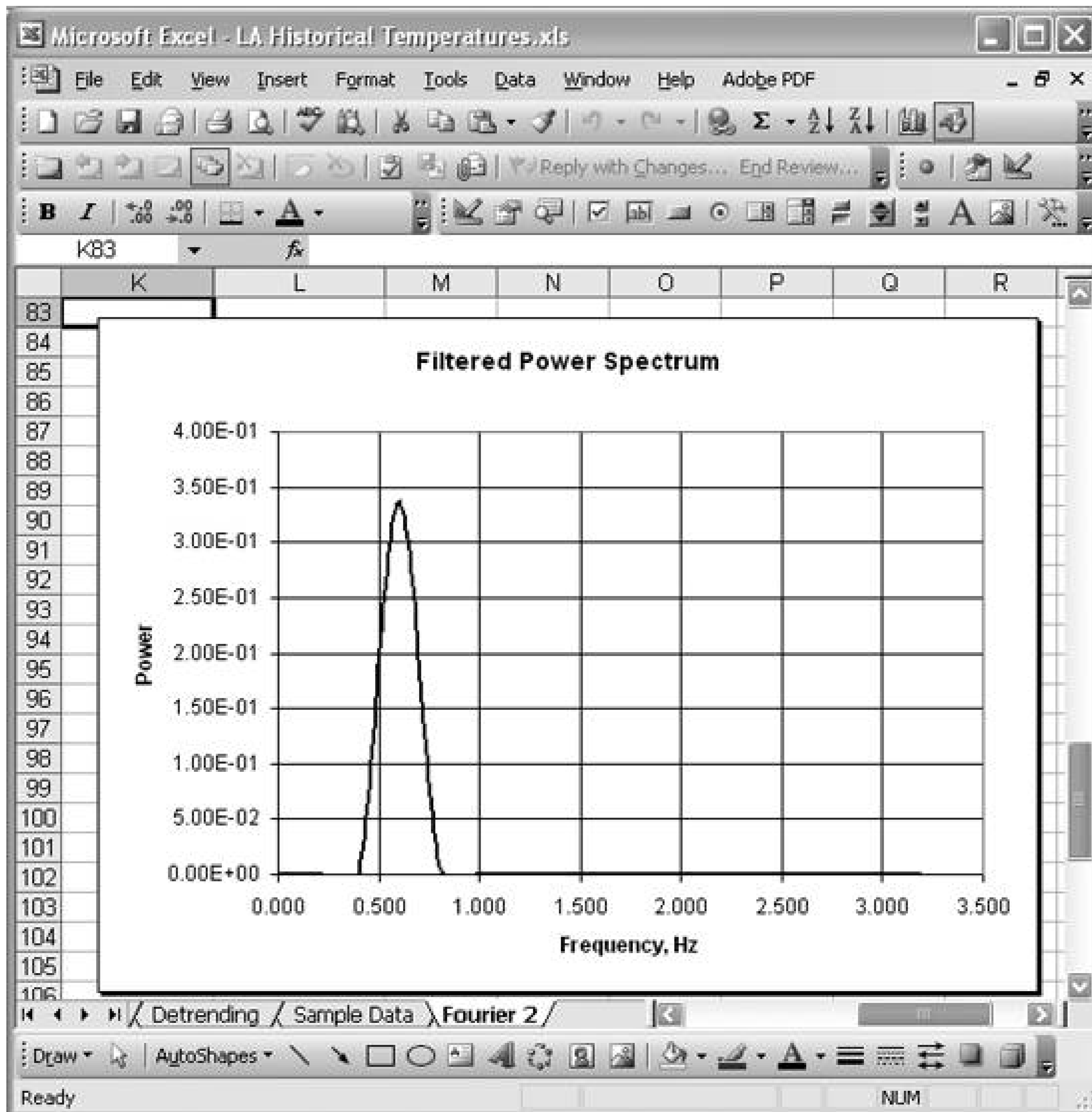


To actually apply this filter, you need to multiply each value in column  $DFT(Y)$  by the corresponding value in the *Filter* column. Remember, however, that the values contained in the  $DFT(Y)$  column are actually text values representing complex numbers. Therefore, you have to use one of Excel's built-in complex number functions to perform the multiplication. The appropriate function is the *IMPRODUCT*

function. For example, the cell formulas contained in the column labeled  $DFT(Y) * Filter$  in [Figure 6-36](#) are of the form  $=IMPRODUCT(F5,G5)$ , where the corresponding values in columns  $DFT(Y)$  and  $Filter$  are used as arguments. The results are complex numbers (text values) contained in the column  $DFT(Y) * Filter$ .

At this point, you can construct another power spectrum based on these filtered results to see the relative strength of the frequency components. I generated the filtered power spectrum shown in [Figure 6-38](#) using the same techniques discussed earlier for constructing a power spectrum.

Figure 6-38. Filtered power spectrum

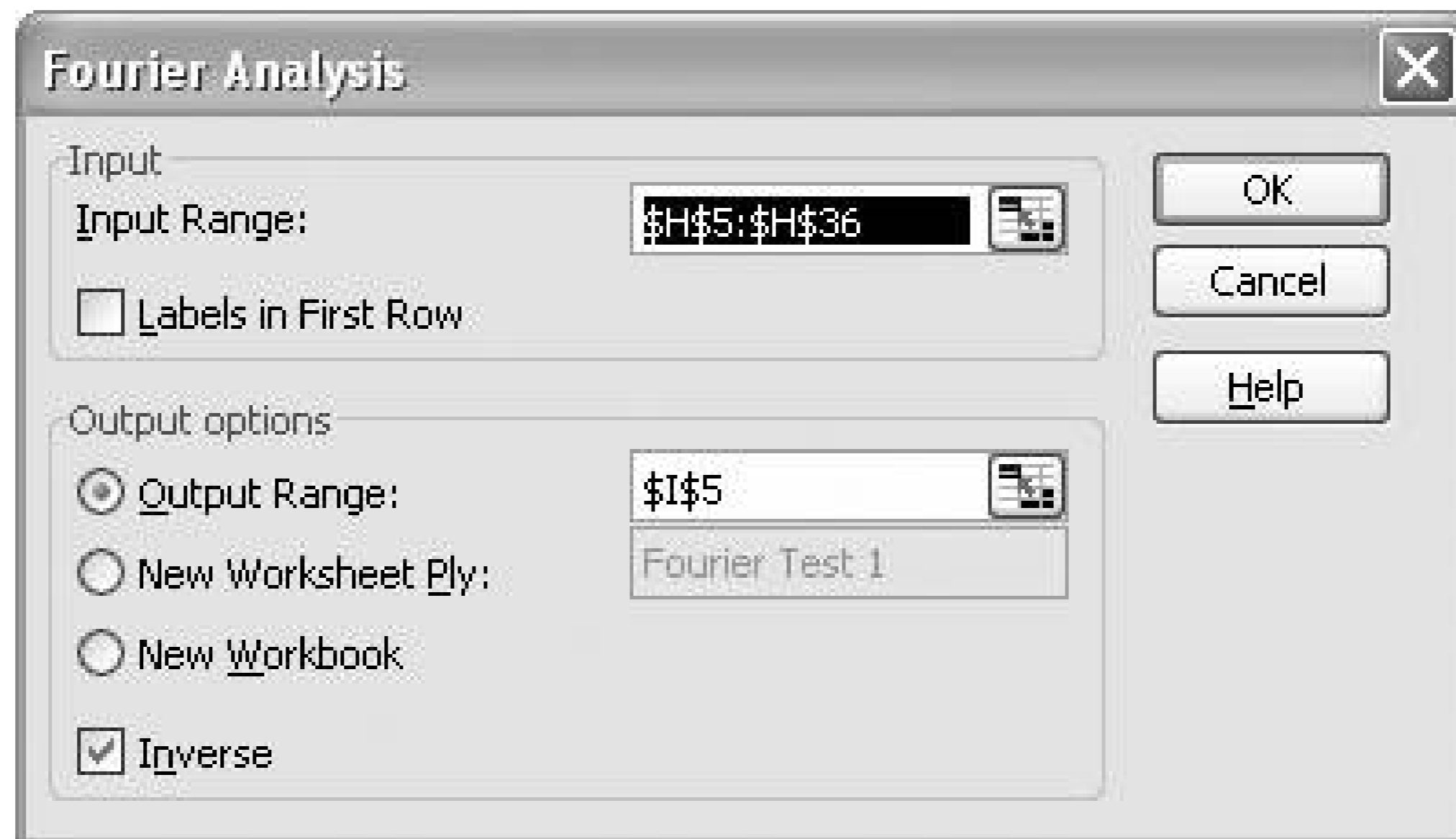


As you can see, this filter effectively isolates the 0.637 Hz frequency component. To see the filtered results back in the time domain, you have to perform an inverse transform.



Open the Fourier Analysis tool as explained earlier. Then, referring to [Figure 6-36](#), select the cell range containing the column labeled  $DFT(Y) * Filter$  as the input cell range. Select cell I5 as the output cell. Since this is an inverse transform, be sure to check the Inverse checkbox. These settings are shown in [Figure 6-39](#).

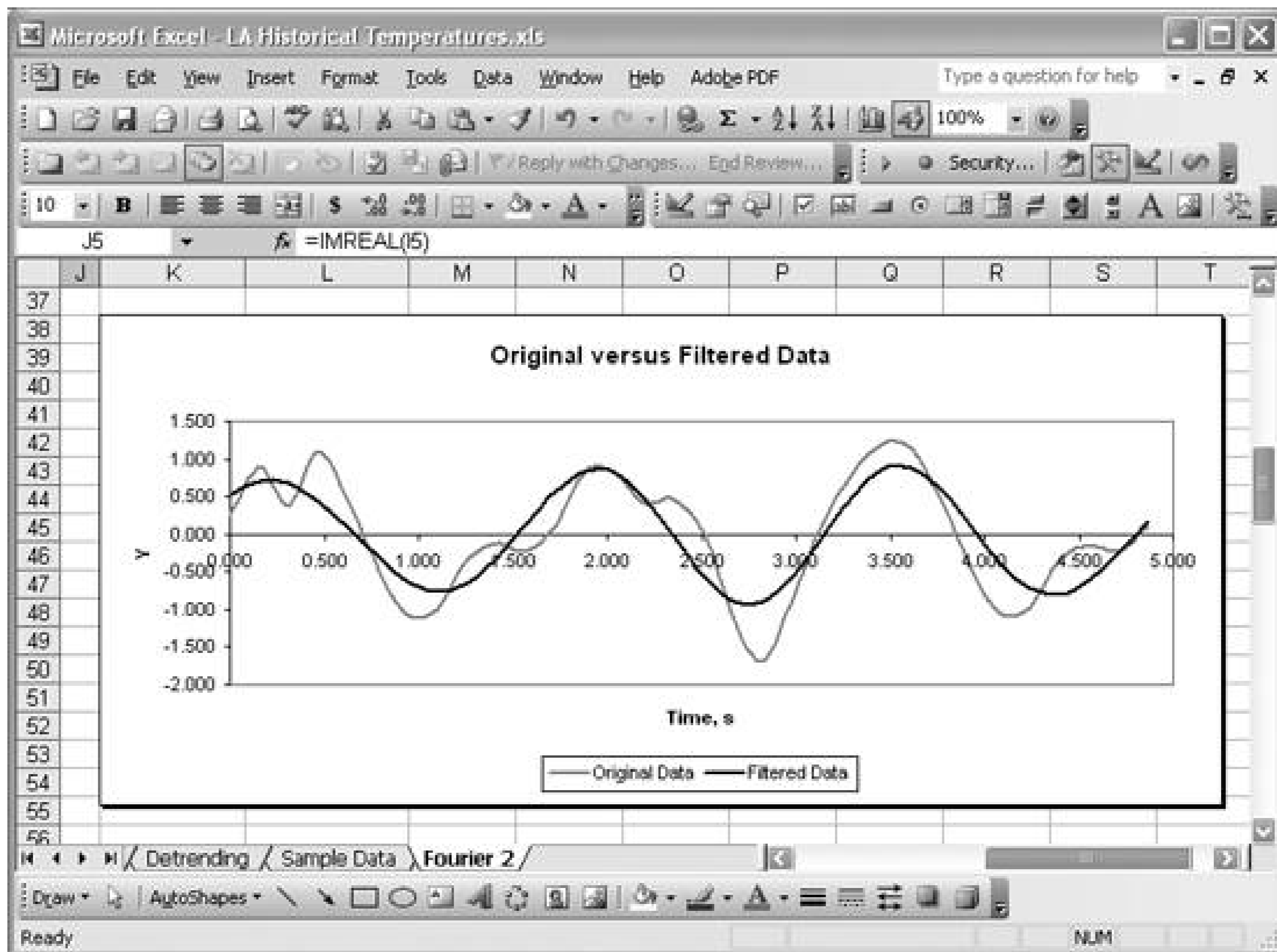
Figure 6-39. Inverse transform settings



Press OK when you're done and you should see results that look like those shown in column  $Inverse DFT(Y)$  in [Figure 6-36](#). The results of the inverse transform are real numbers in this example; however, Excel formatted them as text (i.e., as the real part of an imaginary number). Use the `IMREAL` function to convert these imaginary numbers to values you can use. Column  $Yf(t)$  in [Figure 6-36](#) contains the results. The cell formulas in that column are of the form `=IMREAL(I5)`.

Now you can plot these filtered results and see how they compare to the original time series. [Figure 6-40](#) shows the comparison.

Figure 6-40. Original versus filtered data



The bold curve shows the filtered time series, where it's clear the high frequency components have been removed.

You can easily replace the band pass filter used in this example with a low- or high-pass filter. For example, the following cell formula creates a low-pass filter: `=if(ABS(Ef)<fo, 1, EXP(-(((ABS(E5)-fo)/sig)^2)))`. Using cell formulas in this manner allows you to construct a wide variety of filters. You can use these same techniques to create window functions if your analysis requires windowing. You can create even more elaborate functions using VBA and call them directly from cell formulas, just as we did the weighted average function in [Recipe 6.4](#).

# Chapter 7. Mathematical Functions

[Section 7.0. Introduction](#)

[Recipe 7.1. Using Summation Functions](#)

[Recipe 7.2. Delving into Division](#)

[Recipe 7.3. Mastering Multiplication](#)

[Recipe 7.4. Exploring Exponential and Logarithmic Functions](#)

[Recipe 7.5. Using Trigonometry Functions](#)

[Recipe 7.6. Seeing Signs](#)

[Recipe 7.7. Getting to the Root of Things](#)

[Recipe 7.8. Rounding and Truncating Numbers](#)

[Recipe 7.9. Converting Between Number Systems](#)

[Recipe 7.10. Manipulating Matrices](#)

[Recipe 7.11. Building Support for Vectors](#)

[Recipe 7.12. Using Spreadsheet Functions in VBA Code](#)

[Recipe 7.13. Dealing with Complex Numbers](#)

[← PREV](#)

## 7.0. Introduction

[Recipe 1.10](#) introduced you to Excel's built-in mathematical functions. These functions include things such as taking square roots, computing sines and cosines, inverting matrices, and performing summations. Excel's mathematical functions are pretty much the bread and butter of many of the computations discussed throughout this book. You can think of these as calculator functions, in that they offer the same functionality you'd expect from a scientific calculator. In this regard, these functions are essential for scientific and engineering calculations in Excel. This chapter covers many of Excel's built-in mathematical functions; others are treated in different chapters where they seem to fit better in specific contexts.

[← PREV](#)

## Recipe 7.1. Using Summation Functions

### Problem

You're performing calculations that require summations of large amounts of data (for example, for least-squares curve fitting) and would like to use built-in functions that make such summation computations easy.

### Solution

Use Excel's built-in summation functions such as `SUM`, `SUMSQ`, `SUMPRODUCT`, and `SUMX2MY2`.

### Discussion

Excel includes several built-in functions that make performing summations of large (or small) amounts of data very easy. Perhaps the most common sum function is `SUM`, which simply adds all values contained in a range of cells. For example, `=SUM(B2:B24)` adds all the values contained in cells B2 to B24. The range of cells does not have to be contiguous either. For example, `=SUM(B2:B24,C2:C24)` adds all the values contained in each cell range (you separate cell ranges with commas). You can even include explicit values like this `=SUM(B2:B24,1)`. In this case, the values contained in the range plus the specified value, 1, are added.

Most of Excel's functions that take cell ranges or multiple values as arguments allow you to mix cell references and values, using commas as separators.

[Table 7-1](#) contains a list with short descriptions of several of Excel's convenient summation functions.

Table 7-1. Common summation functions

Function	Syntax	Description
<code>SUM</code>	<code>=SUM(<i>n1</i>, <i>n2</i>, <i>n3</i>, ...)</code>	Returns the sum of all given numbers, which may include cell ranges and explicit numbers.

Function	Syntax	Description
SUMPRODUCT	<code>=SUMPRODUCT(array1, array2, ...)</code>	Adds the products of corresponding values in given cell ranges (called <i>arrays</i> ). The ranges must have the same number of elements.
SUMSQ	<code>=SUMSQ(n1, n2, n3, ...)</code>	Returns the sum of the squares of given numbers, which may include cell ranges and explicit numbers.
SUMX2MY2	<code>=SUMX2MY2(array_x, array_y)</code>	Returns the sum of the difference of squares of values; i.e., $(x^2 - y^2)$ .
SUMX2PY2	<code>=SUMX2PY2(array_x, array_y)</code>	Returns the sum of the sum of squares of values; i.e., $(x^2 + y^2)$ .
SUMXMY2	<code>=SUMXMY2(array_x, array_y)</code>	Returns the sum of squares of differences of values; i.e., $(x - y)^2$ .

## See Also

Excel's Insert Function tool makes it easy to browse for built-in functions. For more information, see [Recipe 1.10](#).

Excel also contains a family of functions that allow you to perform conditional sums, and other operations, on data. For example, you can sum all the values in a range of database-type records that meets a specific criterion. These functions are discussed in [Chapter 5](#).

## Recipe 7.2. Delving into Division

### Problem

You already know how to perform normal division using the `/` operator, but need to perform other division operations, such as modulo division.

### Solution

Use Excel's various division functions, which are summarized in [Table 7-2](#).

Table 7-2. Common division functions

Function	Syntax	Description
MOD	<code>=MOD(<i>numerator</i>,<i>denominator</i>)</code>	Returns the remainder of <i>numerator</i> / <i>denominator</i> .
QUOTIENT	<code>=QUOTIENT(<i>numerator</i>,<i>denominator</i>)</code>	Returns the integer part of <i>numerator</i> / <i>denominator</i> .
GCD	<code>=GCD(<i>n1</i>, <i>n2</i>, <i>n3</i>, ...)</code>	Returns the greatest common divisor of the given integer numbers.

### See Also

Excel also supports division of complex numbers. See [Recipe 7.13](#) for more information. Take a look at [Recipe 7.3](#) for a neat little trick you can do in Excel to quickly scale or shift a selected set of data.

## Recipe 7.3. Mastering Multiplication

### Problem

You already know how to perform normal multiplication using the \* operator, but need to perform other multiplication operations.

### Solution

Use Excel's various multiplication functions, which are summarized in [Table 7-3](#).

Table 7-3. Common multiplication functions

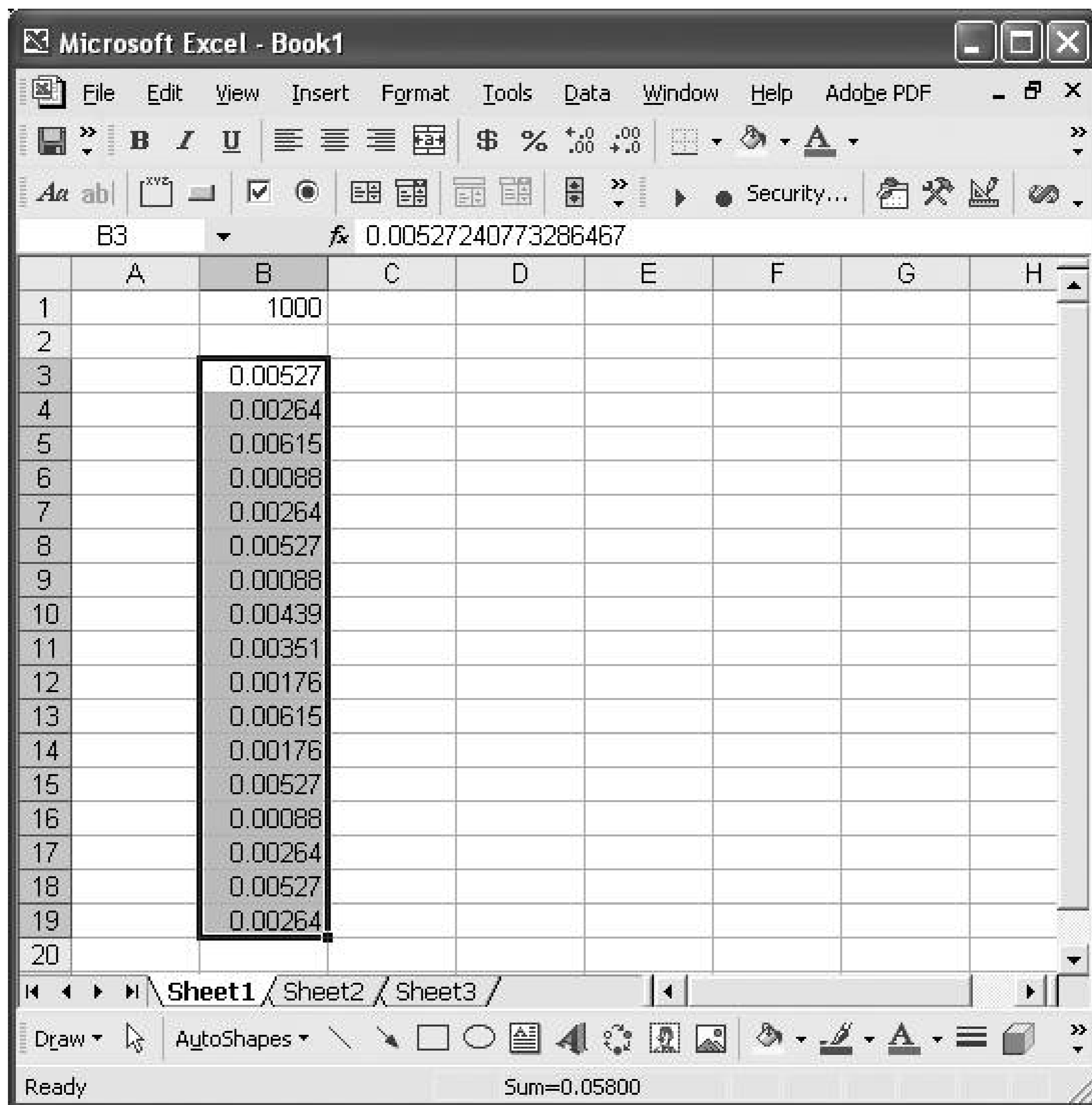
Function	Syntax	Description
PRODUCT	=PRODUCT( <i>n1</i> , <i>n2</i> , <i>n3</i> , ...)	Returns the product of all given numbers, which may include cell ranges and explicit numbers.
LCM	=LCM( <i>n1</i> , <i>n2</i> , <i>n3</i> , ...)	Returns the least common multiple of the given integer numbers.
SUMPRODUCT	=SUMPRODUCT( <i>array1</i> , <i>array2</i> , ...)	Adds the products of corresponding values in given cell ranges (called <i>arrays</i> ). The ranges must have the same number of elements.

### Discussion

Excel provides a handy utility to multiply a range of numbers by some value in one fell swoop. For example, say you have a range of data like that shown in [Figure 7-1](#), which you'd like to scale by a factor of 1,000.

Figure 7-1. Data to be scaled





Follow these steps to quickly scale the data in the range B3:B19 by 1,000. Type the scale factor, 1000 in this case, in a cell (in [Figure 7-1](#), I put it in cell B1). Select the cell containing the scale factor. Select Edit → Copy from the main menu bar. Now, with the mouse, select the range containing the data to be scaled (range B3:B19 in this example). From the main menu bar, select Edit → Paste Special... to open the Paste Special dialog box shown in [Figure 7-2](#).

Figure 7-2. Paste Special dialog box



Now select the Multiply option and press the OK button. The selected data will be multiplied by the scale factor in place; i.e., the scaled data will replace the original data.

Excel will revise the formulas to include the scale factor if the cells to be scaled contain formulas instead of values. For example, if cell B3 in the example discussed here contained the formula  $=A3+A4$ , then after scaling it would contain the formula  $=(A3+A4)*1000$ .

You'll notice from [Figure 7-2](#) that Paste Special also provides other operations including adding, subtracting, and dividing. You can use the same technique with these other operations to conveniently shift or scale data.

## See Also

Excel provides support for matrix multiplication and complex number multiplication. See [Recipes 7.10](#) and [7.13](#) for more information.

## Recipe 7.4. Exploring Exponential and Logarithmic Functions

### Problem

You need to compute logarithms in your calculations but aren't familiar with Excel's support for such calculations.

### Solution

Excel provides nine built-in functions for computing logarithms and one exponential function (excluding the special class of exponential functions known as hyperbolic functions, for which Excel provides six built-in functions). A few of these logarithm functions are related to the log-normal probability distribution, while others are related to complex numbers. See [Chapter 6](#) for recipes dealing with probability and check out [Recipe 7.13](#) for dealing with complex numbers. Read the following discussion to learn about the remaining logarithmic functions.

### Discussion

[Table 7-4](#) shows several of Excel's built-in logarithmic and exponential functions.

Table 7-4. Common logarithmic and exponential functions

Function	Syntax	Description
LOG	=LOG( <i>n</i> , <i>base</i> )	Returns the log of the number <i>n</i> to the specified <i>base</i>
LOG10	=LOG10( <i>n</i> )	Returns the log of the number <i>n</i> to the base 10
LN	=LN( <i>n</i> )	Returns the natural logarithm of <i>n</i>
EXP	=EXP( <i>n</i> )	Returns $e^n$
POWER	=POWER( <i>n</i> , <i>p</i> )	Returns $n^p$ and is the same as the syntax $n^p$ using the ^ operator
SINH	=SINH( <i>n</i> )	Returns $(e^n - e^{-n})/2$
COSH	=COSH( <i>n</i> )	Returns $(e^n + e^{-n})/2$
TANH	=TANH( <i>n</i> )	Returns $(e^n - e^{-n}) / (e^n + e^{-n})$

Function	Syntax	Description
ASINH	=ASINH( <i>n</i> )	Returns the inverse hyperbolic sine, $\ln(n + \sqrt{n^2 + 1})$ , where <i>n</i> is any real number
ACOSH	=ACOSH( <i>n</i> )	Returns the inverse hyperbolic cosine, $\ln(n + \sqrt{n^2 - 1})$ ; <i>n</i> must be greater than 1
ATANH	=ATANH( <i>n</i> )	Returns the inverse hyperbolic tangent, $\ln((1+n)/(1-n))/2$ ; <i>n</i> must be between -1 and 1

These functions pretty much do what you'd expect functions related to logarithms and natural logarithms to do. You may wonder why there's a LOG10 function when the LOG function with a specified base of 10 performs the same computation. Moreover, if you leave the base out of a call to LOG, then base 10 is assumed. It seems redundant, but perhaps some people prefer typing LOG10(*n*) to typing LOG(*n*,10) or even LOG(*n*).

Excel also provides the exponential function EXP(*n*) to compute  $e^n$ , which is of course the inverse of the natural logarithm function.

Aside from straightforward logarithmic and exponential computations, these functions are useful when curve fitting. For example, you may have data through which you'd like to fit a logarithmic curve. You could use Excel's built-in logarithmic curve-fitting features (see [Chapter 8](#)) or you could transform the data and use Excel's linear curve-fitting functions. For example, LINEST(*y values*, LOG(*x values*)) uses Excel's linear curve-fitting function on the transformed data, which results in a logarithmic curve fit. Moreover, sometimes it's more convenient or insightful to plot the log of a dataset instead of the dataset in its original form. You can use these functions to transform your data as desired.

Excel also has a function called POWER that you may find useful. The syntax for POWER is =POWER(*n*, *power*), where *n* is a given number and *power* is the power to which *n* will be raised. You can also use the  $\wedge$  operator to raise a number to a power, as discussed in [Recipe 1.9](#). Conversely, you can use POWER to reverse a LOG operation on a value. For example =POWER(10, LOG(3)) simply returns a value of 3.

This all seems straightforward enough, and it is, but there's potential for confusion when working with Excel's built-in logarithmic functions and the Log function available in VBA (Visual Basic for Applications; see [Chapter 2](#) for a quick introduction). VBA provides the function Log(*n*), which returns the *natural logarithm* of the given number *n*, not the base 10 logarithm as you might expect. This clearly is not consistent with Excel's built-in logarithmic functions, where LOG means base 10 log and LN means natural log.

VBA does not provide another function for computing the base 10 logarithm of a number (or any other base, for that matter). However, when working in VBA you can compute the logarithm of a number to any base by taking the ratio of the natural logarithm of the number to the natural logarithm of the desired base. For example, to compute the base 10 logarithm of *n*, you'd write `Logn = Log(n) / Log(10.0)`, where Logn is just a variable you'd define to store the result (you can name it whatever you want, of course).

## See Also

Take a peek at [Recipe 7.13](#) for more information on Excel's support for taking logarithms of complex numbers.



## Recipe 7.5. Using Trigonometry Functions

### Problem

You need to become familiar with Excel's support for trigonometric functions.

### Solution

Explore the trigonometric functions summarized in [Table 7-5](#).

Table 7-5. Trigonometric functions

Function	Syntax	Description
SIN	=SIN( <i>angle</i> )	Returns the sine of the given <i>angle</i> , where <i>angle</i> is in radians.
COS	=COS( <i>angle</i> )	Returns the cosine of the given <i>angle</i> , where <i>angle</i> is in radians.
TAN	=TAN( <i>angle</i> )	Returns the tangent of the given <i>angle</i> , where <i>angle</i> is in radians.
ASIN	=ASIN( <i>n</i> )	Returns the inverse sine of the given number, <i>n</i> , where <i>n</i> must be within the range -1 to 1. The returned angle is in radians and within the range $\pi/2$ to $\pi/2$ .
ACOS	=ACOS( <i>n</i> )	Returns the inverse cosine of the given number, <i>n</i> , where <i>n</i> must be within the range -1 to 1. The returned angle is in radians and within the range 0 to $\pi$ .
ATAN	=ATAN( <i>n</i> )	Returns the inverse tangent of the given number, <i>n</i> . The returned angle is in radians and within the range $-\pi/2$ to $\pi/2$ .

Excel's function `PI( )` returns the value of  $\pi$  (i.e., 3.14159265358979...), which you might find useful in some calculations.

### Discussion

The trigonometric functions take or return angles in radians. However, often degrees are more convenient to work with. Excel provides two functions making conversions between radians and

degrees easy: `RADIANS` and `DEGREES`. Use `RADIANS(angle)` to convert an angle from degrees to radians. Use `DEGREES(angle)` to convert an angle from radians to degrees. For example, `=COS(RADIANS(45))` returns the cosine of the angle 45 degrees.





## Recipe 7.6. Seeing Signs

### Problem

You want to capture only the sign of an expression, not its value. Or you want to take the absolute value of an expression.

### Solution

Use the `SIGN` function in the former case and the `ABS` function in the latter case.

### Discussion

Situations often arise in scientific calculations when you're more interested in the sign of an expression than in the resulting value. For example, the formula

```
=F3-E8 / (ABS(G2/F2)) * ABS(G2/F2)^0.1 * SIGN(G2-F2)
```

uses the `SIGN` function to compute the algebraic sign of the expression `(G2-F2)` and multiplies the rest of the formula by that sign (as a unit value). If `(G2-F2)` evaluates to a negative value, then the formula is multiplied by -1; if `(G2-F2)` evaluates to a positive number, then it is multiplied by +1 instead.

In this particular case, the expression `(G2-F2)` represents some change in value of a dimensional parameter and the direction of that change (i.e., positive or negative) is used as a scale factor to scale some other parameter from a baseline using an empirical formula. In other examples, you may be interested in whether some value is greater or less than zero, which you can determine by checking the sign of the value. The `SIGN` function allows you to perform such a check very easily and use the result in other computations or logical functions like the `IF` function.

You'll notice that in the example formula discussed a moment ago, I also used the absolute value function, `ABS`. The syntax for this function is `=ABS(value or expression)`, where you can supply a value, cell reference, or some mathematical expression.



[← PREVIOUS](#)

## Recipe 7.7. Getting to the Root of Things

### Problem

You need to compute the square root of an expression. Moreover, you'd like to know how to compute other roots, like the cube root, of an expression.

### Solution

Use the `SQRT` function for taking square roots and use the `POWER` function or the `^` operator to handle other roots.

### Discussion

The syntax for the square root function is simply `=SQRT(n)`, where *n* is a number or an expression containing cell references that evaluates to a number. `SQRT` returns the positive square root of *n*.

Excel does not provide other specific functions for taking cube roots and so on; however, you can take such roots using the `POWER` function or the `^` operator, which performs the same function as the `POWER` function. To take the cube root of a number, use `=POWER(n, 1/3)` or `n^(1/3)`. Taking the cube root of a number is the same as raising the number to the power 1/3. So for any other roots you may find in your calculations, you can raise a number to the reciprocal of the desired root using `POWER` or `^`.

### See Also

See [Recipe 7.4](#) for more information related to the `POWER` function. Also, take a look at [Recipe 1.9](#) for more on `^` and other operators.

## Recipe 7.8. Rounding and Truncating Numbers

### Problem

Your calculations call for you to either round or truncate values in your spreadsheet.

### Solution

Use one or more of the built-in functions shown in [Table 7-6](#).

Table 7-6. Excel functions for rounding and truncating

Function	Syntax	Description
ROUND	=ROUND( <i>n</i> , <i>digits</i> )	Rounds the number <i>n</i> to the specified number of <i>digits</i>
ROUNDUP	=ROUNDUP( <i>n</i> , <i>digits</i> )	Rounds the number <i>n</i> up to the specified number of <i>digits</i>
ROUNDDOWN	=ROUNDDOWN( <i>n</i> , <i>digits</i> )	Rounds the number <i>n</i> down to the specified number of <i>digits</i>
MROUND	=MROUND( <i>n</i> , <i>multiple</i> )	Rounds the number <i>n</i> to the nearest <i>multiple</i>
CEILING	=CEILING( <i>n</i> , <i>multiple</i> )	Rounds the number <i>n</i> up to the nearest <i>multiple</i>
EVEN	=EVEN( <i>n</i> )	Rounds the number <i>n</i> to the nearest even integer
ODD	=ODD( <i>n</i> )	Rounds the number <i>n</i> to the nearest odd integer
INT	=INT( <i>n</i> )	Rounds the number <i>n</i> to the nearest integer
trUNC	=trUNC( <i>n</i> , <i>digits</i> )	Truncates the number <i>n</i> at the specified number of digits

### Discussion

In Excel you can format any cell displaying a value to display that value to a specified number of decimal places. For example, you can format a cell containing the value 4.5837450 to display the number to only two decimal places (that is, 4.58). Formatting a cell to display only a certain number of decimal places does not change the underlying value; it only changes what's shown in the cell. To actually change the value, you need to use the rounding or truncating functions shown in [Table 7-6](#).

I pretty much use the `ROUND` function as my workhorse for rounding numbers. For example, to round the value 3.213 to one decimal place use `=ROUND(3.213, 1)`; this returns 3.2. To round a value to the nearest integer, specify 0 for the number of digits. For example, to round 3.213 to the nearest

integer use `=ROUND(3.213, 0)` this returns 3.

You can also use `ROUND` to round a value to the nearest significant digit to the left of the decimal place. To do so, specify a negative number of digits. For example, `=ROUND(121.986, -1)` returns a value of 120. The formula `=ROUND(12839.31, -3)` returns a value of 13,000.



## Recipe 7.9. Converting Between Number Systems

### Problem

You need to convert numbers between number systems, for example, from hexadecimal to decimal.

### Solution

Use Excel's number system conversion functions such as `HEX2DEC` and `DEC2HEX`.

### Discussion

When debugging software, I use the heck out of hexadecimal to decimal (and vice versa) conversions. Excel makes such conversions easy with its `HEX2DEC` and `DEC2HEX` functions.

The syntax for `HEX2DEC` is `=HEX2DEC(n)`, where *n* is the hexadecimal to be converted. For example, the formula `=HEX2DEC("1F0CC3")` returns the decimal value 2,034,883.

Conversions from decimal to hexadecimal use the syntax `DEC2HEX(n, digits)`, where *n* is the hexadecimal to be converted and *digits* is the number of characters to use for the resulting hexadecimal number. For example, the formula `=DEC2HEX(64000, 8)` returns the hexadecimal value 0000FA00, whereas `=DEC2HEX(64000, 4)` returns FA00.

### See Also

Excel also supports conversions to and from the binary and octal systems. For example, `OCT2HEX` converts a number from octal to hexadecimal and `DEC2BIN` converts a decimal number to binary. Check out the Excel help topic "Engineering Functions" for more information on these other number system conversion functions.

## Recipe 7.10. Manipulating Matrices

### Problem

You need to manipulate matrices in your calculations and would like to know what support Excel offers for dealing with matrices.

### Solution

Excel has several built-in matrix functions for computing matrix inverses and determinants and for multiplying or transposing matrices.

### Discussion

[Table 7-7](#) summarizes Excel's built-in matrix functions.

Table 7-7. Excel's built-in matrix functions

Function	Syntax	Description
<code>MDETERM</code>	<code>=MDETERM(<i>cell range</i>)</code>	Returns the determinant of the matrix argument, which must be a square matrix.
<code>MINVERSE</code>	<code>=MINVERSE(<i>cell range</i>)</code>	Returns the inverse of the matrix argument, which must be a square matrix.
<code>MMULT</code>	<code>=MMULT(<i>cell range 1</i>, <i>cell range 2</i>)</code>	Returns the matrix product of the two matrix arguments. The resulting matrix has the same number of rows as the first argument and the same number of columns as the second.
<code>TRANSPOSE</code>	<code>=TRANSPOSE(<i>cell range</i>)</code>	Returns the transpose of the matrix argument.

`MDETERM` returns a scalar, which means you enter the function as a normal function, by pressing Enter. The other three functions are array functions, which means you first have to select an appropriate number of cells, then type the function and press Ctrl-Shift-Enter to enter the function.

[Figure 7-3](#) shows how these functions can be used in a spreadsheet.

Figure 7-3. Matrix functions

	B	C	D	E	F	G	H
4		2	3	-4			
5	[A] =	3	-1	-2			
6		4	-7	-6			
7							
8	Det[A] =	82					
9							
10		-0.098	0.561	-0.122			
11	Inverse [A] =	0.122	0.049	-0.098			
12		-0.207	0.317	-0.134			
13							
14		2	3	4			
15	Transpose [A] =	3	-1	-7			
16		-4	-2	-6			
17							
18							
19	[M] =	1	4	3			
20		5	2	6			
21							
22		7	12				
23	[N] =	11	8				
24		9	10				
25							
26							
27	[M][N] =	78	74				
28		111	136				

To enter a matrix in a spreadsheet, just type the matrix elements in a series of cells as shown in [Figure 7-3](#). Matrix A is a 3 x 3 matrix whose elements are contained in cells C4 through E6.

To compute the determinant of A, you can enter the formula `=MDETERM(C4:E6)`. This formula was entered in cell C8 of the spreadsheet shown in [Figure 7-3](#). The result is 82.

The inverse of A is shown in cells C10 to E12. To compute the inverse, select the cell range from C10

to E12, type the formula `=MINVERSE(C4:E6)`, and press Ctrl-Shift-Enter.

You can use a similar procedure to compute the transpose of A, as shown in cells C14 to E16. The formula to enter is `=trANSPPOSE(C4:E6)`.

Cells C19 to E20 contain a 2 x 3 matrix, M. Another matrix, N, is contained in cells C22 to D24. N is a 3 x 2 matrix. To compute the matrix product MN, select a 2 x 2 range of cells and type the formula `=MMULT(C19:E20,C22:D24)`. Press Ctrl-Shift-Enter to enter the formula. The results for this example are contained in cells C27 to D28.

## See Also

Take a look at [Recipe 9.4](#) for an example application of matrix operations.



[← PREV](#)

## Recipe 7.11. Building Support for Vectors

### Problem

You want to perform vector-based calculations, like vector cross and dot products, but can't find any vector functions in Excel.

### Solution

Excel does not provide any built-in functions to perform vector calculations; however, you can add your own custom functions without too much effort.

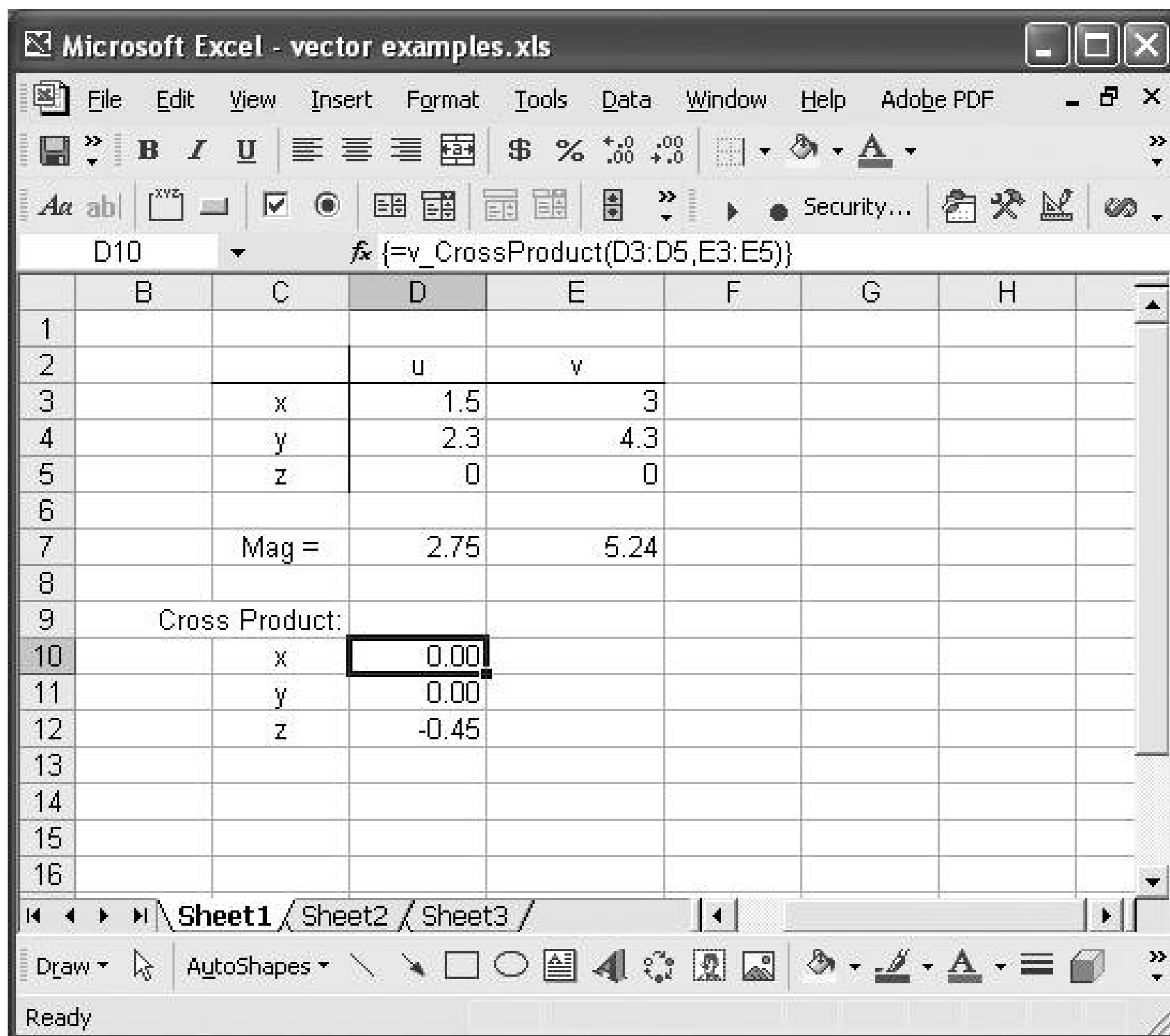
### Discussion

Since VBA allows you to write custom functions that can then be called directly from your Excel spreadsheet, you can easily add functions to perform vector calculations.

[Figure 7-4](#) shows an example spreadsheet containing two vectors,  $u$  and  $v$ , whose  $x$ ,  $y$ , and  $z$  components are stored in separate cells.

Figure 7-4. Vector example





Cells D7 and E7 show the magnitudes of these vectors. These magnitudes are computed using a custom function called `v_Mag`. The syntax for `v_Mag` is `=v_Mag(cell range)`, where `cell range` is a cell reference consisting of three cells containing the vector components, `x`, `y`, and `z`. [Example 7-1](#) shows the VBA code for `v_Mag`. (See [Recipe 2.2](#) for more information on writing VBA functions.)

### Example 7-1. `v_Mag`

```

Public Function v_Mag(v As Range) As Double
    Dim x As Double
    Dim y As Double
    Dim z As Double

    x = v.Cells(1).Value
    y = v.Cells(2).Value
    z = v.Cells(3).Value

    v_Mag = Sqr(x ^ 2 + y ^ 2 + z ^ 2)
End Function

```

The only parameter for this function is a cell range, `v`, that represents the selected cell range containing the three vector components, `x`, `y`, and `z`. The first three lines of this function declare three local variables in which to store the vector components. You really don't need to use local variables in this example, but I did so to make the remainder of the code more readable.

The next three lines set the values of the local variables to the values contained in the input cell range. Since `v` is of type `Range`, you can access its `Cells` property to retrieve the value for each cell. Each cell in the range is referred to using a 1-based array index as shown.

Finally, the last line of this function computes the magnitude of the vector and returns the result as a `Double` type scalar value.

The VBA function for square root is `Sqr`, whereas the Excel function for square root is `SQRT`. This is another example of inconsistencies between VBA and Excel function names.

Computing the vector cross product is a little more involved than simply computing the magnitude of a vector. For the cross product, we need to return the three components of the resulting vector as an array in Excel. This means we need to write a VBA function that returns an array instead of a scalar. The resulting function will be an array function that must be entered in a spreadsheet using Ctrl-Shift-Enter just like Excel's built-in array functions.

In [Figure 7-4](#), cells D10 through D12 contain an array formula called `v_CrossProduct`. This is a custom function I created to compute the vector cross product of two vectors passed in as cell ranges. If you take a look at the formula bar in [Figure 7-4](#), you'll see how this function is used; specifically, the formula is `{=v_CrossProduct(D3:D5, E3:E5)}`, where the braces indicate the function is an array function.

To enter this function, you have to select a range consisting of three contiguous cells, either vertically or horizontally oriented. The function we'll write will detect how the input cells are oriented and will return the results accordingly.



If you select a range of fewer than three contiguous cells, then only those selected will contain results. If you select a range of more than three contiguous cells, then the cells after the first three will contain #N/A.

[Example 7-2](#) shows the VBA code for `v_CrossProduct`.

## Example 7-2. `v_CrossProduct`

```
Public Function v_CrossProduct(u As Range, v As Range)
    ' Declare local variables:
    Dim ux As Double
    Dim uy As Double
    Dim uz As Double
    Dim vx As Double
    Dim vy As Double
    Dim vz As Double
    Dim ReturnArray(3)
    Dim DoTranspose As Boolean

    ' Determine whether or not the selected
    ' output range is a row or a column array:
    If Application.Caller.Rows.Count > 1 Then
        DoTranspose = True
    Else
        DoTranspose = False
    End If

    ' Get the vector components:
    ux = u.Cells(1).Value
    uy = u.Cells(2).Value
    uz = u.Cells(3).Value

    vx = v.Cells(1).Value
    vy = v.Cells(2).Value
    vz = v.Cells(3).Value

    ' Compute the cross product:
    ReturnArray(0) = (uy * vz - uz * vy)
    ReturnArray(1) = (uz * vx - ux * vz)
    ReturnArray(2) = (ux * vy - uy * vx)

    ' If the selected output range is a column of cells then transpose the result:
    If DoTranspose Then
        v_CrossProduct = Application.WorksheetFunction.Transpose(ReturnArray)
    Else
        v_CrossProduct = ReturnArray
    End If
End Function
```

```
End Function
```

`v_CrossProduct` takes two parameters. Both are cell ranges that must contain three contiguous cells, which themselves contain the `x`, `y`, and `z` components for each vector. The first six lines in this function declare six local variables to store the components for each input vector.

The second to last local variable, called `ReturnArray`, is a three-element array that is used to store the result of the vector cross product operation. The final local variable, `DoTranspose`, is a `Boolean` flag that is used to control whether or not the resulting vector is transposed upon output.

The next commented block of code checks to see whether the selected output range is a row vector or a column vector. If the selected output range is a column vector, then its row count will be greater than 1 and we'll have to transpose the resulting vector. Otherwise, the selected output range is a row vector and we don't have to transpose the result.

The next block of code stores the components of the input vectors in local variables. Then it computes the cross product with each component of the resulting vector stored in the three elements of the local array `ReturnArray`, which we declared to store the result.

The final step is to check the `DoTranspose` flag. If it is `true`, then `ReturnArray` must be transposed using the `transpose` worksheet function. If it is false, you can simply return the `ReturnArray` without further modification.

The techniques discussed here can be used to construct custom functions to perform all the usual vector operations.

← PREV

## Recipe 7.12. Using Spreadsheet Functions in VBA Code

### Problem

You're writing a VBA function and want to call a built-in Excel function from within VBA code.

### Solution

Use the `WorksheetFunction` object to gain access to Excel's functions from within VBA code. Read [Recipe 2.13](#) for more information on the `WorksheetFunction` object.

← PREV

## Recipe 7.13. Dealing with Complex Numbers

### Problem

You need to perform complex algebra (e.g., when working on electrical engineering problems involving alternating current), and would like to know how Excel supports complex numbers so you can use them in your spreadsheets.

### Solution

Excel supports complex numbers in the form  $a + bi$  or  $a + bj$  ( $j$  is often used to indicate a complex number instead of  $i$ , as is common in electrical engineering applications). Further, Excel provides 18 built-in functions that can be used to manipulate complex numbers and perform complex algebra.

You need to have the Analysis ToolPak loaded in order to use Excel's complex number functions. See the introduction to [Chapter 5](#) for instructions on loading the Analysis ToolPak.

### Discussion

In Excel, a complex number is entered into a cell in the form  $a + bi$  or  $a + bj$ . In reality, Excel treats complex numbers entered this way as text. To enter a complex number, simply type it in as you would text. For example, to enter the complex number  $6 + 2i$ , simply select a cell, type in `6+2j` and then press Enter. Or you can use the `COMPLEX` function to let Excel format the complex number, given the real and imaginary parts you provide. For example, entering the formula `=COMPLEX(6,2)` in a cell results in the cell displaying the text `6+2i`.

Excel provides a suite of special-purpose imaginary number functions to manipulate complex numbers in these forms. There are 18 functions that are designed to work specifically with complex numbers. All but one of these functions start with the prefix `IM` (for example, `IMABS`, `IMPRODUCT` and so on). The one function that does not start with the `IM` prefix is `COMPLEX`. These and the other complex number functions are summarized in [Table 7-8](#).

Table 7-8. Complex number functions

Function	Syntax	Description
COMPLEX	=COMPLEX( <i>real part</i> , <i>imaginary part</i> )	Takes the real and imaginary arguments and returns text representing the complex number
IMABS	=IMABS( <i>complex number</i> )	Returns the modulus of the complex number
IMAGINARY	=IMAGINARY( <i>complex number</i> )	Returns the imaginary part of a complex number
IMARGUMENT	=IMARGUMENT( <i>complex number</i> )	Returns the argument $\theta$ , in radians, of the complex number, where $\theta$ is from the exponential form of complex numbers, $r e^{j\theta}$
IMCONJUGATE	=IMCONJUGATE( <i>complex number</i> )	Returns the complex conjugate of a complex number
IMCOS	=IMCOS( <i>complex number</i> )	Returns the cosine of a complex number
IMDIV	=IMDIV( <i>n1</i> , <i>n2</i> )	Returns the quotient of two complex numbers where the first argument is the numerator and the second is the denominator
IMEXP	=IMEXP( <i>complex number</i> )	Returns the exponential of a complex number
IMLN	=IMLN( <i>complex number</i> )	Returns the natural logarithm of a complex number
IMLOG10	=IMLOG10( <i>complex number</i> )	Returns the base 10 logarithm of a complex number
IMLOG2	=IMLOG2( <i>complex number</i> )	Returns the base 2 logarithm of a complex number
IMPOWER	=IMPOWER( <i>complex number</i> , <i>n</i> )	Raises the complex number to the <i>n</i> th power
IMPRODUCT	=IMPRODUCT( <i>n1</i> , <i>n2</i> , <i>n3</i> , ...)	Returns the product of two or more complex numbers
IMREAL	=IMREAL( <i>complex number</i> )	Returns the real part of a complex number
IMSIN	=IMSIN( <i>complex number</i> )	Returns the sine of a complex number
IMSQRT	=IMSQRT( <i>complex number</i> )	Returns the square root of a complex number
IMSUB	=IMSUB( <i>n1</i> , <i>n2</i> )	Subtracts the complex number <i>n2</i> from <i>n1</i>
IMSUM	=IMSUM( <i>n1</i> , <i>n2</i> , <i>n3</i> , ...)	Returns the sum of two or more complex numbers

# Chapter 8. Curve Fitting and Regression

[Section 8.0. Introduction](#)

[Recipe 8.1. Performing Linear Curve Fitting Using Excel Charts](#)

[Recipe 8.2. Constructing Your Own Linear Fit Using Spreadsheet Functions](#)

[Recipe 8.3. Using a Single Spreadsheet Function for Linear Curve Fitting](#)

[Recipe 8.4. Performing Multiple Linear Regression](#)

[Recipe 8.5. Generating Nonlinear Curve Fits Using Excel Charts](#)

[Recipe 8.6. Fitting Nonlinear Curves Using Solver](#)

[Recipe 8.7. Assessing Goodness of Fit](#)

[Recipe 8.8. Computing Confidence Intervals](#)



[← PREV](#)

## 8.0. Introduction

A very common exercise when analyzing data is fitting a curve through that data. *Curve fitting* is the process of trying to find the curve (which is represented by some model equation) that best represents the sample data, or more specifically the relationship between the independent and dependent variables in the dataset. When the results of the curve fit are to be used for making new predictions of the dependent variable, this process is known as regression. Sometimes, you may want to fit a curve just to interpolate a set of data. At other times, you may actually want to extrapolate beyond the sample data. In still other cases, curve fitting can be used to predict parameters of some known model (an equation) given a set of observed data. In this chapter, I discuss several different techniques for performing linear and nonlinear curve fitting in Excel.

[← PREV](#)

## Recipe 8.1. Performing Linear Curve Fitting Using Excel Charts

### Problem

You'd like to generate a best-fit straight line for a set of data.

### Solution

Use Excel's chart trendline feature to perform a linear curve fit of your data. Plot your data using an XY scatter chart (see [Chapter 4](#)). Once your chart has been created, right-click on the data series and select Add Trendline from the pop-up menu.

### Discussion

Linear curve fits are easily generated using the trendline feature built into Excel's XY scatter chart. Once you've plotted your data using an XY scatter chart, you can generate a trendline that will be displayed on your chart, superimposed over your data. You can also include the resulting equation for the best-fit line on your chart.

Consider the data shown in [Table 8-1](#). This data represents experimentally obtained angular deflections of an angular spring resulting from prescribed applied torques.

Table 8-1. Angular spring data

Applied torque (N-m)	Measured deflection (degrees)
-1.993	-1.547
-1.411	-1.178
-1.440	-1.167
-0.443	-0.318
-0.428	-0.416
-0.444	-0.327

Applied torque (N-m)	Measured deflection (degrees)
0.016	-0.035
0.013	0.032
0.049	0.036
0.830	0.762
0.854	0.687
0.851	0.698
1.730	1.412
1.703	1.399
1.698	1.419
2.044	1.675
2.063	1.624
2.034	1.627

This data should exhibit a linear trend when plotted. (Assuming the spring obeys Hooke's law.) Indeed this is the case, as is revealed by plotting this data on an XY scatter chart. Once that chart has been created, right-click on the data series and select Add Trendline from the pop-up menu. This will open the Add Trendline dialog box shown in [Figure 8-1](#).

Excel's trendline feature allows you to add nonlinear trendlines as well. See [Recipe 8.5](#) for more information.

Select the Linear Trend/Regression type as shown. Before pressing OK, click the Options tab (see [Figure 8-2](#)).

Select "Display equation on chart" and "Display R-squared value on chart." The former will display the resulting best-fit equation on your chart, while the latter will also include the R-squared value, allowing you to assess the goodness of the fit (see [Recipe 8.7](#)). Press OK to go back to your chart and see the resulting trendline.

Figure 8-1. Add Trendline dialog box

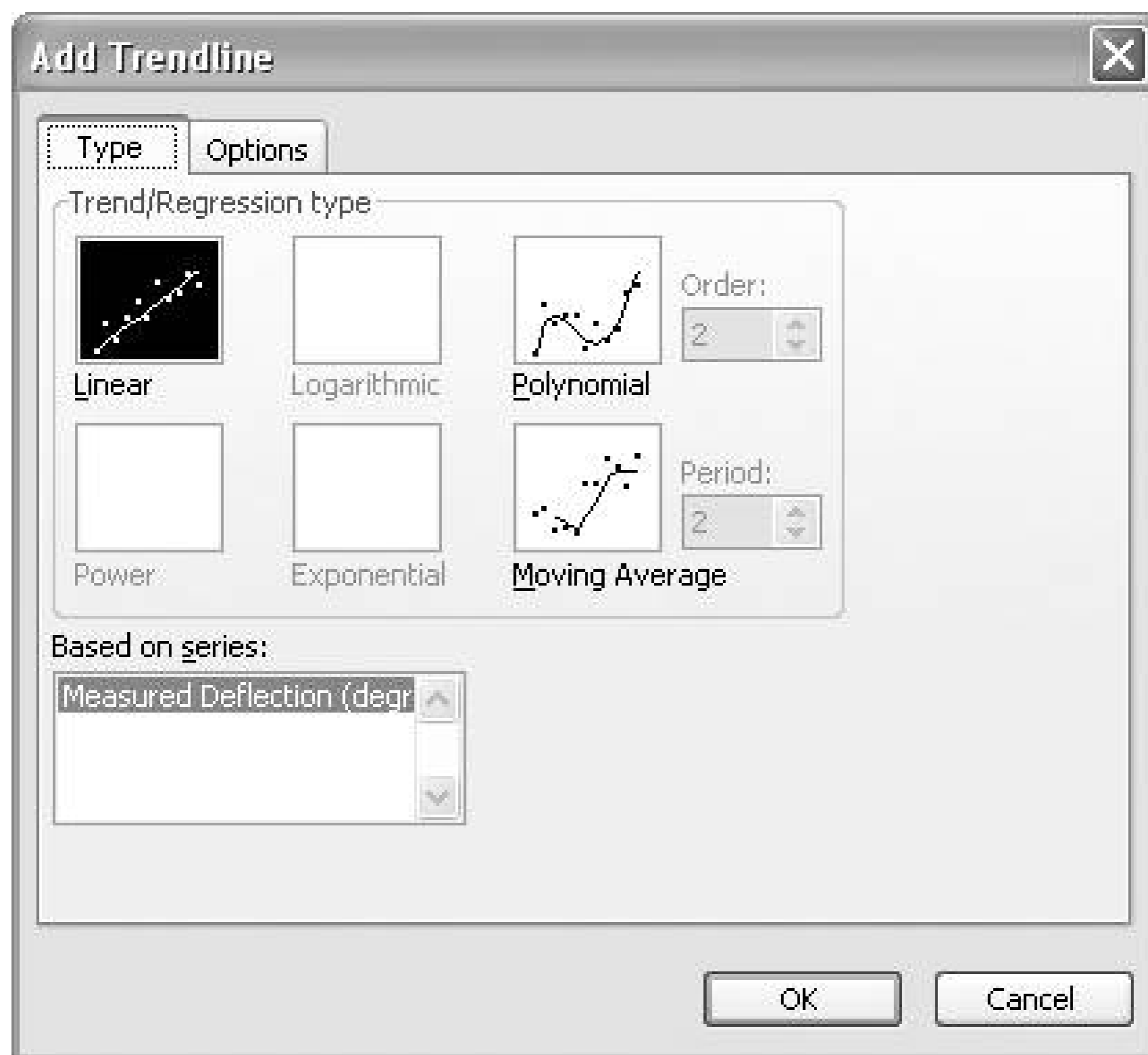
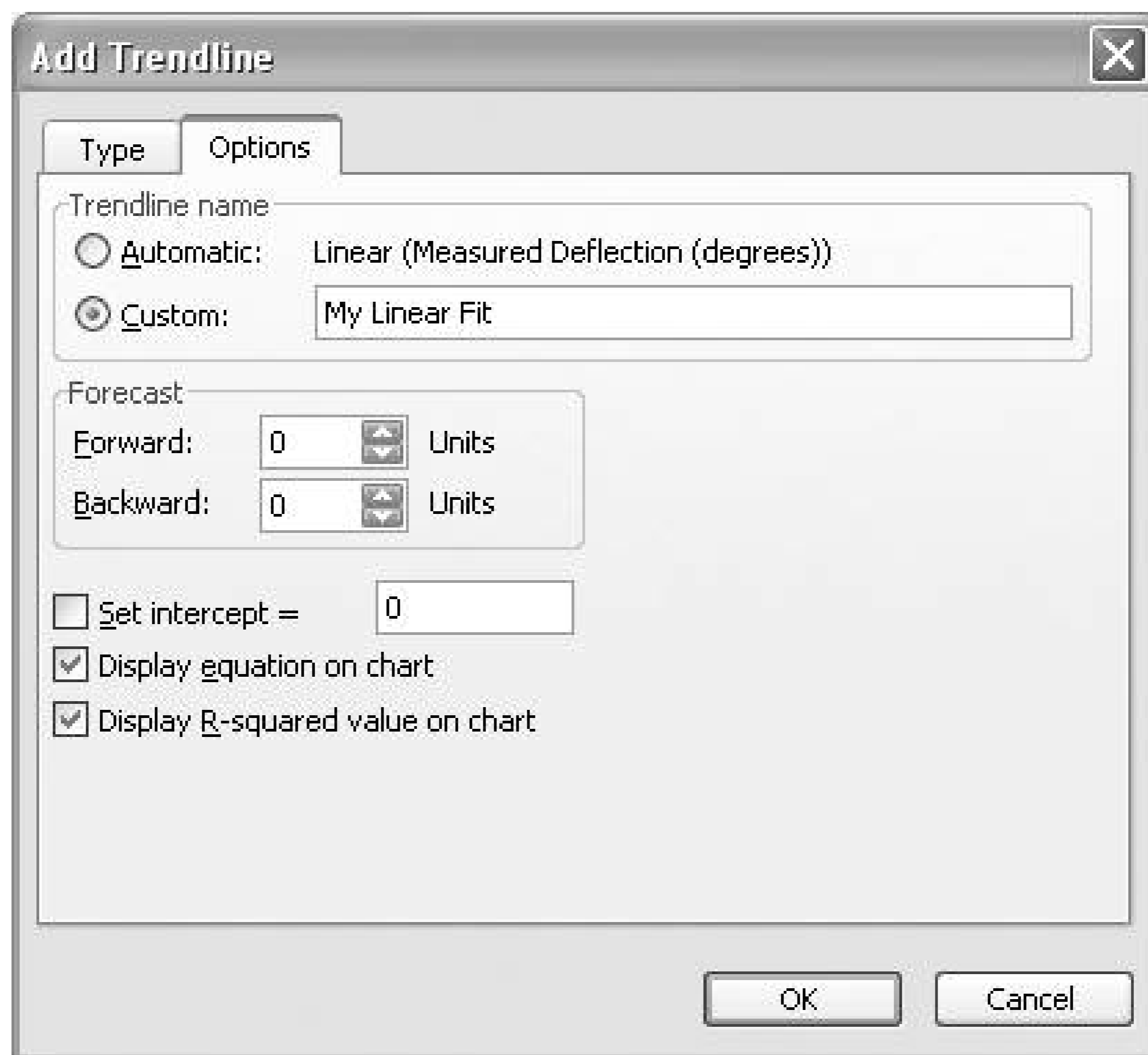
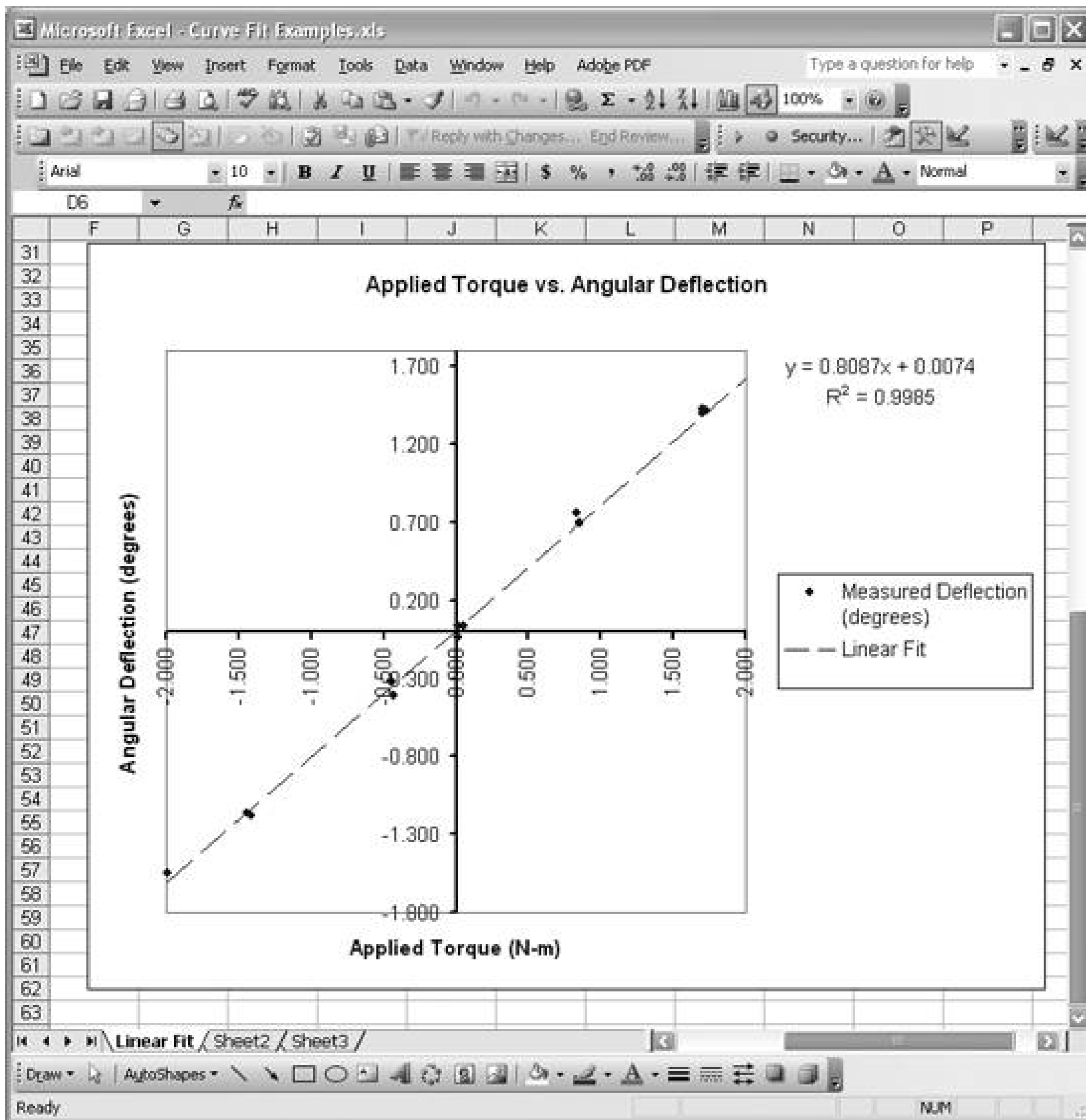


Figure 8-2. Add Trendline Options tab



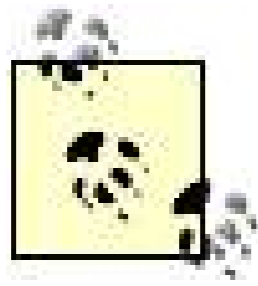
[Figure 8-3](#) shows the resulting best-fit line for the example data contained in [Table 8-1](#). The best-fit line is shown as the dashed line in [Figure 8-3](#). The original data points are shown as dots. Clearly the data follows a straight line trend very well.

Figure 8-3. Linear fit



Excel's trendline feature can give you poor results if your data consists of very large numbers that differ only slightly relative to their magnitudes (for example, if your  $x$ -values look something like 100,000,001, 100,000,001.5, 100,000,002.3, and so on). If your data looks like this, then you should center it or scale it before performing any sort of curve fit. You can easily center data by subtracting the mean value from each data point.

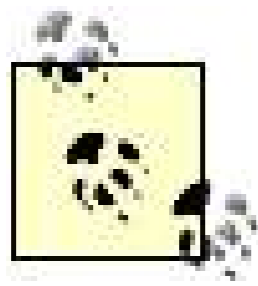
The equation for the best-fit line is displayed in the upper-right of the chart. The R-squared value for the trendline is very close to 1, indicating a good fit.



You can edit your trendline by selecting it and pressing Ctrl-1. To delete the trendline, select it and then right-click. Choose Clear from the pop-up menu to get rid of it.

In practice, you could use the slope from this best-fit line to estimate the spring constant for the spring being tested. You have to be careful with units here, because we've plotted deflection versus torque, resulting in units for the slope of degrees per N-m.

In reality, you'd probably want the reciprocal of this value, to yield a slope with units of N-m per degree, following the standard units for spring constants. You can force your best-fit line to go through the origin by specifying the intercept on the Options tab of the Add Trendline dialog box (see [Figure 8-2](#)). Perhaps a better approach would be to plot the angular deflections on the horizontal axis with the torque data on the vertical axis, and then fit a line through the points.



The trendline equation shown on the chart in [Figure 8-3](#) uses Y as the dependent variable and X as the independent variable. These correspond to angular deflection and applied torque, respectively, for the example considered here. To make your chart less confusing, you can edit the equation text shown on the chart, changing Y to angular deflection and X to applied torque. Simply select the equation and then click again to position the cursor where you'd like to start editing the text. Then make your changes. Take a look at [Figure 8-5](#) for an example.

## See Also

See [Chapter 4](#) for recipes covering creating and manipulating charts in Excel. Take a look at [Recipe 8.7](#) to learn more about the R-squared value of trendlines. Also, check out the other recipes in this chapter to learn about alternative methods of performing linear curve fitting.

[← PREV](#)

## Recipe 8.2. Constructing Your Own Linear Fit Using Spreadsheet Functions

### Problem

You want to perform a linear curve fit using standard least-squares formulas instead of Excel's linear trendline.

### Solution

Use Excel's built-in formulas such as `COUNT`, `SUM`, `SUMSQ`, and `SUMPRODUCT` to make it easy to apply the standard least-squares formulas.

### Discussion

The standard equation for a straight line is:

The standard least-squares formulas used to determine the slope,  $m$ , and intercept,  $b$ , of the fit line are:

In these equations,  $n$  is the number of data points. Excel has several built-in functions that make it very easy to compute the various sums that appear in the least-squares equations. These functions include:

#### `COUNT`

This function counts the number of cells containing numbers in a range of cells.

#### `SUM`

This function adds all the numbers in a range of cells.



### SUMSQ

This function returns the sum of squares of the numbers contained in a range of cells.

### SUMPRODUCT

This function sums the products of entries in corresponding ranges of cells.

Let's reconsider the data used in the example in [Recipe 8.1](#). Instead of using a chart trendline to determine the best-fit line, we'll use the least-squares equations and built-in Excel functions. [Figure 8-4](#) shows a simple spreadsheet I set up for this example.

The columns labeled **x** and **y** contain the same data as before. I just renamed them to make it easier to follow the application of the formulas used in this example. Cells B23 through B27 contain formulas to compute the various quantities required by the least-squares formula. The formulas in these cells are as follows:

#### *Cell B23*

The formula in this cell, `=COUNT(A4:A21)`, computes the number of data points.

#### *Cell B24*

The formula in this cell, `=SUM(A4:A21)`, computes the sum of **x**-values.

#### *Cell B25*

The formula in this cell, `=SUM(B4:B21)`, computes the sum of **y**-values.

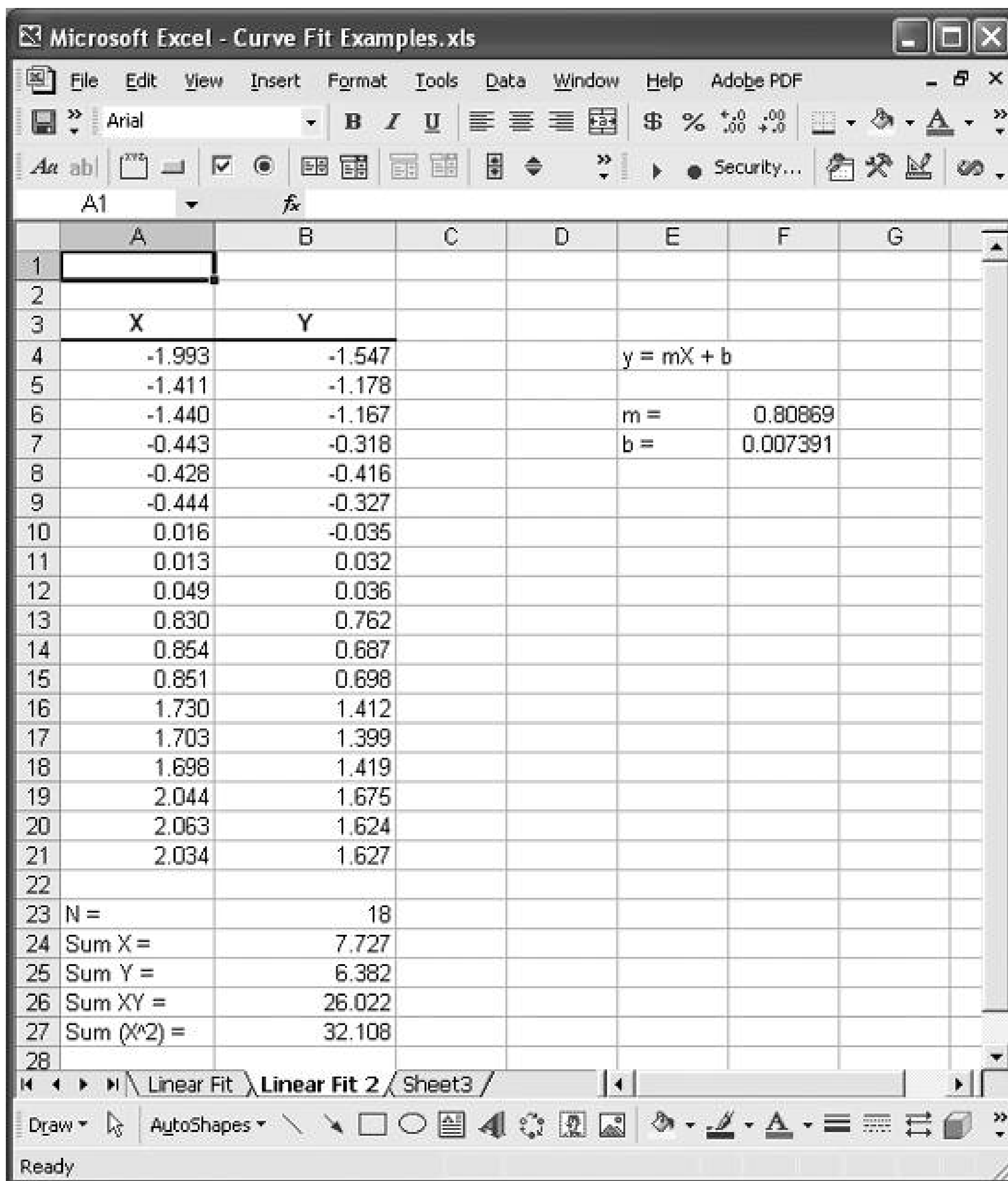
#### *Cell B26*

The formula in this cell, `=SUMPRODUCT(A4:A21,B4:B21)`, computes the sum of the products on values in the **x** and **y** columns.

#### *Cell B27*

The formula in this cell, `=SUMSQ(A4:A21)`, computes the sum of squares of **x**-values.

Figure 8-4. Least-squares fit



These cells contain all of the data we need to apply the least-squares formulas.

Cell F6 computes the slope of the best-fit line. The formula in cell F6 is  $= (B23*B26 - B24*B25) / (B23*B27 - B24^2)$ . The formula in cell F7,  $= (B25*B27 - B24*B26) / (B23*B27 - B24^2)$ , computes the  $y$ -intercept for the best-fit line.

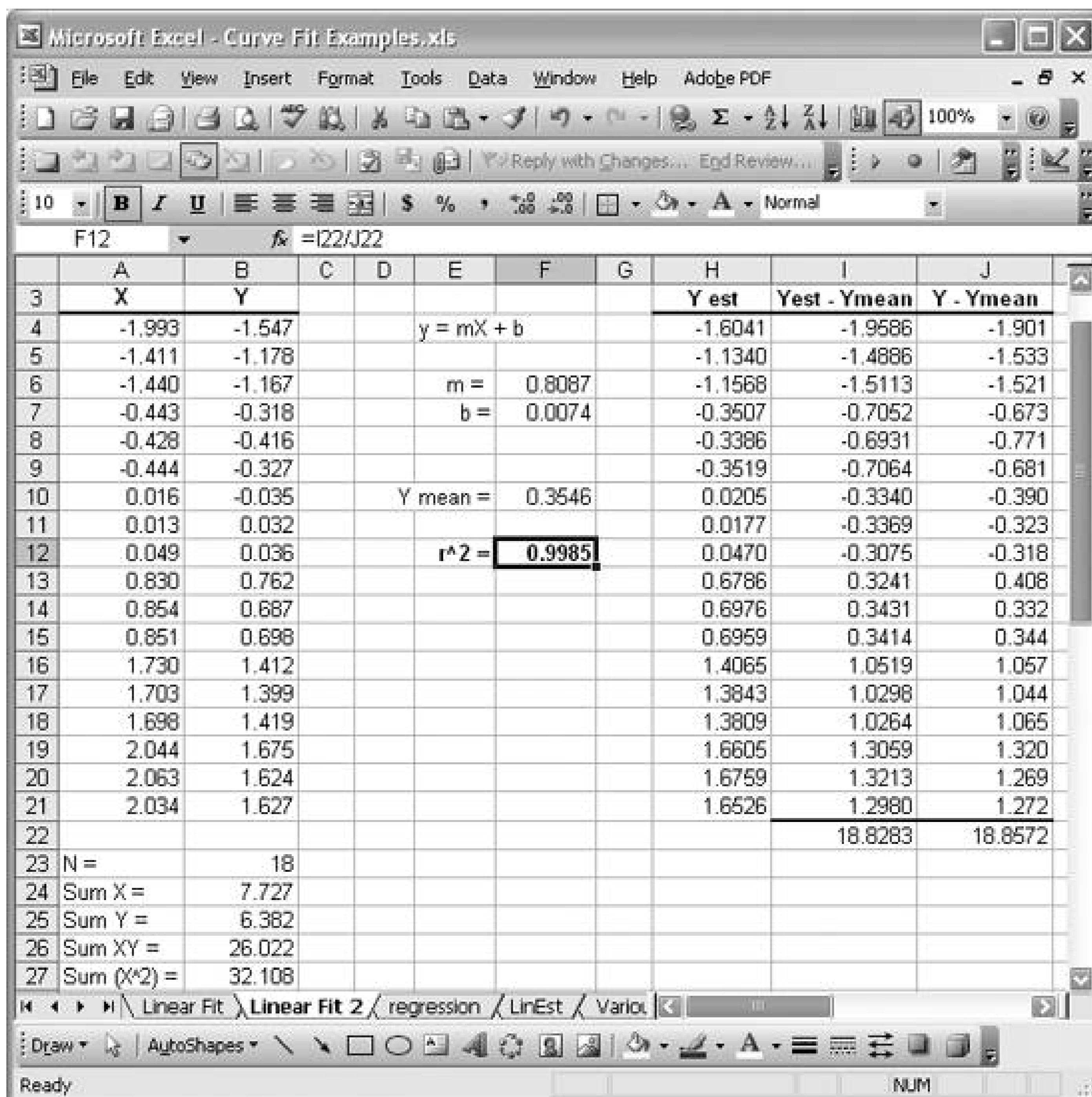
The resulting slope and intercept are 0.80869 and 0.007391, respectively. These values agree very well with those obtained using a trendline as discussed in [Recipe 8.1](#).

To assess how well this fit equation represents the data, you can compute the R-squared value just

as Excel does when it computes chart trendlines. The closer R-squared is to 1, the better the fit. Read [Recipe 8.7](#) to learn more about the R-squared value.

For this example, you need to perform a few more calculations in order to compute the R-squared value. Namely, you need to compute the mean of the  $y$ -values, the estimated  $y$ -values, and the residuals. [Figure 8-5](#) shows these new calculations for this example. Cell F10 computes the mean  $y$ -value using Excel's **AVERAGE** function. Column H contains the estimated  $y$ -values using the equation for a straight line, along with the computed slope and intercept. Columns I and J are used to calculate the differences between the estimated  $y$ -values and the mean and between the actual  $y$ -values and the mean, respectively. The sums of the squares of these differences are computed in cells I22 and J22 using Excel's **SUMSQ** function.

Figure 8-5. R-squared computation



The R-squared value is computed in cell F12 by simply taking the ratio of the values in cells I22 and J22. In [Figure 8-5](#), cell F12 is selected and you can see the cell formula for computing R-squared in the formula bar. For this example, R-squared comes out to 0.9985, which indicates a good fit and agrees very well with the value obtained using a chart trendline as discussed in [Recipe 8.1](#).

## See Also

Take a look at [Recipe 8.7](#) in this chapter to learn more about R-squared. Also, check out the other recipes in this chapter to learn about alternative methods of performing linear curve fitting.



## Recipe 8.3. Using a Single Spreadsheet Function for Linear Curve Fitting

### Problem

You want to perform a quick linear curve fit without using chart trendlines and without having to write the least-squares formulas yourself.

### Solution

Use Excel's `LINEST` function.

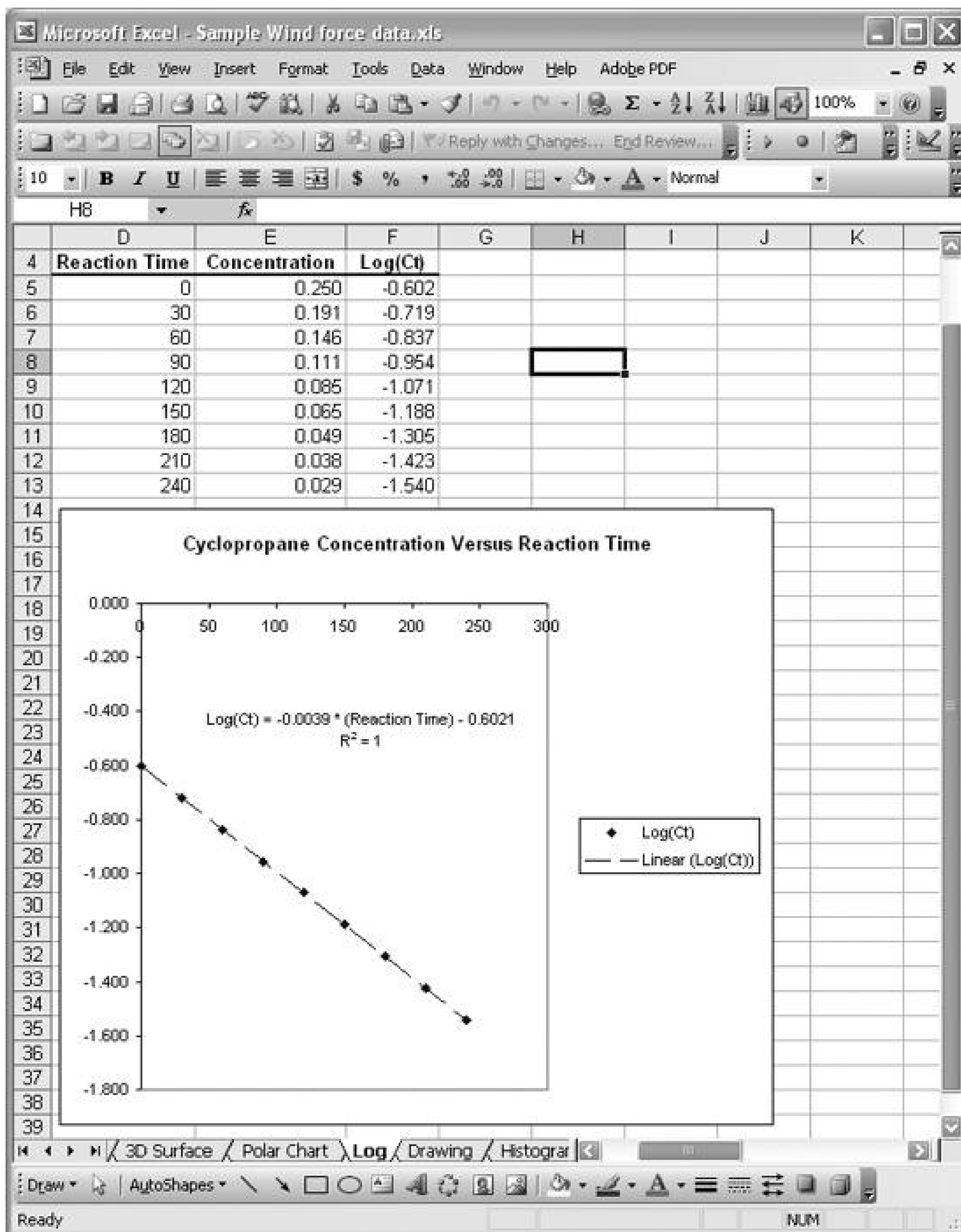
### Discussion

`LINEST` computes statistics for a least-squares straight line through a given set of data. The syntax for `LINEST` is `{= LINEST(y-value cell range, x-value cell range, compute intercept, compute statistics)}`. Note the braces surrounding this formula since it is an *array* formula. When you type this formula into a cell, you have to press Ctrl-Shift-Enter to enter it. Further, you have to select a 2 x 5 grid of cells before typing and entering the formula. This is because `LINEST` returns an array of data containing the various statistics computed for the best-fit line.

The first argument in `LINEST` is a cell range containing the  $y$ -values for the data to be fit, and the second argument is a cell range containing the  $x$ -values. The third argument is a logical value (`true` or `false`) specifying whether or not to force the intercept of the fit line to pass through zero. If `TRUE`, the intercept is calculated in the usual least-squares manner. If `false`, the intercept is forced to zero, with the slope computed accordingly. The fourth argument is a logical value indicating whether or not to display extended statistics for the best-fit line. These extended statistics include such things as standard errors and residual sums. (See the "LINEST" help topic in Excel's online help for more information on these statistics. Also, take a look at Recipes 8.7 and 8.8.)

Consider the chemical reaction data discussed in [Recipe 4.5](#). Instead of plotting concentration versus reaction time on a semilog scale, you can plot the log of the concentration versus reaction time on a linear scale. The plot should look like a straight line. [Figure 8-6](#) shows the log of the concentration versus reaction time data, along with an XY scatter plot of the data using linear axes.

Figure 8-6. Log concentration versus reaction time

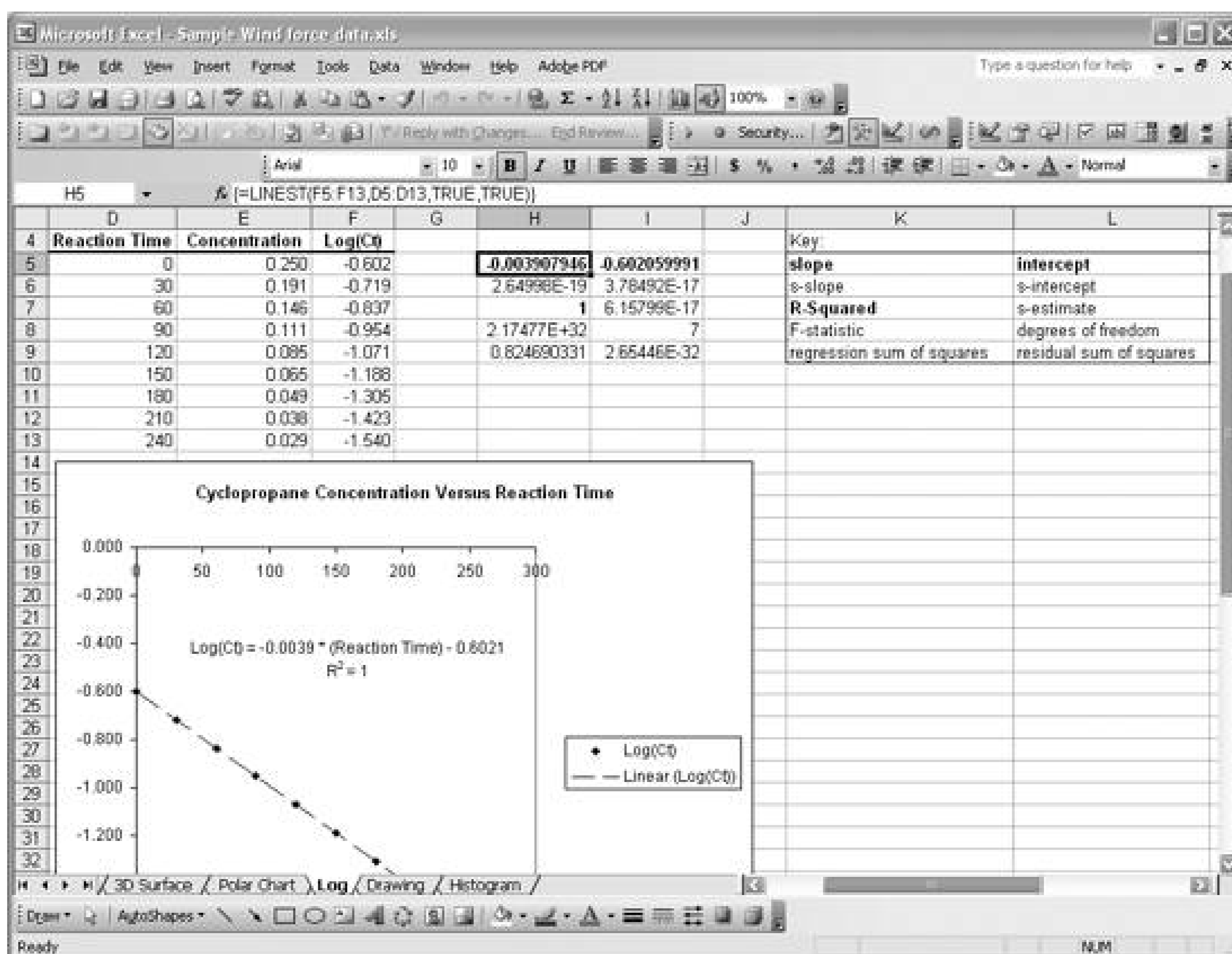


I also included a linear trendline on this chart (see [Recipe 8.1](#)). The linear trendline represents the data very well.

[Figure 8-7](#) shows the results of using **LINEST** to perform a linear fit. The data returned by **LINEST** is contained in cells H5 to I9. The key to the right describes what each returned value represents. The three values in bold type are the slope, intercept, and R-squared value. These values agree very well

with those returned by the chart trendline.

Figure 8-7. Linear fit using LINEST

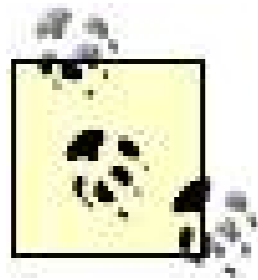


You'll also notice that I have cell H5 selected. This is the first cell in the range that I selected before entering the formula. The formula I entered is displayed in the formula bar. The actual formula is `=LINEST(F5:F13,D5:D13,TRUE,TRUE)`. The  $y$ -values are contained in the range F5:F13. These are the log of concentration values. The  $x$ -values are the reaction times in the range D5:D13. I set the final two arguments to `TRUE` so that `LINEST` computes the intercept and returns extended statistics.

Here are the steps for using `LINEST` as I did in this example:

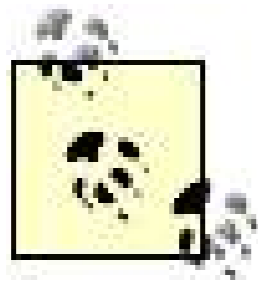
1. With the mouse, select the cell range H5:I9. (You can select any 2 x 5 range you'd like). If you want to see only the slope and intercept, select only two side-by-side cells.
2. Type the `LINEST` formula using the syntax described earlier.
3. Don't press Enter! Press Ctrl-Shift-Enter to enter the formula as an array formula.

That's all there is to it.



Excel also includes a function called **LOGEST** that performs an exponential fit. It's used in much the same way as **LINEST**. See the Excel help topic "LogEst" for more information.

**LINEST** is also capable of performing multiple linear regression where the equation fit is of the form  $y = m_1x_1 + m_2x_2 + m_3x_3 + \dots + m_jx_j + b$ . See [Recipe 8.4](#) for an example.



The Analysis ToolPak add-in contains a regression tool that is capable of performing linear and multiple linear regressions. This tool actually uses **LINEST** to compute the curve fits, and it also returns a boatload of other statistics including ANOVA (analysis of variance) results and confidence intervals. It also computes residuals, making it convenient to check them while assessing the fit. It also prepares a few plots for you. The Analysis ToolPak is installed on your version of Excel if you see the menu item Data Analysis under the Tools menu. Otherwise, select Tools → Add-Ins... from the main menu bar and select Analysis ToolPak from the list of available add-ins.

## See Also

Read the help topic "LinEst" in Excel's online help guide for more details on the **LINEST** function.



## Recipe 8.4. Performing Multiple Linear Regression

### Problem

You want to perform multiple linear regression, where an independent variable depends linearly on more than one independent variable.

### Solution

Use Excel's `LINEST` function. `LINEST` computes statistics for a least-squares straight line through a given set of data. It can be used for linear regression or multiple linear regression. See [Recipe 8.3](#) for an example using `LINEST` for linear regression. The steps for using `LINEST` for multiple linear regression are almost identical; the only differences are the model being fit to the data and the number of cells you have to select when entering the array formula.

### Discussion

As discussed in [Recipe 8.3](#), the syntax for `LINEST` is `{= LINEST(y-value cell range, x-value cell range, compute intercept, compute statistics)}`. Braces surround this formula because it is an *array* formula. When you type this formula into a cell, you must press Ctrl-Shift-Enter to enter it. Further, you must select a grid of cells before typing and entering the formula. This is because `LINEST` returns an array of data containing the various statistics computed for the best-fit line. For multiple linear regression, the grid of cells you have to select must be  $n + 1$  columns wide by 5 rows high, where  $n$  represents the number of independent variables in the data (the  $x$ -variables).

The first argument in `LINEST` is a cell range containing the  $y$ -values for the data to be fit, and the second argument is a cell range containing the  $x$ -values, which should be in multiple adjacent columns, corresponding to the number of independent variables in the data being fit. The third argument is a logical value (`true` or `false`) specifying whether or not to force the intercept of the fit line to pass through zero. If `TRUE`, the intercept is calculated in the usual least-squares manner. If `false`, the intercept is forced to zero, with the slope computed accordingly. The fourth argument is a logical value indicating whether or not to display extended statistics for the best-fit line. These extended statistics include such things as standard errors and residual sums.

Let's consider a standard benchmark dataset. I downloaded the data shown in [Figure 8-8](#) from the NIST web site (<http://www.nist.gov>).<sup>[\*]</sup> The data represents labor statistics where the dependent variable,  $y$ , is the Total Derived Employment and the independent variables are GNP Implicit Price Deflator with Year 1954 = 100, Gross National Product, Unemployment, Size of Armed Forces, Non-Institutional Population over Age 14, and Year, corresponding to the variables  $x_1$ ,  $x_2$ ,  $x_3$ ,  $x_4$ ,  $x_5$ , and  $x_6$ , respectively.

[\*] NIST stands for National Institute of Standards and Technology. The data used here is from the NIST statistical reference library, which is freely available online. The data was originally prepared by J. W. Longley in 1967. See J. W. Longley, "An Appraisal of Least Squares Programs for the Electronic Computer from the Viewpoint of the user." *Journal of the American Statistical Association* 62 (1967): 819-841.

The model to be fit is of the form  $y = m_6x_6 + m_5x_5 + m_4x_4 + m_3x_3 + m_2x_2 + m_1x_1 + b$ . This model has seven parameters (the  $m$ s and  $b$ ) that are determined by the fitting process.

The steps for applying **LINEST** in this example are as follows:

1. With the mouse, select the cell range D25:J29. (You can select any  $(n + 1) \times 5$  range you'd like).
2. Type the **LINEST** formula using the syntax shown in the formula bar in [Figure 8-8](#).
3. Press Ctrl-Shift-Enter to enter the formula as an array formula.

The results are shown in [Figure 8-8](#). I labeled the model parameters for clarity. Notice the R-square value shown in bold in [Figure 8-8](#). A value of 0.995 represents a very good fit and agrees very well with the results published by the NIST for this dataset.

I also calculated the estimated  $y$ -values as shown in column K ([Figure 8-8](#)). [Figure 8-9](#) shows a chart of the estimated  $y$ -value along with the actual  $y$ -value. Visually, the fit very reasonably approximates the underlying data.

Figure 8-8. Multiple linear regression using LinEst

Microsoft Excel - Curve Fit Examples.xls

File Edit View Insert Format Tools Data Window Help Adobe PDF

D25  $f_x$  (=LINEST(D7:D22,E7:J22,TRUE,TRUE))

	D	E	F	G	H	I	J	K
6	<b>y</b>	<b>x1</b>	<b>x2</b>	<b>x3</b>	<b>x4</b>	<b>x5</b>	<b>x6</b>	<b>y-est</b>
7	60323	83	234289	2356	1590	107608	1947	60055.66
8	61122	88.5	259426	2325	1456	108632	1948	61216.01
9	60171	88.2	258054	3682	1616	109773	1949	60124.71
10	61187	89.5	284599	3351	1650	110929	1950	61597.11
11	63221	96.2	328975	2099	3099	112075	1951	62911.29
12	63639	98.1	346999	1932	3594	113270	1952	63888.31
13	64989	99	365385	1870	3547	115094	1953	65153.05
14	63761	100	363112	3578	3350	116219	1954	63774.18
15	66019	101.2	397469	2904	3048	117388	1955	66004.7
16	67857	104.6	419180	2822	2857	118734	1956	67401.61
17	68169	108.4	442769	2936	2798	120445	1957	68186.27
18	66513	110.8	444546	4681	2637	121950	1958	66552.06
19	68655	112.6	482704	3813	2552	123366	1959	68810.55
20	69564	114.2	502601	3931	2514	125368	1960	69649.67
21	69331	115.7	518173	4806	2572	127852	1961	68989.07
22	70551	116.9	554894	4007	2827	130081	1962	70757.76
23								
24	m6	m5	m4	m3	m2	m1	b	
25	1829.1515	-0.051104	-1.033227	-2.02023	-0.035819	15.06187	-3482259	
26	455.4785	0.226073	0.214274	0.4884	0.033491	84.91493	890420.4	
27	0.995479	304.8541						
28	330.28534	9						
29	184172402	836424.1						

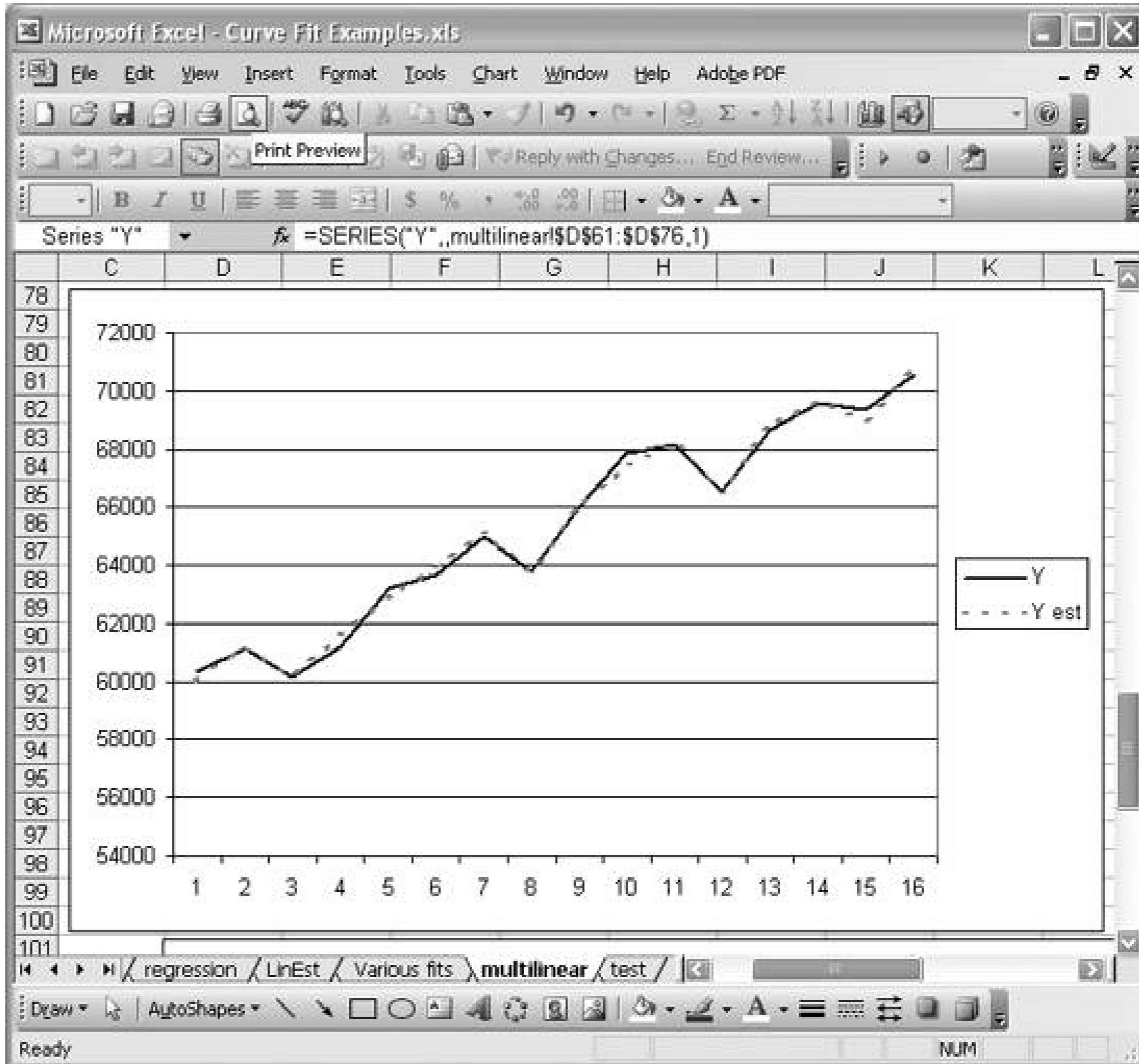
Ready NUM

One of the challenges with multiple regression is determining which independent variables best predict the dependent variable without duplicating characteristics. That is, you don't want to use independent variables that are highly correlated with each other. See [Recipe 5.4](#) for more information on conducting correlation analyses in Excel.

## See Also

Read the help topic "LinEst" in Excel's online help for more details on the **LINEST** function.

Figure 8-9. Y and Y-estimated



## Recipe 8.5. Generating Nonlinear Curve Fits Using Excel Charts

### Problem

You'd like to generate best-fit curves nonlinear curves through your data.

### Solution

Use Excel's chart trendline feature to perform nonlinear curve fits of your data. Plot your data using an XY scatter chart (see [Chapter 4](#)). Once your chart has been created, right-click on the data series and select Add Trendline from the pop-up menu.

### Discussion

The procedure for adding nonlinear trendlines to charts is the same as that used for adding linear trendlines, as discussed in [Recipe 8.1](#). Once you've plotted your data using an XY scatter chart, you can generate a trendline, which will be displayed on the chart, superimposed over your data. You can also include the resulting equation for the best-fit line on your chart.

When you add a trendline using the procedure discussed in [Recipe 8.1](#), you can choose from several different Trend/Regression types (as shown in the Add Trendline dialog box in [Figure 8-1](#)). Depending on the range of values in your data, you'll be able to choose from six different trendlines. Of these six, four are nonlinear trendlines: Logarithmic, Polynomial, Power, and Exponential. The other two include the Linear trendline already discussed and a Moving Average trendline.

The Logarithmic trendline uses the equation:

where  $c$  and  $b$  are parameters to be determined during the curve-fitting process.

Polynomial trendlines use the equation:

where the  $c$ -coefficients and  $b$  are parameters of the curve fit. Excel supports polynomial fits up to sixth order. You can select the order to use in the Add Trendline dialog box (See [Figure 8-1](#)).

Excel's Power trendline uses the equation:

where  $c$  and  $b$  are parameters of the fit.

Finally, the Exponential trendline uses the equation:

$$y = ce^{bx}$$

where  $c$  and  $b$  are parameters of the fit.

The natural question that arises upon learning of these available models is which one to use. There's no single answer to this question, as it depends on your purpose for performing the fit and the nature of your data. If you're simply fitting a curve through some data so you can conveniently interpolate the data, then choose whichever model best replicates your data. In this case, you're not so concerned with smoothing the data or with statistical rigor. To assess which model works, you can look at the R-squared value (see [Recipe 8.7](#)) or you can eyeball the trendline against your data. Further, you can plot the residuals differences between your actual  $y$ -value and the estimated  $y$ -value and examine them to assess how well your data is represented. You can compute percentage differences to gauge how well your data is replicated. Further, you should examine how the trendline behaves between data points used to generate the trendline to make sure there are no unrealistic oscillations between data points. This can happen when you try to fit a higher-order polynomial trendline.

On the other hand, if you're genuinely interested in modeling the data from a statistical standpoint, or are trying to gain insight in a physical process, then I suggest you read [Recipe 8.7](#). Further, you must also consider what your data represents. If your data represents some physical process, then let physics be your guide and choose a model that best represents the physical relationship between the variables underlying the data, if it is indeed known. If none of Excel's trendline models work for you, then you're left to your own devices to craft a suitable model. In this case, take a look at [Recipe 8.6](#).

## See Also

Take a look at [Recipes 8.6](#) and [8.7](#). Also, be sure to read [Recipe 8.1](#) for a refresher on adding trendlines to charts.

← PREV

## Recipe 8.6. Fitting Nonlinear Curves Using Solver

### Problem

You'd like to perform a curve fit using a model (an equation) that's not included in Excel's suite of curve fit functions and trendlines.

### Solution

Perform a least-squares curve fit using Solver.

### Discussion

Solver is a fantastic Excel add-in that allows you to perform constrained optimization calculations. Solver uses the nonlinear Generalized Reduced Gradient optimization algorithm developed by Leon Lasdon and Allan Waren. Go to the Tools → Solver menu to open the Solver dialog box shown in [Figure 8-10](#).

Figure 8-10. Solver dialog box

If you don't see the Solver menu item, then go to Tools → Add-Ins... and look for Solver in the available add-ins list. Select its checkbox to make it available.

Solver allows you to choose a target cell and either maximize it, minimize it, or attempt to set its value to some specified value by changing values in a given range of cells. The controls in the Solver dialog box allow you to set these parameters. After setting these parameters in the Solver dialog box press Solve to initiate the iterative calculation. Solver will iterate until it finds a solution or until certain limits are reached (these effectively put the brakes on the iterative calculation so it will not continue forever). You can set these limits by opening the Options dialog via the Options button.

The target cell could be any cell containing a formula. That formula could contain references to other cells, which could themselves contain formulas, and so on. Ultimately, the value in the target cell will be affected by certain parameters contained in a select group of cells or a single cell. You can construct arbitrarily complex spreadsheets that somehow relate the target cell to one or more cells containing values, such as constants or coefficients. This is really powerful. You can create spreadsheets to compute something subject to a given set of inputs and then you can use Solver to find the optimum set of inputs to maximize, minimize, or specify the output. It's this facility that we'll leverage to use Solver for nonlinear curve fitting.

Ultimately, curve fitting involves minimizing some error, which can be expressed as the sum of squared differences between actual and predicted values as in the least-squares method. You could use some other merit function instead of the sum of squared residuals. For example, you could choose to maximize the coefficient of determination, R-squared, instead.

The basic approach to using Solver for curve fitting is to set up a spreadsheet containing sample data for your dependent and independent variables. The dependent variable will be predicted using some model (i.e., an equation) that you devise. Set up another column for predictions of your dependent variable using your model equation. Add another column containing the residuals (i.e., the difference between your predicted value and the actual value of the dependent variable). Compute the sum of squared differences in another cell. In the case of minimizing the sum of squared differences, you can set this cell as the target cell in the Solver dialog box. Set the option to minimize that cell. Your model equation must contain parameters, or coefficients. These should be located conveniently in another set of cells. Use references to these cells as the cells to change in the Solver dialog box. Solver will then attempt to minimize the sum of squared differences with the end result, if it succeeds, being the set of parameters that yield the optimum solution. This gives you the curve fit.

Let's look at an example. We'll consider a standard reference set of data available from the NIST web site. In this case, the data is from an NIST study involving semi-conductor electron mobility where the dependent variable,  $\mu$ , is a measure of electron mobility, and the independent variable,  $x$ , is the natural log of the density.<sup>[\*]</sup> The model for this data is nonlinear and of the following form:

[\*] This dataset is credited to R. Thurber of NIST and is available for download from <http://www.nist.gov>.

The  $b$ s in this equation are the parameters to be determined through the fitting processes. There are seven of them. Clearly, this nonlinear, rational function is not included in Excel's suite of curve-fitting tools. This illustrates the generality of using Solver for curve fitting you can use any model equation you want.

[Figure 8-11](#) shows a portion of the data used for this example. There are 37 data points contained in columns D and E. As you can see, I set up a little table of  $b$ -coefficients off to the left. These are set



up with some initial values that you must choose. These cells don't contain formulas. Just type the numbers in directly; Solver will change them during its calculations.

Figure 8-11. Example data

	A	B	C	D	E	F	G	H	I	J	K	L	M
1													
2													
3													
4	b1	1288.242		X	Y		Y est	Y - Yest					
5	b2	1479.152		-3.0670	80.6740		76.0301	-4.5439		Sum Residuals-squared =		5662.30	
6	b3	574.492		-2.9810	84.2480		84.28334	0.0353		Residual Standard Deviation =		13.74	
7	b4	73.735		-2.9210	87.2640		88.23812	0.9741		Number of Parameters =		7.00	
8	b5	0.957		-2.9120	87.1950		88.71732	1.5223		Number of Observations =		37.00	
9	b6	0.393		-2.8400	89.0760		91.601	2.5250		Degrees of freedom =		30.00	
10	b7	0.048		-2.7970	89.6080		92.61114	3.0031		Average Y =		783.21	
11				-2.7020	89.8680		93.39718	3.5292		R-Squared =		0.99932	
12				-2.6990	90.1010		93.39665	3.2957		Confidence =		0.95	
13				-2.6330	92.4050		93.12856	0.7236		t-value =		2.04	
14				-2.4810	95.8540		92.13305	-3.7210		Confidence Interval =		28.06	
15				-2.3630	100.6960		93.43915	-7.2569					
16				-2.3220	101.0600		94.8369	-6.2231					
17				-1.5010	401.6720		393.6037	-8.0683					
18				-1.4600	390.7240		424.5296	33.8056					
19				-1.2740	567.5340		574.4731	6.9391					
20				-1.2120	635.3160		626.0346	-9.2814					
21				-1.1000	733.0540		718.2712	-14.7828					
22				-1.0460	759.0870		761.5975	2.5105					
23				-0.9150	894.2080		861.5842	-32.6218					

You have to provide an initial guess at the model parameters, the  $b$ s in this example. This is the case with many iterative algorithms. You must choose these wisely given the problem you are considering and in light of your understanding of it. It's best to make as good an educated guess as possible. If your model is highly nonlinear, there may be several solutions. As with nonlinear optimization problems in general, there could exist local minima and maxima in the merit function domain within which Solver may converge on a solution. I recommend you try several different sets of initial values to make sure you're converging on a global optimum as opposed to some local optimum.

Column G contains the estimated  $\mu$ -value for each  $x$ -value in the dataset. This estimated  $\mu$ -value

uses the model equation. In this case, I coded the model equation in a custom VBA function called `CalcYest` (see [Recipe 2.2](#)). `CalcYest` is shown in [Example 8-1](#).

### Example 8-1. CalcYest

```
Function CalcYest(x As Double, b1 As Double, b2 As Double, b3 As Double,
                 b4 As Double, b5 As Double, b6 As Double,
                 b7 As Double) As Double

    CalcYest = (b1 + b2 * x + b3 * x ^ 2 + b4 * x ^ 3) /
                (1 + b5 * x + b6 * x ^ 2 + b7 * x ^ 3)

End Function
```

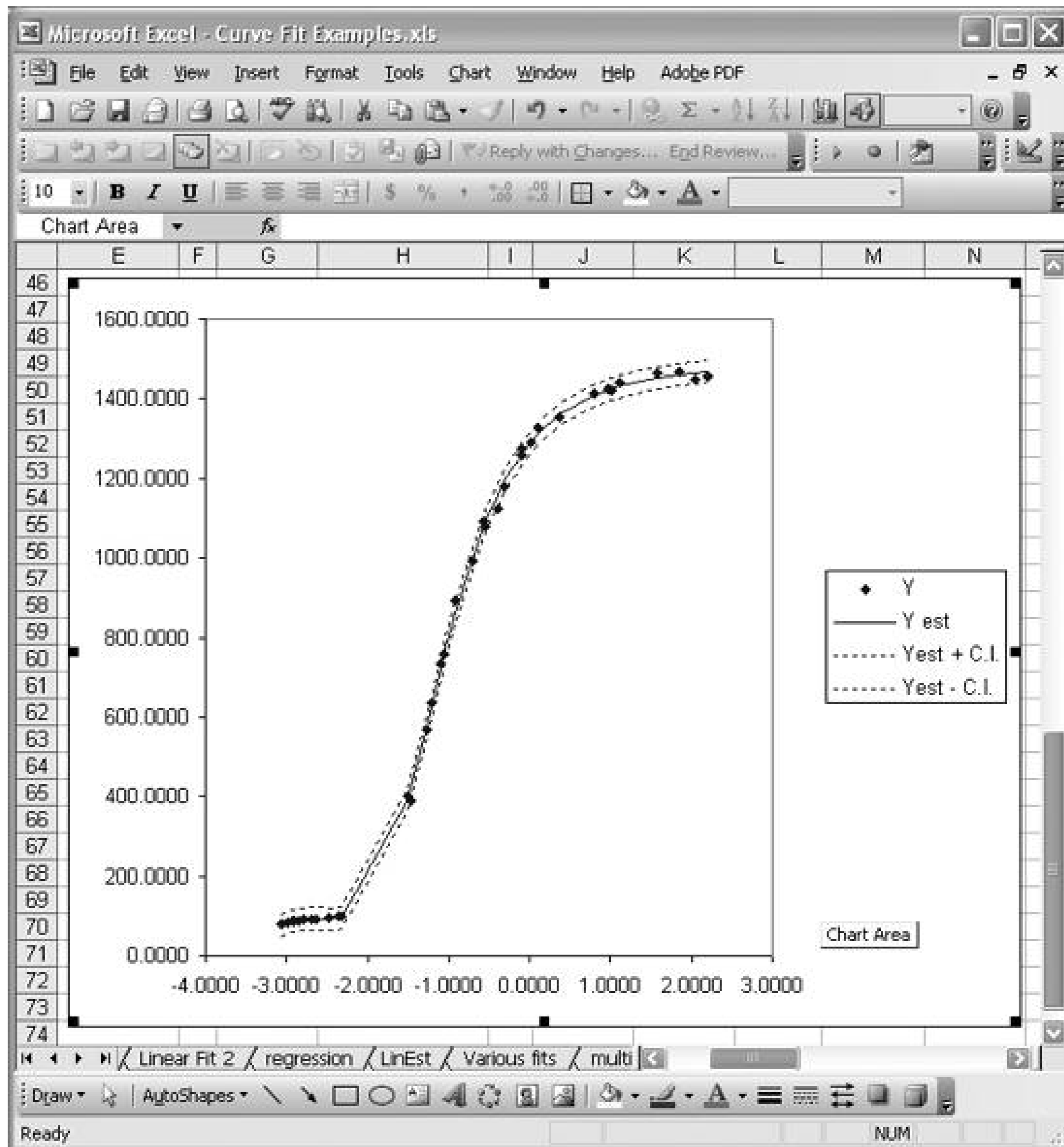
This function takes the  $x$ -value along with all seven  $b$ -parameters as arguments and returns the estimated  $y$ -value. You'll notice in [Figure 8-11](#) that cell G5 is selected. Take a look at the formula bar to see how `CalcYest` is actually used as a cell formula.

The table off to the right in [Figure 8-11](#) contains several statistics that I computed for this example. I'll address most of these later (see [Recipes 8.7](#) and [8.8](#)), so for now just look at cell M5. That cell contains the sum of squared residuals. The residuals are computed in column H and are simply the estimated  $y$ -values minus the sampled  $y$ -values. The sum of squared residuals is computed in cell M5 with the formula `=SUMSQ(H5:H41)`. `SUMSQ` is Excel's built-in function that computes the sum of the squares of the values contained in the given range. It's this value we want to minimize for the curve fit.

With things set up as described, you can now open Solver and set the parameters. First set the target cell to M5, the sum of squared residuals. Next select the "Equal to: Min" option, letting Solver know we want to minimize the value contained in M5. Enter the cell range B4:B10 in the By Changing Cells field. Press the Solve button. At this point, another dialog box will open, showing the results of Solver's calculations. Solver will tell you whether or not it converged on a solution and will give you the option of saving its results or restoring your original values (the values of the parameters you allowed Solver to change).

The converged results for this example are already shown in [Figure 8-11](#). The  $b$ -parameters computed here agree very well with the NIST benchmark results. Further, the R-squared value for this example is 0.99932, indicating a good fit. [Figure 8-12](#) shows the results plotted on a chart along with the original data. The original data points are shown with diamond point markers, and the estimated values are shown with the solid curve. The two other curves above and below the estimate curve represent the 95% confidence interval for the predicted values. See [Recipe 8.8](#) to learn how to compute these curves.

Figure 8-12. Solver curve-fit results



Clearly, this approach using Solver yields a nice curve fit to a fairly nonlinear set of data. I must say that not all nonlinear curve fits are this straightforward. I've had some very unruly cases involving multiple nonlinear models, where a different local minimum was converged upon for virtually every set of guessed input parameters I could imagine. In other cases, the model was such that Solver could not converge on a solution at all. These aren't necessarily a defect in this approach or a deficiency in Solver. Those of you who've experienced such difficulties firsthand know that it's just the nature of some highly nonlinear problems to make your life difficult.

## Recipe 8.7. Assessing Goodness of Fit

### Problem

You need to assess how well an equation fits the underlying data.

### Solution

If you're simply fitting a curve through some data so you can conveniently interpolate the data, then choose whichever model best replicates your data. In this case you're not so concerned with smoothing the data or with statistical rigor. You can plot your data along with estimates using the fit curve and eyeball them to see how well they compare. You can also plot the residuals differences between your actual  $y$ -value and the estimated  $y$ -value and examine them to assess how well your data is represented or to determine if the residuals exhibit some unexpected structure. You can compute percentage differences to gauge how well your data is replicated. Further, you should examine how the trendline behaves between data points used to generate the trendline to make sure there are no unrealistic oscillations between data points. This can happen when you try to fit a higher-order polynomial trendline.

If you're interested in modeling the data from a statistical standpoint, or are trying to gain insight into a physical process, then you can perform some standard statistical calculations to help assess your trendline. Moreover, you should consider what your data represents. If your data represents some physical process, then let physics be your guide and choose a model that best represents the physical relationship between the variables underlying the data, if it is indeed known. In the discussion to follow, I show you how to perform several standard calculations for assessing goodness of fit in Excel.

### Discussion

As already mentioned, one way to see how well your equation fits the underlying data is to plot the fit equation over the data and observe how well it appears to capture the relationship between the variables considered. The *correlation coefficient* is a quantitative way to assess how well your equation models the relationship between the underlying variables.

Any standard text on statistical analysis that covers regression and correlation theory can provide details on the derivation of the correlation coefficient. In a nutshell, the correlation coefficient is a ratio of variations between the variables estimated using the fit equation and the actual variables. The basic formula for correlation coefficient is:

The term in the denominator under the radical is called the *total variation* of  $Y$ . It's a measure of the deviations of  $Y$  from the mean value of  $Y$  in the dataset. Total variation is composed of two components: the *explained variation* and the *unexplained variation*. Explained variation is a measure of the deviations of the estimated  $Y$  values (those predicted from the fit curve) from the mean value of  $Y$  in the dataset. Unexplained variation is a measure of the deviations of the estimated  $Y$  value from the actual or measured  $Y$  value. These variations are assumed random and it's these variations you aim to minimize when fitting a curve to a set of data.



The formula for the correlation coefficient can be and often is written in different forms. For example, in some cases it's written in terms of standard deviations.

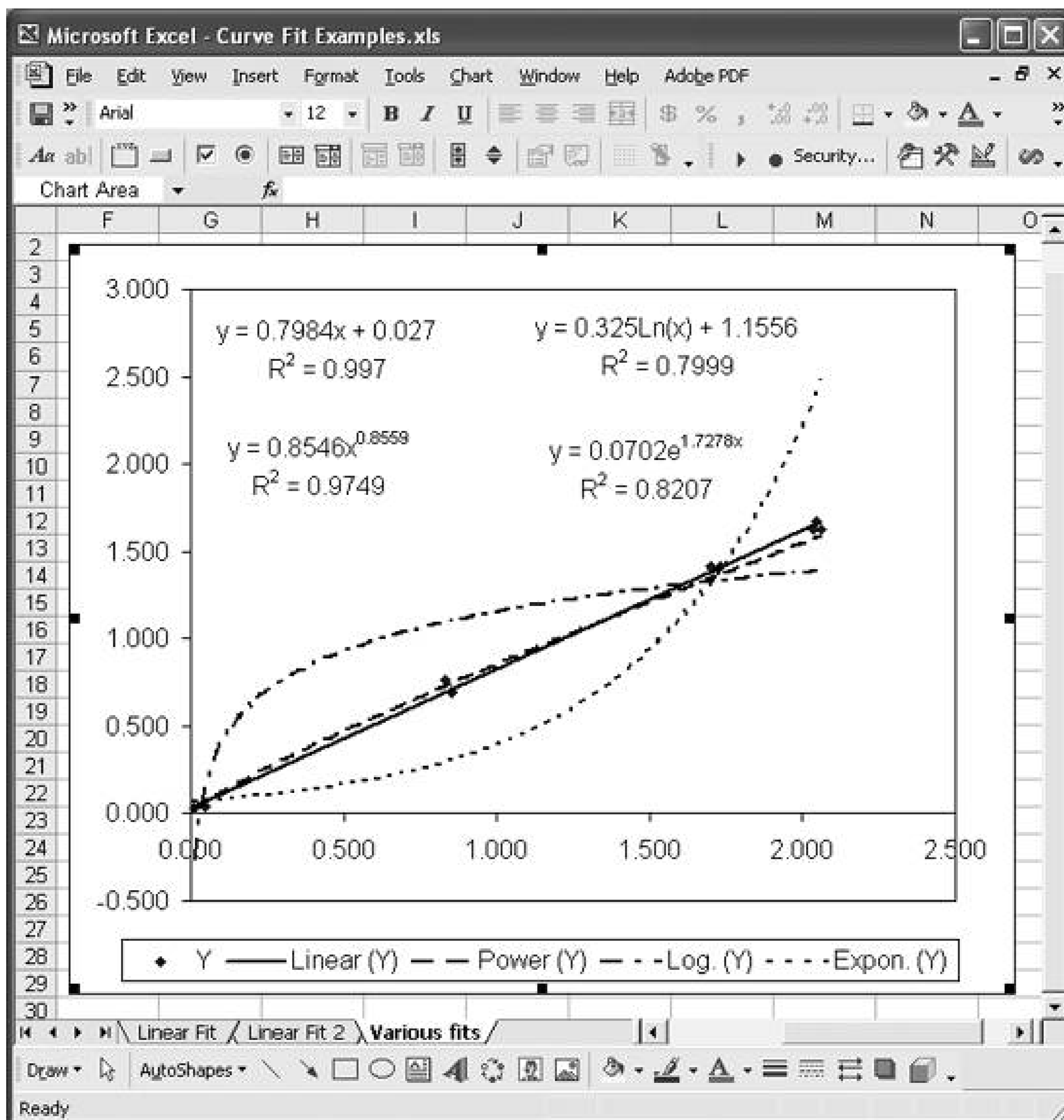
As you can see from the formula shown a moment ago, the correlation coefficient is the ratio of the explained variation to the total variation. The greater the explained variation, the smaller the unexplained variation, and therefore the closer the correlation coefficient approaches a value of  $\pm 1$ . The R-squared value is just the square of the correlation coefficient. Therefore, the closer R-squared is to a value of 1, the smaller the unexplained variation and the better the fit. Excel's curve fit and trendline functions compute the R-squared value, which is also called the *coefficient of determination*. I've mentioned this R-squared value several times in other recipes already. The closer the R-squared value is to 1, the better the equation fits the underlying data.

You can display the R-squared value for a chart trendline by checking "Display R-squared value on chart" under the Options tab in the Add Trendline dialog box. See [Figure 8-2](#) in [Recipe 8.1](#).

For illustration purposes, I prepared a dataset that exhibits a linear relationship and had Excel compute various trendlines, including one linear and three nonlinear trendlines. The results are shown in [Figure 8-13](#).

The equations, along with the corresponding R-squared values, are also shown on the chart. The linear trendline best fits the data, as expected, with an R-squared value of 0.997. The next best fit is from the power fit, with an R-squared value of 0.9749, which is still not a bad fit. The other two trendlines the log and exponential

Figure 8-13. Various trendlines



trendlines exhibit poor fits, with R-squared values of 0.7999 and 0.8207, respectively. You can also visually see that the log and exponential fits poorly represent the data under consideration.

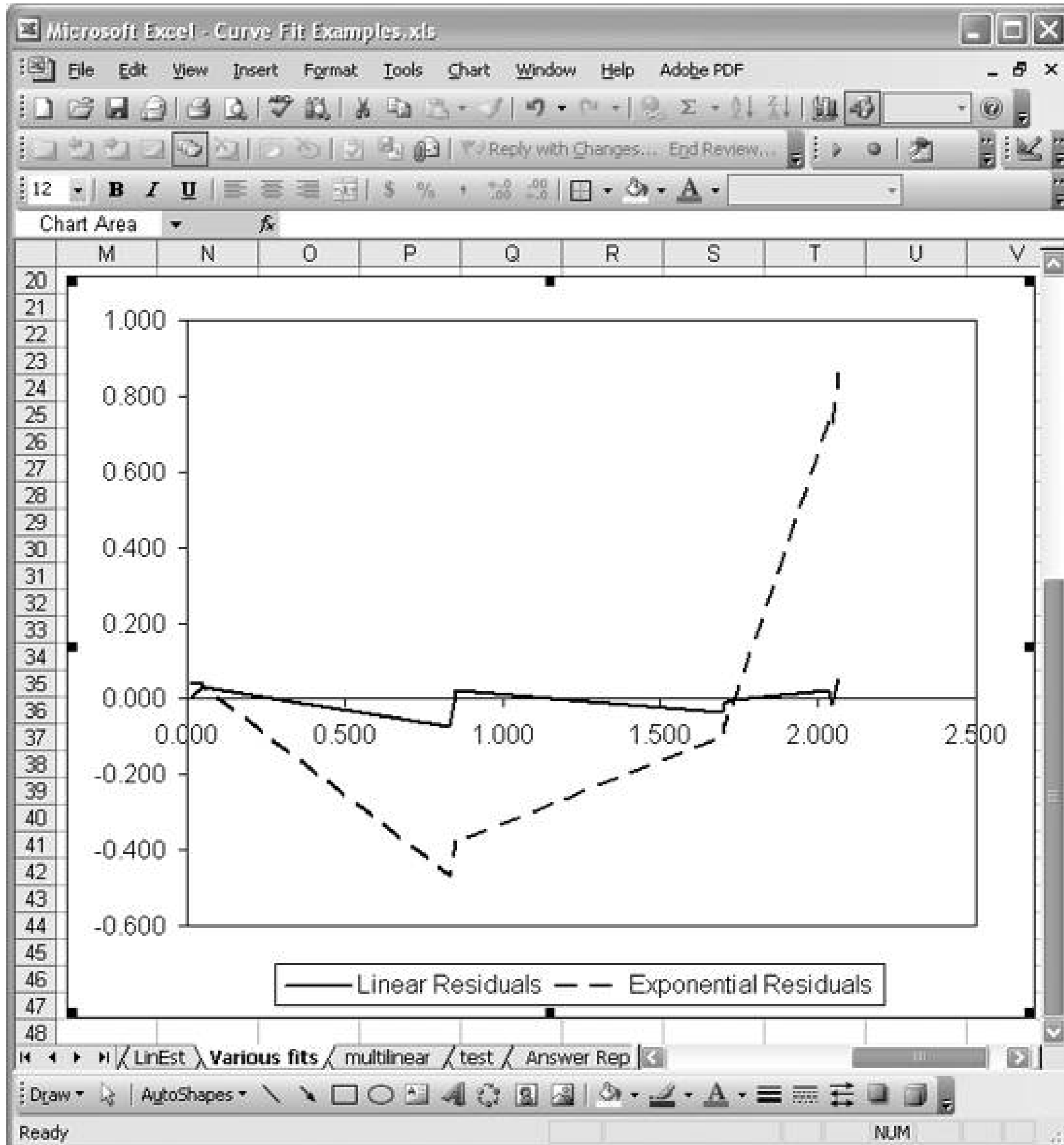
Examining these results brings up a point of concern. Say you initially fit a power curve to the data shown here. You might reasonably assume the power curve is the best-fit curve, given the resulting R-squared value of 0.9749. However, we happen to know a little something about the physical phenomenon this data represents and know that this is a linear problem. A linear fit indeed shows a better fit, with an R-squared value of 0.997. These results reinforce the importance of the need to use your judgment and understanding of the problem being analyzed when choosing an appropriate model.

As a further check on how well a model equation fits the data, you can plot the residuals. (See [Recipe 8.6](#) for an example showing how to compute residuals.) These represent the unexplained variation between the estimated  $\mu$ -values and the given  $y$ -values. Unexplained variations are supposed to be

random, and the residuals, when plotted, should confirm this behavior. If you plot the residuals and see some structure to the residual plot that does not appear random, you should take a closer look at your assumed model.

As an illustrative example, consider the data shown in [Figure 8-12](#) along with the linear and exponential trendlines. The residuals between the linear fit and the exponential fit are plotted in [Figure 8-14](#).

Figure 8-14. Residual plot



The residual curve from the linear fit does seem to exhibit a random pattern. On the other hand, the

residual curve from the exponential fit exhibits a very distinct pattern, resembling an exponential curve that's sort of been shifted down a bit. Such a structure in the residual plot indicates a fundamental misrepresentation of the underlying data. This is an exaggerated example, as we already knew the exponential curve was a poor fit. However, the point is that examining the residual plot can give you some insight into the goodness of the fit.

Another check on how well a model represents a set of data is known as the *F-test*. This test comes from a technique known as *analysis of variance* (ANOVA) developed by R. A. Fisher. The formal details of this test can be found in standard statistical texts that cover ANOVA.

Excel makes it easy to perform this F-test because its regression functions, such as `LINEST`, return the statistics required for this test, and Excel also contains other required probability-related functions. The F-test basically involves computing a statistic known as the F-statistic and comparing that value to a critical value obtained from the standard F-distribution using Excel's built-in functions. If the computed F-statistic is greater than the critical statistic, then you can conclude the fit is a good one or, more formally, statistically significant.

[Figure 8-15](#) shows an example linear curve fit using Excel's `LINEST` function (see [Recipe 8.3](#)). The results of using `LINEST` on the given dataset are contained in the cell range E8:F12. I've included a key explaining what each returned value represents. The value in cell E11 is the F-statistic computed by `LINEST`.

Cells F25 to F30 contain calculations I added to compute the critical F-value for comparison. Cell F25 contains the number of parameters in the regression equation; 2 in this case, since it's linear. Cell F26 contains the number of samples. I used `=COUNT(A8:A25)` to determine the number of samples (of course, for this small dataset they can be easily counted manually). Cell F27 contains the *numerator degrees of freedom* required for computing F-critical; this value is equal to the number of parameters in the regression equation. Cell F28 contains the denominator degrees of freedom, which is equal to  $n - (k + 1)$ , where  $n$  is the number of samples and  $k$  is the number of regression parameters. Cell F29 contains the probability of significance for which we're interested in computing F-critical. F-critical is computed using the formula `=FINV(F29,F27,F28)`. In this case, we get an F-critical value of 3.6823, which is far lower than the F-statistic computed for the curve fit. This indicates that curve fit is significant and useful in representing the underlying data.

`FINV` is a built-in formula that returns the inverse of the F-distribution. See Excel's help topic "FINV" and [Recipe 5.6](#) for more information.

## See Also

Assessing goodness of fit is certainly not limited to the techniques I've discussed here. I'm sure a professional statistician could easily explain a half-dozen more statistical tests that could be used to validate a particular curve fit.

Figure 8-15. F-test example



Microsoft Excel - Curve Fit Examples.xls

File Edit View Insert Format Tools Data Window Help Adobe PDF

10 B I U \$ % , %0 %0 A Normal

F30 =FINV(F29,F27,F28)

	A	B	C	D	E	F
1					Key:	
2					<b>slope</b>	<b>intercept</b>
3					s-slope	s-intercept
4					<b>R-Squared</b>	s-estimate
5					F-statistic	degrees of freedom
6					regression sum of	residual sum of squares
7	<b>Applied Torque (N-m)</b>	<b>Measured Deflection (degrees)</b>				
8	-1.993	-1.547			0.8087	0.0074
9	-1.411	-1.178			0.0079	0.0106
10	-1.440	-1.167			0.9985	0.0425
11	-0.443	-0.318			10411.8096	16.0000
12	-0.428	-0.416			18.8283	0.0289
13	-0.444	-0.327				
14	0.016	-0.035				
15	0.013	0.032				
16	0.049	0.036				
17	0.830	0.762				
18	0.854	0.687				
19	0.851	0.698				
20	1.730	1.412				
21	1.703	1.399				
22	1.698	1.419				
23	2.044	1.675				
24	2.063	1.624				
25	2.034	1.627			# Fit Parameters	2
26					# Samples	18
27					df1	2
28					df2	15
29					Probability	0.05
30					Fcritical	3.6823
31						

regression LinEst Various fits multilineal

Draw AutoShapes

Ready NUM

## Recipe 8.8. Computing Confidence Intervals

### Problem

You want to compute confidence intervals along with your curve fits.

### Solution

See the following discussion.

### Discussion

You can compute confidence intervals for the estimated values from your curve fit, or you can compute confidence intervals for the parameters in the model itself.

#### Confidence intervals for estimated values

You can readily compute confidence intervals for the values predicted by a regression equation. In this case, you'd report estimated values as  $y \pm \mathcal{J}_c$ , where  $y$  is the estimated value and each  $\pm \mathcal{J}_c$  is the confidence limit corresponding to some degree of confidence (probability),  $c$ . To compute confidence limits, we'll utilize the *Student's t-distribution*, which comes from an area of statistics called *small sampling theory*. Excel provides support for the t-distribution through several built-in functions. See [Recipe 5.6](#) for more information.

Let's reconsider the example from [Recipe 8.6](#). [Figure 8-12](#) shows the result of a nonlinear curve fit, along with upper and lower confidence limits. [Figure 8-11](#) shows the original data along with the fit parameters. It also shows several other statistics I computed for this example. The ones of interest here are t-value and Confidence Interval, shown in cells M13 and M14.

Before computing these values, you must set up some auxiliary calculations. First you must compute the sum of squared residuals as shown in cell M5. The formula used in this example is `=SUMSQ(H5:H41)`, which is just the sum of the squares of the residuals already computed in column H.

Next you need to compute the residual standard deviation. This value is computed in cell M6 using the formula `=SQRT(M5/M9)`, where the cell in the numerator is the sum of squared residuals and the cell in the denominator is the degrees of freedom for this example. Degrees of freedom is computed in cell M9 using the formula `=M8-M7`, which is the number of samples minus the number of parameters in the fit (7 in this case, for the seven  $\mathcal{b}$ -parameters).

Now you can compute the t-statistic using the formula `=TINV((1-M12), M9)`, the result of which is

shown in cell M13. The function `TINV` returns the inverse of the Student's t-distribution corresponding to a specified probability. In this case, I set the degree of confidence to 95% (.95 probability) in cell M12 and used `1-M12` as the probability passed in as the first argument in `TINV`. The second argument to `TINV` represents the problem's degrees of freedom.

Finally, the confidence limits are found by multiplying the t-statistic by the residual standard deviation. Cell M14 contains this result, using the formula `=M13*M6`. To plot the upper and lower confidence limits on the same chart as the original data and fit curve, you must set up two new columns. The first column contains the estimated  $\mu$ -value plus the confidence limit, and the second column contains the estimated  $\mu$ -value minus the confidence limit. You can then add these two new series to your chart to obtain the results shown in [Figure 8-12](#).

## Confidence intervals for curve parameters

Sometimes it's the parameters of a curve fit that are of scientific interest, instead of the estimated values predicted from the fit line. For example, you may compute a best-fit straight line through a set of experimental data where the slope of the best-fit line represents some aspect of the underlying physical process, such as reaction rate or spring constant. Curve fitting to estimate the model parameters yields statistical estimates of those parameters given the sample, which we assume represents the entire population. We can use statistical techniques to estimate confidence intervals for these parameters just as we did when estimating confidence intervals for predicted values.

[Figure 8-16](#) shows some experimental data that has been the subject of a linear least-squares curve fit. In this case, the parameters of the curve fit (the slope and intercept) are of scientific importance and we'll compute confidence limits for each of these corresponding to a 95% degree of confidence.

To begin with, you must fit a straight line through the data, using any of the linear curve-fitting techniques discussed in this chapter. Then you need to compute the standard error for the estimate by using Excel's built-in function `STEYX`. Cell F12 contains the formula `=STEYX(B4:B10,A4:A10)`, which computes the standard error for a given set of  $y$ - and  $x$ -values. In this case, the  $y$ - and  $x$ -values are contained in columns B and A, respectively.

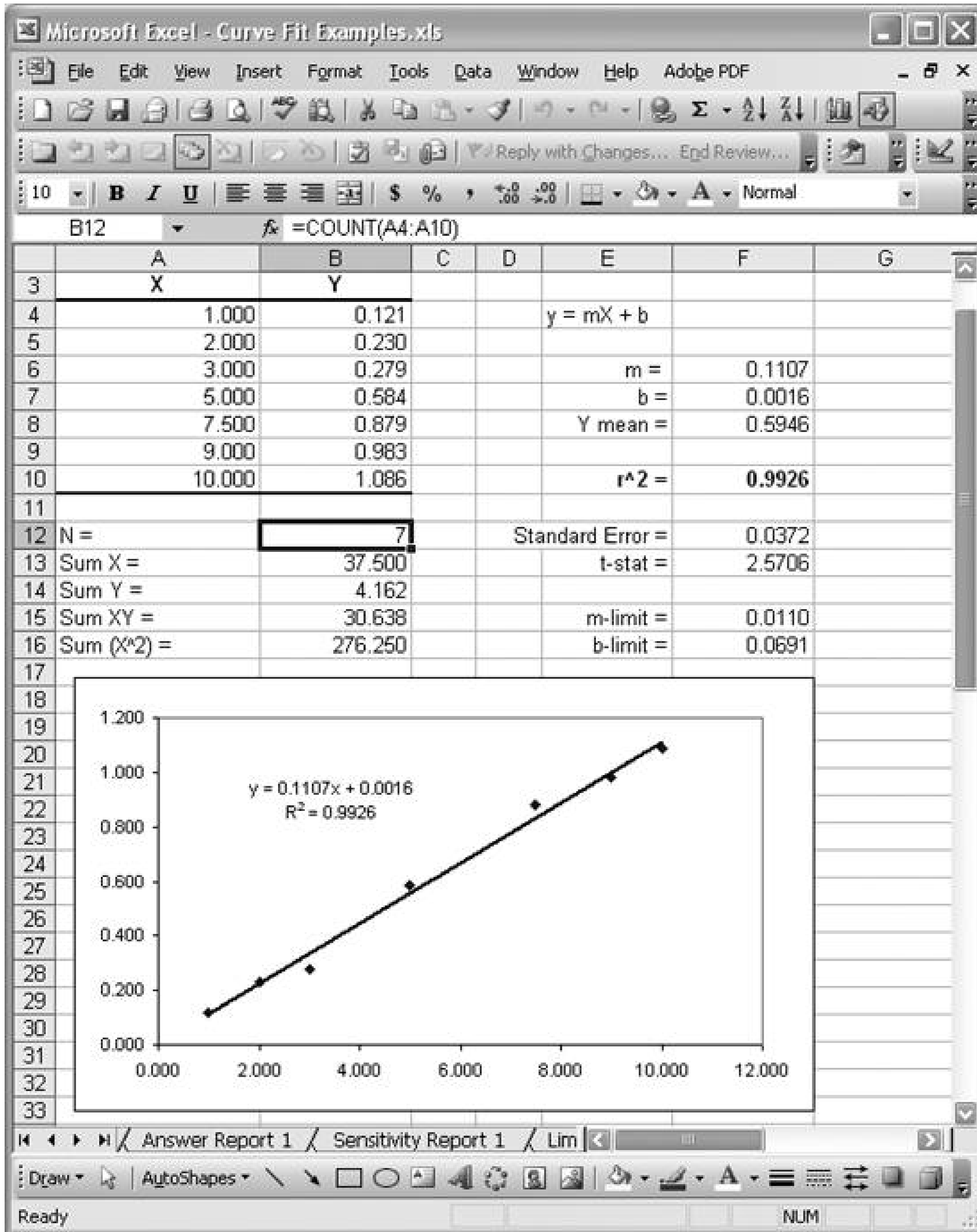
`STEYX` computes the standard error assuming a linear fit. See Excel's help topic "STEYX" for more information.

The t-statistic is computed in a manner similar to that discussed earlier. In this case, cell F13 computes the t-statistic using the formula `=TINV(0.05,(B12-2))`.

Now you need to set up a column computing the  $x - x_{\text{mean}}$  for each sample. These aren't shown in [Figure 8-16](#), but all you need to do is set up another column containing the formula `=A4-AVERAGE($A$4:$A$10)` for each  $x$ -value contained in cells A4 through A10. Once that's set up, you can compute the sum of squares of the values in that column by using the formula `=SUMSQ(L4:L10)`. In my spreadsheet, I put the  $x - x_{\text{mean}}$  calculations in the cell range L4:L10.

Finally, the confidence limits for the slope and intercept are computed in cells F15 and F16, respectively. Cell F15 contains the formula `=F13*F12/SQRT(L11)`, which

Figure 8-16. Parameter confidence intervals example



yields a confidence limit of  $\pm 0.0110$  for the slope. Cell F16 contains the formula  $=F13 * F12 * \text{SQRT}(1 / (B12 + \text{AVERAGE}(A4:A10)^2 / L11))$ , which yields a confidence limit of  $\pm 0.0691$  for the intercept.

The results of this exercise would be reported as follows: slope  $m = 0.1107 \pm 0.0110$ ; intercept  $b = 0.0016 \pm 0.0691$ .

## See Also

See [Recipes 5.3](#) and [5.6](#) for more information on Excel's support for the Student's t-distribution.



# Chapter 9. Solving Equations

[Section 9.0. Introduction](#)

[Recipe 9.1. Finding Roots Graphically](#)

[Recipe 9.2. Solving Nonlinear Equations Iteratively](#)

[Recipe 9.3. Automating Tedious Problems with VBA](#)

[Recipe 9.4. Solving Linear Systems](#)

[Recipe 9.5. Tackling Nonlinear Systems of Equations](#)

[Recipe 9.6. Using Classical Methods for Solving Equations](#)

## 9.0. Introduction

In this chapter we're going to focus on solving equations using Excel. Equation solving, in a general sense, can range from finding roots of single equations, to finding values of independent variables that yield a specific value of a dependent variable, to solving systems of nonlinear equations. There are many traditional hand and computerized methods for solving equations. Newton's method for iteratively finding roots of nonlinear equations and Gaussian elimination with back substitution for solving linear systems are just a couple of examples of classical equation-solving methods. Hildebrand's classic book *Introduction to Numerical Analysis* presents several classical methods for equation solving. The *Numerical Recipes* books also present algorithms in a variety of programming languages suitable for equation solving.<sup>[\*]</sup> The methods presented in these and other standard textbooks on numerical methods are just fine and have served scientists and engineers very well. However, in this chapter I want to show you how easy it is to leverage Excel's built-in features to solve equations with very little programming, and in some cases no programming at all beyond basic spreadsheet formulas. After seeing what Excel has to offer, if you still want to program a classic method, I'll show you how to use VBA with Excel to implement a few classic techniques.

[\*] See F. B. Hildebrand, *Introduction to Numerical Analysis*, 2nd ed. (New York: Dover Publications, 1974) and the *Numerical Recipes* books by Press, Teukolsky, Vetterling, and Flannery, published by Cambridge University Press.

There's not much to gain by using Excel or some other tool to solve equations that can readily be solved by hand after a few algebraic manipulations. Therefore, I'm going to focus almost exclusively on nonlinear equations for the examples presented in this chapter.

I should also add that while most of the example equations are presented in the form of mathematical expressions, this need not be the case for applying the techniques discussed herein. For example, you might have an "equation" composed of several formulas with a few tabular lookups and so on, that can easily be set up in a spreadsheet. This spreadsheet may then represent a nonlinear equation that you'd like to solve. You can even have calculations span multiple spreadsheets. Ultimately, whether you're dealing with a neat mathematical expression or a complicated spreadsheet, you have both independent variables and dependent variables. Much of what we try to achieve when solving these sorts of equations is finding specific values for the independent variables that yield certain dependent variables. In some cases these dependent variables are indirectly related to the equation being solved, in that we may be trying to find a root for some measure of merit (for example, in least-squares curve fitting).

This sort of calculation also falls within or overlaps the subjects of "what-if" analyses and optimization (see [Chapter 13](#)). In both of these cases, you often have to resort to the same iterative techniques used for nonlinear equation solving.

Because the recipes in this chapter rely heavily on the use of Excel's built-in tools Goal Seek and Solver, I want to go over them now to make sure you're aware of their capabilities and differences.

Goal Seek and Solver are handy tools that allow you to easily perform iterative calculations without having to program iterative solution algorithms yourself. For me, these tools are workhorses, and I use them for problems ranging from solving nonlinear systems of equations to optimization analysis to forecasting. I've even used Solver to solve the equations of motion for a hydrofoil boat that I built

and tested.

On the surface it appears as though Goal Seek and Solver do the same thing; that is, they both allow you to iteratively find a target value in a target cell by changing a value in some other related cell. There are, however, certain key differences between Goal Seek and Solver that you should be aware of when deciding which tool to use for a particular problem. The discussion to follow describes these key differences.

## Goal Seek

Goal Seek is quick and simple to use. Simply specify a target value for a target cell and specify the independent cell to vary in order to reach the target. [Figure 9-1](#) shows Goal Seek's interface. (You can access Goal Seek from the Tools menu; select Tools → Goal Seek... from the main menu bar.)

The "Set cell" field contains the target cell whose value you want to set to the value contained in the "To value" field. You tell Goal Seek which cell to change by specifying a cell in the "By changing cell" field. The target cell must contain a formula that must also directly or indirectly refer to the cell specified in the "By changing cell" field.

### Figure 9-1. Goal Seek window

You can also use Goal Seek from within VBA subroutines, so you don't have to access it manually. This comes in handy when you need to use Goal Seek repeatedly. See [Recipe 9.3](#) for an example.

You can vary only one cell using Goal Seek; thus Goal Seek is only for univariate problems. Further, you must specify the target value explicitly; that is, you have to give a numerical value as the target value. Also, Goal Seek does not allow you to impose any constraints on the independent cell (the cell in the "By changing cell" field).

According to a short article on Microsoft's support web site (Article ID 100782), Goal Seek employs a linear search algorithm using initial guesses around the value initially set in the independent cell. This means that the value contained in the independent cell when you first start Goal Seek in effect serve:



as your initial guess. This is important to remember, especially when solving problems with more than one solution. For example, if you're trying to find the roots of a cubic equation, your initial guess may result in Goal Seek converging on one solution, while a different initial guess will result in convergence on the other solution. This also implies that if Goal Seek fails to converge on a solution, you should try a different initial guess.

Goal Seek considers convergence successful if the maximum amount of change between iterations falls below some threshold. By default, this threshold is 0.001. Goal Seek will also stop if the number of iterations exceeds some maximum (100 by default). You can change these criteria by selecting Tools → Options... from the main menu bar, which opens the Options window as shown in [Figure 9-2](#).

Click the Calculation tab to reveal the calculation options. Toward the middle of the window you'll see an Iteration checkbox along with "Maximum iterations" and "Maximum change" fields. Specify appropriate values in these fields and check the Iteration checkbox so they take effect.

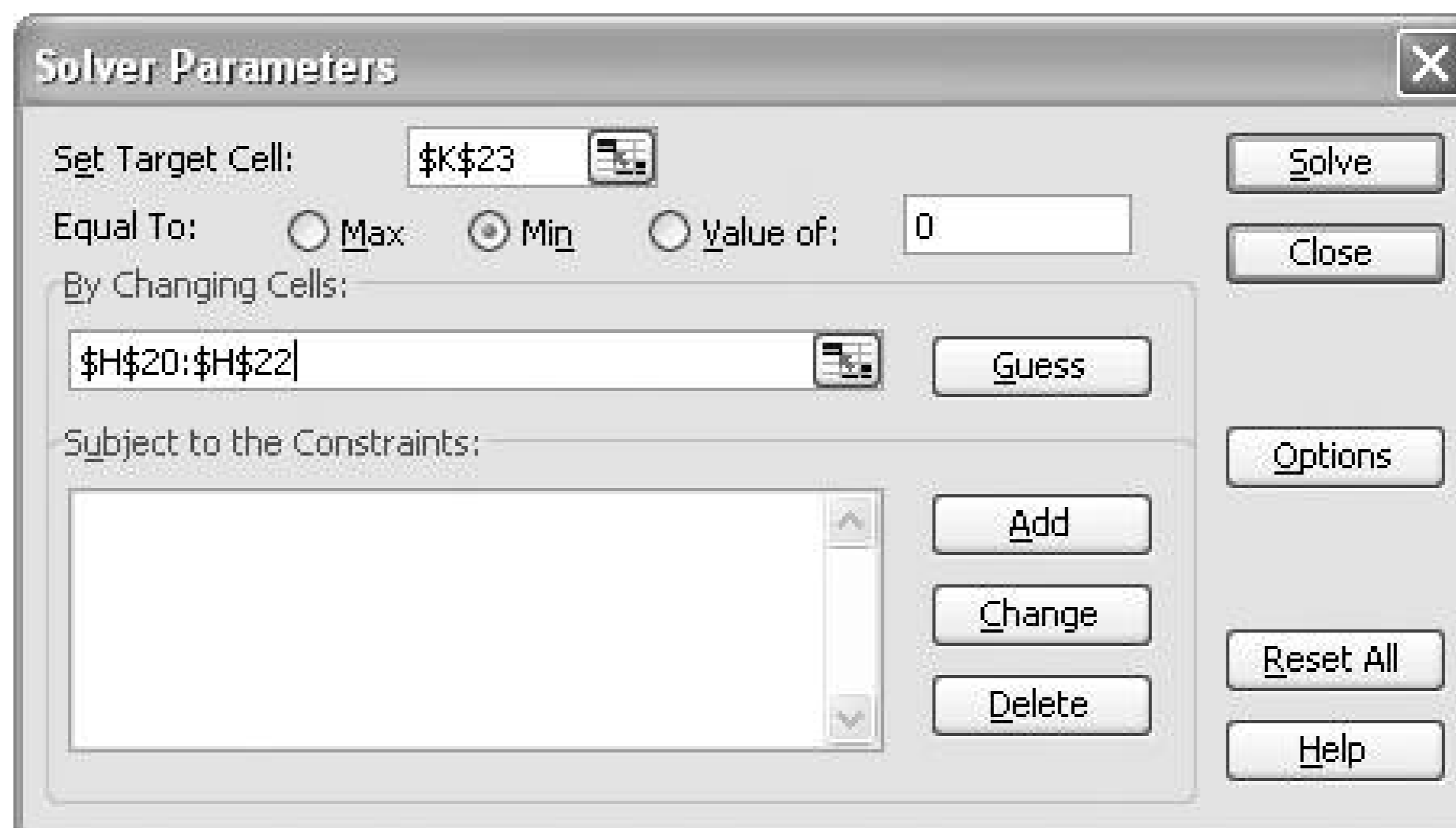
For simple iterative problems, Goal Seek is quite capable and it's quick and easy to use. However, for more complex problems or where more control over the iterative process is required, Solver is the tool of choice.

Figure 9-2. Options window

## Solver

Solver is similar to Goal Seek in that you can iteratively find a target value in a target cell by changing values in an independent cell, but Solver's features go well beyond those of Goal Seek. This is reflected in the more complex interface to Solver, which is shown in [Figure 9-3](#). (You can access Solver from the Tools menu. If you don't see it listed there, go to Tools → Add-Ins... and select Solver Add-in to make it available; then go to Tools → Solver... to open it.)

Figure 9-3. Solver Parameters window



You can use Solver for multivariate problems, where you want to vary more than one independent cell to obtain some target value in a target cell. You can specify the cells to vary in the By Changing Cells field. You can use any valid cell reference format here, with multiple cell references separated by commas. In the version of Solver that's bundled with Excel, you can select up to 200 cells to change.

You can use the Guess button to let Solver attempt to "guess" the independent cells to change for your problem. I usually shy away from this feature, though. All it does is select all nonformula cells that are referred to by the formula contained in the target cell. For simple formulas with a few independent variables where you want to change them all, this may be useful. However, for the problems I've encountered I find it selects way too many cells, many of which I don't want to vary at all (e.g., certain constants or values that I want to fix for one reason or another). On the other hand, if you're dealing with a large number of variables, it might be easier to let Solver select them all and then manually remove the ones you don't want from the By Changing Cells field.

In Solver you don't even have to specify a target value. Instead you can choose to maximize or minimize the value in the target cell. The target cell is referred to in the Set Target Cell field (it must be a single cell that contains a formula). The Equal To radio buttons allow you to specify whether to

maximize or minimize the target cell or set its value to a specific numerical value.

Solver does not stop there. You can even specify constraints on cells that are part of the problem being solved. These constraints allow you to impose equalities or upper and lower bounds on values within cells. This is a handy feature when you're trying to bracket a solution or when performing constrained optimization problems. [Recipe 9.4](#) shows how to use constraints with Solver to solve simultaneous equations. Also, [Chapter 13](#) contains recipes on using Solver for constrained optimization. Solver allows up to 100 constraints for nonlinear problems and up to 200 constraints for linear problems (in the version that's bundled with Excel).

Solver also has more available solution algorithms than does Goal Seek. For linear problems, Solver uses the *simplex* method. Solver is also capable of handling nonlinear problems by employing a *generalized reduced gradient method*. In cases where the problem being solved requires integer-valued variables, Solver also employs a *branch and bound* algorithm.

You do have a little control over these algorithms in Solver via the Options window. Press the Options button on the Solver Parameters window to display the Solver Options window (see [Figure 9-4](#)).

### Figure 9-4. Solver Options window

As you can see, there are several more options here, as opposed to the two available for Goal Seek. Solver's options include:

#### *Max Time*

Max Time represents the maximum amount of time to allow Solver to obtain a solution, after which the process will be aborted. Solver will display a message box saying that it could not find a solution within the allowable time. You can set this value as high as 32,767 seconds (jus

over 9 hours), though I doubt you'll ever have to do so.

### *Iterations*

Iterations specifies the maximum number of iterations to allow Solver while attempting to find a solution. If the maximum number of iterations is reached before Solver finds a solution, the process will be aborted and Solver will display a message box saying that it could not find a solution within the allowable number of iterations. You can set this value as high as 32,767, though in my experience I've never had to set it above 1,000 and usually leave it at the default value of 100.

### *Precision*

Precision is specified as a decimal number between 0 and 1, with a default value of  $1.0 \times 10^6$ . It is used by Solver to determine when a constraint is satisfied. The lower the value specified here, the higher the precision. Rarely do I have to change this value in practice.

### *Tolerance*

Tolerance is used to determine whether or not an integer-valued constraint is satisfied. Tolerance is expressed as a percentage, with a default value of 5%. If you're not using integer constraints, then Tolerance does not apply.

### *Convergence*

Convergence is used to determine whether or not Solver has converged on a solution. Solver considers a solution converged upon when the optimal values are within the value specified in the Convergence field over the last five solution iterations. Solver will display a message box after it converges on a solution. Solver will continue trying to converge so long as the specified maximum number of iterations or maximum time has not been reached. If either limit is reached, Solver will display a message box indicating that it could not converge.

### *Assume Linear Model*

In cases where you know your problem consists of a linear model you can specifically instruct Solver to use the simplex method of solution rather than the generalized reduced gradient method. In the event that you do specify this option, Solver will perform some checks to see if your model is indeed linear according to its criteria. If not, a message box will appear, warning you that your model does not fit the criteria for a linear model.

### *Assume Non-Negative*

You can check this to instruct Solver to automatically set lower limit constraints on all variable cells. If you know that your variables should always be greater than 0 and you want to constrain them to be so, check this option. This way you can avoid having to add a constraint manually for each variable.

### *Use Automatic Scaling*

In some problems the values of the independent variables and dependent variables may differ widely in magnitude. In these cases it's prudent to scale the model so that the magnitudes of the input and output values are of the same order. If you didn't already set up your model with scaling, then you can select this option to allow Solver to perform the scaling for you. In this case, it will scale your input and output values by dividing them by the initial value specified in the variable and target cells. This makes your initial guesses extra important for problems where scaling is required. It is always best to set your initial values to something realistic and reasonable for the problem under consideration.

### *Show Iteration Results*

If you want to follow Solver's progress as it iterates when solving a problem, you can check the Show Iteration Results option. When you do, a dialog box will appear, saying that Solver is paused and that the current iteration's results are displayed on the active worksheet. This option is useful when you want to see the results of each iteration step; for example, when Solver can't find a solution, you can use this option to try to gain insight as to why not. Under normal circumstances I do not check this option, since you actually have to press a button each time Solver is paused in order for it to proceed to the next step. This can become tedious if the solution for a particular problem requires much iteration.

### *Estimates*

Here you can specify how Solver will estimate initial values of the independent variables. Tangent uses linear extrapolation, whereas Quadratic uses quadratic extrapolation and is said to speed convergence for nonlinear problems. Quite frankly, given today's high-speed processors, I notice little difference on convergence time for typical problems.

### *Derivatives*

Solver uses finite differences to compute gradients and provides two options for your control. You can use either a forward differencing scheme or a central differencing scheme. Central differencing requires more computations but is generally more accurate. Here again, the difference in processing time using either method may be imperceptible on today's processors. I generally just use central differencing. (See [Recipe 10.6](#) for more discussion on forward differencing versus central differencing in general numerical work.)

### *Search*

You can choose to instruct Solver to use either Newton's method or the conjugate gradient method for solving a problem. Solver uses these methods to determine search directions during each iteration. Newton's method requires fewer computations than the conjugate method; however, Newton's method requires more memory. If memory is a concern for example, if you're dealing with a very large problem with many variables and constraints or if your computer memory is limited then you might want to use the conjugate method.

There are two other available Solver options that don't affect the solution process for Solver, though they are useful nonetheless. These options are Load Model and Save Model. Basically, Solver allows

you to save Solver models and reload them. This can be quite handy when you're trying different combinations of variables and constraints along with solution options to see which model works best for a given problem. It also allows you to recall a model later (without having to remember all the model settings) if you decide to come back and redo an analysis.

Press the Save Model button to save a model. When you do, a small dialog box will appear, asking you to select a range of cells within which the model data will be saved. You should select a range of cells away from other data and formulas in your spreadsheet so that you don't overwrite anything. You need only select a single cell and Solver will fill in a column of data, starting with the selected cell, consisting of a number of cells totaling 3 plus the number of constraints in your model.

To recall a model later, simply go to the Solver Options window and press the Load Model button. You will then see a small dialog box asking you to select the range of cells containing the saved model. You must select the entire range of cells previously saved. Selecting only the uppermost cell will not work.

Another handy Solver feature not included with Goal Seek is the Solver Results feature. Upon finding a solution, Solver will notify you via a message box like that shown in [Figure 9-5](#).

### Figure 9-5. Solver Results window

Here you can choose to keep the solution found by Solver, or revert back to your original values. Moreover, you can choose to generate any of three Solver reports: Answer, Sensitivity, or Limits. Click on each report name that you want. You can click one or all or any combination. When you press the OK button, Solver will create a new worksheet for each selected report. These reports are useful for interpreting the results of a Solver solution, particularly for constrained optimization problems. I talk more about these reports in [Chapter 13](#), which covers optimization.

You can always refer to the Solver help documents in Excel by pressing the help buttons that appear on the various Solver-related windows. For more interesting reading about the history and inner workings of Solver, check out the paper "Design and Use of the Microsoft Excel Solver," by Daniel Fylstra, Leon Lasdon, John Watson, and Allan Waren, published in *Interfaces* 28, no. 5 (1998).

## Recipe 9.1. Finding Roots Graphically

### Problem

You want to find the roots of an equation graphically.

### Solution

Write the equation in the form  $y = g(x)$ , compute values for  $y$  for a given range of  $x$ -values, and plot the results using Excel's charting features. Refer to [Chapter 4](#) for more information on charting in Excel.

### Discussion

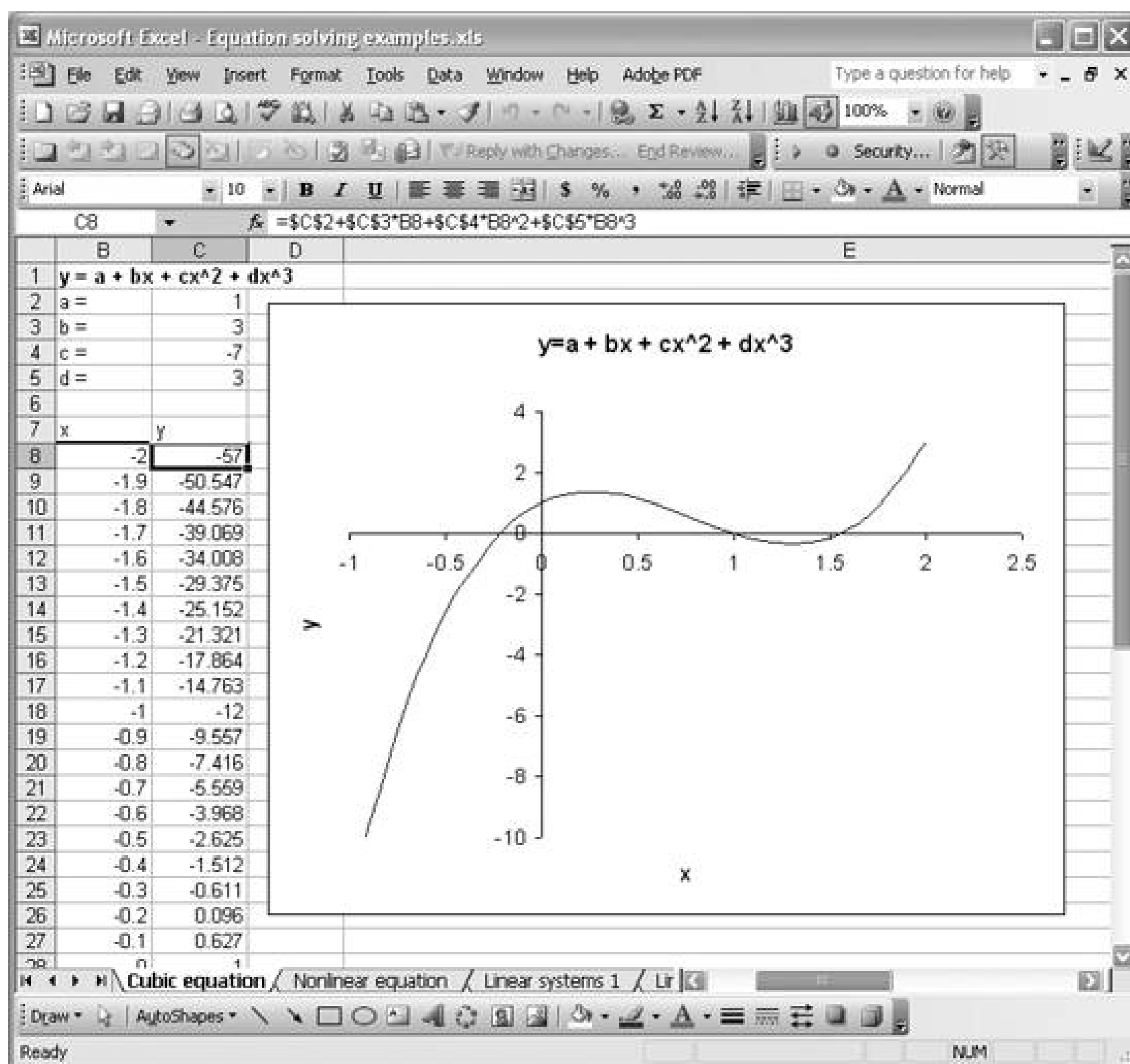
When faced with finding roots to an equation that you can't solve by hand, the first thing I recommend you do is chart it. This will give you valuable insight into the nature of the equation and where its roots lie. If you can write the equation in the form of  $y = g(x)$ , where  $x$  is the independent variable and  $y$  the dependent variable, then you can easily set up a few columns computing  $y$  for various values of  $x$ .

This may all seem trivial and frankly, in terms of spreadsheet operations, it is very simple to perform. However, when dealing with a particularly ugly equation with potentially multiple roots, you'd better have a good idea of where those roots lie. This is because most iterative methods for solving equations require an initial guess as to where the roots lie. Some methods require two initial guesses while others require you to bracket the roots. The success of these various methods for finding roots lies in the quality of your initial guess. (See the other recipes in this chapter for specific examples.)

Consider the third-order polynomial equation:

This equation has, of course, three roots. That is, there are three values of  $x$  that yield a  $y$ -value of zero. This is readily confirmed by plotting the curve for  $y$  versus  $x$  in Excel. [Figure 9-6](#) shows how this particular equation looks when plotted in Excel.

Figure 9-6. Third-order polynomial plot



As you can see, there are indeed three zero crossings; that is, there are three values of  $x$  that yield zero for  $y$ . To plot this curve, I set up two columns: one containing given values for  $x$  and the other containing calculated values for  $y$ . These are shown in columns B and C, respectively, in [Figure 9-6](#). I assumed arbitrary coefficients for this example and placed them in cells C2 through C5. The formula bar shows how I constructed the formula for computing  $y$ -values corresponding to each  $x$ . (Cell C8 is selected in the screenshot, so you see its formula in the formula bar.) The formula is of the form  $=\$C\$2+\$C\$3*B8+\$C\$4*B8^2+\$C\$5*B8^3$ .

At this point, you have a clear picture of how this equation behaves and where its roots lie. You have a few options now to actually find the particular values for  $x$  that yield  $y = 0$ . You can approximate the  $x$ s by interpolating between the tabulated values contained in columns B and C. Essentially you look for the  $x$ s corresponding to where the  $y$ -values go from negative to positive and positive to negative and then interpolate between those tabulated values. Alternatively, you can use iterative techniques to find the  $x$ -values. Such techniques are illustrated in [Recipes 9.2](#) and [9.3](#). [Recipe 9.6](#) shows how to find the roots of the cubic equation discussed here, using Newton's method and the secant method.

Now, let me ask you what happens when you're dealing with an equation that you can't write in the



form  $y = g(x)$ . Well, that presents a problem when you attempt to plot it using the straightforward approach discussed in this recipe. Fortunately, Excel offers other tools that you can leverage to get around such a difficulty. Check out [Recipe 9.2](#) to see what I mean.



## Recipe 9.2. Solving Nonlinear Equations Iteratively

### Problem

You'd like to solve a nonlinear equation that requires an iterative solution and are not sure how to approach it in Excel.

### Solution

Use Goal Seek or Solver. See the introduction to this chapter for more information on how these two tools differ as well as their respective advantages and disadvantages.

### Discussion

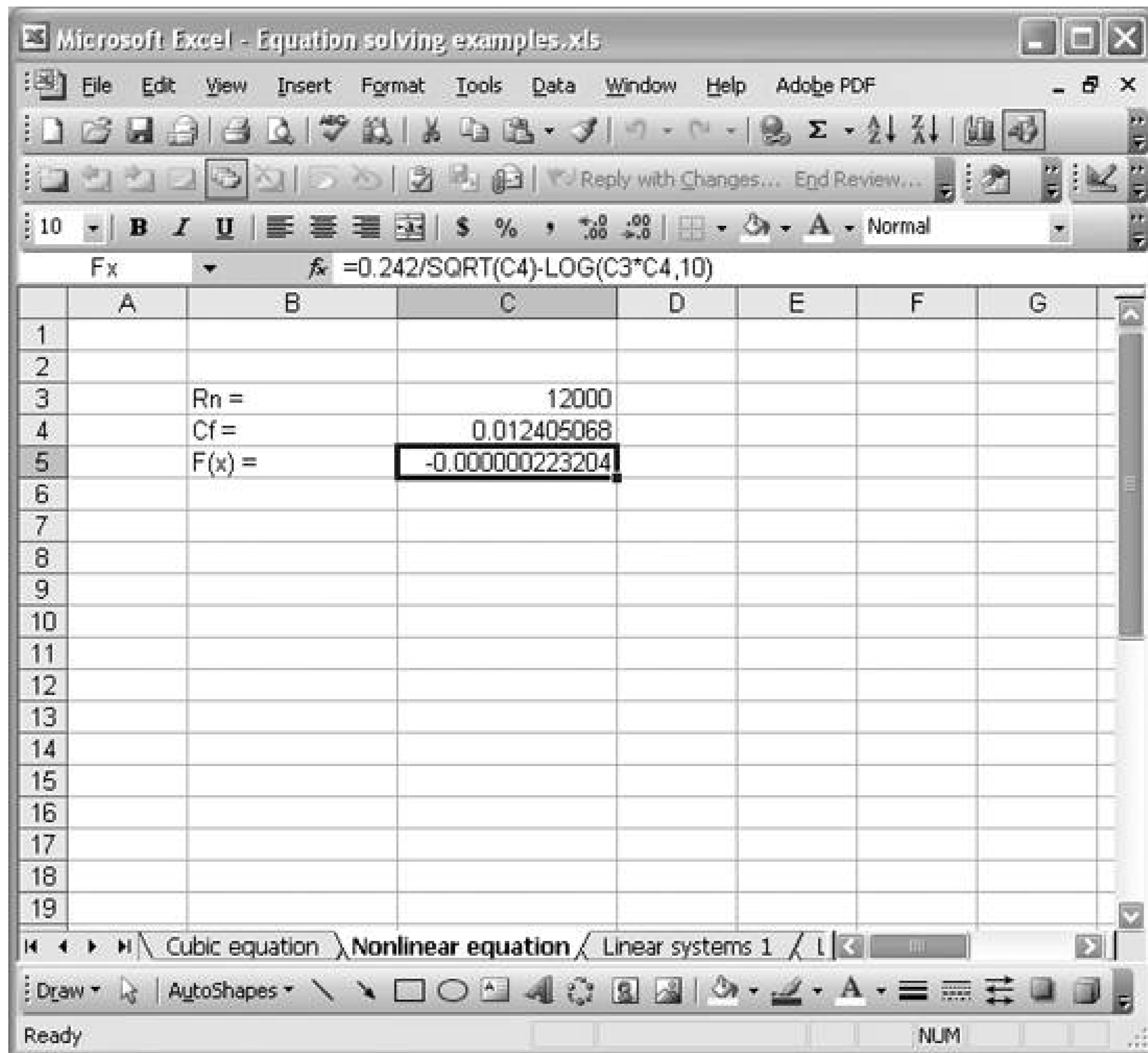
In this recipe, I want to show you by way of example how to use Excel's built-in Solver and Goal Seek tools to solve a nonlinear equation. The equation we'll consider is:

This equation is used to estimate a frictional drag coefficient,  $C_f$ , as a function of the *Reynolds number*,  $R_N$ , for some ship resistance calculations. There are other equations for this same purpose but I chose this classical one because it can't be written in an explicit form of  $y = g(x)$ . Instead, you have to resort to some iterative method to find a value of  $C_f$  corresponding to some given value of  $R_N$ .

I'll show you how to use both Goal Seek and Solver to solve this example problem. In both cases the equation must first be rearranged in the following form:

Now we can use Solver or Goal Seek to iteratively find a value for  $C_f$  given a value for  $R_N$  that makes the righthand side of this equation equal to 0. But first, you need to enter the righthand side of this equation as a spreadsheet formula, as shown in [Figure 9-7](#).

Figure 9-7. Nonlinear equation example

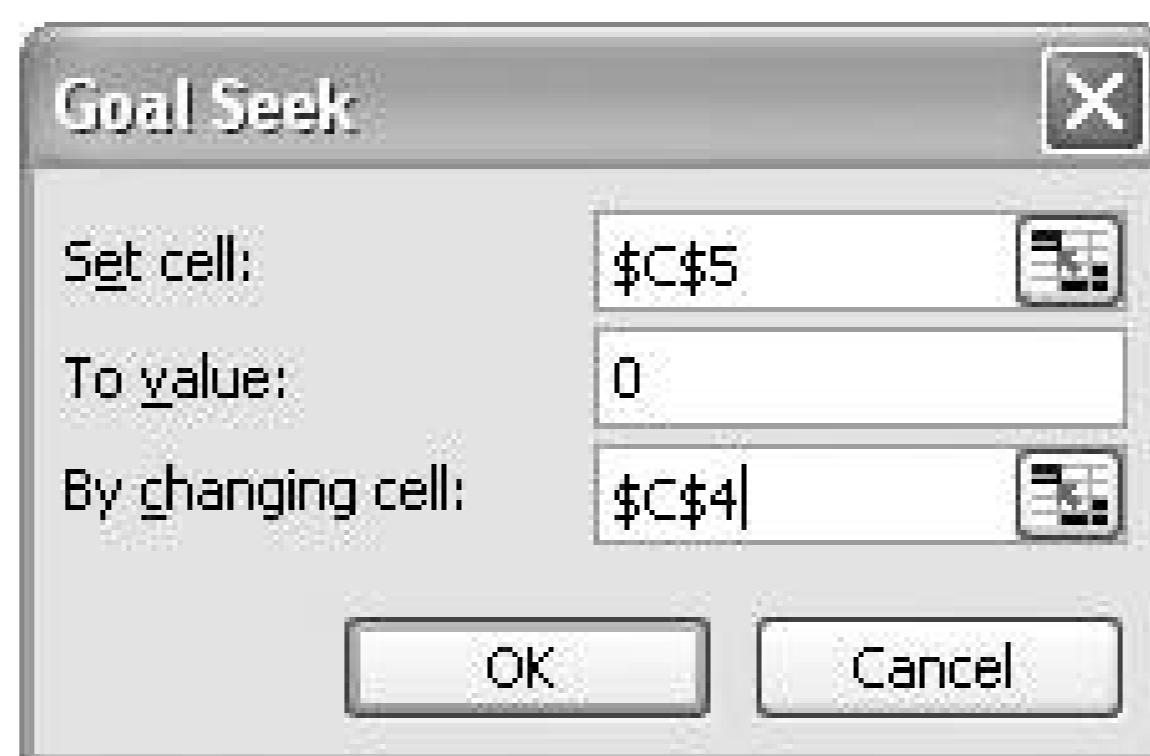


Cell C5 contains a worksheet formula representing the righthand side of our equation. The formula is  $=0.242/\text{SQRT}(C4)-\text{LOG}(C3*C4,10)$ , as shown in the formula bar in [Figure 9-7](#). Cell C3 contains the given Reynolds number, while cell C4 contains the corresponding friction coefficient, found using either Solver or Goal Seek. Before actually using Solver or Goal Seek, you should enter your initial guess for the friction coefficient in cell C4. As mentioned in the introduction to this chapter, you should pick an initial guess that is reasonable for the problem at hand.

### Finding $C_f$ with Goal Seek

To use Goal Seek to find a solution for this example, select Tools → Goal Seek... from the main menu bar to open the Goal Seek window shown in [Figure 9-8](#).

Figure 9-8. Goal Seek window for nonlinear equation example



In the "Set cell" field, type in **C5**, or press the little icon to the right of the edit field to temporarily go back to the spreadsheet and click cell C5 to select it (press Enter when you've selected it to return to the Goal Seek window). Remember, cell C5 contains the formula representing the righthand side of our equation, and we want the result of this formula to be zero. Therefore, in the "To value" field, type in **0**. That's our target value. Now, set the "By changing cell" field to C4, which contains the friction coefficient value. When you press the OK button, Goal Seek will vary the value in C4 until the formula in C5 is approximately 0. The result for this example is shown in [Figure 9-7](#), where the residual shown in cell C5 is  $-2.23 \times 10^{-7}$ .

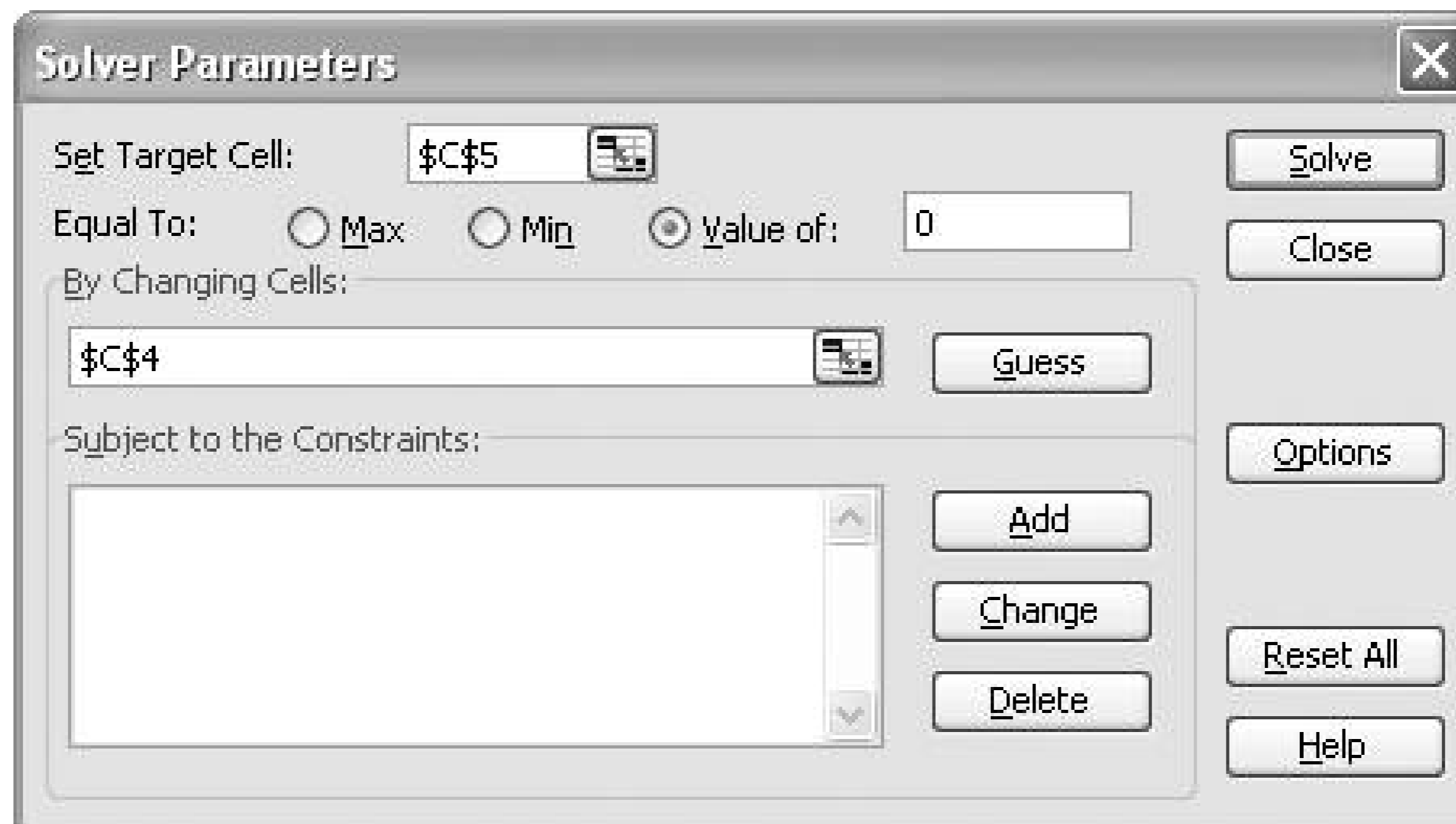
I should mention that my initial guess (i.e., the value I started with in cell C4) was  $1 \times 10^7$ . I would have chosen 0 since I know the friction coefficient should be a pretty small number, but I didn't, because it would have resulted in a divide-by-zero error for this problem. Running Goal Seek with the divide-by-zero error would result in Goal Seek failing with a "Formula in cell must result in a number" error.

A residual of  $-2.23 \times 10^{-7}$  isn't that bad, but we can do a little better. As discussed in the introduction, you can change the convergence settings for Goal Seek by going to the Options window **Select Tools > Options...** from the main menu bar to open the Options window (shown earlier in [Figure 9-2](#)). On the Calculation panel, I checked Iteration, set "Maximum iterations" to 5,000, and set "Maximum change" to  $1 \times 10^{-8}$ . With these settings, Goal Seek finds a solution with a much smaller residual,  $1.2 \times 10^{-13}$ . The corresponding coefficient of friction for the given Reynolds number is 0.0124.

## Finding $C_f$ with Solver

Goal Seek works well for this simple example, and in practice you would not have to rely on Solver for this problem. For illustration purposes, however, I want to show you how to apply Solver to the same problem. The spreadsheet set up in this case is the same as in the case of using Goal Seek. All you do differently is select **Tools > Solver...** from the main menu bar to open Solver instead of Goal Seek. The Solver Parameters window for this example is shown in [Figure 9-9](#).

Figure 9-9. Solver Parameters window for nonlinear equation example



Here, I set the Set Target Cell field to C5 and set the Equal To option to a value of 0. Again, C5 contains the righthand side of our equation, which we want equal to 0. Next, I set the By Changing Cells field to cell C4, which contains the initial value for the friction coefficient. Pressing the Solve button initiates the solution process.

Solver does indeed find a solution, yielding a friction coefficient of 0.0124 with a residual of  $-4.87 \times 10^{-12}$ . These values are comparable with the results obtained using Goal Seek.

Just for fun, try this example again, but this time use an initial guess for the friction coefficient of 1 or 5 or some large number. In this case, Solver actually fails to find a solution! What happens is that Solver overshoots the actual solution and ends up trying a value for the friction coefficient of 0 for an iteration. This of course yields a divide by zero error, which causes Solver to fail prematurely. This little exercise highlights the importance of selecting a reasonable initial guess. It further shows that if your first attempt fails, you should try a different initial guess as the new one might work.

## Recipe 9.3. Automating Tedious Problems with VBA

### Problem

You now know how to use Solver and Goal Seek to solve nonlinear equations, but the trouble is you need to solve such an equation over a range of values, making manual, repeated application of Solver and Goal Seek tedious.

### Solution

Automate this process using VBA, where you can programmatically call Solver or Goal Seek.

### Discussion

Solver and Goal Seek are very handy tools for solving a wide variety of problems in Excel. What's even cooler is that you can use these tools directly in VBA code, without ever having to manually open their respective windows, click cells, and so on. This means you can automate many tasks that require Solver or Goal Seek, saving yourself a lot of time.

Let's reconsider the example equation from the [Recipe 9.2](#). The equation is repeated here for convenience:

What if you wanted to plot this equation? If you wanted to plot a curve of  $C_f$  versus  $R_M$ , you'd have to use Solver or Goal Seek a number of times for each point along the curve you intend to plot. Maybe you don't want to plot this curve, but instead want to generate a lookup table with  $C_f$  over a range of  $R_M$  values. In either case, what a pain it would be to have to manually apply Solver or Goal Seek repeatedly for every value of  $R_M$ . Imagine an even more complicated case where there's another parameter in the equation and you want to vary it systematically as well. Now you're really taking about tedium!

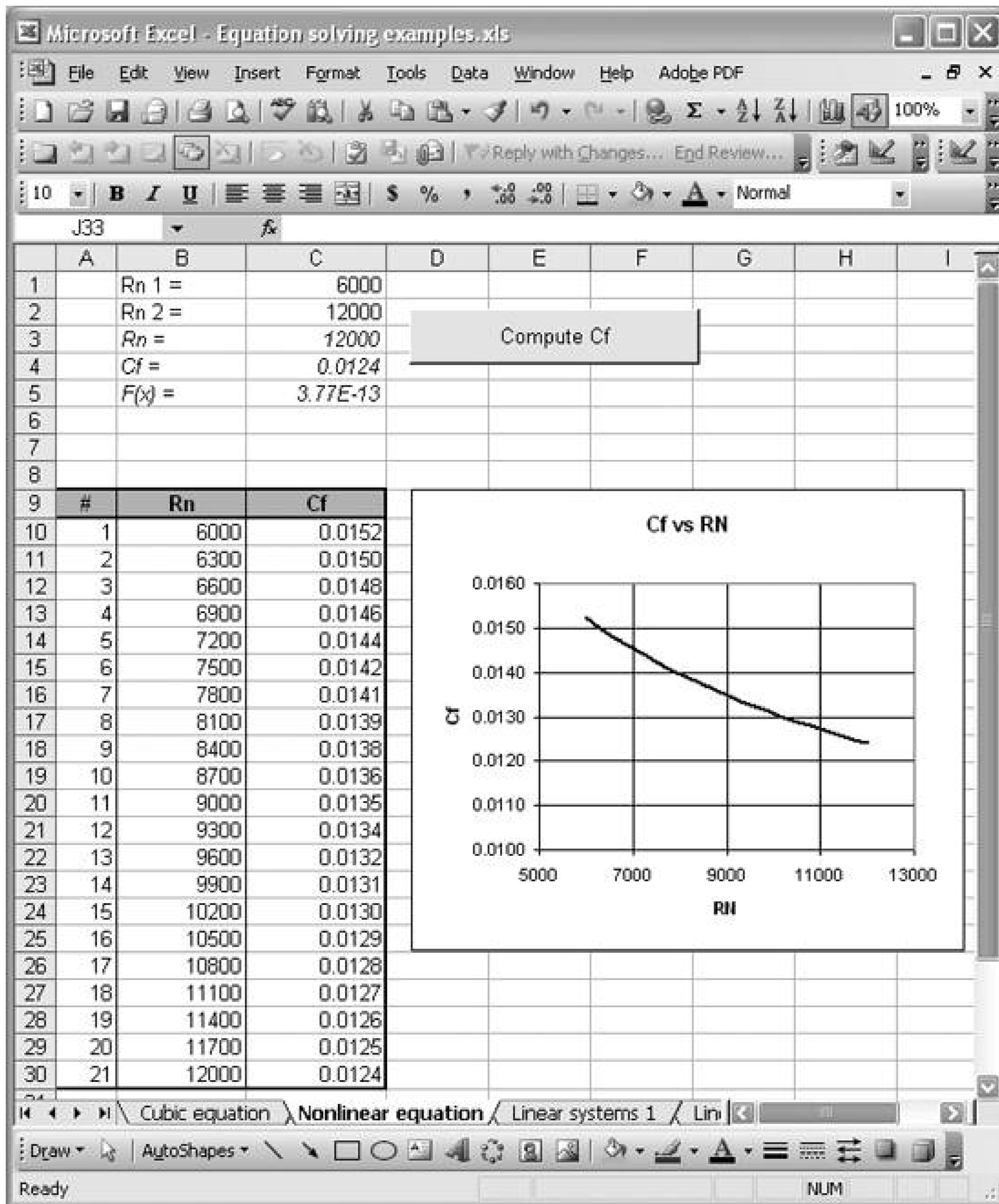
Thankfully, the developers of Solver and Goal Seek have exposed their functionality to us in VBA so we can automate their use.

### Automating Goal Seek

To show you how to automate Goal Seek, I'll start with the spreadsheet shown in [Figure 9-7](#) and

make some modifications to it. What I want to do is add a VBA subroutine that makes calls to Goal Seek to compute values for  $C_f$  for a range of  $R_M$  values. In order to call the VBA subroutine, I want to add a button control to the spreadsheet that can be clicked to start the solution process. Also, I want to set up a table that will be filled in automatically with the  $C_f$  and  $R_M$  values as they are computed. Finally, I'd like to add a chart to show the resulting curve. The modified spreadsheet is shown in [Figure 9-10](#).

Figure 9-10. Goal Seek automation example



As you can see, there are quite a few additions here, so I'll explain each one. The original spreadsheet elements are shown in italics in [Figure 9-10](#); everything else is new for this example.

The first thing I added were entries in cells C1 and C2 to store starting and ending values for the range of  $R_N$  values I wanted to consider. I also added text labels to the left of these cells.

Then I constructed a table in cells A9 through C30 to hold values of  $C_f$  versus  $R_N$ . Initially, this table contained only the headings in row 9 and row number labels in column A. The  $C_f$  and  $R_N$  cells were left blank because they will get filled in later with a custom VBA subroutine that I'll show you shortly.



With this table set up, I then created a chart to plot  $C_f$  versus  $R_M$ . Since the table is initially empty, the chart didn't display any curve. However, having it set up ahead of time means that when the custom VBA subroutine is finished computing the required data, the chart will automatically be updated, showing the resulting curve. (Refer to [Chapter 4](#) to learn how to prepare charts in Excel if you need a refresher.)

The final element to add is the button shown in [Figure 9-10](#), labeled Compute Cf. Excel allows you to add several standard windows controls to spreadsheets, enabling you to prepare fairly rich GUI (graphical user interface) forms like the ones you see in standard Windows applications. For the purposes of this example, I'll show you how to add a simple button control and associate it with some VBA code, so that when the button is pressed it will actually do something. In this case, the VBA code will compute  $C_f$  over a range of  $R_M$  and fill in the data table.

To add the button, you need to make sure the Control Toolbox toolbar is displayed in Excel. The Control Toolbox toolbar is shown in [Figure 9-11](#).

Figure 9-11. Control Toolbox toolbar



If you don't see it, select View → Toolbars → Control Toolbox from Excel's main menu bar.

To add a button, control-click the Command Button icon (see [Figure 9-11](#)) and then click anywhere in your spreadsheet where you'd like to place the button. Upon doing so, you should see a new button on your spreadsheet. Initially, it will have some resizing handles (circles) located around it so you can click and drag to resize the button. You can also click and drag the button itself to relocate it.

Initially, when you place a new control on a spreadsheet, Excel enters *design mode* and the Design Mode button on the Control Toolbox toolbar is depressed to indicate this. In design mode you can resize and move controls, whereas when you're not in design mode, clicking on a control will activate it. You can toggle design mode by pressing the Design Mode button on the toolbar.

The controls available from the Control Toolbox are ActiveX controls. While in design mode, you can right-click on the control and select Properties from the pop-up menu to open a window displaying all of the control's properties. You can edit many characteristics of the control from the properties window. For example, the Caption property allows you to change the text shown on the control itself. In this example, I changed the caption to Compute Cf, as you can see in [Figure 9-10](#).

With the button added, you need to associate it with some VBA code so that it will actually do something when you click it. So the next step is to add a custom VBA subroutine and then associate it with the new button.

[Example 9-1](#) shows the custom VBA subroutine I prepared for this example. (Refer to [Chapter 2](#) to learn how to add VBA subroutines such as this one to your spreadsheets.)

### Example 9-1. ComputeCf

```
Public Sub ComputeCf( )
    Dim inc As Double

    With Worksheets("Nonlinear equation")
        inc = (.Range("Rn_2") - .Range("Rn_1")) / 20

        For i = 0 To 20
            .Range("Rn") = .Range("Rn_1") + (inc * i)
            .Range("Fx").GoalSeek goal:=0, ChangingCell:=.Range("Cf")
            .Cells(10 + i, 2) = .Range("Rn")
            .Cells(10 + i, 3) = .Range("Cf")
        Next i
    End With
End Sub
```

This subroutine, called `ComputeCf`, extracts the  $R_N$  range from cells C1 and C2 in the spreadsheet and then computes 20 corresponding values of  $C_f$  using Goal Seek. As each  $C_f$  value is computed, the results are placed in the table contained in cells A9 through C30 of the spreadsheet.

The first line of code within the subroutine declares a local variable, `inc`, that is the increment (to be computed later) between successive  $R_N$  values in the range.

The next statement is a `With` statement. `With` statements are useful in object-oriented languages such as VBA where members of objects are accessed with the `.` syntax that you see in this example. `With` statements allow you to specify which object to use as the default object when accessing members. This way you can leave off the parent object name, saving yourself some typing. For example, here I'm using the statement `With Worksheets("Nonlinear equation")`. I'm telling VBA that I want access to the worksheet object named "Nonlinear equation." That's the name I gave to the worksheet we're using in this example. You can see this name in the worksheet tab at the bottom of [Figure 9-10](#). Now, every time I use a statement like `.Range("Rn_2")`, in this case accessing the range object, VBA defaults to accessing the range object of the `Nonlinear equation` worksheet object. Without using the `With` statement, you'd have to write `Worksheets("Nonlinear equation").Range("Rn_2")`.

`With` statements must be bracketed with an `End With` statement, as shown in [Example 9-1](#). This limits the scope of the `With` statement to apply to everything within the `With` and `End With` statements.

The next statement after the `With` statement computes `inc` by accessing the cells via range objects

containing the starting and ending  $R_N$  values, as discussed earlier. Notice that I'm using the name labels `Rn_2` and `Rn_1` here to access the cells. I find names far more readable than standard cell references. Take a look at [Recipe 1.14](#) to learn how to add cell names like these.

After `inc` is computed, the subroutine enters a `for` loop iterating over the range of  $R_N$  values. For each  $R_N$ , Goal Seek is used to iteratively find a corresponding  $C_f$  value. Let me explain each line of the `for` loop:

```
.Range("Rn") = .Range("Rn_1") + (inc * i)
```

This line computes the next  $R_N$  value and stores it in the cell named `Rn` (cell C3).

```
.Range("Fx").GoalSeek goal:=0, ChangingCell:=.Range("Cf")
```

This is the line that actually calls Goal Seek for the cell named `Fx` (cell C4). `Fx` is the target cell whose value we want to set to 0. In the VBA code shown here, the target value is specified by `goal:=0`. We also specify the By Changing Cell with `ChangingCell:=.Range("Cf")`, telling Goal Seek to vary  $C_f$  until `Fx` is zero.

```
.Cells(10 + i, 2) = .Range("Rn")
```

This line adds the `Rn` value for this iteration in our data table starting at cell B10 and incrementing by 1 row at each iteration through the loop. Notice here that I'm using R1C1-style cell references for convenience. (See [Recipe 1.6](#) for more information on this style.)

```
.Cells(10 + i, 3) = .Range("Cf")
```

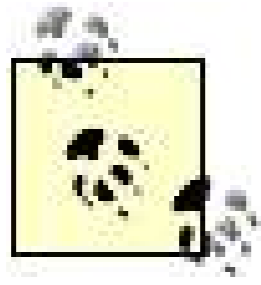
This line adds the  $C_f$  value found by Goal Seek to the data table. Here again, I'm using the R1C1 reference style.

With the `ComputeCf` subroutine written, you can now associate it with the button we added earlier. Enter design mode for the control as discussed earlier and then right-click on the button to reveal a pop-up menu. Select View Code from this menu. This will switch you over to the VBA editor, where you should see a new subroutine for this button. Our new subroutine is shown in [Example 9-2](#).

### Example 9-2. New button subroutine

```
Private Sub CommandButton1_Click( )
End Sub
```

This is the click event handler for the button you added earlier. The button's default name is `CommandButton1`, so this subroutine is named `CommandButton1_Click`.



You can change the name of a control by changing the `Name` property in the ActiveX control's property window. In design mode, right-click on the control and select Properties from the pop-up menu to display the Properties window. You can then make the appropriate changes to the control's properties.

Initially, this subroutine is empty, so you need to add a call to our custom subroutine, `ComputeCf`, as shown in [Example 9-3](#).

### Example 9-3. Add call to ComputeCf

```
Private Sub CommandButton1_Click( )
    ComputeCf
End Sub
```

When you click the button in Excel now (with design mode off), it will execute the custom subroutine computing  $C_f$  values over the desired  $R_N$  range. You'll also see the results displayed in the table and on the chart. Should you decide you want to compute new  $C_f$  values over a different range of  $R_N$  values, you need only enter the new range in cells C1 and C2 and press the button again.

Excel has another set of form controls available from the Forms toolbar (select View → Toolbars → Forms to show it). These controls look like the ActiveX controls we used in this example, but give you more control. The Form controls work a little differently in terms of how you edit their properties and assign functionality to them. In general, the ActiveX controls are newer and more powerful, while the Forms controls are older and maintained for backward compatibility.

## Automating Solver

You can automate Solver in a manner similar to the way in which we automated Goal Seek earlier. In fact, we can modify the earlier example slightly to use Solver instead of Goal Seek. All of the spreadsheet elements can remain the same as before and we need only make some VBA code modifications in the `ComputeCf` subroutine. The new `ComputeCf` subroutine is shown in [Example 9-4](#).

### Example 9-4. ComputeCf using Solver

```

Public Sub ComputeCf( )
  Dim inc As Double

  With Worksheets("Nonlinear equation")
    inc = (.Range("Rn_2") - .Range("Rn_1")) / 20

    For i = 0 To 20
      .Range("Rn") = .Range("Rn_1") + (inc * i)

      ' New code starts here:
      .Range("Cf") = 0.001

      SolverOK SetCell:=Range("Fx"), MaxMinVal:=3, ValueOf:=0, _
        ByChange:=Range("Cf")

      SolverSolve UserFinish:=True

      SolverFinish KeepFinal:=1

      .Cells(10 + i, 2) = .Range("Rn")
      .Cells(10 + i, 3) = .Range("Cf")
    Next i
  End With
End Sub

```

The new code consists of the four lines of code following the ' `New code starts here:` comment. The new lines are:

```
.Range("Cf") = 0.001
```

This line sets the initial value for `cf` to 0.001. This serves as our initial guess prior to calling Solver.

```
SolverOK SetCell:=Range("Fx"), MaxMinVal:=3, ValueOf:=0, ByChange:=Range("Cf")
```

This line calls the Solver subroutine `SolverOK` to initialize the Solver model. The target cell is set to `Fx` by `SetCell:=Range("Fx")`. We want to set it to a specific value, so we set `MaxMinVal:=3`. If we wanted to maximize the target, we'd set `MaxMinVal` to 1; if we wanted to minimize it, we'd set `MaxMinVal` to 2. The target value is set via `ValueOf:=0`, which tells Solver to set the target value to 0. `ByChange:=Range("Cf")` tells Solver to vary the value in the cell named `cf` to find a solution.

```
SolverSolve UserFinish:=True
```

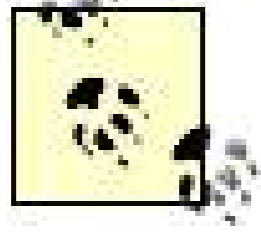
Calling `SolverSolve` actually starts Solver. We specify `UserFinish:=True` to let Solver know that

we do not want it to display the Results dialog box every time it finds a solution. If we set this to `false`, Solver will display the Results dialog, which can be useful for debugging.

```
SolverFinish KeepFinal:=1
```

This line tells Solver that we want to keep the results it finds.

Upon making these code changes and running the code by pressing the button on the example spreadsheet, you should see results that are very similar to those obtained using Goal Seek.



If after adding the new code you attempt to run the subroutine and get a "Sub or Function not Defined" error, you need to make sure Solver is being referenced in your VBA editor. To do so, in the VBA editor select Tools References... from the main menu bar. In the References window that appears, make sure SOLVER is checked and press OK.

## See Also

You can control many more aspects of Solver using VBA code than those I showed here. For more information, see the Solver help in Excel and check out Microsoft's support web site, <http://support.microsoft.com>, where you can search for technical articles on Solver (among many other Microsoft product-related topics). The article "How to create Visual Basic macros by using Excel Solver in Excel 97" (Article ID 843304) should get you well on your way to becoming a Solver power user.

## Recipe 9.4. Solving Linear Systems

### Problem

You'd like to solve a linear system of equations using Excel.

### Solution

You can take one of several approaches to solving linear systems in Excel. You could, for example, use VBA to program any of the standard numerical algorithms for solving systems of equations. While this approach would undoubtedly work, it's the most time-consuming since you'd have to program and test the algorithm. Since Excel provides built-in support for matrix operations, you could frame the problem in the form of a matrix problem and perform the necessary matrix inversion to find a solution. An easier way is to use Solver. In the discussion that follows, I'll show you the latter two approaches.

### Discussion

If you can frame the system of equations in the form of a matrix equation, then you can use direct matrix inversion to find a solution. If you can't or don't want to do so, or if you're dealing with nonlinear equations, you can use one of Solver's several approaches.

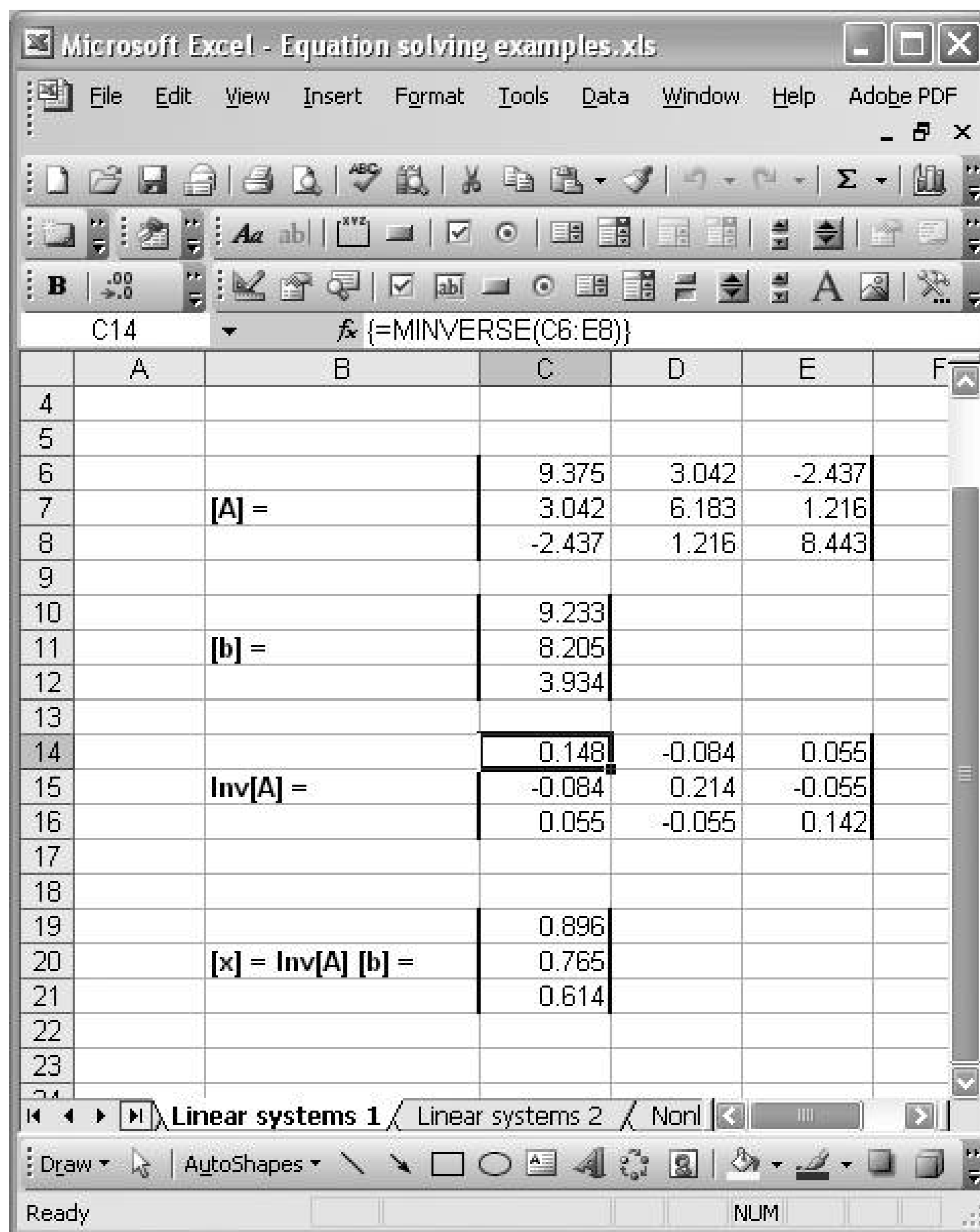
### Matrix inversion

In [Recipe 7.10](#), I showed you how to use Excel's built-in matrix functions to multiply and invert matrices. You can use these same functions to solve a linear system of equations of the form:

The approach to solving this problem is a straightforward application of the built-in matrix functions, where you can find the unknown  $x$ s that satisfy this equation by solving the following matrix equation:

For example, let's say you're given  $[A]$  and  $[b]$  matrices as shown in [Figure 9-12](#).

Figure 9-12. Solving a matrix equation



The first step in the solution is to invert the  $[A]$  matrix. To do so, enter the matrix formula `=MINVERSE(C6:E8)`. Remember, this is a matrix formula entered in a range of cells; you have to press Ctrl-Shift-Enter to enter the formula. The results of this matrix inversion are shown in [Figure 9-12](#) in cells C14 through E16.

The next and final step is to multiply the inverse of the  $[A]$  matrix by the  $[b]$  matrix using the formula `=MMULT(C14:E16,C10:C12)`. Here again, this is a matrix formula entered in a range of cells. The final results are shown in [Figure 9-12](#) in cells C19 through C21.

### Using Solver with constraints



Instead of using matrix functions, which can get a little unwieldy for large matrices, you can use Solver to solve a linear system (or a nonlinear one, as discussed in [Recipe 9.5](#)). In this example, I'll show you how to solve the same linear system discussed earlier by using Solver with constraints. [Figure 9-13](#) shows how this problem is set up.

Figure 9-13. Solving a linear system with Solver by using constraints

	C	D	E	F	G	H	I	J	
5	Using Solver with Constraints:								
6	$a_1x_1 + a_2x_2 + a_3x_3 = b$								
7									
8		a1	a2	a3	x	b	a1*x1 + a2*x2 + a3*x3		
9	(1)	9.375	3.042	-2.437	0.896	9.233	9.233		
10	(2)	3.042	6.183	1.216	0.765	8.205	8.205		
11	(3)	-2.437	1.216	8.443	0.614	3.934	3.934		
12									
13									
14									
15									

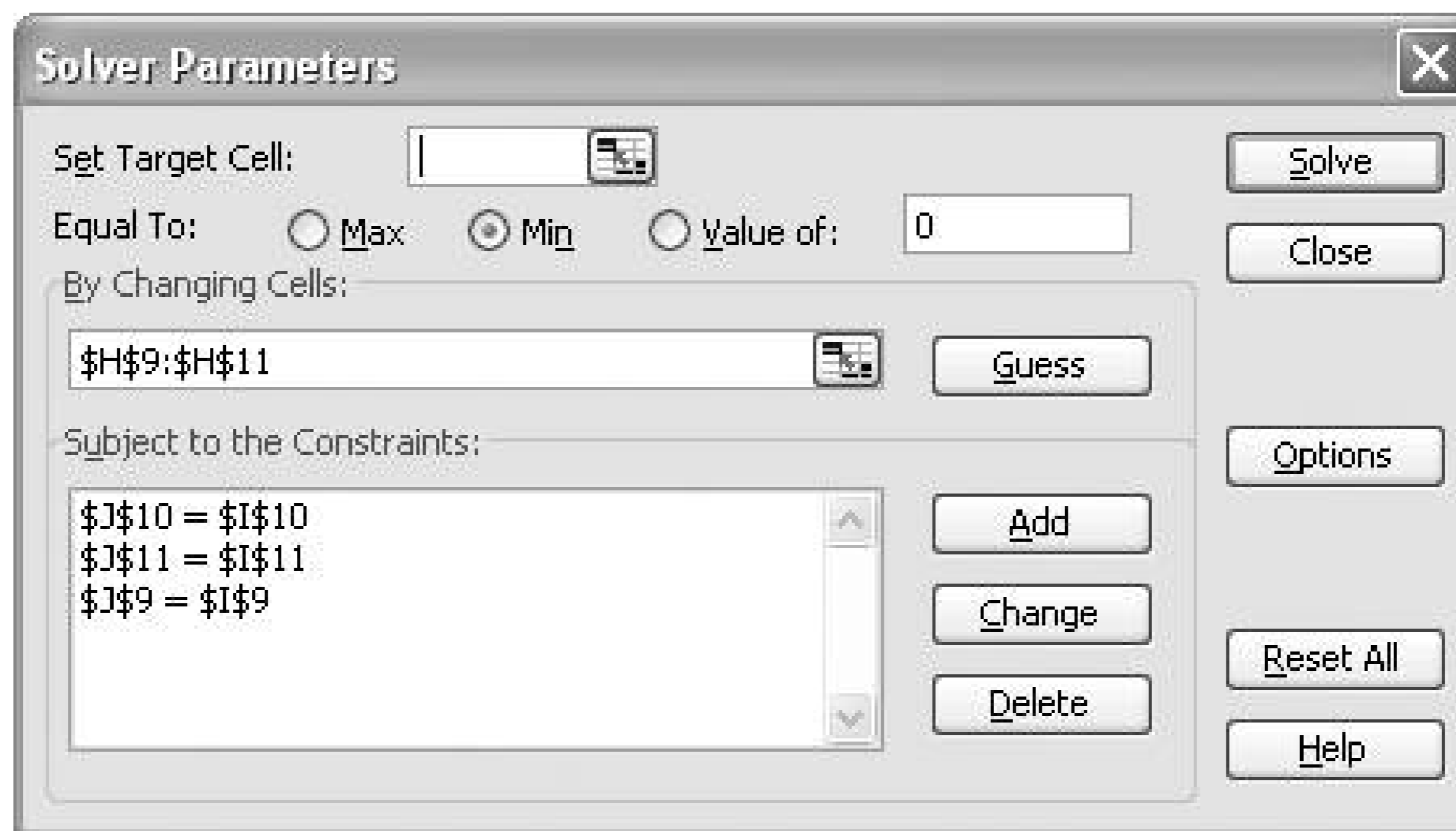
This problem essentially deals with three equations and three unknowns, the  $x$ s. The equations are of the form  $a_1x_1 + a_2x_2 + a_3x_3 = b$ . In this approach, I consider the system in this equation form rather than in the matrix form shown earlier. It's the same system, just with different notation and format.

The first thing I do in this approach is set up a table containing the given  $a$ -coefficients and  $b$ -coefficients along with initial guesses for the corresponding  $x$ s. Referring to [Figure 9-13](#), I entered the  $a$ -coefficients in cells D9 through F11, the  $b$ -coefficients in cells I9 through I11, and the  $x$ s in cells H9 through H11.

Next, I set up a column of cells (J9 through J11) containing formulas of the form  $=D9*\$H\$9+E9*\$H\$10+F9*\$H\$11$ . These formulas compute  $b$  for each equation, given the  $a$ -coefficients and the initial guesses for the  $x$ s. Ideally, the results in this column should equal the given  $b$ -values shown in the adjacent column. Since initially the  $x$ s are only guesses, the  $b$ -results will differ. However, we can use Solver to find  $x$ s that will make these two columns of  $b$ -values the same, thus resulting in the solution to this system of equations.

Open Solver by selecting Tools Solver... from the main menu bar. Instead of maximizing, minimizing, or setting a target value for some target cell, we're going to leave the target cell blank, as shown in [Figure 9-14](#).

Figure 9-14. Solver with constraints



We will set the By Changing Cells fields to the range of cells containing the  $x$ -values, cells H9 through H11 in this case. Now it's time to add some constraints.

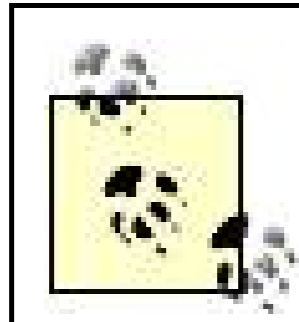
Basically, we want to let Solver vary the  $x$ s subject to the constraints that the computed  $b$ -values in cells J9 through J11 are equal to the given  $b$ -values in cells I9 through I11. So we have to add a constraint for each  $b$ -value.

[Figure 9-14](#) shows the constraints I set up in Solver. To add a constraint, press the Add button to open the Add Constraint window shown in [Figure 9-15](#).

Figure 9-15. Add Constraint window

In the Cell Reference field, select one of the cells containing a computed  $b$ -value. Then in the Constraint field, select the cell containing the corresponding given  $b$ -value. In the drop-down listbox in the center of the window, select the equals sign ( $=$ ). Press the Add button to add the constraint. Pressing Add does not close the Add Constraint window, so you can go ahead and add the other two constraints as well. Press Cancel when you're done to close the window. Upon doing so, you should

see the constraints as shown in [Figure 9-14](#).



You might be wondering why we didn't add a target cell. It turns out that Solver will add a dummy target cell if one is omitted. We can take advantage of that fact to simply add our constraints, since the actual value of the dummy cell is irrelevant to the solution.

After everything is set up, you can press Solve to actually solve the problem. Solver should quickly find a solution as in [Figure 9-14](#), where the column of  $x$ -values (H9 through H11) represents the solution to the given system of equations. Notice that these  $x$ -values are equal to those found earlier using matrix inversion.

## Using Solver to minimize residuals

An alternative approach to using constraints in Solver is to frame the problem in the form of minimizing residuals. We can easily do this by adding a column to the setup shown in [Figure 9-14](#). The new setup is shown in [Figure 9-16](#).

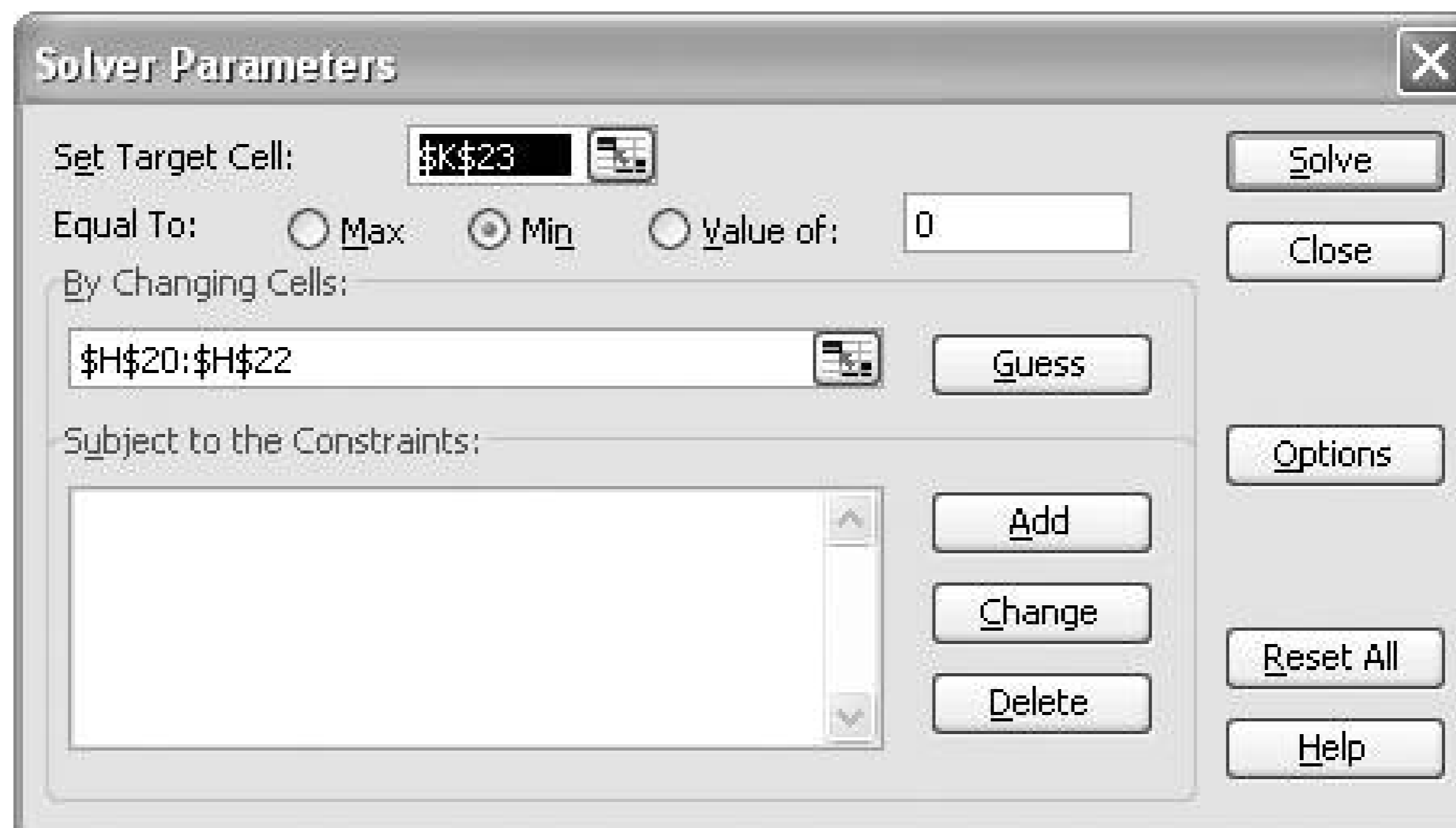
Figure 9-16. Solving a linear system with Solver by minimizing residuals

Column K contains the square of residuals between the given  $b$ -values (column I) and the computed  $b$ -values (column J). The formulas in column K are of the form  $= (I20 - J20)^2$ . Cell K23 contains the sum of these squared residuals, using the formula  $= \text{SUM}(K20 : K22)$ .

This is now essentially a least-squares problem, where we want to minimize the sum of squared residuals. We can readily use Solver to minimize this sum by changing the  $x$ -values contained in column H.

This is a straightforward application of Solver using techniques already discussed in this chapter. [Figure 9-17](#) shows the Solver model for this example.

Figure 9-17. Solver without constraints



Set the target cell to the cell containing the sum of squared residuals, cell K23. In this case you want to minimize the value in cell K23. Select the cells containing the  $x$ -values to vary in the By Changing Cells field. Notice that there are no constraints this time.

Upon pressing Solve, you should see the results already shown in [Figure 9-16](#). Here again, the resulting  $x$ -values are the same as those found using the two other approaches discussed earlier.

Which approach is better depends on your preferences and the size of your problem. For larger systems I'd probably use this third method Solver without constraints since it would be the quickest one to set up. This is also a good example of how you might find many different, equally successful, approaches to solving the same problem in Excel.

[← PREV](#)

## Recipe 9.5. Tackling Nonlinear Systems of Equations

### Problem

You have a system of nonlinear equations that you need to solve.

### Solution

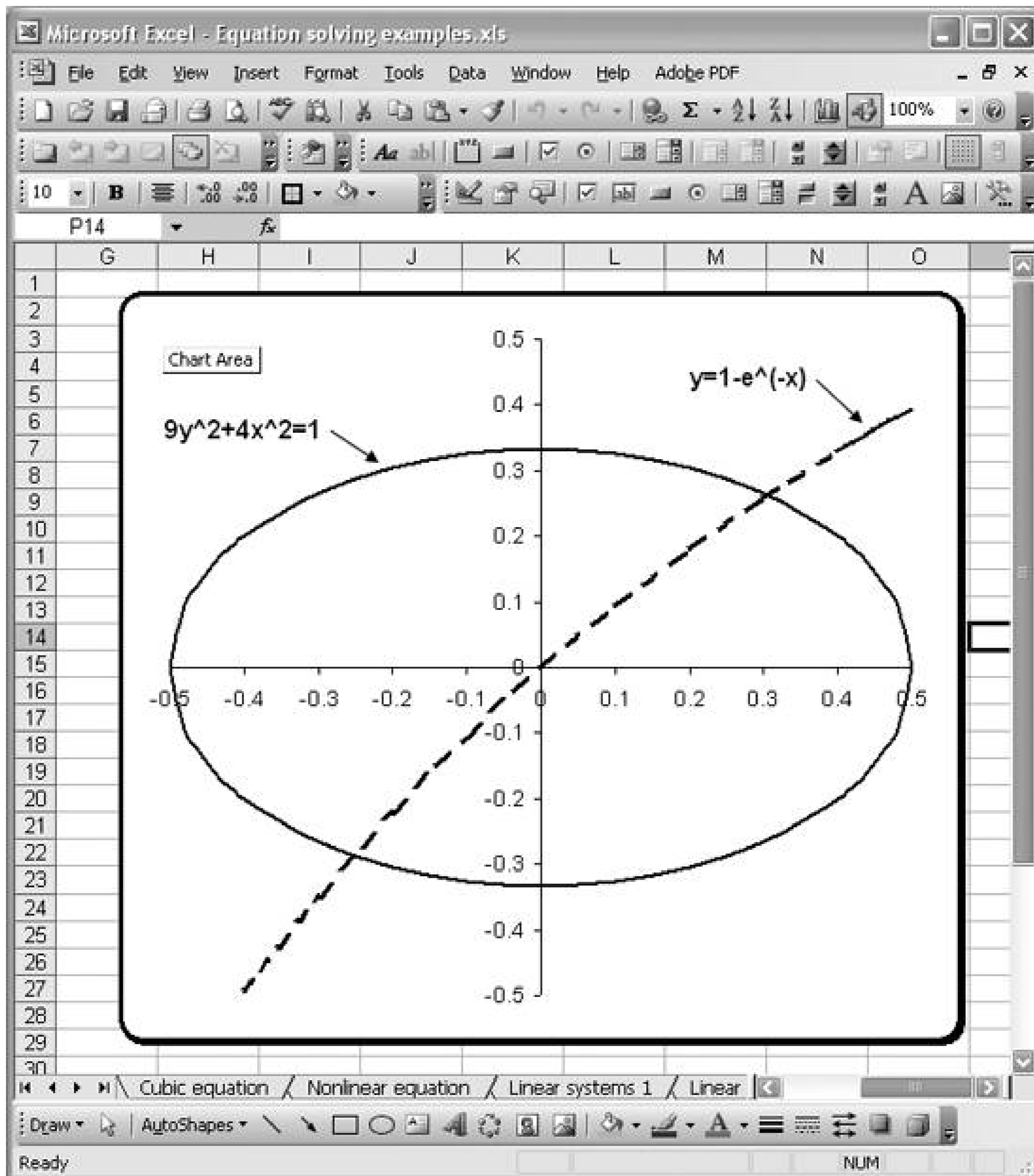
Frame the problem in terms of residuals and use Solver to minimize the residual, thus arriving at a solution.

### Discussion

You can use the same Solver-based techniques discussed in [Recipe 9.4](#) to solve systems of nonlinear equations. By way of example, consider the two equations:

Let's assume you want to find a value, or values, of  $x$  satisfying both equations simultaneously. As you can see from the plot of these two equations shown in [Figure 9-18](#), there are actually two solutions.

Figure 9-18. Nonlinear equations plot



You have some choices as to how you approach a solution to this problem. If you can readily solve these equations for  $y$  (which you can in this case), then you can equate them and rewrite them in the form  $0 = f(x) - g(x)$  and then use the techniques discussed in [Recipe 9.2](#) to find a solution.

If you have more than two equations or you can't easily solve them in terms of a common variable, then you can use the minimization of residuals method discussed in [Recipe 9.4](#), specifically in the section "[Using Solver to minimize residuals](#)," to find a solution. Using this latter approach, I found two  $x$ s that satisfy this system, namely,  $x = 0.306$  and  $x = -0.253$ . In order to get Solver to find both solutions, I had to assume two different initial guesses for  $x$ . In one case I chose an initial  $x$ -value greater than 0.5, and in the other I chose an initial  $x$ -value less than -0.5.



## Recipe 9.6. Using Classical Methods for Solving Equations

### Problem

You've seen how to leverage Excel's built-in features to solve linear and nonlinear equations and systems, but you'd still like to know how to go about implementing some classical equation-solving algorithms in Excel.

### Solution

You can use VBA to program any algorithm, just as you might in other programming languages. In the discussion to follow, I'll show you how to implement two classical methods in VBA: Newton's method and the secant method.

### Discussion

The examples in this recipe will show how you might implement Newton's method and the secant method. These are well-known methods and are treated in most advanced mathematics and numerical methods texts. Further, you can find implementations of these methods in various languages, such as Fortran and C, on the Web. It's not a great leap to implement these methods in VBA in the form of a custom function, which you can then call from an Excel spreadsheet.

For the examples in this recipe, let's reconsider the cubic polynomial equation already discussed in Recipe 9.1. The equation is repeated here for convenience:

A quick plot of this equation, as shown earlier in Figure 9-6, shows about where these three roots lie. You could easily use Solver to accurately find each of these roots. Or, you could use Newton's method or the secant method (or any of a number of other methods you could program), as discussed in the remainder of this recipe.

### Newton's method

Newton's method is based on the idea of estimating values of  $x$  that are roots of the equation by taking the intersection of a line tangent to the curve under consideration at some assumed starting  $x$ -value. Subsequent estimates are made repeatedly using this approach until the estimate  $x$ -value converges on the true root. The basic formula for predicting each new  $x$ -value as you approach a root is:

Notice that in Newton's method you must be able to compute, or estimate, the first derivative of the curve under consideration. In the example considered here, we have an equation that allows you to explicitly compute the derivative. In other cases where you can't easily compute the derivative, or when



it is impossible to do so explicitly (such as when part of your function relies on tabular data), you have to resort to numerical derivatives. Recall from the introduction to this chapter that Solver uses numerical derivatives and allows you to select either forward differencing or central differencing (refer to Recipe 10.6 for a discussion of these and other techniques).

The structure of the above equation should give you some hints as to how you might implement Newton's method. First, we need a subroutine to evaluate the function under consideration. Second, we need another subroutine to evaluate the derivative of that function. Third, we need a subroutine that will iteratively apply the above equation starting from some initial guess until it converges on a root. I've prepared just such subroutines for this example. The first one is shown in Example 9-5 and it is used to evaluate the function under consideration at some given  $x$ -value.

### Example 9-5. Function Fx

```
Public Function Fx(x As Double) As Double
    Fx = 1 + x * (3 + x * (3 * x - 7))
End Function
```

This subroutine is actually a `Function`, and therefore it returns a value to the calling subroutine. In this case, it returns the value of the cubic equation we're considering for this example. (If you were dealing with some other equation, you would change the function `Fx` accordingly.)

The next subroutine, shown in Example 9-6, is a `Function` that returns the derivative of our cubic equation for some given value of  $x$ .

### Example 9-6. Function dFx

```
Public Function dFx(x As Double) As Double
    dFx = 3 + x * (9 * x - 14)
End Function
```

The final required subroutine is shown in Example 9-7 and actually implements Newton's method, making periodic calls to `Fx` and `dFx`.

### Example 9-7. Newton's method

```

Public Function NewtonsMethod(x0 As Double, e As Double, n As Integer) As Double
    Dim i As Integer
    Dim err As Double
    Dim xn As Double
    Dim xn1 As Double

    i = 0
    err = 9999
    xn = x0

    While (err > e) And (i < n)
        xn1 = xn - Fx(xn) / dFx(xn)
        err = Abs(xn1 - xn)
        xn = xn1
    Wend
    NewtonsMethod = xn
End Function

```

Notice again that this subroutine, `NewtonsMethod`, is actually a `Function`. This means you'll be able to call it directly in a spreadsheet cell formula and it will return a result in the calling cell. I'll show you how shortly, but first I want to go through the function `NewtonsMethod`.

`NewtonsMethod` takes three parameters. The first one, `x0`, serves as your initial guess at a root. The second parameter, `e`, is a convergence tolerance, which should be a small real number (e.g., `0.001`). `e` is used to determine when the function has converged on a possible root. The third and final parameter, `n`, is an iteration limit. It is used to prevent the function from iterating indefinitely in the event it can't converge on a root.

Several variables are declared upon entering the function. These include `i`, which is just a counter variable; `err`, which is used to store the difference between successive estimates for  $x$ ; `xn`, which is used to store estimate  $x_n$ ; and `xn1`, which is used to store estimate  $x_{n+1}$ .

Next, these variables are initialized. `i` is set to `0`. `err` is set to some large number. And `xn` is set to the initial guess `x0`.

The `While` loop is where Newton's method is applied. The `While` loop will iterate until the computed error, `err`, is less than the convergence tolerance, `e`, or until the maximum number of iterations, `n`, has been reached.

A few things take place within the `While` loop. First, Newton's formula is applied, calling `Fx` and `dFx` to estimate the next  $x$ -value given the current estimate. Then the error is computed. Finally, the current  $x$ -estimate is updated before starting the next iteration.

When the `While` loop ends due either to converging on a root or reaching the iteration limit, the current estimate for  $x$  is returned by the function.

To call this function from a spreadsheet cell, you could enter something like `=NewtonsMethod(0, 0.0001, 100)` in any given cell. Excel will automatically call `NewtonsMethod` every time the spreadsheet needs updating (there's no need to use a button control, because `NewtonsMethod` is a `Function`). In this case,

the cell will contain the value -0.215, which is indeed a root to the equation under consideration. Notice the initial guess I used here of 0.

To capture the other roots, you could change your initial guess. Changing the initial guess to 0.5 yields the root at  $x = 1.0$ . Again, changing the initial guess to 1.75 yields the root at  $x = 1.549$ .

## Secant method

An alternative to Newton's method is the secant method. The secant method has the advantage that you do not need to compute the derivative of the equation under consideration. As a trade-off, you do have to supply two initial guesses so that a secant line can be constructed instead of a tangent line for making  $x$  predictions. The formula for estimating  $x_{n+1}$  using the secant method is:

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

To implement the secant method, we still need the function `Fx` as shown in Example 9-5, but we do not need the function `dFx`.

The VBA Function that implements the secant method for this example is shown in Example 9-8.

### Example 9-8. Secant method

```
Public Function SecantMethod(x0 As Double, x1 As Double, e As Double, n As Integer) As Double
    Dim i As Integer
    Dim err As Double
    Dim xn As Double
    Dim xn1 As Double
    Dim xnml As Double

    i = 0
    err = 9999
    xn = x1
    xnml = x0

    While (err > e) And (i < n)
        xn1 = xn - Fx(xn) * (xn - xnml) / (Fx(xn) - Fx(xnml))
        err = Abs(xn1 - xn)
        xnml = xn
        xn = xn1
    Wend
    SecantMethod = xn
End Function
```

This function is substantially similar to the function `NewtonsMethod`. There are a few differences that I'll highlight. First, notice the additional parameter, `x1`, which represents the second initial guess for  $x$ . Remember that for the secant method we need two initial guesses for  $x$ .

The remaining differences are found within the `while` loop. As you can see, the expression for predicting `xn1` has been replaced with the secant formula. And each iteration requires updating both  $x_n$  and  $x_{n-1}$ , which are stored in the variables `xn` and `xnm1`, respectively.

You can use `SecantMethod` in a cell formula just as you did `NewtonsMethod`. For example, I entered the following three formulas in three different cells in a spreadsheet:

```
=secantmethod(0, -0.1, 0.0001, 100)  
=secantmethod(0.5, 0.6, 0.0001, 100)  
=secantmethod(1.75, 1.7, 0.0001, 100)
```

These three formulas returned the values -0.215, 1.000, and 1.549, respectively. As you see, these roots are identical to those obtained using Newton's method, and in both cases the results agree well with what you can glean by examining the plot of the example cubic equation in Figure 9-6.

[← PREV](#)

[← PREV](#)

# Chapter 10. Numerical Integration and Differentiation

[Section 10.0. Introduction](#)

[Recipe 10.1. Integrating a Definite Integral](#)

[Recipe 10.2. Implementing the Trapezoidal Rule in VBA](#)

[Recipe 10.3. Computing the Center of an Area Using Numerical Integration](#)

[Recipe 10.4. Calculating the Second Moment of an Area](#)

[Recipe 10.5. Dealing with Double Integrals](#)

[Recipe 10.6. Numerical Differentiation](#)

## 10.0. Introduction

Numerical integration is the process of approximating integrals in lieu of evaluating them analytically. This subject is very rich in terms of the number of different techniques, history, applications, and even debate as to which approaches are better for given applications. Many of the methods commonly used are derived from *Newton-Cotes* integration formulas. Most modern books on numerical methods include some chapter(s) devoted to these methods. Some well-known Newton-Cotes integration formulas are the *trapezoidal rule*, *Simpson's first and second rules*, and *Bode's rule*.

There are other methods that are typically referred to as *Gaussian methods* or *Gaussian quadratures*. This class of methods includes such formulas as the *Gauss-Legendre formula*, the *Gauss-Chebyshev formula*, and the *Gauss-Hermite formula*.

Of course, there are other methods that combine elements of these classes, expand upon them, or take different approaches altogether, for example, *Monte Carlo integration*.

In this chapter I'll show you how to implement some of the more popular numerical integration rules and formulas to solve specific problems. My aim is to show you how to implement such rules and formulas in Excel, not to show how to implement every rule, formula, or variation available. In general, most of these rules and formulas follow similar patterns in the way they are constructed, with many of them differing only by coefficients and factors. Therefore, the techniques I discuss here on how to implement some common methods in Excel can be easily extended to deal with other rules and formulas.

Additionally, I've included a recipe here on numerical differentiation. As differentiation is the inverse of integration, I felt it appropriate to address numerical differentiation here as well, and to highlight some of the particular challenges associated with numerical differentiation. In other chapters of this book that deal with solving differential equations, I'll address some other numerical methods that involve differentiation.

## Recipe 10.1. Integrating a Definite Integral

### Problem

You have an analytic function that you need to integrate numerically.

### Solution

Use the *trapezoidal rule* of numerical integration.

### Discussion

As discussed in the introduction to this chapter, there exists a multitude of numerical integration formulas and rules to choose from. Your choice depends on several factors including, but not limited to, desired accuracy and ease of implementation. For the purposes of this example, I want to use one of the easiest to implement numerical integration rules, namely the trapezoidal rule. This rule exemplifies the main steps in setting up numerical integration in a spreadsheet without clouding the issue with more complicated mathematics.

In this example, I'll consider the integration of a given analytic function of the form:

where the analytic form of  $f(x)$  is known. Even though you know  $f(x)$ , you may not be able to integrate it analytically, and this is one reason why you would resort to numerical integration.

The trapezoidal rule is well documented in virtually every calculus, engineering analysis, or numerical methods book I've ever read. Basically, this technique approximates the function under consideration with a sequence of linear curves. The general formula (also called *extended* or *composite* formula) for the trapezoidal rule is:

Here,  $s$  is the distance along the  $x$ -axis between samples. This formula assumes that all of the  $y$ -values are computed, or correspond, to equally spaced  $x$ -values. Since we have an analytic form of  $f(x)$ , we can easily choose the spacing and thus the number of  $x$ -values between the bounds of integration  $a$  and  $b$ . The smaller the spacing and the greater the number of samples, the higher the accuracy.

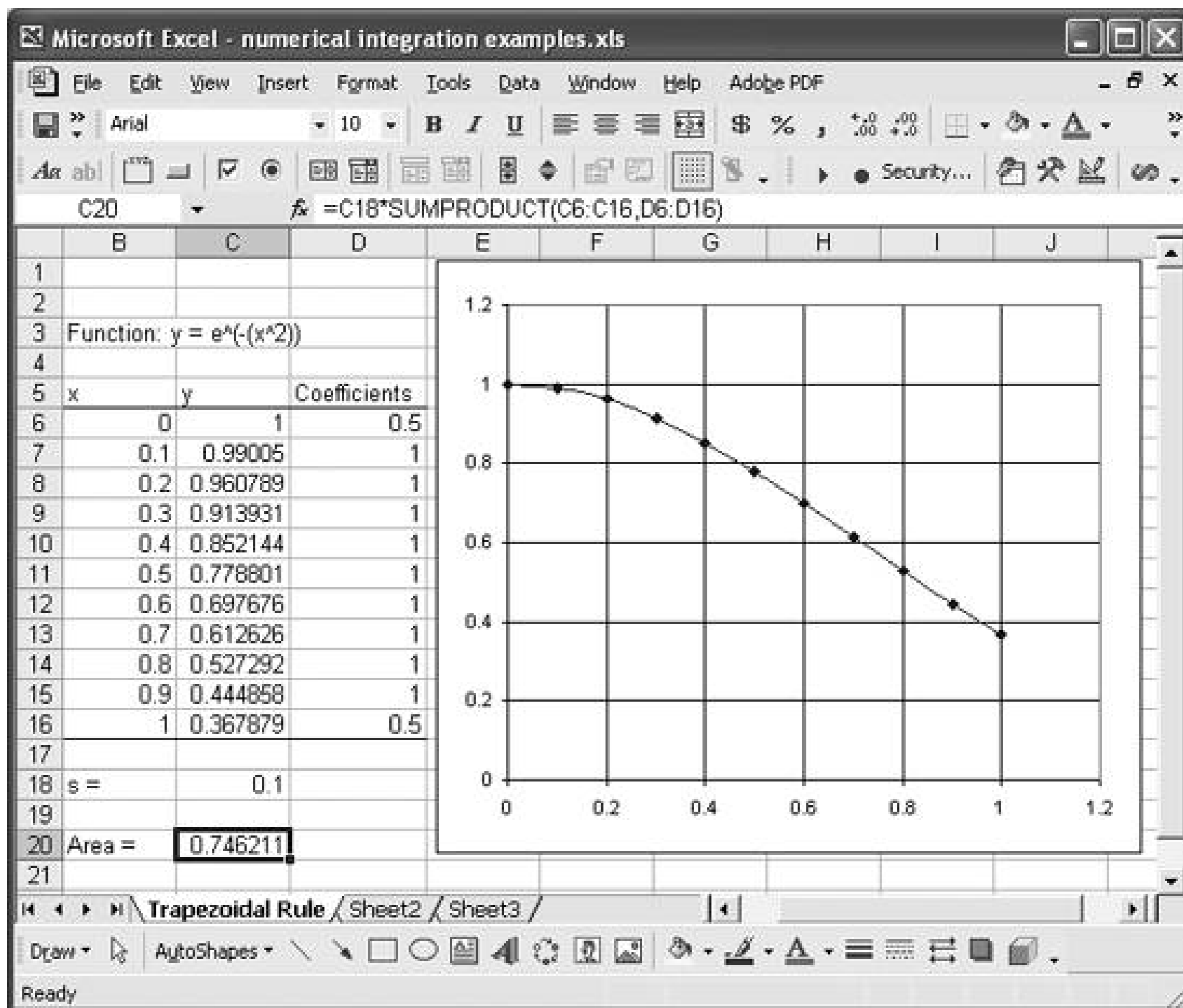
Excel is very well suited to integration formulas like the trapezoidal rule. In fact, many popular integration rules are of similar form, in that they consist of the sum of products between coefficients and  $y$ -values. This sort of computation is easily set up in Excel.

Figure 10-1 shows a spreadsheet that uses the trapezoidal rule to compute the integral of the function:

$$y = e^{-x^2}$$

for  $x$  in the interval  $[0, 1]$ .

Figure 10-1. Trapezoidal rule example



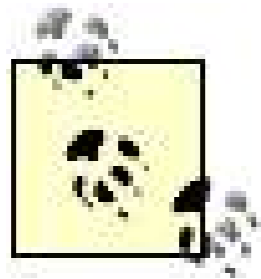
The first step in performing this integration is to set up a table containing the  $x$ -values and corresponding  $y$ -values. I selected a spacing of 0.1 between  $x$ -values and put them in column B. The  $y$ -values are computed in column C using formulas like  $=\text{EXP}(-(\text{B6}^2))$ .

The next step is to set up a column containing the coefficients that appear in the integration formula. This is almost trivial in the case of the trapezoidal rule, since all but the first and last coefficients are



1, but it's a very handy way to organize the coefficients for other rules that don't involve a bunch of ones.

I put the  $x$ -spacing value ( $s = 0.1$ ) in cell C18, just for clarity. Cell C20 contains the actual trapezoidal rule formula. This cell is selected in the screenshot so you can see the formula in the formula bar toward the top of the spreadsheet. The formula is `=C18*SUMPRODUCT(C6:C16,D6:D16)` and it uses the `SUMPRODUCT` worksheet function to compute the sum of the products of the  $y$ -values and the coefficients. The result in this case is 0.746 and represents the area under the curve from  $x = 0$  to  $x = 1$ .



In many cases, you won't necessarily have an analytic function with which to work. You may be dealing with experimental data or functions represented by tabular values. You can apply the technique discussed in this recipe in just the same way. Instead of evaluating the given function to populate a  $y$ -column as shown in [Figure 10-1](#), you would simply use your sampled or tabulated  $y$ -data. The rest of the calculation is the same as discussed here. See [Recipes 10.3](#) and 10.4 for some examples using tabulated data.

The trapezoidal rule shown here is very easy to implement, which makes it very attractive. The theoretical error of the trapezoidal rule is proportional to spacing between samples to the third power. If you increase the number of samples so as to decrease the spacing by a factor of 2, then the error will decrease by a factor of 8. Looking at this from another angle, for the example discussed here, reducing the spacing by a factor of 2 from 0.1 to 0.05 only affects the result beyond the fourth digit after the decimal place. The result goes from 0.746211 to 0.746671.

Increasing the number of samples (thus decreasing the spacing) is one way to improve accuracy when using the trapezoidal rule. Some practitioners take an iterative approach and stop increasing the number of samples when changes in the result from successive iterations fall below some tolerance (see [Recipe 10.2](#)).

There are other ways to improve the accuracy of numerical integration. An obvious way is to use a rule or method that is inherently more accurate. One very well-known rule is called *Gaussian quadrature* or *Gaussian integration*.

There are many other, inherently more accurate rules, many of which are easier to implement than Gaussian quadrature and which are better suited for tabulated data in lieu of having a known analytic function. [Recipes 10.3](#) and [10.4](#) discuss examples using *Simpson's rule* for tabulated data, which is just as easy to implement as the trapezoidal rule, and more accurate.

For now, let's consider the same problem discussed earlier but this time solved using Gaussian integration. The idea behind Gaussian integration is that an integral of a function over a standard interval can be approximated by the weighted sum of functional values over that interval. In equation form this looks like:

There are a couple of tricks to applying this method. First, for integrals over nonstandard intervals, you have to transform (usually using a linear transformation) the actual interval to the standard interval, using a change of variables. Also, you need to derive the proper coefficients by which to multiply the functional values over the interval.

Fortunately, there are several different standard Gaussian integration formulas that have already been derived and are readily available from standard texts on the subject of numerical methods. Once such standard formula is the *fifth-order formula*:

$$\int_{-1}^1 f(t) dt = \frac{5}{9} f\left(-\sqrt{\frac{3}{5}}\right) + \frac{8}{9} f(0) + \frac{5}{9} f\left(\sqrt{\frac{3}{5}}\right)$$

To apply this formula to the example function discussed in this recipe, the first thing you need to do is transform from variable  $x$  over the interval from 0 to 1 to a variable  $t$  over the interval from -1 to 1. To do this, let  $t = 2x - 1$ . Then  $x = (t + 1)/2$ , which gives  $dx = (1/2) dt$ . Now we have:

$$f(t) = e^{-\frac{1}{4}(t+1)^2}$$

And the integral to be evaluated is now:

$$\frac{1}{2} \int_{-1}^1 f(t) dt$$

The 1/2 factor comes from the change of variables  $dx = (1/2) dt$ . Applying the Gaussian formula in a spreadsheet cell formula looks like this:

$$=0.5 * (5/9 * f\_of\_t(-SQRT(3/5)) + 8/9 * f\_of\_t(0) + 5/9 * f\_of\_t(SQRT(3/5)))$$

This formula uses a custom VBA function named `f_of_t`, which simply evaluates the function  $f(t)$  shown earlier. The VBA code for this function is shown in [Example 10-1](#).

### Example 10-1. VBA code for `f_of_t`

```
Public Function f_of_t(t As Double) As Double
    f_of_t = Exp(-(1 / 4#) * (t + 1) ^ 2)
End Function
```

You don't have to use a custom VBA function as I did here. I did so only to make the spreadsheet formula more readable.

The end result for this example is 0.746814. Comparing this result to that obtained using the trapezoidal rule shows that the results are identical out to the third decimal place. To make the result from the trapezoidal rule equal that obtained from the Gaussian formula out to the fourth decimal place, you'd have to increase the number of samples to 53. To make the result identical to the fifth decimal place, you'd have to increase the number of samples to 67. (I obtained these values iteratively.)

There are two ways to view these results. On one hand, it's clear that the required number of samples when using the trapezoidal rule is higher for the same level of accuracy. On the other hand,

in this example at least, the higher number of samples results in an insignificant increase in computational time, whereas applying the Gaussian formula is certainly more effort.

I have to be honest here. I've never applied the Gaussian formula to solve any real-world problem. I've always found it too easy to simply increase the number of samples when using the trapezoidal rule, or more often Simpson's rule, and live with the few extra milliseconds it takes to compute the result, given the power of modern desktop computers. This works for me, but it's not to say the same will work for the type of problems that require you to implement numerical integration.

## See Also

If you're dealing with a known analytic function, then you might consider using VBA to carry out the integration, as discussed in [Recipe 10.2](#).





## Recipe 10.2. Implementing the Trapezoidal Rule in VBA

### Problem

You see how easy it is to apply the trapezoidal rule in a spreadsheet, but you'd like to implement the rule in VBA to save having to set up the columns of data as was done in the previous recipe. You'd also like to be able to quickly change the spacing between  $x$ -values to improve accuracy.

### Solution

Add two custom VBA functions as discussed here. See Recipe 2.2 to learn the basics of writing VBA functions if you don't know them already.

### Discussion

Let's reconsider the example from Recipe 10.1. This time, instead of setting up the table of data shown in Figure 10-1 and using the `SUMPRODUCT` function in cell C20, you need only write two simple VBA functions.

The first function we need is simply a function to compute a  $y$ -value for a given  $x$ -value using the known analytic function under consideration. Example 10-2 shows just such a function for computing the analytic function from the previous example.

#### Example 10-2. VBA function for computing $y = f(x)$

```
Public Function F(x As Double) As Double
    F = Exp(-(x ^ 2))
End Function
```

As you can see, it's a fairly simple function. It takes a given  $x$ -value as an argument and returns the corresponding  $y$ -value using the given analytic function. The other required VBA function is shown in Example 10-3.

#### Example 10-3. VBA function for trapezoidal rule

```

Public Function Trapezoidal(xMin As Double, xMax As Double, n As Integer) As Double
    Dim dx As Double
    Dim sum As Double
    Dim y As Double

    dx = (xMax - xMin) / n

    sum = (F(xMin) + F(xMax)) / 2#

    For i = 1 To (n - 1)
        y = F(xMin + (dx * i))
        sum = sum + y
    Next i

    sum = sum * dx

    Trapezoidal = sum
End Function

```

This function takes three arguments: the lower and upper bounds of the interval of integration, `xMin` and `xMax`, and the number of intervals over which to divide the range of  $x$ -values, `n`.

The first task upon entering this function is to declare a few local variables. `dx` will store the computed spacing of the  $x$ -values given the number of intervals, `n`. The computed  $y$ -value is stored in the variable `y`. `sum` is used to accumulate the products of the coefficients and the `y` values.

After declaring these local variables, the function goes on to compute `dx`, which is just the  $x$ -range divided by the number of intervals. `sum` is also initialized to the average of the results of the function evaluated at the upper and lower bounds of integration. This step takes care of the first and last terms in the trapezoidal rule formula involving the 1/2 coefficient. Next, the function enters a `For` loop on the index variable `i` from 1 to `n-1` to accumulate the function evaluated at each computed  $x$ -point. This step takes care of all the middle sums in the trapezoidal rule formula.

The first calculation within the `For` loop evaluates `y` for a given `x` value by calling the function `F` as defined in Example 10-2. Here, `x` is computed by multiplying `dx` and `i`. Upon exiting the loop, `sum` is multiplied by `dx` to yield the final result.

To actually use these functions in a spreadsheet you need only type the formula `=trapezoidal(0,1,10)` into any given cell. Here, I used the same range and number of samples as used in the example from Recipe 10.1. As expected, the result in this case is exactly the same as before, that is, 0.746.

Using the VBA approach discussed here allows you to easily change the number of samples used in the computation to improve the accuracy of the result. Increasing the argument `n` results in more samples at smaller  $x$ -spacing, resulting in greater accuracy. You can also change the upper and lower bounds of integration very easily using this approach. You can achieve both of these changes by taking the spreadsheet approach, but it requires you to reconstruct your columns of data.

You can integrate other analytic functions using the VBA functions shown here. All you need to do is change the `F` function to use your new analytic expression; the function `trapezoidal` remains unaffected.

I mentioned in Recipe 10.1 that some practitioners apply an iterative approach using the trapezoidal rule to adjust the number of sample intervals until the end result stops changing by an amount greater than some given tolerance. You can easily implement such a scheme using the VBA functions shown in this recipe. As an illustration, I added another custom VBA function as shown in Example 10-4.

#### Example 10-4. IterateTrapezoidal function

```
Public Function IterateTrapezoidal(xMin As Double, xMax As Double, n As Integer, tol As Double) As Double
    Dim value As Double
    Dim previousValue As Double
    Dim difference As Double
    Dim counter As Integer
    Dim nNew As Integer

    previousValue = Trapezoidal(xMin, xMax, n)
    difference = 99999
    counter = 0
    nNew = n

    While ((counter < 100) And (difference > tol))
        nNew = nNew * 2
        previousValue = value
        value = Trapezoidal(xMin, xMax, nNew)
        difference = Abs(value - previousValue)
        counter = counter + 1
    Wend

    IterateTrapezoidal = value
End Function
```

This function, called `IterateTrapezoidal`, takes four arguments. The first three are the same as discussed earlier for the function `trapezoidal`. The fourth argument is new and represents the tolerance, `tol`, which tells the iterative algorithm when to stop.

The scheme goes like this: first, we compute a result using `Trapezoidal` just as we did earlier, using the supplied default number of sample intervals, `n`. Then we double `n`, recompute the result, and compare it with the previous result to see how much the result changed given the change in `n`. If the change is greater than `tol`, then we double `n` again and recompute. This repeats until the difference is less than `tol`, at which time the loop is exited and the last computed result is returned. This algorithm works pretty well and is one way to automate the process of choosing an appropriate `n` when applying the trapezoidal rule.

You do have to be careful with this approach. You'll notice that I put a counter in the `While` loop shown in Example 10-4. You could encounter a situation where the specified tolerance is too low and the solution never converges. The counter will eventually kick execution out of the loop, to prevent the function from getting stuck in an infinite loop. You do also have to keep in mind round-off error and precision. Basically don't specify an unrealistically small tolerance, and keep in mind the magnitude of the result you're

expecting. If you're expecting a result on the order of  $10,000 \pm 100$ , a tolerance of  $1e-5$  is probably way too small!

## See Also

Refer back to Recipe 10.1 for more discussions on the trapezoidal rule.





## Recipe 10.3. Computing the Center of an Area Using Numerical Integration

### Problem

You need to integrate a function represented by tabular values to determine the area enclosed by the function and the center of that area.

### Solution

Apply a numerical integration technique in your spreadsheet (as discussed in earlier recipes) to compute the area, and take the *first moment* of the area to compute the centroid.

### Discussion

It often happens in science and engineering that you need to compute the area under a curve or the area of some nonprimitive geometric shape. Often the center of that area is of some significance. When working with probability distributions, the area and moments of the area yield important information. In structural mechanics, areas and moments of areas are important for computing stress. Naval architects compute areas and moments of areas to assess how ships float. These are just a few examples of how areas and moments of areas are commonly found in science and engineering.

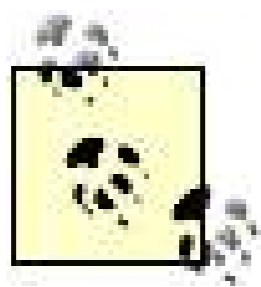
In this recipe, we'll consider the first moment of an area, which is computed according to the formula:

This formula yields the first moment of the area about the  $y$ -axis. The center of area is computed according to the formula:

In this formula,  $M_y$  is the first moment of the area and  $A$  is the area.

Now, let's consider an example. This time, instead of using the trapezoidal rule, I'll use another common integration rule known as *Simpson's rule*.





There's another Simpson's rule known as *Simpson's second rule* or the 3/8ths rule.

The general formula for Simpson's rule is:

$$A = \frac{s}{3}(1y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + 4y_5 + \dots + 2y_{n-2} + 4y_{n-1} + 1y_n)$$

This formula is a little different from the formula for the trapezoidal rule. Notice the coefficients in this formula; they follow the pattern 1, 4, 2, 4, 2, 4, 1. The first and last coefficients are always 1, while the middle terms alternate between 4 and 2, with the second and second to last coefficients always 4. Generally, Simpson's rule yields more accurate results than the trapezoidal rule for the same number of samples because Simpson's rule approximates the function, or curve, being integrated using a second-order polynomial as opposed to a linear approximation as is used in the trapezoidal rule. Application of Simpson's rule requires an odd number of samples,  $n$ , in the general formula, which makes  $n-1$  an even number. Further, just as in the trapezoidal rule, the spacing,  $s$ , between  $x$ -values must be uniform.

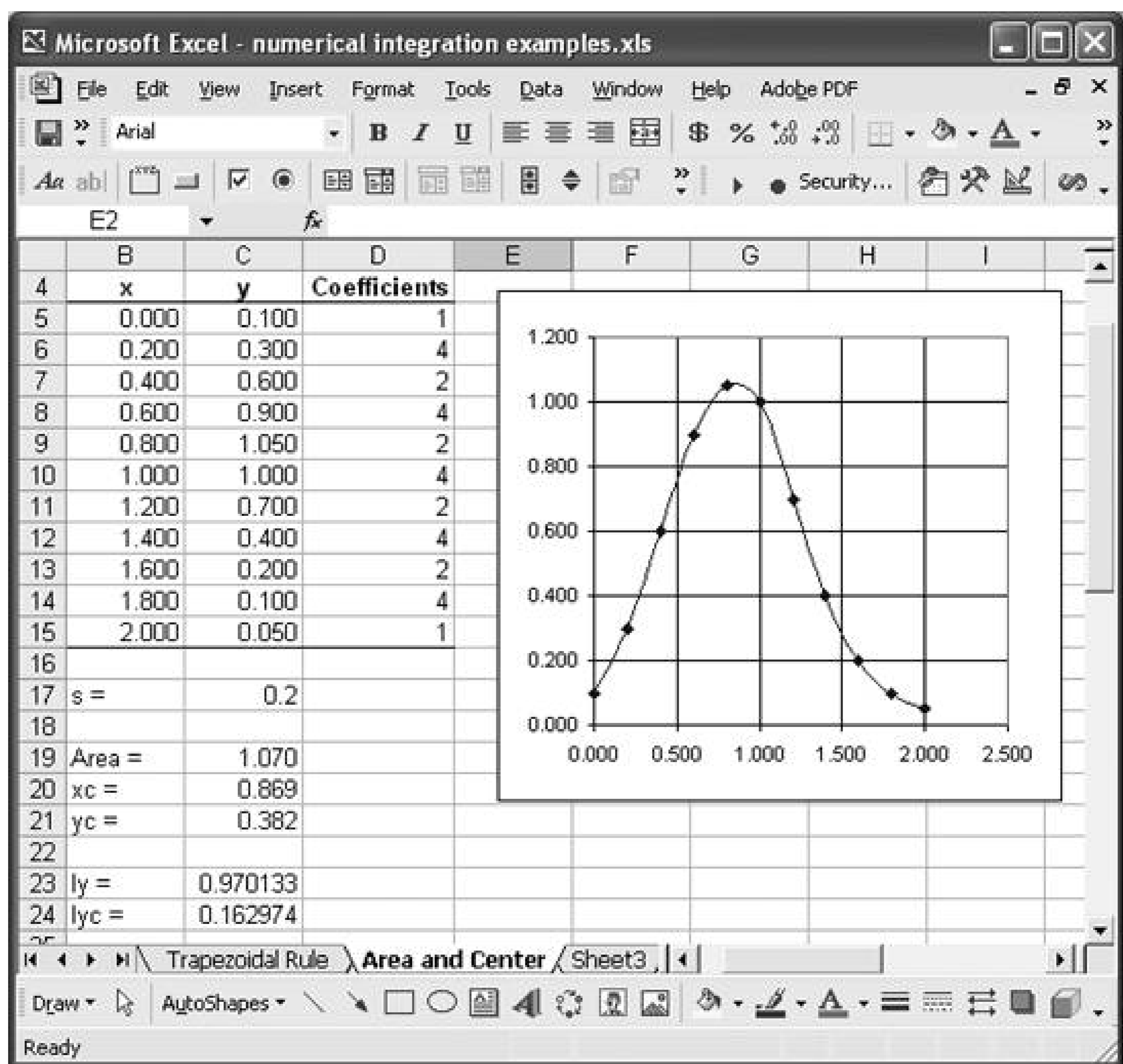
In this example, we're not dealing with a known analytic function. Instead we have tabular  $y$ -values corresponding to a set of  $x$ -values. In the real world, these could be measured coordinates from some geometry, or experimentally obtain data, for example.

[Figure 10-2](#) shows the data for this example.  $x$ - and  $y$ -values are shown in columns B and C, and the coefficients are shown in column D. The little chart on the right shows the shape of the curve represented by this data.

Calculating the area under this curve proceeds in a manner similar to that discussed earlier in [Recipe 10.1](#). Area is computed in cell C19 using the formula `=C17/3*SUMPRODUCT(C5:C15,D5:D15)`. Here again, I use the `SUMPRODUCT` formula to simultaneously sum the products of the  $x$ - and  $y$ -values.

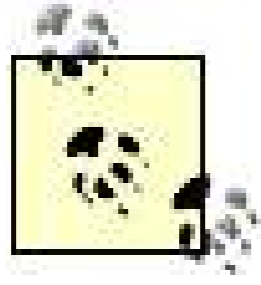
The first step in computing the center of the area is to compute the first moment of the area about each axis. This allows you to compute both the  $x$ - and  $y$ -coordinates of the center of area. To get the coordinates, you have to divide the first moment of the area by the area.

Figure 10-2. Area and area moments example



The  $x$ -coordinate of the center of area is computed in cell C20 using the formula  $= (C17/3 * \text{SUMPRODUCT}(B5:B15, C5:C15, D5:D15)) / C19$ . The numerator in this formula computes the first moment of the area about the  $y$ -axis using Simpson's rule. This is achieved by again using the **SUMPRODUCT** formula to compute the sum of the products of  $x$ ,  $y$ , and the coefficients. Dividing by the area, contained in cell C19, yields the  $x$ -coordinate to the center of area relative to the origin.

The  $y$ -coordinate of the center of area is computed in a similar manner in cell C21 using the formula  $= (C17/3 * \text{SUMPRODUCT}(C5:C15, C5:C15/2, D5:D15)) / C19$ . This formula is generally similar to the one used to compute the  $x$ -coordinate; however, this time you don't use the  $x$ -value at all. To compute the first moment about the  $x$ -axis, you multiply the  $y$ -value by the distance from the  $x$ -axis to the center of the ordinate, which is just half of  $y$ . Take a look at the third parameter in the **SUMPRODUCT** term; it includes a cell range divided by 2. Excel interprets this as an instruction to divide each value in the cell range **C5:C15** by 2 before multiplying it by the other values from the other ranges in the **SUMPRODUCT** function. This is convenient, as it saves having to set up another column of data to compute  $y/2$ .



You can use other mathematical operators and functions in a similar manner. Clever use of this sort of syntax allows you to construct concise formulas.

Dividing the first moment of the area about the  $x$ -axis by the area yields the  $y$ -coordinate to the center of area. The result is shown in cell C21 in [Figure 10-2](#).

## See Also

You can easily extend the example discussed here to compute the second moment of the area. See [Recipe 10.4](#) for more information.





## Recipe 10.4. Calculating the Second Moment of an Area

### Problem

You've learned how to compute area and center of area using numerical integration as discussed in the previous recipe and now you want to learn how to compute the second moment of the area.

### Solution

Use the same technique as discussed in the previous recipe for computing the first moment of the area, but this time use the product of  $x^2$  and  $y$  instead of  $x$  and  $y$ .

### Discussion

You can easily extend the example shown in [Figure 10-2](#) from the previous recipe to include computing the second moment of the area. In mechanics, the second moment of an area is called the *moment of inertia* and is computed according to the following formula:

This yields the moment of inertia about the  $y$ -axis. The formula for computing the second moment about the  $x$ -axis is very similar; the  $x$ s and  $y$ s are just swapped.

Often, you're more interested in the moment of inertia about an axis that passes through the center of an area. In that case you need to apply the well-known *parallel axis theorem*.

Here,  $I_{na}$  is the moment of inertia of the area about an axis parallel to the  $y$ -axis, but passing through the centroid of the area.  $A$  is the area, and  $d$  is the distance from the  $y$ -axis to the axis passing through the centroid.

[Figure 10-2](#) includes the results of these computations for the area considered earlier. Cell C23 computes the moment of inertia about the  $y$ -axis using the formula `=C17/3*SUMPRODUCT(C5:C15,B5:B15^2,D5:D15)`. This formula looks very similar to that shown earlier for computing the first moment of the area. However, in this case we square the  $x$ -values using the argument `B5:B15^2`.

Application of the parallel axis theorem yields the moment of inertia about an axis that passes through the center of the area. Cell C24 computes this value using the formula `=C23-C19*C20^2`. In this case, the  $d$  term from the parallel axis theorem is the distance along the  $x$ -axis to the center of area, which was already computed in the example from the previous recipe.

## See also

Read [Recipe 10.3](#) for more information on applying Simpson's rule to calculate areas and centers of areas.



## Recipe 10.5. Dealing with Double Integrals

### Problem

You need to numerically integrate a double integral (for example, to compute the volume of some arbitrary shape or the volume under a surface).

### Solution

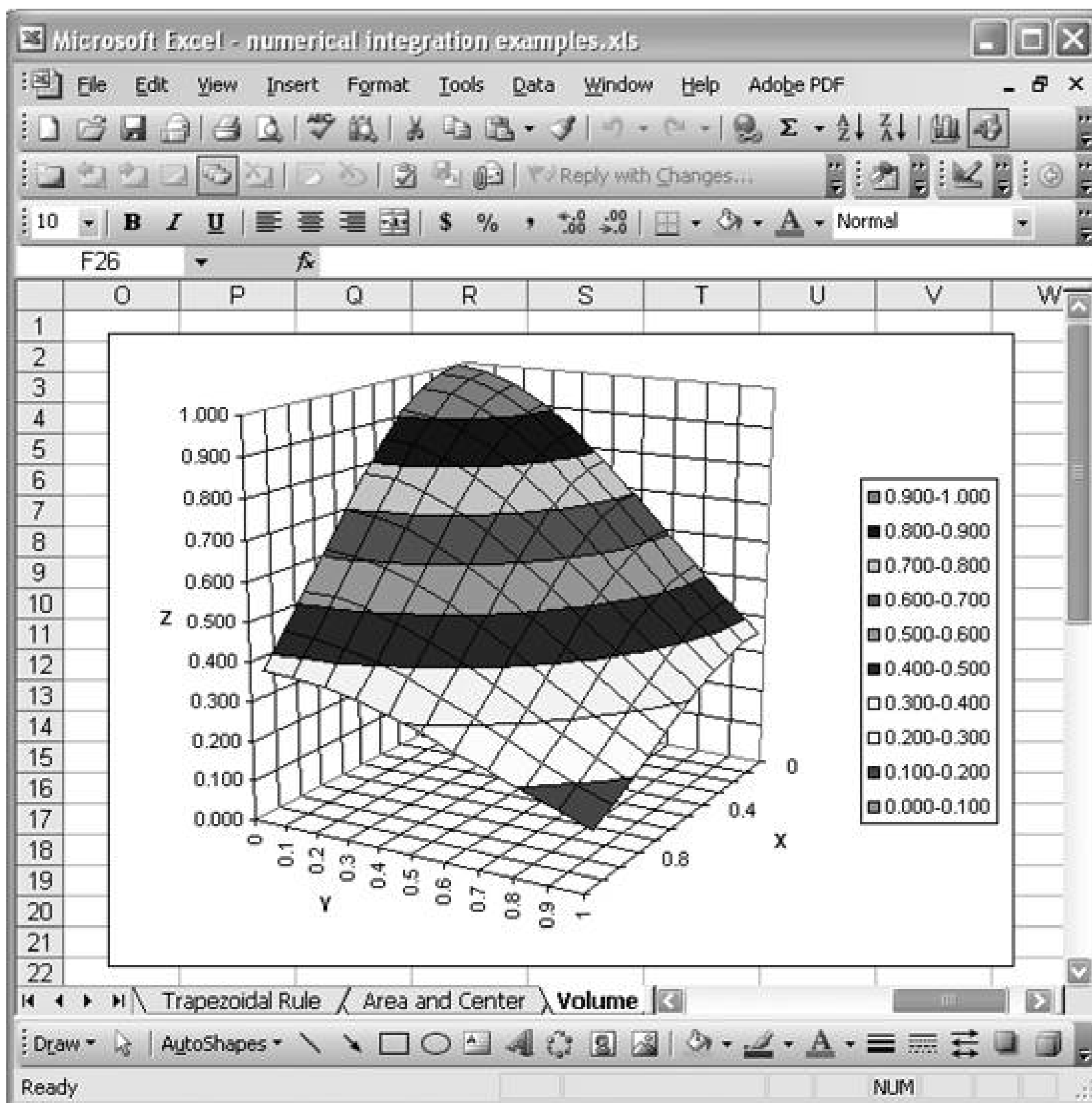
Separate the problem into two parts, each involving only single integrals, and then apply the numerical integration techniques discussed earlier in this chapter.

### Discussion

Computing multiple integrals numerically can be challenging, especially when triple or higher integral and involved. To the best of my knowledge, there are no general-purpose numerical integration formulas that handle multiple integrals. There is a method called *Monte Carlo* integration that involves taking a lot of random samples within the volume or domain being integrated and using the distribution of those samples to estimate the volume or whatever is of interest in higher dimensions. Alternatively, you can break up a multiple integral into successive single integrals and apply the standard numerical integration techniques discussed in this chapter.

To demonstrate this latter approach, consider the surface shown in [Figure 10-3](#).

Figure 10-3. 3D surface



Let's assume you want to compute the volume under that surface bounded by  $y = [0,1]$  and  $x = [0,1]$ . This is a three-dimensional problem, but instead of computing a multiple integral, you can compute a series of single integrals.

The first step is to consider cross-sections of this surface at successive  $y$ -values. Each cross-section, at each  $y$ -value, will yield a curve. We can compute the area under each of these curves using a standard integration rule. You'll end up with a new set of data that represents cross-sectional areas as a function of  $y$ . You can then integrate the resulting curve of areas to get the volume under the surface.

[Figure 10-4](#) shows the data that represents the surface in [Figure 10-3](#).

The table contains  $z$ -values for corresponding  $x$ - and  $y$ -values. As outlined a moment ago, the first task is to compute cross-sectional areas corresponding to cross-sections

Figure 10-4. Data for 3D surface

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	
1			sx =	0.1											
2			sy =	0.1											
3															
4	Coefficients		x		0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
5	0.5		0		1.000	0.990	0.961	0.914	0.852	0.779	0.698	0.613	0.527	0.445	0.368
6	1		0.1		0.990	0.980	0.951	0.905	0.844	0.771	0.691	0.607	0.522	0.440	0.364
7	1		0.2		0.961	0.951	0.923	0.878	0.819	0.748	0.670	0.589	0.507	0.427	0.353
8	1		0.3		0.914	0.905	0.878	0.835	0.779	0.712	0.638	0.560	0.482	0.407	0.336
9	1		0.4		0.852	0.844	0.819	0.779	0.726	0.664	0.595	0.522	0.449	0.379	0.313
10	1		0.5		0.779	0.771	0.748	0.712	0.664	0.607	0.543	0.477	0.411	0.346	0.287
11	1		0.6		0.698	0.691	0.670	0.638	0.595	0.543	0.487	0.427	0.368	0.310	0.257
12	1		0.7		0.613	0.607	0.589	0.560	0.522	0.477	0.427	0.375	0.323	0.273	0.225
13	1		0.8		0.527	0.522	0.507	0.482	0.449	0.411	0.368	0.323	0.278	0.235	0.194
14	1		0.9		0.445	0.440	0.427	0.407	0.379	0.346	0.310	0.273	0.235	0.198	0.164
15	0.5		1		0.368	0.364	0.353	0.336	0.313	0.287	0.257	0.225	0.194	0.164	0.135
16			Areas:		0.746	0.739	0.717	0.682	0.636	0.581	0.521	0.457	0.393	0.332	0.275
17			Coefficients:		0.5	1	1	1	1	1	1	1	1	1	0.5
18															
19			Volume =		0.557										

taken of the surface at each  $y$ -value. This means that for each  $y$ -value, held constant, we can integrate along the  $x$ -axis to compute the area under the cross-section curve.

The row labeled *Areas* on row 16 of the spreadsheet shows the results of integrating the cross-section curves for each  $y$ -value. I applied the trapezoidal rule here, using formulas like `=D$1*SUMPRODUCT($A$5:$A$15,D5:D15)`. Cells D16 through N16 contain formulas just like this one to compute each cross-sectional area, that is, at each  $y$ -value.

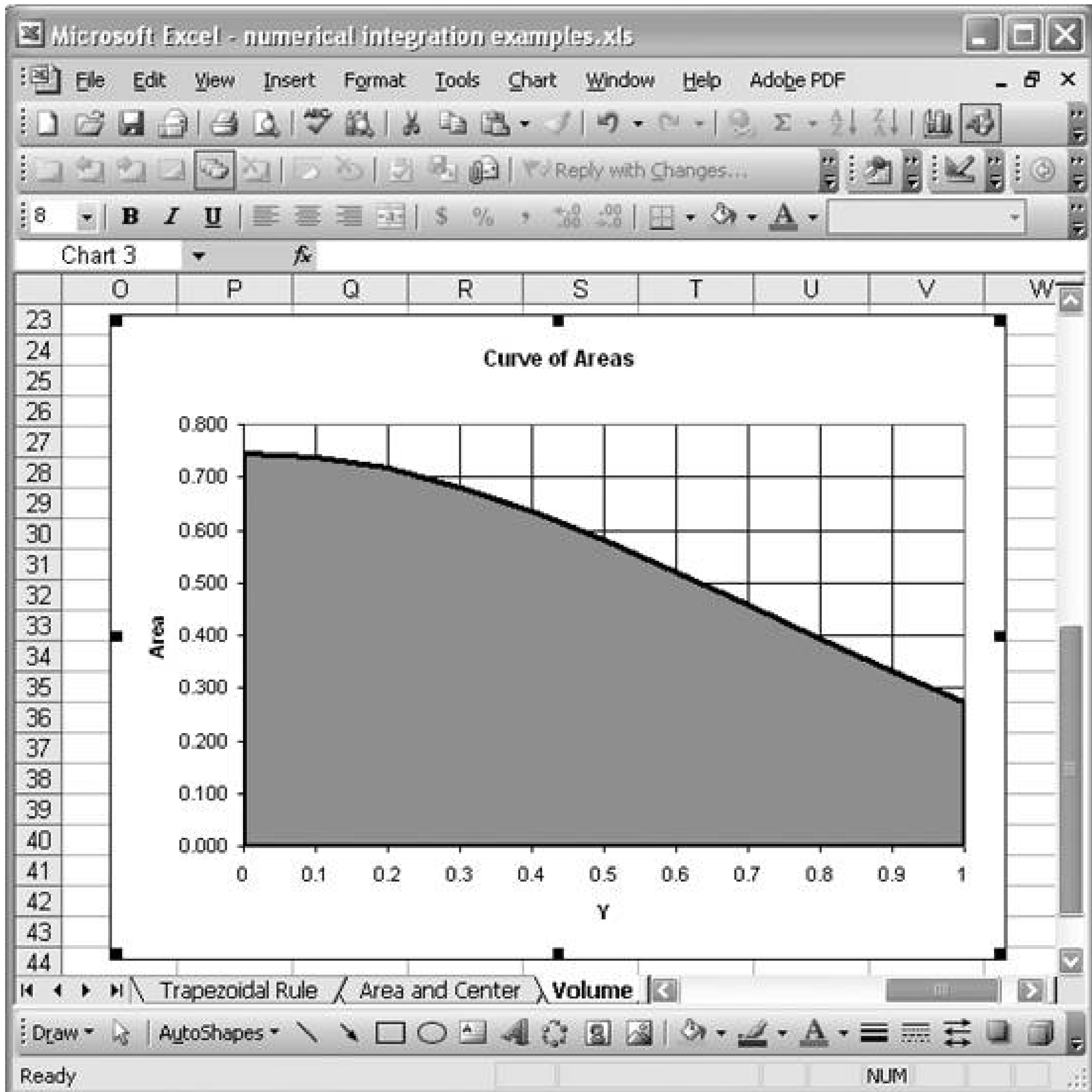
If you plot the resulting cross-sectional areas versus the  $y$ -values, you end up with a curve like that shown in [Figure 10-5](#).

This plot was generated using the  $y$ -data in the cell range D4:N4 and the area data in the cell range D16:N16. The area under this curve is the volume under the surface. So now all you need to do is integrate this cross-sectional area curve to get the volume. Cell D19 contains the formula `=D2*SUMPRODUCT(D17:N17,D16:N16)`, which performs the integration of the cross-sectional area curve, yielding the volume under the surface.

That's all there is to it. This technique is fairly straightforward and readily implemented in Excel.

Figure 10-5. Cross-sectional area curve





[← PREV](#)

## Recipe 10.6. Numerical Differentiation

### Problem

You have a table of experimentally obtained data representing some function and you'd like to approximate the derivatives of that function.

### Solution

Use numerical differentiation in your spreadsheet.

### Discussion

A well-respected professor once told me that numerical differentiation is death. That's a pretty strong statement, and what he meant was that once you start taking finite differences (a way to approximate derivatives numerically), accuracy goes downhill fast, ruining your results. This is because numerical differentiation can be very inaccurate due to its high sensitivity to inaccuracies in the values of the function being differentiated. This is in contrast to numerical integration, which is far more insensitive to functional inaccuracies because it has a smoothing effect that diminishes the effect of inaccuracies in the values of the function being integrated.

If you have an analytic form of the function under consideration, then you're far better off deriving an analytic expression for the derivative, as opposed to resorting to numerical differentiation. On the other hand, if you have experimentally obtained data with no analytic expression for the function, then you may have to resort to numerical differentiation. This latter case is the one we'll consider here.

By definition, the derivative of a function is:

When given tabulated data representing a function, we don't have the luxury of allowing  $h$  to go to zero. In this case, we have the function evaluated at some point  $x$  and some other point  $x + \delta x$  that's a finite distance,  $\delta x$ , away from  $x$ . We can approximate the derivative of a function using so-called finite differences, where we have:

It then follows that as  $\delta x$  gets smaller, this approximation should become more accurate. The trouble with tabulated data is that often  $\delta x$  is not as small as we'd like it to be for purposes of approximating derivatives. That said, we can use this approach to at least approximate the derivative of a function.

The question then arises: when dealing with a tabulated set of data of the form  $(x_i, y_i)$ , what set of values do you use to approximate the derivative at point  $i$ ? Specifically, do you use a *forward difference*,  $(f(x_{i+1}) - f(x_i))/\delta x$ , or a *backward difference*,  $(f(x_i) - f(x_{i-1}))/\delta x$ ? The answer is that you could use either approach or better yet, neither. An even better approach uses a so-called *central difference*,  $(f(x_{i+1}) - f(x_{i-1}))/2\delta x$ .

Let's consider each of these in the context of an example using a known analytic function so we can compare the approximate results to known theoretical results. Consider the function  $f(x) = x^3$ . The derivative of this function is easily derived and is  $f'(x) = 3x^2$ . Figure 10-6 shows tabulated values for this function and its theoretical derivative over the range  $x = [0, 1]$ .

I've also included several columns corresponding to approximations of the function's derivative using several different approaches, namely, forward differences, backward differences, central differences, and a so-called 5-point formula, which I'll come back to shortly.

Figure 10-6. Comparison of finite differencing schemes

	C	D	E	F	G	H	I	J
1	Function: $y = x^3$			h =		0.1		
2								
3	x	y	dy/dx	Forward Difference	Backward Difference	Central Difference	5 Point Formula	
4	0	0	0	0.01				
5	0.1	0.001	0.03	0.07	0.01	0.04		
6	0.2	0.008	0.12	0.19	0.07	0.13	0.12	
7	0.3	0.027	0.27	0.37	0.19	0.28	0.27	
8	0.4	0.064	0.48	0.61	0.37	0.49	0.48	
9	0.5	0.125	0.75	0.91	0.61	0.76	0.75	
10	0.6	0.216	1.08	1.27	0.91	1.09	1.08	
11	0.7	0.343	1.47	1.69	1.27	1.48	1.47	
12	0.8	0.512	1.92	2.17	1.69	1.93	1.92	
13	0.9	0.729	2.43	2.71	2.17	2.44		
14	1	1	3		2.71			
15								
16								
17								

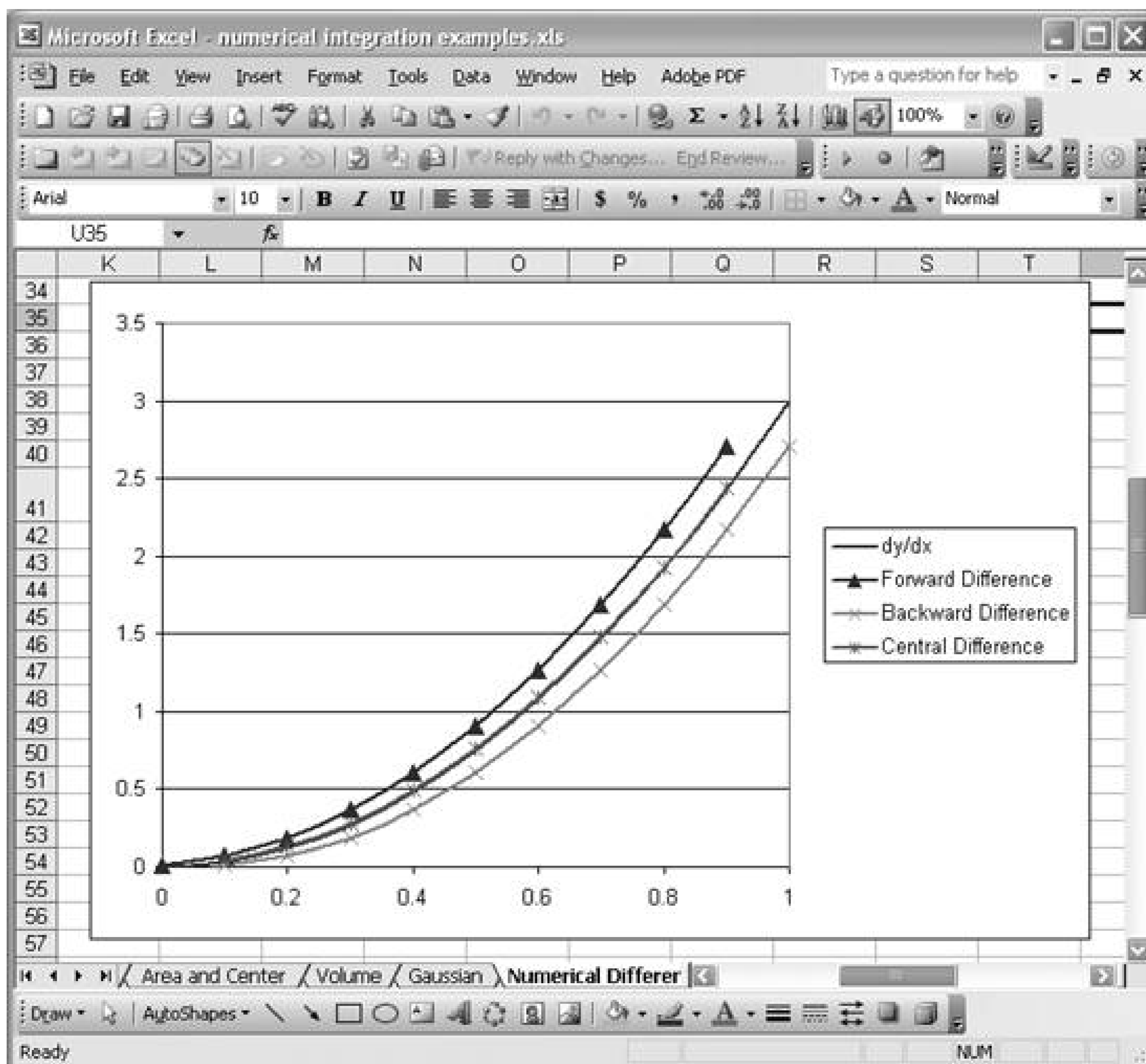
The formulas in column F for the forward difference approximation are of the form  $= (D5 - D4) / \$G\$1$ , where the difference  $D5 - D4$  is the forward difference in  $y$ -values and G1 contains the value for  $\delta x$  (labeled  $h$  in the spreadsheet). Similarly, the formulas in column G for the backward difference approximation are of the form  $= (D5 - D4) / \$G\$1$ . The central difference formulas in column H are of the form  $= (D6 - D4) / (2 * \$G\$1)$ .

Plotting these results along with the theoretical derivatives yields the chart shown in [Figure 10-7](#).

The  $dy/dx$  curve is the theoretical derivative. The others are the various approximations. Clearly, the central difference approach yields far better results than either forward differencing or backward differencing. This is true in general, and central differences are usually preferred over the other two approaches. For this particular function, the central difference approximation curve lies directly on top of the theoretical curve, implying very close agreement. You can also see this from the tabulated results shown in [Figure 10-6](#).

The central difference approach has the effect of smoothing the derivative at a point as it considers the values of the function both ahead of and behind the point under consideration. The 5-point formula mentioned earlier extends this idea, taking into account even

Figure 10-7. Chart comparing approximate to theoretical results



more points around the specific point under consideration. The 5-point formula is commonly presented in texts on numerical methods and the formula is as follows:

In this example, the 5-point formula is of the form  $= (1 / (12 * \$G\$1)) * (D4 - 8 * D5 + 8 * D7 - D8)$ . This formula corresponds to the approximation in cell I6.

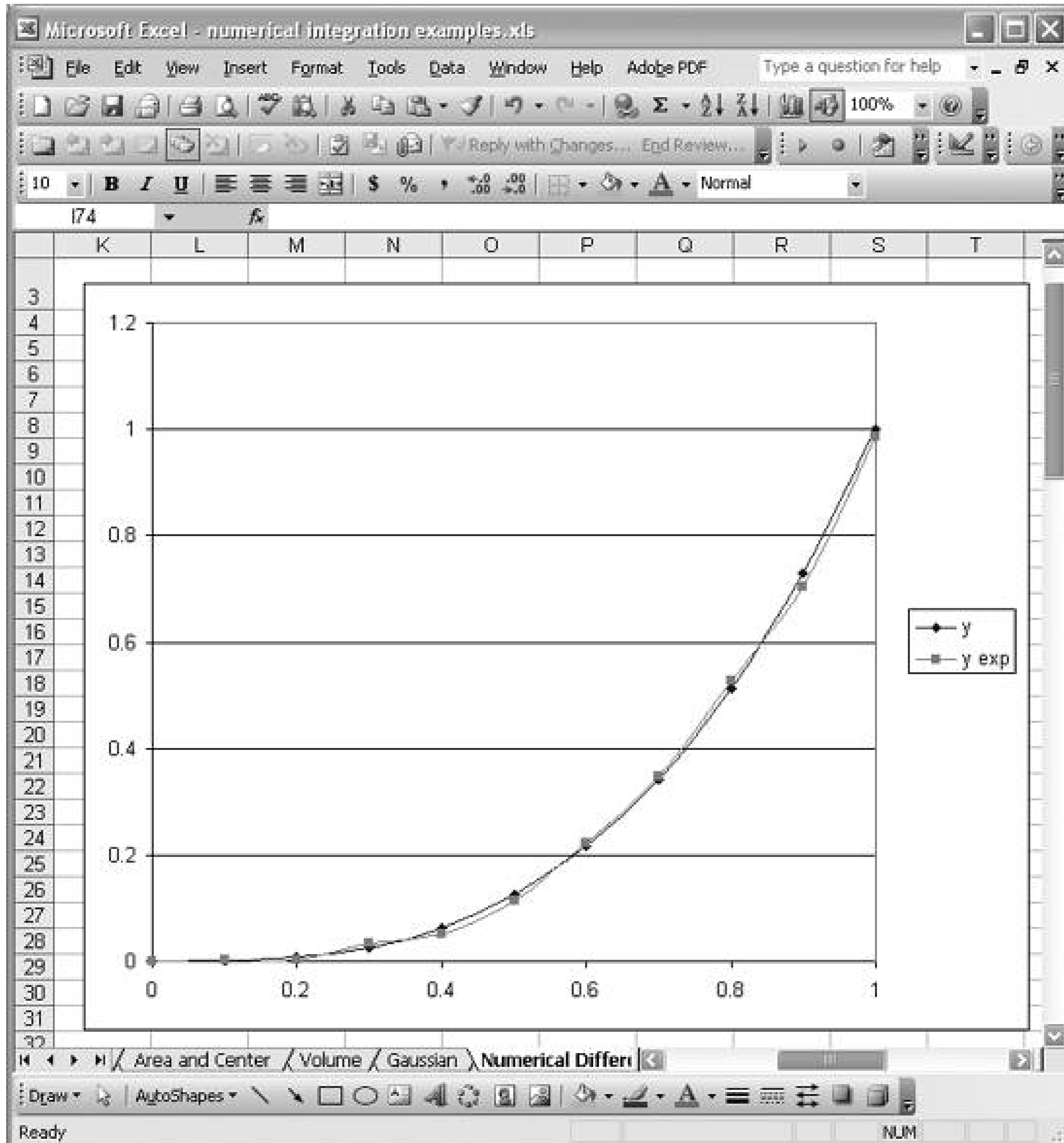
For this particular example, the 5-point formula yields results exactly matching the theoretical derivatives. In practice, this isn't always the case, but it does show how much better the 5-point formula is than forward or backward differencing for this example.

As mentioned earlier, numerical differentiation is very sensitive to errors in the values of the function being considered, and this is one of the main sources of error in numerical derivatives. The example so far has included exact values for the function under consideration.

In practice, if this data was measured experimentally, you'd expect some error in values. To simulate such error, I computed some random deviations for the example function. [Figure 10-8](#) shows the

simulated experimental data plotted against the theoretical values from the function under consideration.

Figure 10-8. Simulated experimental data



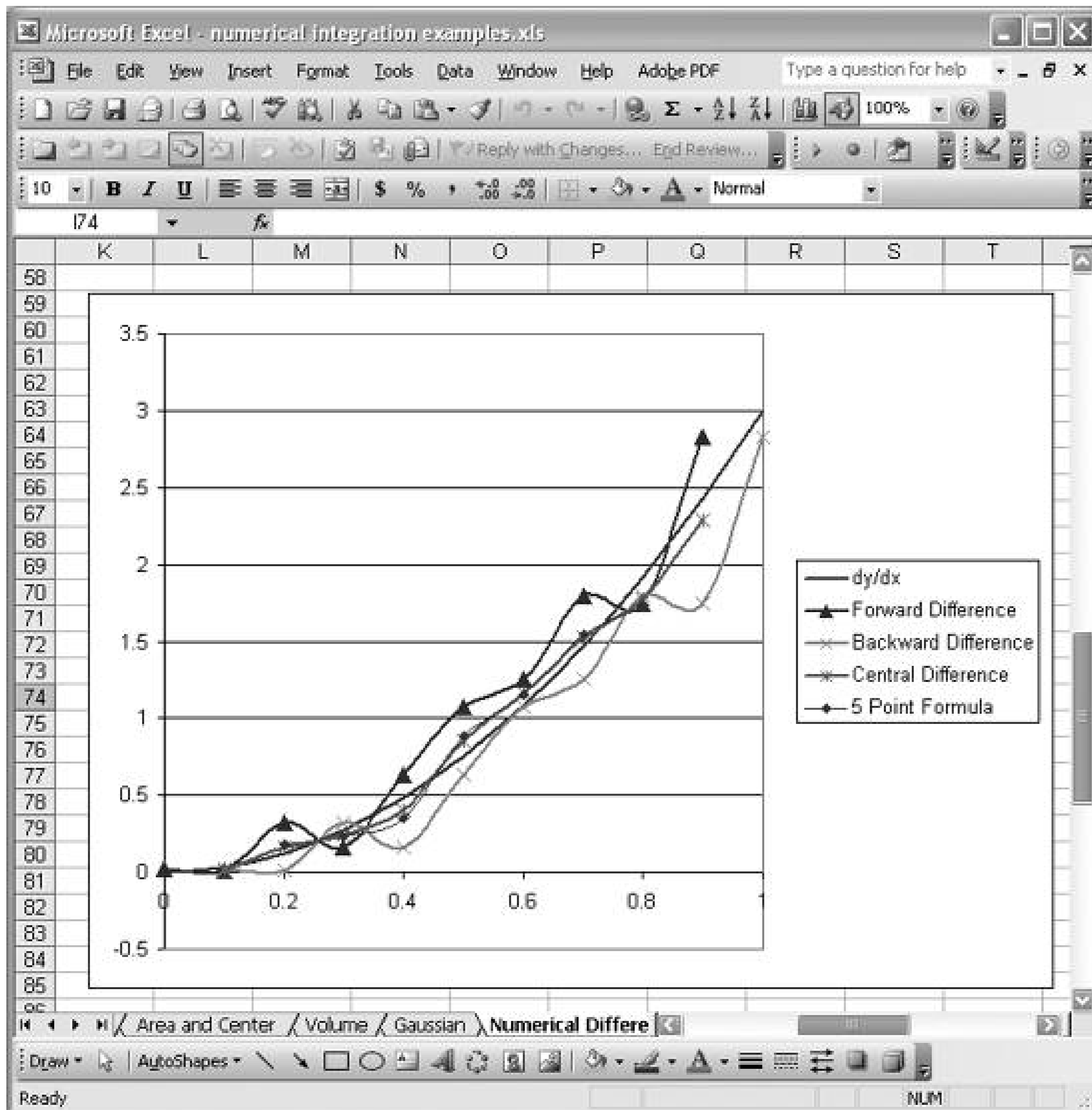
As you can see, the curve labeled  $y \text{ exp}$  fairly closely approximates the theoretical curve of  $y = x^2$ , but with some small errors. Now, let's look at the effect of these small errors on the derivatives as approximated using the numerical differentiation formulas shown earlier. [Figure 10-9](#) shows the results.

This chart is dramatically different from that shown in [Figure 10-7](#). Look at how wildly the forward

and backward difference curves deviate from the curves shown in [Figure 10-7](#). This is characteristic of numerical derivatives in light of small errors in the values of the function under consideration. Any small errors in the values of the function from point to point are amplified in the numerically obtained derivative.

These results are motivation for using the central difference or 5-point formulas, as they tend to smooth out these oscillations by considering more neighboring values of

Figure 10-9. Approximate derivatives for experimental data



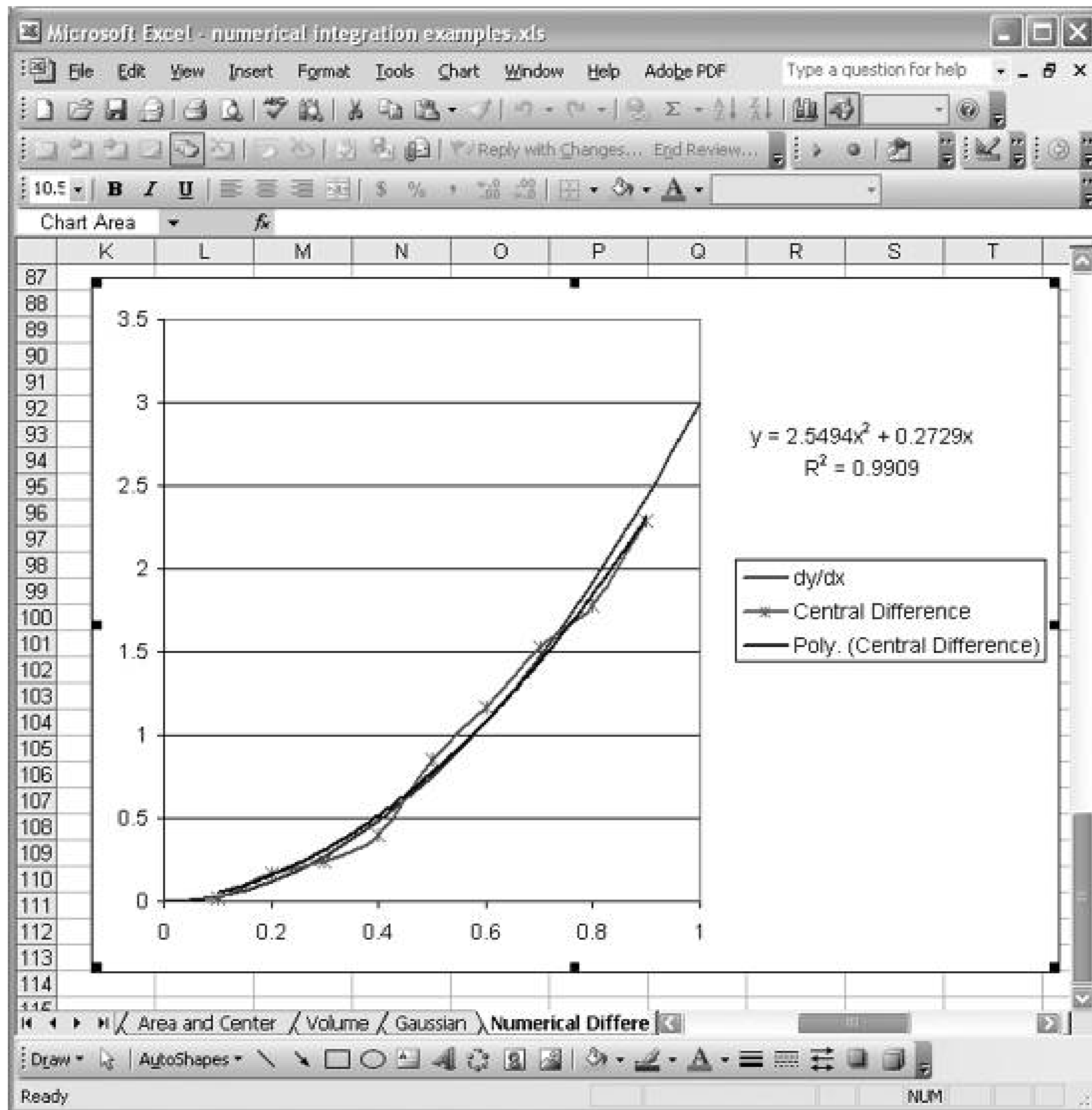
the function. [Figure 10-9](#) illustrates that these latter methods do indeed yield far better approximations. The curves of the derivative using these two approaches still aren't as smooth as the theoretical curve, but they are far better than either the forward or backward difference curves.

The smoothing effect offered by formulas like the central difference and 5-point formulas has inspired other techniques for approximating derivatives. Indeed, it would seem plausible to smooth the tabulated functional values before computing numerical derivatives in an effort to increase accuracy. Moreover, you can fit an interpolation function to a given set of data and then use the derived analytic expression for the fit curve to derive an expression for the derivative, thus giving much smoother results for the derivative. (See [Chapter 8](#) to learn how to fit equations in Excel.) In fact, this is the basis for many numerical methods that require derivative computations.

Another approach is to compute the numerical derivatives as discussed here and then fit a curve through those results to yield a smooth derivative curve. [Figure 10-10](#) shows the central difference results from [Figure 10-9](#) plotted along with a second-order least-squares fit polynomial. This polynomial was fit using Excel's trendline feature (see [Chapter 8](#)) on the central difference curve.

Figure 10-10. Smoothed derivative curve





The resulting trendline is smooth and approximates the theoretical derivative fairly well, giving a much better picture of the derivative than the raw derivative curve computed using central differences.

There are many schemes and formulas for computing numerical derivatives besides the ones discussed here. Further, you can apply these techniques to compute higher derivatives such as the second and third derivatives of a function. Such formulas are widely published in books on numerical methods, which include numerical differentiation.

## See Also

Take a look at [Chapter 8](#) to learn how to fit curves to data using Excel. You can use those fit curves to approximate derivatives as discussed here.



← PREV

# Chapter 11. Solving Ordinary Differential Equations

[Section 11.0. Introduction](#)

[Recipe 11.1. Solving First-Order Initial Value Problems](#)

[Recipe 11.2. Applying the Runge-Kutta Method to Second-Order Initial Value Problems](#)

[Recipe 11.3. Tackling Coupled Equations](#)

[Recipe 11.4. Shooting Boundary Value Problems](#)

← PREV

## 11.0. Introduction

In this chapter we're going to focus on numerically solving ordinary differential equations using Excel

There are many methods available for numerically solving ordinary differential equations. Solution methods for *initial value problems* include such standard methods as *Euler's method*, the *improved Euler method*, the *Runge-Kutta method*, the *leap frog method*, various implicit schemes, as well as various adaptive schemes. Texts on numerical methods are full of such methods.

Other classes of problems dealing with ordinary differential equations include boundary value problems and initial-boundary value problems. Methods available for boundary value problems include the *shooting method* and variants, the *band matrix method*, *Richardson extrapolation*, various *finite difference methods*, and many others.

I aim to show you a few selected methods in this chapter to show you how you might implement some of these well known methods in Excel. Numerical methods texts contain detailed descriptions of these, as well as other methods. The Excel techniques I'll show you in this chapter will allow you to implement many of the standard methods in Excel.

## Recipe 11.1. Solving First-Order Initial Value Problems

### Problem

You need to numerically solve a first-order differential equation of the form:

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

$$y(0) = a$$

### Solution

This is a standard initial value problem and you can implement any of a number of standard numerical integration techniques to solve it using Excel and VBA.

### Discussion

Consider a specific initial value problem such as:

$$y'(x) = e^x - x - 1$$

The exact solution for this problem is:

$$y = e^x - x - 1$$

[Figure 11-1](#) shows a plot of this exact solution.

Normally, if you can find an exact solution, you wouldn't resort to numerical integration. However, I chose this problem so that you can compare the numerical results with the exact solution.

In this recipe, I will show you how to implement Euler's method. This isn't necessarily the best method, since it requires small step sizes to ensure accuracy. However, it's simple enough to show you the mechanics of implementing such a method in Excel and VBA without clouding the issue. In [Recipe 11.2](#), I'll show you how to implement a much better method. In [Recipe 11.3](#), I'll show you why Euler's method may not be the best choice for some problems.

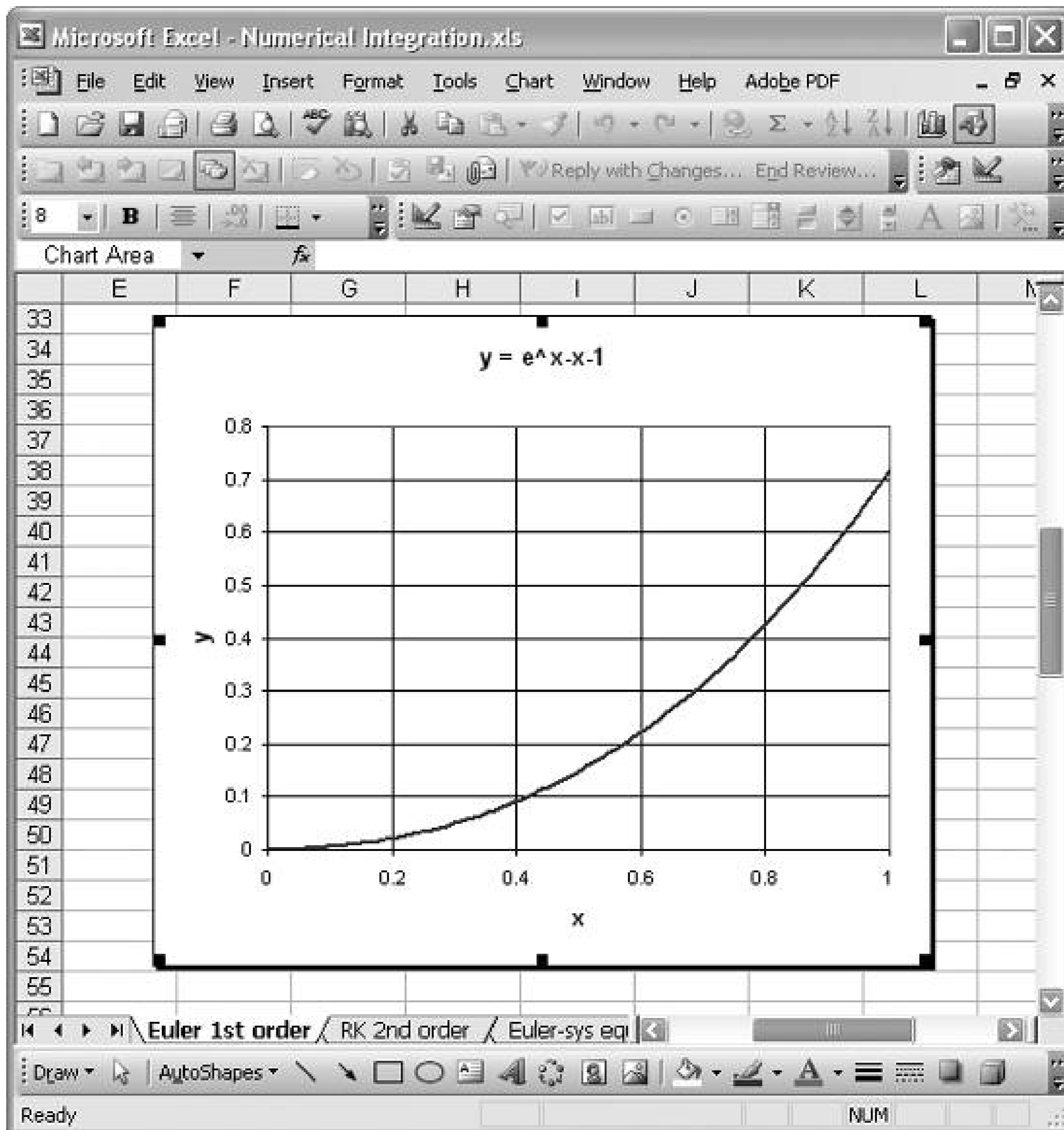
Euler's method is based on taking the first two terms of a *Taylor series expansion* of a function to predict the value of the function at some point, knowing the value of the function at some other point.

plus some information about the derivatives of that function. In this case, you can write the Taylor series expansion of  $y$  as follows:

$$y(x + \Delta x) = y(x) + (\Delta x) \frac{dy}{dx} + \left( \frac{(\Delta x)^2}{2!} \right) \frac{d^2y}{dx^2} + \left( \frac{(\Delta x)^3}{3!} \right) \frac{d^3y}{dx^3} + \dots$$

Euler's method considers only the first two terms on the right side of this equation, ignoring all higher-order terms, which collectively constitute the truncation error for this method. For the problem we're considering here, we know  $dy/dx$  it's the given differential equation therefore, we can use the first two terms in the Taylor series expansion of  $y$  to determine  $y$  at successive increments along  $x$ , starting at the given initial value. This scheme is very easy to implement in Excel and VBA. In fact, you can implement this scheme easily in a standard spreadsheet without the use of VBA, or you can write a VBA subroutine, which I think is a more versatile approach. I'll show you how to do both.

Figure 11-1. Exact solution of an example initial value problem



## Using spreadsheet only

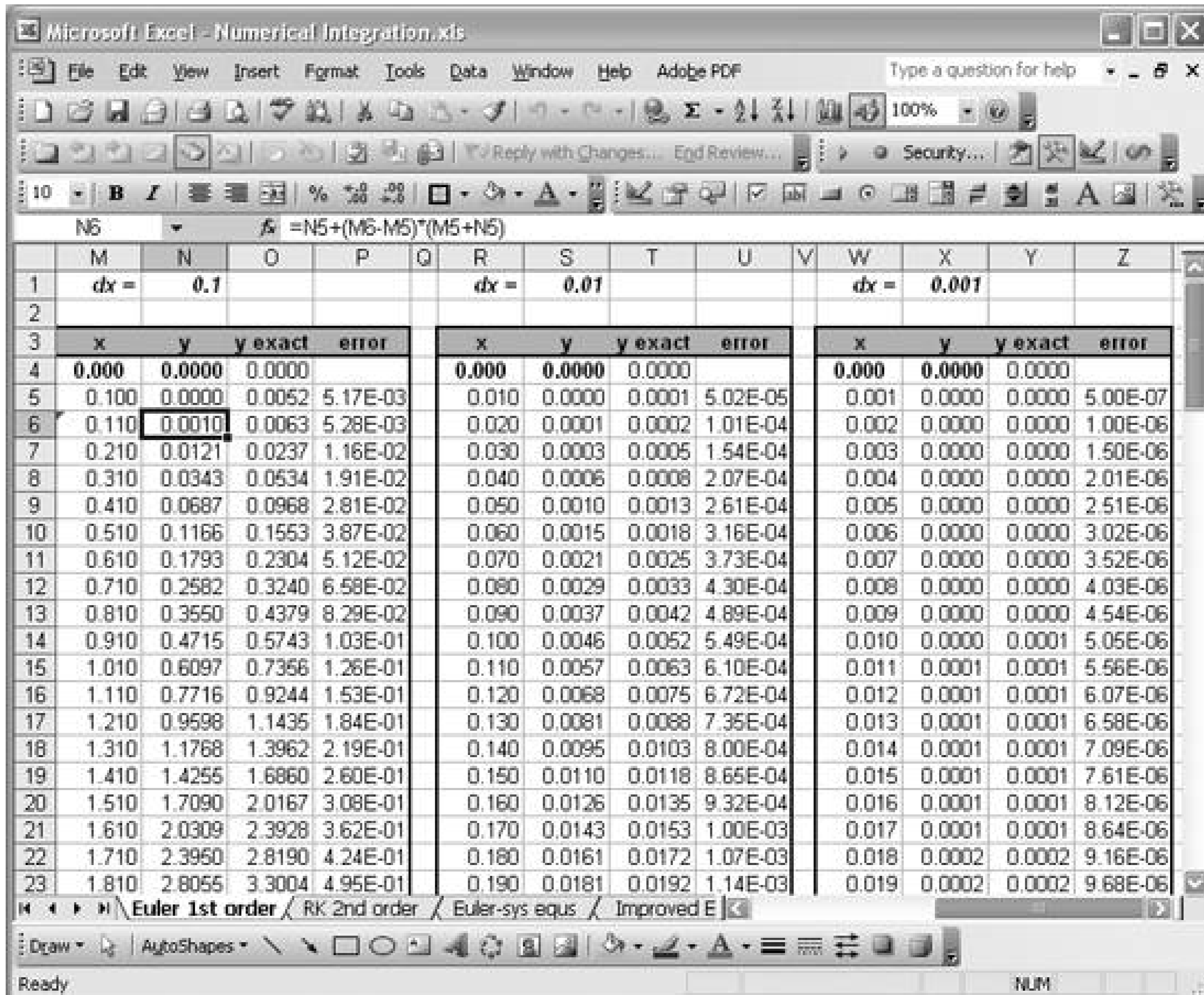
[Figure 11-2](#) shows a spreadsheet I set up implementing Euler's method using only spreadsheet formulas.

The three separate tables you see in [Figure 11-2](#) are solutions to the same problem using three different step sizes,  $\delta x$  (labeled  $\Delta x$  in [Figure 11-2](#)). You need only one table to solve the problem, but I include three to show the differences in results using different step sizes.

The tables are fairly straightforward. The columns labeled  $x$  contain formulas like  $=M5+\$N\$1$ , which simply compute a new  $x$ -value given the previous  $x$ -value plus the step size. The columns labeled  $y$  contain Euler's formula for predicting the next  $y$ -value. The spreadsheet formula looks like  $=N5+(M6-$

$M5) * (M5+N5)$ . The  $(M6-M5)$  part of this formula is the step size, which could have been simply  $N\$1$  as well. The  $(M5+N5)$  part is the derivative of  $y$  from the given differential equation.

Figure 11-2. Euler's method using only spreadsheet formulas



The columns labeled  $y$  exact contain formulas like  $=EXP(M6)-M6-1$ , corresponding to the equation for the exact solution discussed earlier. And finally, the error columns contain formulas like  $=(O6-N6)$ , which is just the difference between the exact solution and the approximate solution.

The error decreases as the step size decreases, as you would expect. Note, however, that you have to perform far more computations with the smaller step size to cover an equivalent range of  $x$ -values. For example, if you wanted to compute the solution over the range  $x = [0,1]$ , you'd need 10 rows of calculations using a step size of 0.1, 100 rows using a step size of 0.01, and 1,000 rows using a step size of 0.001. Clearly, if you require a small step size and want to cover a reasonable range of  $x$ -values for a given problem, your spreadsheet tables will grow very large. This becomes cumbersome, especially when you want to plot the results, change the range, or change your step size. VBA offers a better approach.



## Using VBA

[Example 11-1](#) shows the VBA subroutine I wrote implementing Euler's method for our example problem.

### Example 11-1. DoEuler1stOrder

```
Public Sub DoEuler1stOrder( )
    Dim yn As Double
    Dim yn1 As Double
    Dim xn As Double
    Dim dx As Double
    Dim n As Integer

    yn = 0
    xn = 0
    dx = 0.001
    n = 1000

    For i = 1 To n
        yn1 = yn + (xn + yn) * dx
        xn = xn + dx
        yn = yn1
        ActiveSheet.Cells(i + 1, 1) = xn
        ActiveSheet.Cells(i + 1, 2) = yn
    Next i
End Sub
```

This subroutine computes estimated values for  $y$  over a range of  $x$ -values and places the results in the active spreadsheet starting at row 2 in columns A and B.

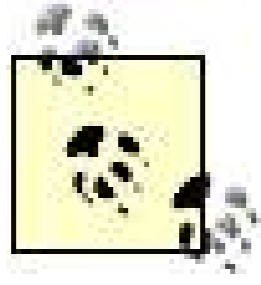
The variables declared at the beginning of this subroutine are used to store  $y_x$  ( $yn$ ),  $y_{x+1}$  ( $yn1$ ),  $x$  ( $xn$ ), and  $\delta x$  ( $dx$ ).  $n$  is a counter variable representing the number of steps to take.

The `For` loop iterates from 1 to  $n$ , computing estimates for  $y$  at each step. The line `yn1 = yn + (xn + yn) * dx` computes the predicted  $y$ -value given the previous  $y$ -value, the step size, and the derivative of  $y$  from the original differential equation. After this calculation and after updating  $xn$  and  $yn$ , the subroutine writes the results to the active spreadsheet. The line `ActiveSheet.Cells(i + 1, 1) = xn` writes the  $x$ -value to row  $i+1$  and column 1 in the R1C1 cell reference style (column 1 corresponds to column A). The line `ActiveSheet.Cells(i + 1, 2) = yn` writes the  $y$ -value to row  $i+1$ , column 2.

I set up a button on a spreadsheet (just like the one I showed in [Recipe 9.3](#)) to call this subroutine. When I press the button, the spreadsheet is populated with the computed results. While this approach works fine, it suffers one of the problems mentioned earlier with using pure spreadsheet formulas: when using a small step size while covering a reasonable range of  $x$ -values, the

spreadsheet contains a large number of data points. The subroutine shown in [Example 11-1](#) would fill 1,000 rows of data, which could become cumbersome, as discussed earlier.

To mitigate this problem, you can readily add a few lines of code to `DoEuler1stOrder` that will output results every 100 steps or so in order to keep the number of output rows manageable.



This is a common technique in numerical and simulation work. For example, real time simulations and video games that use physics-based effects use numerical integration to solve equations of motion. These algorithms typically use very small time steps that would render the simulation or game extremely slow if the graphics were updated every time step. Instead, the underlying computations are computed at small steps but the graphics are updated at a rate that keeps the simulation or game at a real-time pace.

[Example 11-2](#) shows the modified subroutine using this controlled output technique.

### Example 11-2. Modified DoEuler1stOrder

```
Public Sub DoEuler1stOrder( )
    Dim yn As Double
    Dim yn1 As Double
    Dim xn As Double
    Dim dx As Double
    Dim n As Integer
    Dim c As Integer
    Dim k As Integer

    yn = 0
    xn = 0
    dx = 0.001
    n = 1100
    c = n / 10
    k = 1

    For i = 1 To n
        yn1 = yn + (xn + yn) * dx
        xn = xn + dx
        yn = yn1

        If c >= (n / 10) Then
            ActiveSheet.Cells(k + 1, 1) = xn
            ActiveSheet.Cells(k + 1, 2) = yn
            k = k + 1
            c = 0
        Else
            c = c + 1
        End If
    End For
End Sub
```

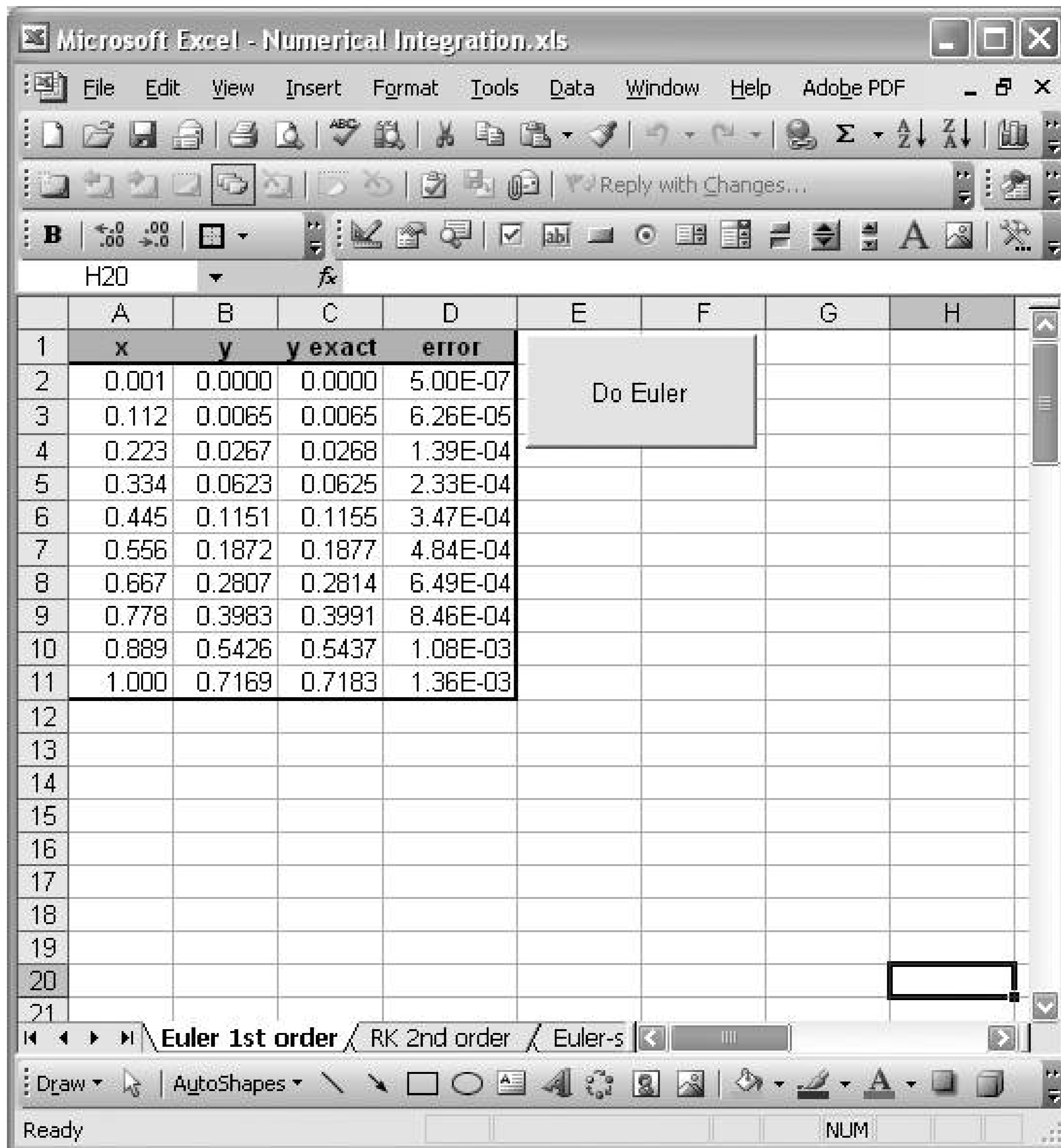
```
Next i  
End Sub
```

I added two new variables, `c` and `k`. `c` is a counter variable used to determine when to output results to the spreadsheet, and `k` is a counter variable used to track the output row.

I decide to output only 10 rows of data while keeping the step size at 0.001. I still wanted to cover a range of  $x$ -values from 0 to 1, which meant I had to make `n` equal to 1,100. Thus, output will be delivered to the spreadsheet every  $n/10$  steps, that is, whenever `c` equals  $n/10$ .

The results of this new subroutine are shown in [Figure 11-3](#).

Figure 11-3. Results using VBA



Here you can see the little table I set up to receive the 10 rows of results. You can also see the button I set up to run the subroutine. This version is much more manageable and gives you greater flexibility when you need to modify your solution algorithm. For example, to change the step size to 0.0001 while keeping the results table the same size, all you need to do is change  $dx$  to 0.0001 and  $n$  to 11,000 in `DoEuler1stOrder` and then rerun it.

If you try Euler's method for a problem and see that it gives poor results, you might then decide to implement a higher-order method such as the Runge-Kutta method. This latter method would be cumbersome to set up in a spreadsheet alone; however, if you're already using VBA, you can modify your solution subroutine without too much effort and then recompute the results.

## See Also

Check out [Recipe 11.2](#) to see how to extend the VBA techniques from this recipe to handle second-order equations.



## Recipe 11.2. Applying the Runge-Kutta Method to Second-Order Initial Value Problems

### Problem

You need to numerically solve a second-order differential equation of the form:

$$A \frac{d^2 y}{dx^2} + B \frac{dy}{dx} = f(x, y)$$

$$y(0) = \alpha$$

$$\frac{dy}{dx}(0) = \beta$$

### Solution

This is a standard initial value problem, and you can implement any of a number of standard numerical integration techniques to solve it using Excel and VBA.

### Discussion

Consider this equation and initial conditions:

Physically, this equation represents the equation of motion for an object moving under some applied thrust,  $T$ .<sup>[\*]</sup> Here,  $m$  represents the mass of the object,  $C_d$  represents a drag factor, and  $s$  represents the displacement (position) of the object. This equation allows us to compute the motion of this object.

[\*] This problem was adapted from an example I used in one of my previous books, *Physics for Game Developers* (O'Reilly).

To solve this equation, it helps to rewrite it in the form of two first-order equations. Let  $v = ds/dt$  represent the velocity of the object. Now we have the following:

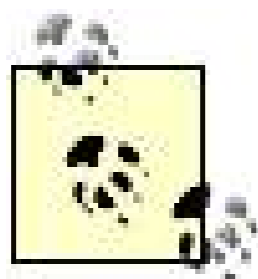
$$m \frac{dv}{dt} = T - C_d v$$

$$\frac{ds}{dt} = v$$

$$s_{t=0} = 0$$

$$v_{t=0} = 0$$

These equations represent two coupled first-order equations, and you can apply any of a number of numerical integration schemes to solve them both.



From physics, you know that acceleration is the time rate of change of velocity,  $a = dv/dt$ . Therefore, you can see that the equation written in terms of velocity is nothing more than an expression of Newton's second law of motion, which states that the sum of all forces acting on an object equals the object's mass times its acceleration.

I decided to use the Runge-Kutta method for this example. This method is based on taking more terms in the Taylor series expansion of a function, as I explained in [Recipe 11.1](#). The caveat here is that you need to apply further Taylor series expansions to estimate the higher-order derivatives. The Runge-Kutta approach reduces the truncation error to something on the order of  $(\Delta t)^5$ , as opposed to  $(\Delta t)^2$  in the case of Euler's basic method ( $\Delta t$  is the step size). This means you can usually use larger step sizes while still maintaining accuracy. The trade-off here is more computations per step. But ultimately, with the larger step size you should be able to reduce the overall computational burden using this method.

The Runge-Kutta integration formulas are a little more complicated than the simple formula we used with Euler's basic method. The general equations are as follows:

$y'$  represents  $dy/dx$  in these equations. You can see that every step requires four intermediate computations prior to predicting the next  $y$ -value, that is,  $y(x + \delta x)$ .

[Example 11-3](#) shows the VBA subroutine I wrote implementing these formulas.

### Example 11-3. Implementation of Runge-Kutta method

```
Public Sub DoRK2ndOrder( )
    Dim T As Double      ' Thrust
    Dim Cd As Double     ' Drag coefficient
    Dim M As Double      ' Mass
    Dim dt As Double     ' Time step size
    Dim F As Double      ' Force
    Dim A As Double      ' Acceleration
    Dim Vn As Double     ' Velocity at time t
    Dim Vn1 As Double    ' Velocity at time t + dt
    Dim Sn As Double     ' Displacement at time t
    Dim Sn1 As Double    ' Displacement at time t + dt
    Dim time As Double   ' Total time
    Dim k1 As Double     ' Runge-Kutta k1
    Dim k2 As Double     ' Runge-Kutta k2
    Dim k3 As Double     ' Runge-Kutta k3
    Dim k4 As Double     ' Runge-Kutta k4
    Dim n As Integer     ' Counter controlling total number of time steps
    Dim C As Integer     ' Counter controlling output of results to spreadsheet
    Dim k As Integer     ' Counter controlling output row
    Dim r As Integer     ' Number of output rows

    ' Extract given data from the active spreadsheet:
    With ActiveSheet
        dt = .Range("dt")
        T = .Range("T")
        M = .Range("M")
        Cd = .Range("Cd")
        n = .Range("n")
        r = .Range("r_")
    End With

    ' Initialize variables:
    k = 1
    time = 0
    C = n / r
    Vn = 0
    Sn = 0

    ' Start iterations:
    For i = 1 To n

        ' Compute k1:
```



```

F = (T - (Cd * Vn))
A = F / M
k1 = dt * A

' Compute k2:
F = (T - (Cd * (Vn + k1 / 2)))
A = F / M
k2 = dt * A

' Compute k3:
F = (T - (Cd * (Vn + k2 / 2)))
A = F / M
k3 = dt * A

' Compute k4:
F = (T - (Cd * (Vn + k3)))
A = F / M
k4 = dt * A

' Compute velocity at t + dt:
Vn1 = Vn + (k1 + 2 * k2 + 2 * k3 + k4) / 6

' Compute displacement at t + dt using Euler prediction:
Sn1 = Sn + Vn1 * dt

' Update variables:
time = time + dt
Vn = Vn1
Sn = Sn1

' Output results to the active spreadsheet:
If C >= n / r Then
    ActiveSheet.Cells(k + 1, 1) = time
    ActiveSheet.Cells(k + 1, 2) = Sn
    ActiveSheet.Cells(k + 1, 3) = Vn
    k = k + 1
    C = 0
Else
    C = C + 1
End If
Next i
End Sub

```

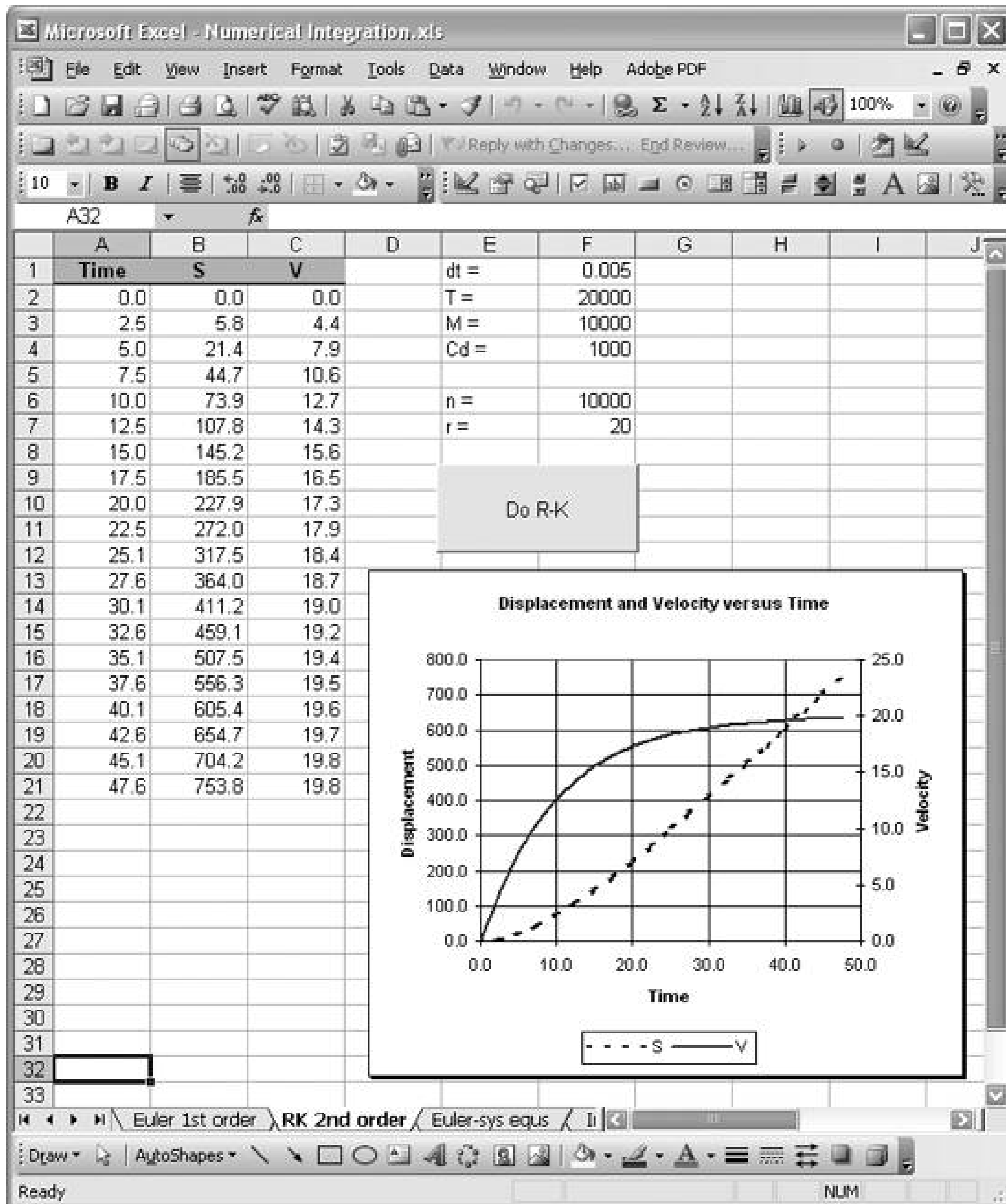
This is a relatively long subroutine compared to the one shown in [Example 11-2](#) implementing Euler's method, so I commented this one to help you navigate through it.

As usual, several local variables are declared at the beginning of the subroutine. There are a bunch of them here, so I labeled each one in [Example 11-3](#) to indicate their respective purposes.

After the variables are declared, several of them are populated with data retrieved from the active

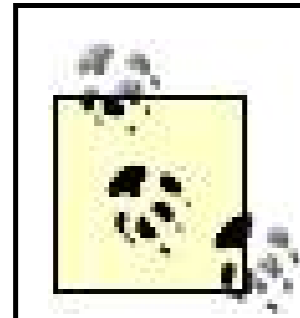
spreadsheet. The spreadsheet is shown in [Figure 11-4](#).

Figure 11-4. Solution using the Runge-Kutta method



Values retrieved from the spreadsheet include the time-step size, thrust, mass, drag coefficient, number of iterations, and number of output rows. The block of code in [Example 11-3](#) containing the `with` statement performs the data retrieval. Notice that I use cell names to refer to the spreadsheet

cells containing the data of interest.



See the discussion of [Example 9-1](#) in [Recipe 9.3](#) for information on the use of `With` statements. Also, take a look at [Recipe 1.14](#) to learn how to define cell names.

The next little section of code simply initializes a few variables. This is where the initial values for displacement and velocity specified in the problem are set.

Next, the `For` loop iterates computing estimates for displacement and velocity at each step. Taking a closer look at the code within the `For` loop, you can see how I applied the Runge-Kutta formulas to compute  $v_{t+dt}$ . To make things a little more intuitive (at least to me), I broke the computations of the  $k$ -values into a few steps, using the analogy of Newton's second law of motion.  $F$  in these computations stands for the sum of forces. For this problem, the sum of forces is composed of the thrust force and the drag force.  $A$  then represents the object's acceleration, which is equal to the net force divided by the object's mass. Finally, the  $k$ -values are computed by multiplying the acceleration by the time step.

I find it easier to think in terms of physical values like force and acceleration whenever possible. Knowing that acceleration is equal to  $dv/dt$ , you can readily see that acceleration corresponds to  $y'$  in the Runge-Kutta formulas for this particular problem. This is because we broke the original equation up and are dealing with the first one involving velocity.

After all of the  $k$ -values are obtained,  $v_{t+dt}$  is estimated using the Runge-Kutta formula discussed earlier. Next, a simple Euler estimate is made for the corresponding displacement,  $s_{t+dt}$ .

Finally, the results are sent to the active spreadsheet in the same manner I showed you in [Recipe 11.1](#). [Figure 11-4](#) shows the results, including a plot of velocity and displacement versus time, using a time step of 0.005. You should play around with this time step to see how the accuracy of the results is affected. (If this were a class, I'd assign that exercise for homework.)

I should also mention that I used a button control, as you can see in [Figure 11-4](#), to call the subroutine when the button is clicked. Refer back to [Recipe 9.3](#) to learn how to add this sort of button control and assign its functionality.

[← PREV](#)

## Recipe 11.3. Tackling Coupled Equations

### Problem

You've seen how to apply both the Runge-Kutta method and Euler's basic method using VBA for first- or second-order differential equations, but now you need to solve a system of coupled equations. You're wondering how to extend what you've learned so far to deal with this more complicated problem.

### Solution

You can apply the same VBA techniques discussed earlier with some modifications to deal with multiple equations.

### Discussion

Consider this system of coupled nonlinear differential equations with initial conditions:

Pretty, huh? This problem originally consisted of two coupled second-order equations that were reduced to four first-order equations using the same technique discussed in Recipe 11.2. This is a common technique for reducing the order of differential equations, making them more amenable to solving. This particular problem was adapted from a problem that appears in *An Introduction to Linear and Nonlinear Finite Element Analysis*, by Prem K. Kythe and Dongming Wei (Birkhauser). The problem represents a finite element solution of the nonlinear vibrations of a metal rod. Since this is a nonlinear problem, it requires numerical solution. In their book, Kythe and Wei present some Matlab code to solve this problem using Euler's method.

## Euler's method

I implemented Euler's method in VBA to illustrate how to solve this same system in Excel. Example 11-4 shows my VBA code.

### Example 11-4. DoEuler for nonlinear system

```
Public Sub DoEuler( )
    Dim y(1 To 4) As Double
    Dim t(1 To 4) As Double
    Dim n As Integer
    Dim dt As Double
    Dim time As Double

    y(1) = 29.8613
    y(2) = 0
    y(3) = 30.0616
    y(4) = 0

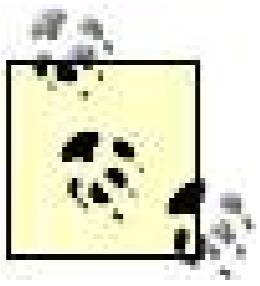
    n = 5000
    dt = 5 / n
    time = 0

    For i = 1 To n
        t(1) = y(1) + dt * dy1dt(y(1), y(2), y(3), y(4))
        t(2) = y(2) + dt * dy2dt(y(1), y(2), y(3), y(4))
        t(3) = y(3) + dt * dy3dt(y(1), y(2), y(3), y(4))
        t(4) = y(4) + dt * dy4dt(y(1), y(2), y(3), y(4))

        y(1) = t(1)
        y(2) = t(2)
        y(3) = t(3)
        y(4) = t(4)

        time = time + dt
        ActiveSheet.Cells(i + 1, 1) = time
        ActiveSheet.Cells(i + 1, 2) = y(1)
        ActiveSheet.Cells(i + 1, 3) = y(2)
        ActiveSheet.Cells(i + 1, 4) = y(3)
        ActiveSheet.Cells(i + 1, 5) = y(4)
    Next i
End Sub
```

As you can see, this is not a particularly complicated subroutine. It's very similar to the code in Example 11-2 that I showed you for solving a first-order equation using Euler's method. The major difference in this case is that we have four coupled equations instead of a single equation.



I'm using VBA array variables in this code for the  $y$ -values and temporary  $t$ -values. In VBA, you declare an array using `Dim y(1 To 4) As Double`. You can access the array using `y(1)` or `y(3)`, and so on.

I have to confess that there's a little more to this code than is shown in Example 11-4. Notice the functions `dy1dt`, `dy2dt`, `dy3dt`, `dy4dt` in the lines of code that implement Euler's integration formula (the line just after the `For` statement). These functions compute the derivatives for each  $y$ . The derivative expressions are taken directly from the system of equations shown earlier. The corresponding VBA functions are shown in Example 11-5.

### Example 11-5. Derivative functions

```
Public Function dy1dt(y1 As Double, y2 As Double, y3 As Double, y4 As Double) As Double
    dy1dt = y2
End Function

Public Function dy2dt(y1 As Double, y2 As Double, y3 As Double, y4 As Double) As Double
    dy2dt = (-15600# * (2# * ((Abs(y1)) ^ -0.74) + 3# * ((Abs(y3 - y1)) ^ -0.74))) _
        / 7# * y1 + (3# * 15600# * ((Abs(y3 - y1)) ^ -0.74)) / 7# * y3
End Function

Public Function dy3dt(y1 As Double, y2 As Double, y3 As Double, y4 As Double) As Double
    dy3dt = y4
End Function

Public Function dy4dt(y1 As Double, y2 As Double, y3 As Double, y4 As Double) As Double
    dy4dt = (-15600# * (-((Abs(y1)) ^ -0.74) - 5# * ((Abs(y3 - y1)) ^ -0.74))) _
        / 7# * y1 - (5# * 15600# * ((Abs(y3 - y1)) ^ -0.74)) / 7# * y3
End Function
```

This code runs just fine. The problem is that this is a particularly troublesome system of equations for Euler's method. The solution diverges dramatically as it marches along in time. Check out Figure 11-5 to see what I mean.

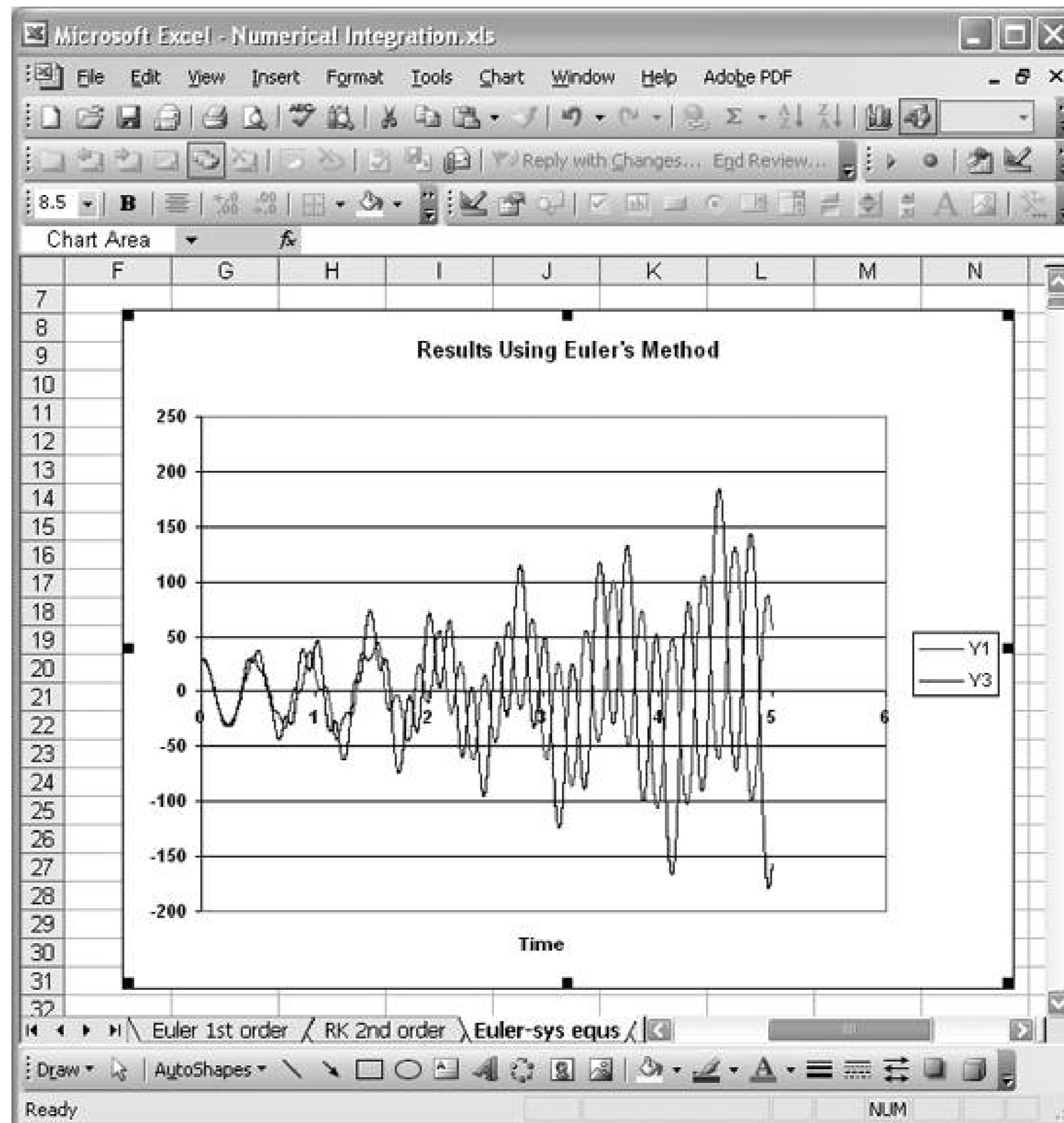
The chart shows  $y_1$  and  $y_3$  the displacements of the rod at its center and free end over time. The results should look sinusoidal. However, you can clearly see that this solution is not a sinusoid. This is wonderful example of the sort of problem where Euler's method, though easy to implement, is a poor choice.

### Runge-Kutta method

The Runge-Kutta method presents a better alternative to Euler's method for this problem. In Recipe 11. I showed you how to reduce a second-order equation to two first-order equations and then use the Runge-Kutta method to solve one and use an Euler step to solve the other. In this new example, I'll show you how to apply the Runge-Kutta method for all four equations at each time step. To this end, I prepared a new subroutine, `DoRungeKutta`, shown in Example 11-6. `DoRungeKutta` also makes calls to the derivative

functions shown earlier in Example 11-5 .

Figure 11-5. Poor results using Euler's method



Example 11-6. Runge-Kutta for nonlinear system

```
Public Sub DoRungeKutta( )
    ' Declar Local variables
    Dim y(1 To 4) As Double
    Dim k1(1 To 4) As Double
    Dim k2(1 To 4) As Double
    Dim k3(1 To 4) As Double
    Dim k4(1 To 4) As Double
    Dim n As Integer
    Dim dt As Double
    Dim time As Double
```

```

Application.ScreenUpdating = False

' Initialize variables
y(1) = 29.8613
y(2) = 0
y(3) = 30.0616
y(4) = 0
n = 5000
dt = 5 / n:
time = 0

' Perform iterations
For i = 1 To n
    k1(1) = dt * dy1dt(y(1), y(2), y(3), y(4))
    k1(2) = dt * dy2dt(y(1), y(2), y(3), y(4))
    k1(3) = dt * dy3dt(y(1), y(2), y(3), y(4))
    k1(4) = dt * dy4dt(y(1), y(2), y(3), y(4))

    k2(1) = dt * dy1dt(y(1)+k1(1)/2#, y(2)+k1(2)/2#, y(3)+k1(3)/2#, _
                y(4)+k1(4)/2#)
    k2(2) = dt * dy2dt(y(1)+k1(1)/2#, y(2)+k1(2)/2#, y(3)+k1(3)/2#, _
                y(4)+k1(4)/2#)
    k2(3) = dt * dy3dt(y(1)+k1(1)/2#, y(2)+k1(2)/2#, y(3)+k1(3)/2#, _
                y(4)+k1(4)/2#)
    k2(4) = dt * dy4dt(y(1)+k1(1)/2#, y(2)+k1(2)/2#, y(3)+k1(3)/2#, _
                y(4)+k1(4)/2#)

    k3(1) = dt * dy1dt(y(1)+k2(1)/2#, y(2)+k2(2)/2#, y(3)+k2(3)/2#, _
                y(4)+k2(4)/2#)
    k3(2) = dt * dy2dt(y(1)+k2(1)/2#, y(2)+k2(2)/2#, y(3)+k2(3)/2#, _
                y(4)+k2(4)/2#)
    k3(3) = dt * dy3dt(y(1)+k2(1)/2#, y(2)+k2(2)/2#, y(3)+k2(3)/2#, _
                y(4)+k2(4)/2#)
    k3(4) = dt * dy4dt(y(1)+k2(1)/2#, y(2)+k2(2)/2#, y(3)+k2(3)/2#, _
                y(4)+k2(4)/2#)

    k4(1) = dt * dy1dt(y(1) + k3(1), y(2) + k3(2), y(3) + k3(3), y(4) + k3(4))
    k4(2) = dt * dy2dt(y(1) + k3(1), y(2) + k3(2), y(3) + k3(3), y(4) + k3(4))
    k4(3) = dt * dy3dt(y(1) + k3(1), y(2) + k3(2), y(3) + k3(3), y(4) + k3(4))
    k4(4) = dt * dy4dt(y(1) + k3(1), y(2) + k3(2), y(3) + k3(3), y(4) + k3(4))

    y(1) = y(1) + (k1(1) + 2 * k2(1) + 2 * k3(1) + k4(1)) / 6
    y(2) = y(2) + (k1(2) + 2 * k2(2) + 2 * k3(2) + k4(2)) / 6
    y(3) = y(3) + (k1(3) + 2 * k2(3) + 2 * k3(3) + k4(3)) / 6
    y(4) = y(4) + (k1(4) + 2 * k2(4) + 2 * k3(4) + k4(4)) / 6

    time = time + dt
    ActiveSheet.Cells(i + 1, 1) = time
    ActiveSheet.Cells(i + 1, 2) = y(1)
    ActiveSheet.Cells(i + 1, 3) = y(2)
    ActiveSheet.Cells(i + 1, 4) = y(3)
    ActiveSheet.Cells(i + 1, 5) = y(4)

```



```
Next i
Application.ScreenUpdating = True
Application.ActiveWorkbook.Save

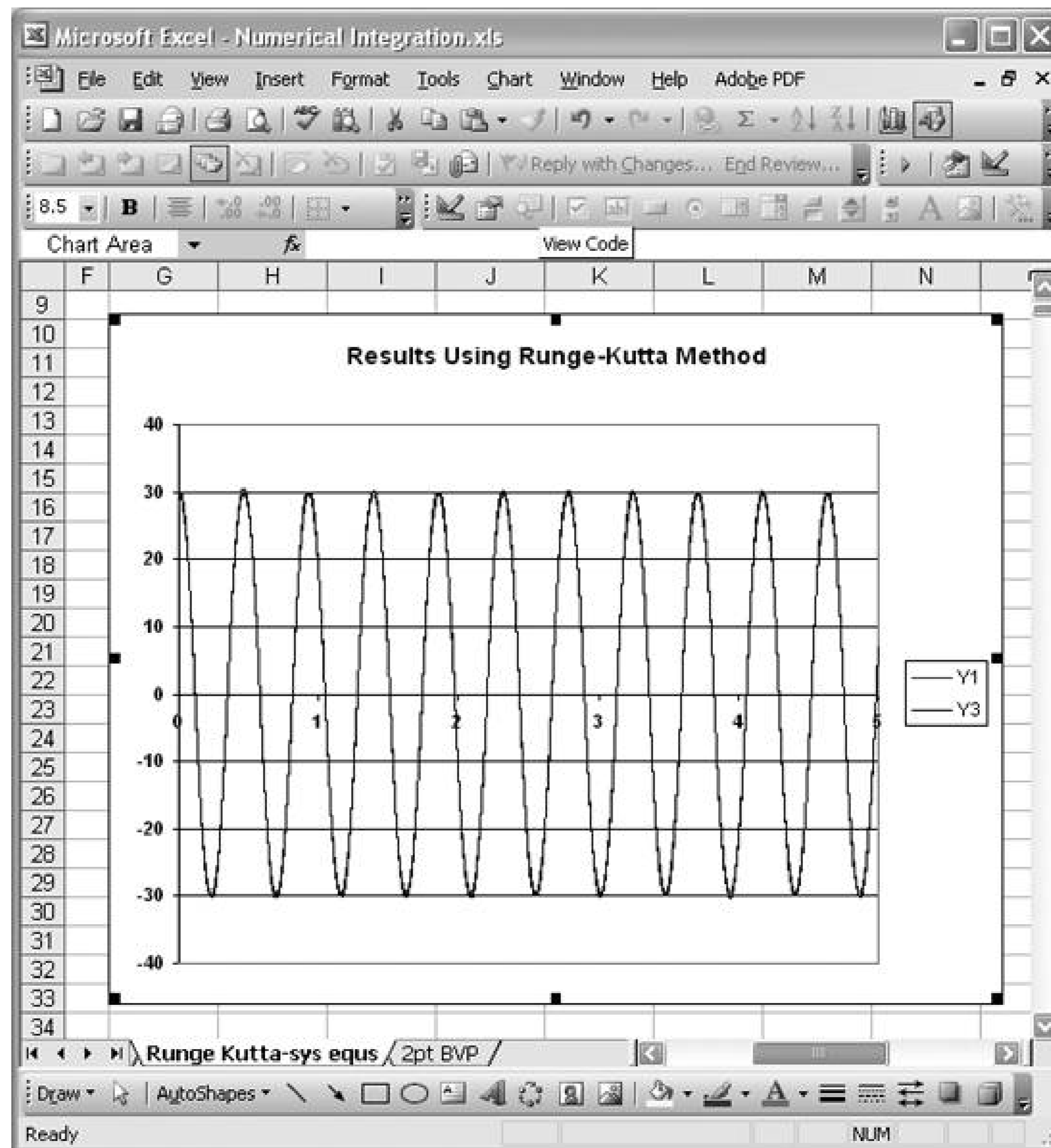
End Sub
```

The line `Dim y(1 To 4) As Double` declares a four-element array variable in which to store the computed  $y$  values at every time step. The next several declarations consist of array declarations for the Runge-Kutta  $k$ -values. Each of the four equations requires four empty  $k$ -values.

After setting the initial  $y$  conditions and initializing other utility variables, the subroutine enters a `For` loop. The guts of the `For` loop contain a straightforward application of the Runge-Kutta formulas for each equation in the system being solved. As in earlier examples, output is returned to the active spreadsheet at each time step.

Running `DoRungeKutta` gives much better results than the earlier attempt that used Euler's method. Figure 11-6 shows the new results.

Figure 11-6. Good results using the Runge-Kutta method



These results look far better than those obtained earlier. Clearly, the results shown in Figure 11-6 for  $y_1$  and  $y_3$  are sinusoidal as expected. The difference between these results and those obtained using Euler method are dramatic and effectively illustrate the relative accuracy of higher-order methods, such as the Runge-Kutta method, over Euler's basic method.

Take note of the lines of code `Application.ScreenUpdating = False` and `Application.ScreenUpdating = True` at the beginning and end of this subroutine, respectively. These lines turn screen updating off and then back on. This subroutine generates a lot of results, so I turned off screen updating to speed up the calculations. At the end, I turned it back on so the final results are displayed. The line `Application.ActiveWorkbook.Save` saves the workbook just in case I forget to do so.

[← PREV](#)

## Recipe 11.4. Shooting Boundary Value Problems

### Problem

You need to numerically solve a boundary value problem where you're given an ordinary differential equation and boundary conditions in the problem domain.

### Solution

You can use the *shooting method* to solve the boundary value problem in Excel.

### Discussion

The shooting method is a well-known iterative method for solving boundary value problems. Consider this example:

This is a second-order equation subject to two boundary conditions, or a standard *two-point boundary value problem*.

Let  $y_2 = u$  and  $y_1 = du/dt = dy_2/dt$  to reduce this second-order equation to two first-order equations:

$$\frac{dy^2}{dt} = y_1$$

$$\frac{dy_1}{dt} = 2t$$

$$y_2(0) = 0$$

$$y_2(1) = 1$$

$$0 < t < 1$$

The shooting method attempts to solve this sort of problem as an initial value problem using a marching algorithm like Euler's method or the Runge-Kutta method, as discussed earlier in this chapter. The problem is that the initial conditions are not fully specified;  $dy_1/dt$  at  $t = 0$  is unknown. This is the same as saying  $y_1(0)$  in the reduced system is unknown. Therefore, the shooting method requires you to make an initial guess at the unknown initial condition and then implement a marching algorithm over the problem domain. Once that is complete, you have to check to see if the values obtained at the end of the marching process satisfy the given boundary conditions at the end of the boundary. If they are not satisfied, you have to assume new values for the unknown initial conditions, redo the marching solution, and then check the end boundary conditions again. You repeat this process until you converge on a solution that satisfies all boundary conditions. This iterative process is ideally suited to Excel's Solver tool.

You can readily set this problem up in Excel and solve it using a combination of Solver and a method such as Euler's method. [Figure 11-7](#) shows the spreadsheet I set up to solve the reduced system of equations shown earlier.

At the top of this spreadsheet you can see where I entered the given boundary conditions into several cells. I divided these into initial conditions that will serve as initial conditions for the marching algorithm, and a boundary condition at the end of the problem domain ( $t = 1$ ). Make sure you notice that the initial condition  $y_1(0)$  is unknown. The value shown in cell D3 is an assumed initial value. We're going to use Solver to find it later.

The meat of this spreadsheet is the table you see in the middle of [Figure 11-7](#), which implements Euler's method to solve the reduced system of equations. The values at  $t = 0$  in this table contain the initial conditions. For example, the cell containing  $y_1$  at  $t = 0$  (cell D12) contains the cell formula `=D3`, referring back to the unknown initial condition. Similarly, the cell containing  $y_2$  at  $t = 0$  (cell E12) contains the cell formula `=D2` referring back to the given initial condition.

Figure 11-7. Two-point boundary value problem setup

Microsoft Excel - Numerical Integration.xls

File Edit View Insert Format Tools Data Window Help Adobe PDF

10 B

Initial Conditions:

$y_2(0) = 0$  ← given

$y_1(0) = 0$  ← unknown, iterate to find

Boundary Condition at  $t=1$ :

$y_2(1) = 1$  ← given

Euler Integration:

$dt = 0.01$

t	y1	y2
0	0	0
0.01	0.0002	0
0.02	0.0006	0.000002
0.03	0.0012	0.000008
0.04	0.002	0.00002
0.94	0.893	0.27683
0.95	0.912	0.28576
0.96	0.9312	0.29488
0.97	0.9506	0.304192
0.98	0.9702	0.313698
0.99	0.99	0.3234
1	1.01	0.3333

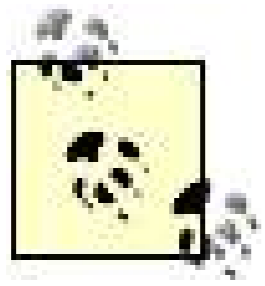
← Initial Conditions

← Must satisfy boundary condition

Runge Kutta-sys equs 2pt BVP

Ready NUM

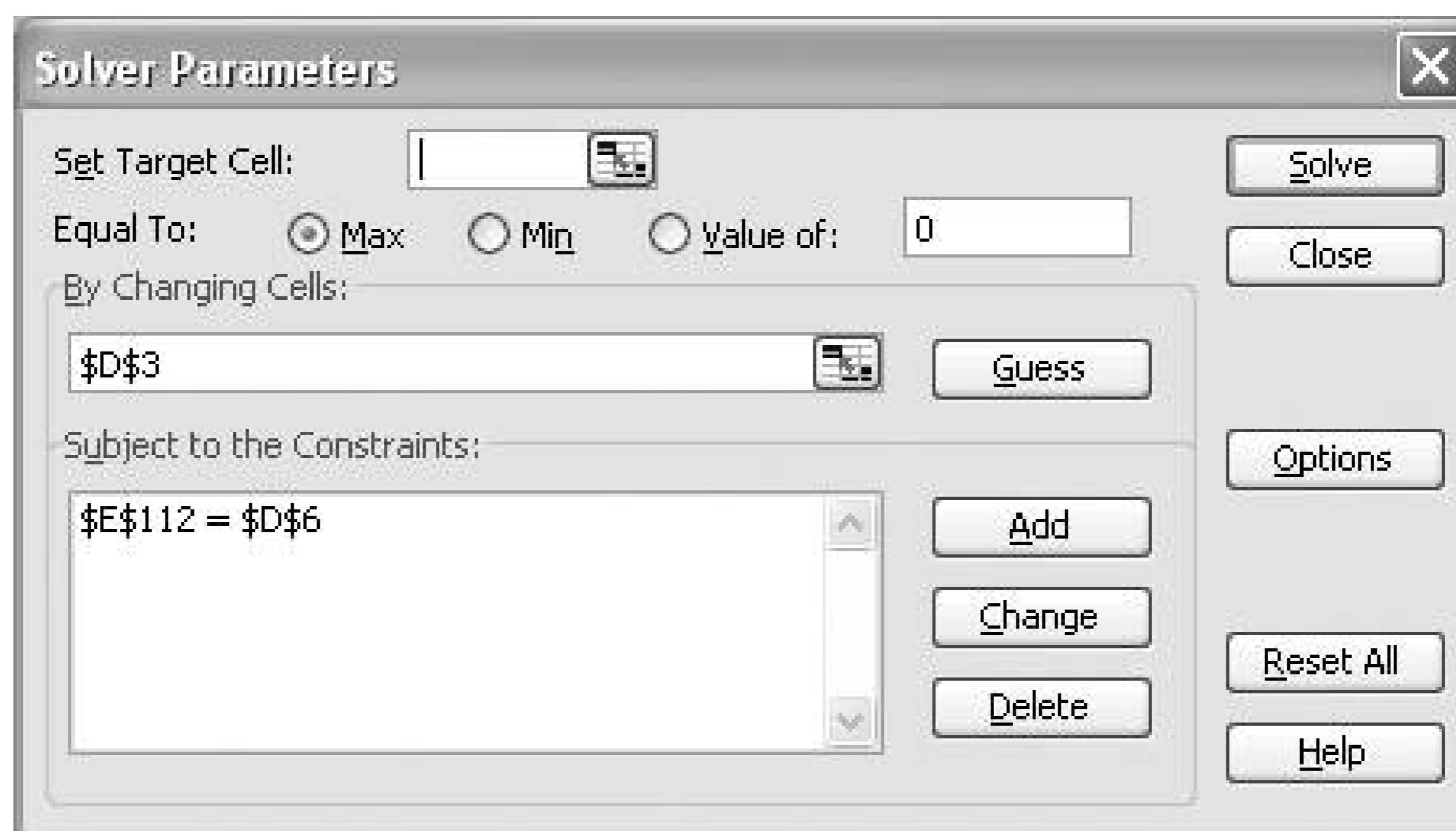
All of the other cells in this table contain formulas like  $=D12+2*C13*$D$9$  for the  $y_1$  column and  $=E12+D12*$D$9$  for the  $y_2$  column. These formulas correspond to the Euler integration formulas for this problem shown here:



The table shown in [Figure 11-7](#) is only a portion of the table I set up. You can see that the row labels skip from row 16 to row 106. What happened to the rows in between? I hid them using Format > Row > Hide from the main menu bar (select the rows you want to hide first). This way I could see the bottom end of the table without having to scroll down.

Notice in [Figure 11-7](#) that the computed value for  $y_2$  at the end of the domain,  $t = 1$ , does not satisfy the given boundary condition. This is because my assumed initial condition,  $y_1(0) = 0$ , is not the correct initial condition. However, we can use Solver to find it iteratively, subject to the constraint that the value in cell E112 must be equal to the value in cell D6 in order to satisfy the boundary conditions. [Figure 11-8](#) shows the Solver model I set up for this problem.

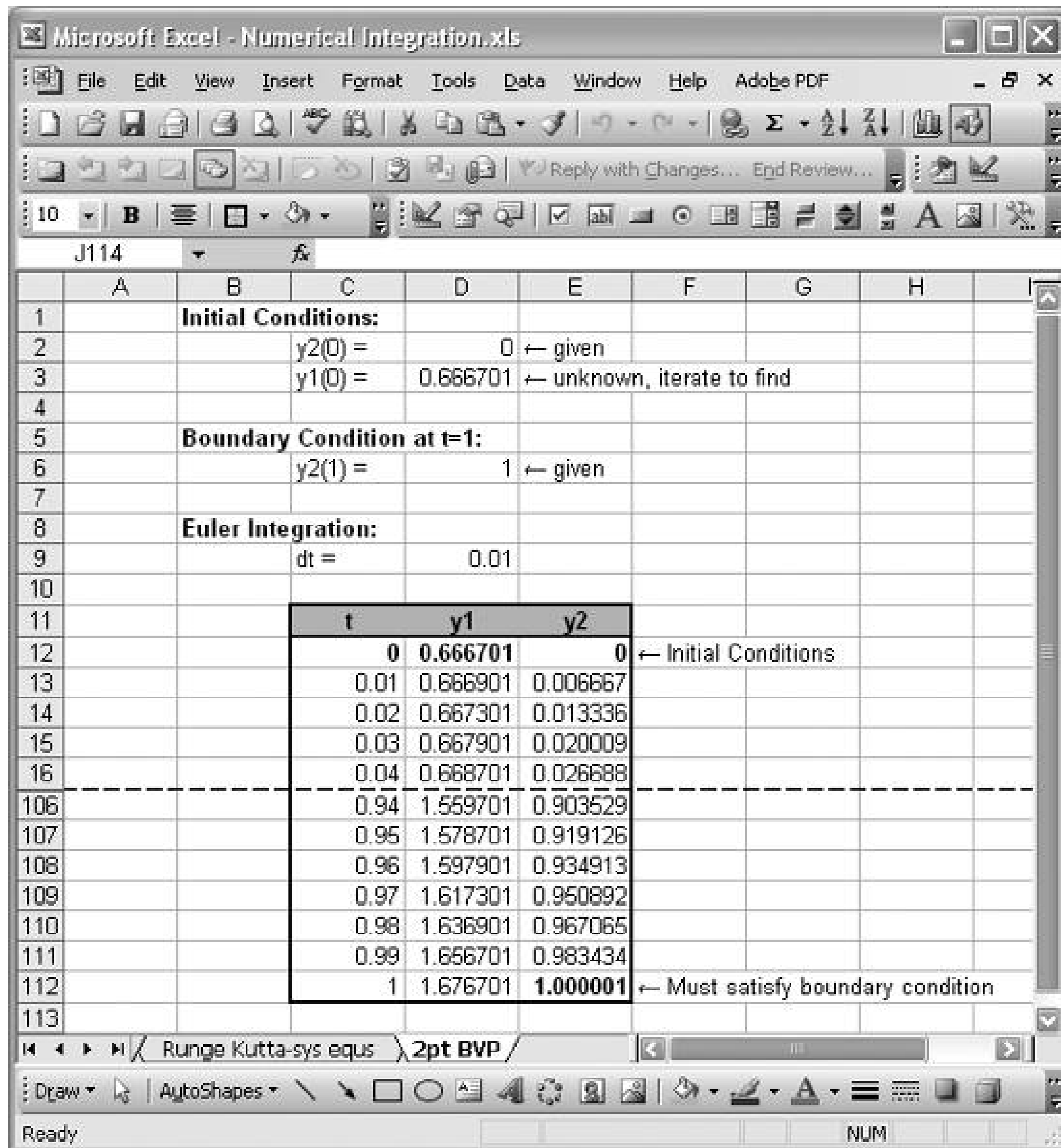
Figure 11-8. Solver model for two-point boundary value problem



Just as in [Recipe 9.4](#), I didn't specify a target cell. Instead, I specified only a cell to change subject to a single constraint. Solver will automatically set up a dummy target cell if one is not specified. The value of that target cell is irrelevant to the solution. The cell to change for this problem is the value of the unknown initial condition. The constraint consists of requiring the computed value of  $y_2$  at the end of the problem domain to equal the given boundary condition. (See [Recipe 9.4](#) to learn how to add constraints in Solver).

Setting up this Solver model and pressing the Solve button starts the iterative solution process, resulting in the solution shown in [Figure 11-9](#).

Figure 11-9. Two-point boundary value problem solution



As you can see, Solver found a solution satisfying the specified constraint. It turns out that the correct initial condition for  $y_1$  is 0.666701. At this point the problem is solved and you can plot the results if you desire. (See [Chapter 4](#) to learn how to create charts in Excel.)

← PREY

# Chapter 12. Solving Partial Differential Equations

[Section 12.0. Introduction](#)

[Recipe 12.1. Leveraging Excel to Directly Solve Finite Difference Equations](#)

[Recipe 12.2. Recruiting Solver to Iteratively Solve Finite Difference Equations](#)

[Recipe 12.3. Solving Initial Value Problems](#)

[Recipe 12.4. Using Excel to Help Solve Problems Formulated Using the Finite Element Method](#)

← PREY



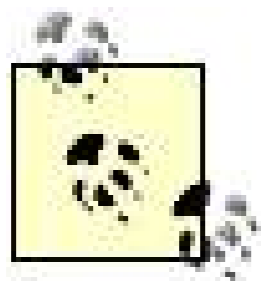
## 12.0. Introduction

Partial differential equations are incredibly important in scientific and engineering problem solving, and it's no wonder a vast array of methods have been devised to solve them. As in the case of ordinary differential equations, you can divide problems involving partial differential equations into two broad classes: *boundary value problems* and *initial value problems* (including initial-boundary value problems).

The best examples for boundary value problems are elliptic equations of the form:

$$u_{xx} + u_{yy} = 0$$

$$u_{xx} + u_{yy} = f(x, y)$$



I'm using the standard subscript notation here to indicate partial derivatives, e.g.,  $u_x$  represents the first partial derivative of  $u$  with respect to  $x$ .

The first equation shown above is the well-known *Laplace* equation, while the second is the *Poisson* equation. Boundary conditions are required for these equations and can consist of specification of  $u$ , derivatives of  $u$ , or a mix of these two on the problem boundary, corresponding to *Dirichlet*, *Neumann*, and *mixed* conditions, respectively. The numerical solution to boundary value problems of this form generally consists of discretizing the problem domain using one of a variety of techniques, such as the *finite difference method*, *finite element method*, or *boundary element method*, and formulating a system of algebraic equations that can be solved for the unknown values of  $u$  at discrete nodal points inside the problem domain or boundary.

Time does not enter into elliptic problems, so the solution sought is a *steady state* or *static* solution. For example, a fluid flow problem can be formulated as a steady state problem, or a structural problem can be formulated as a static problem. The resulting system of equations may or may not be linear and, depending on the size of the problem and how it's discretized, could be a relatively small system of equations or a very large one. These factors combine to guide selection of an appropriate solution algorithm. Small linear systems can be solved directly, but nonlinear or large systems might require an iterative approach.

Time-dependent problems are typically of the form:

The first equation is the one-dimensional *diffusion equation*, and the second is the one-dimensional *wave equation*. These equations can, of course, apply to higher-dimensional problems as well, with

the addition of suitable terms. In terms of standard mathematical classifications, the diffusion equation represents a *parabolic* equation, while the wave equation represents a *hyperbolic* equation.

An important distinction between these equations and the previously mentioned elliptic equations is the dependence on time in the case of parabolic and hyperbolic equations. This has implications with regard to the method of solution. In time-dependent problems, marching or stepping algorithms can be used to propagate solutions in time, just as you saw [Chapter 11](#) for time-dependent ordinary differential equations. Further, in addition to boundary conditions of the types mentioned earlier, you also have to specify initial conditions for time-dependent problems.

Partial differential equations apply to an incredible variety of problems, from describing physical processes to modeling economic processes, and all sorts of problems in between. For many problems exact solutions are difficult if not impossible to find, and numerical methods play an important role in their solution. Given the ready availability of computing power these days, numerical methods for partial differential equations are enjoying much attention and rapid development. It seems the number and variety of methods is almost as varied as are the problems being solved.

In general, Excel will do nothing to formulate a solution to any given partial differential equation. You have to do the math yourself. However, once you have formulated a solution in the form of a nonlinear equation or system of linear or nonlinear equations, either static or time-dependent, you can leverage Excel to help solve these equations. In this regard, the recipes in this chapter provide several examples of how you might use Excel to solve partial differential equations. These are by no means all-inclusive, but I hope they provide some inspiration for taking advantage of Excel's features to help solve some complicated problems.

## Recipe 12.1. Leveraging Excel to Directly Solve Finite Difference Equations

### Problem

You've formulated the finite difference equations for an elliptic boundary value problem and now you need to solve the resulting system of equations.

### Solution

Use Excel's matrix functions to solve the system of finite difference equations.

### Discussion

The finite difference method is a well-known and well-established numerical method suitable for solving partial differential equations. Virtually every book I've ever read dealing with partial differential equations covers the finite difference method to some degree. DuChateau and Zachmann give a pretty good introduction, with examples, on the application of the finite difference method to partial differential equations in their book *Applied Partial Differential Equations*<sup>[\*]</sup>

[\*] Paul DuChateau and David Zachmann, *Applied Partial Differential Equations* (Mineola, N.Y.: Dover Publications, 2002; originally published 1989 by Harper & Row).

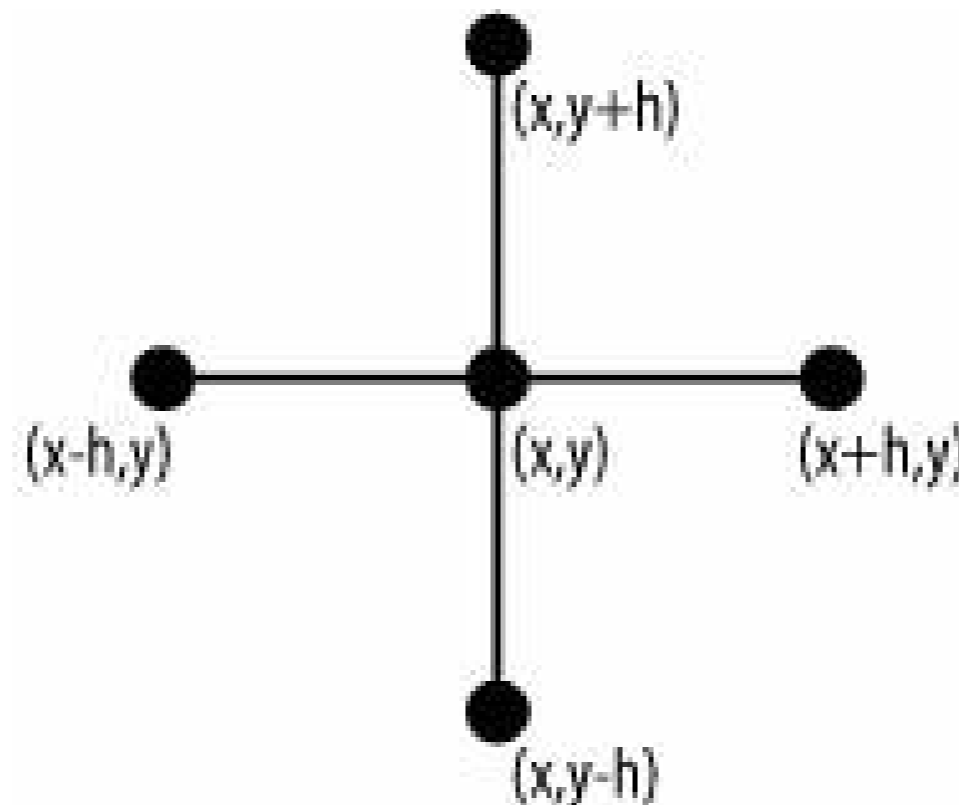
The basic idea of the finite difference method is to divide the problem domain into a finite number of cells, forming a grid. The intersections of grid lines then represent computational nodes where the value of the function sought will be approximated. The governing differential equation is then written in terms of finite differences (see [Recipe 10.6](#)). These finite differences could be forward differences, backward differences, or central differences, and they are used to approximate the derivatives appearing in the governing partial differential equations at each node in the computational grid.

Consider the Laplace equation :

If we discretize the problem domain as describe a moment ago, then at each node illustrated in [Figure 12-1](#), we can approximate the second partial derivative of  $u$  with respect to  $x$  using the following finite difference approximation:

$$u_{xx}(x, y) = \frac{1}{h^2} [u(x+h, y) - 2u(x, y) + u(x-h, y)]$$

Figure 12-1. Finite difference nodes



Similarly, the finite difference approximation for  $u_{yy}$  is:

Here, I'm assuming the node spacing is uniform and the same in both the  $x$  and  $y$  directions. Substitution of these finite difference approximations into the Laplace equation yields the following finite difference equation:

This equation must be satisfied at every node. Nodes adjacent to boundary nodes with Dirichlet boundary conditions will include the specified value at the boundary node. Typically, a ghost particle technique is used in the case of Neumann boundary conditions. In any case, writing this finite difference approximation at each node in the computation grid results in a system of equations that must be solved to find the  $u$ -values at each node. It's this resulting system of equations that can be solved in Excel.

For example, a rectangular domain divided into nine equal-sized cells would contain four interior nodes where values of  $u$  are sought. Applying the above finite difference equation at each node would result in a system of four equations and four unknowns. In this case, with such a small number of equations, you can use matrix formulas discussed in [Recipe 9.4](#) to solve for the unknown  $u$ -values.

Finer computational grids, or meshes, result in a larger number of nodes and thus a larger system of equations. In these cases, it's usually more efficient and easier to use iterative methods to solve the system. [Recipe 12.2](#) explains such an approach.

## See Also

The body of literature covering finite difference methods is vast. Consult your favorite numerical

methods book for more information. I found the *Applied Partial Differential Equations* book cited earlier in this recipe very readable and comprehensive in its coverage of the finite difference method to elliptic, parabolic, and hyperbolic problems.



## Recipe 12.2. Recruiting Solver to Iteratively Solve Finite Difference Equations

### Problem

You've set up a finite difference solution to an elliptic boundary value problem, but instead of directly solving a system of algebraic equations, you want to use an iterative approach.

### Solution

Set up your finite difference equations in Excel and use Solver to iteratively find a solution.

### Discussion

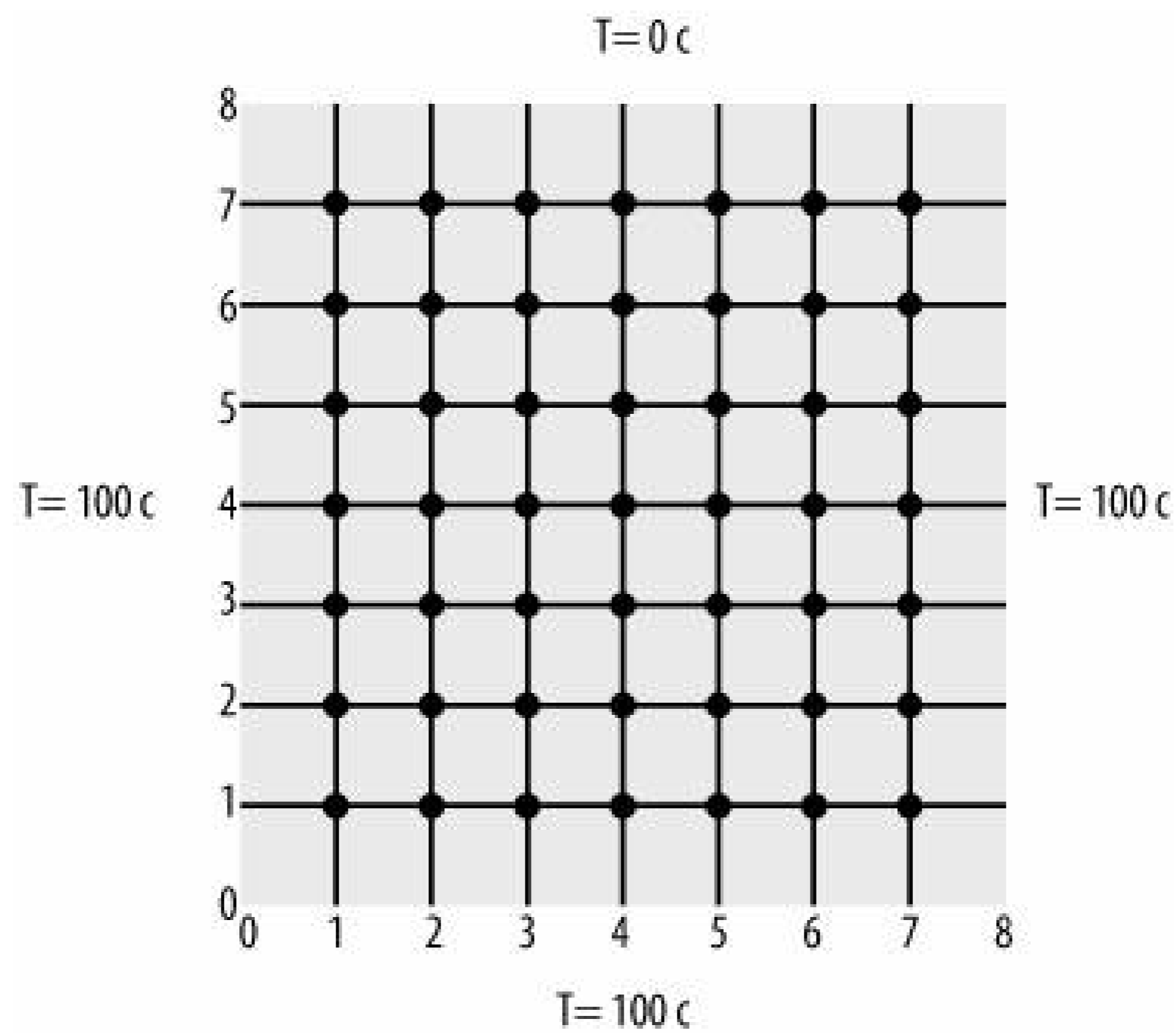
In [Recipe 12.1](#), I explained how you can use Excel's matrix functions to directly solve systems of finite difference equations. For small systems, direct matrix operations are viable; however, for large systems the use of matrix functions in Excel is cumbersome. Other standard direct methods include *Gauss elimination* and *Fourier's method*, among others. These methods could be programmed in Excel; however, there's another class of methods known as *iterative methods* that are more attractive.

Iterative solutions to finite difference equations abound. Various standard methods such as *Jacobi's method*, *Gauss-Seidel iteration* (also called *Liebmann's method*), and the *alternating direction implicit (ADI) method* are commonly used in lieu of direct methods. Iterative methods have an advantage over direct methods in that they typically require less memory and are relatively easy to program. Moreover, when using Excel to help solve your problem, you already have access to a powerful iterative tool, Solver. This means that you can use Solver to find a solution to your finite difference equations without any programming beyond setting up a spreadsheet.

By way of example, consider the steady state temperature distribution in a square plate. The left, right, and bottom edges of the plate are maintained at a constant 100°C, whereas the top of the plate is maintained at a constant 0°C. The temperature distribution in the interior of the plate is governed by the Laplace equation shown earlier, where  $u$  represents temperature. The Laplace equation is repeated here for convenience:

A finite difference mesh for this problem might look something like that shown in [Figure 12-2](#).

Figure 12-2. Finite difference grid for temperature distribution in square plate



Each little circle shown in the grid in [Figure 12-2](#) represents a computational node where the following finite difference equation, from the previous recipe, applies:

In this example, I'm assuming the grid spacing,  $h$ , is uniform in both  $x$  and  $y$  directions and equal to 1 cm.

If you rearrange this finite difference equation, solving for  $u(x, y)$ , you get the following:

You can see that  $u$  (the temperature) at each node is simply the average of the temperatures of adjacent nodes. For nodes adjacent to the plate boundary, the specified boundary conditions are included in the average.

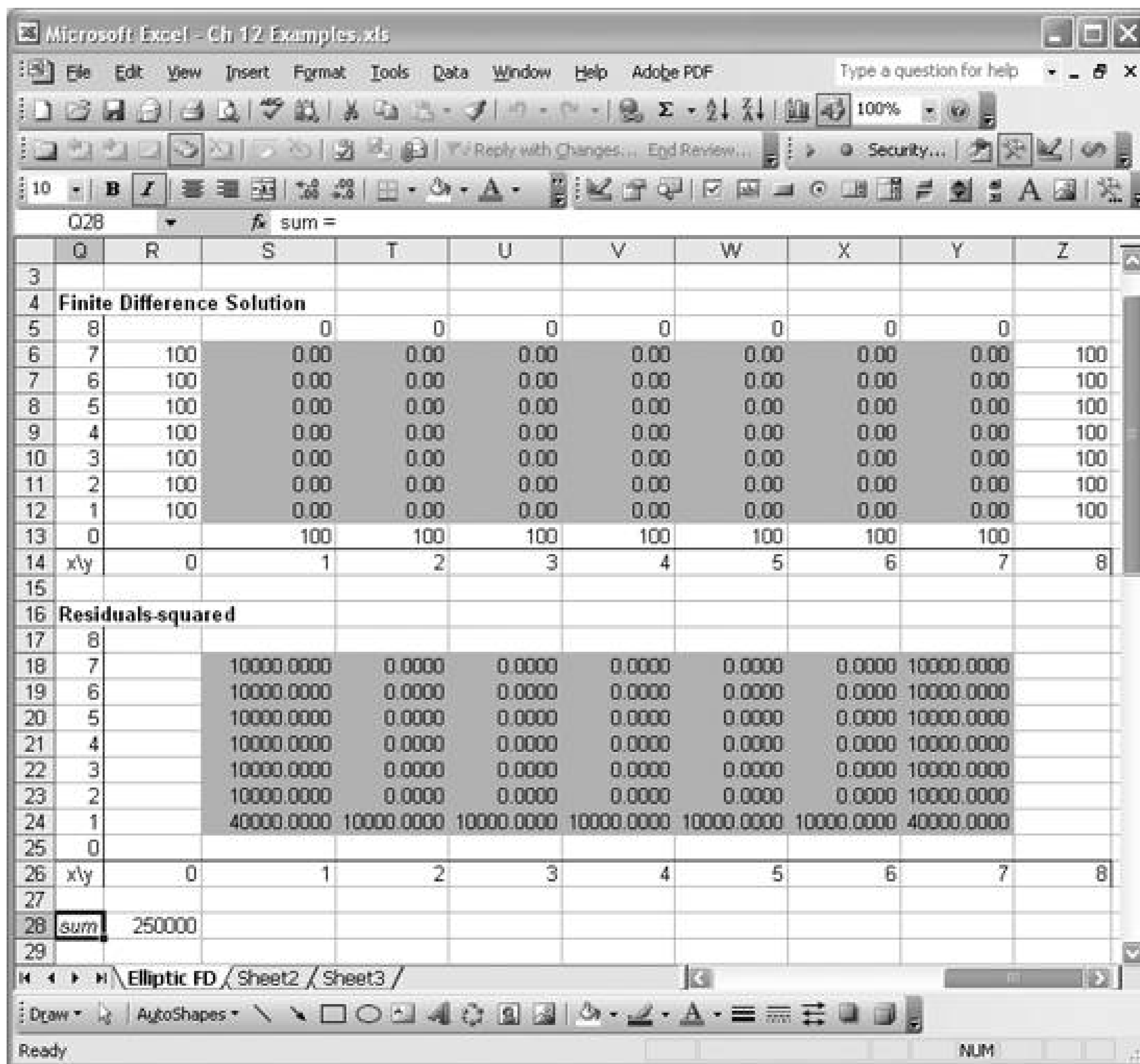
This example results in 49 finite difference equations with 49 unknown temperatures. A temperature distribution is sought that satisfies the above equation at each node.

Another way to look at this problem is to reconsider this form of the finite difference equation:

This is the same equation shown earlier, prior to solving for  $u(x, y)$ . This form of the equation

represents the residual at each node that is, the temperature minus the estimated temperature at each node. This residual must ideally be 0. If you take all these residuals, one for each node, and square them and then sum the squares, you end up with a least-squares minimization problem. You can use Solver to minimize the sum of squared residuals to arrive at a solution that satisfies the governing equations. [Figure 12-3](#) shows the spreadsheet I set up to solve this problem in Excel.

Figure 12-3. Finite difference spreadsheet setup



The table under the heading **Finite Difference Solution** represents the temperature distribution satisfying the governing equations and boundary conditions. At the moment, the table is filled with initial guesses except at the boundaries, where the given boundary conditions are applied. Each of the shaded cells in this table corresponds to a computational node in the finite difference grid illustrated in [Figure 12-2](#). The nonshaded cells bounding the shaded cells correspond to boundary nodes with specified boundary conditions.

The lower table in [Figure 12-3](#), under the heading **Residuals-squared**, contains a similar set of shaded

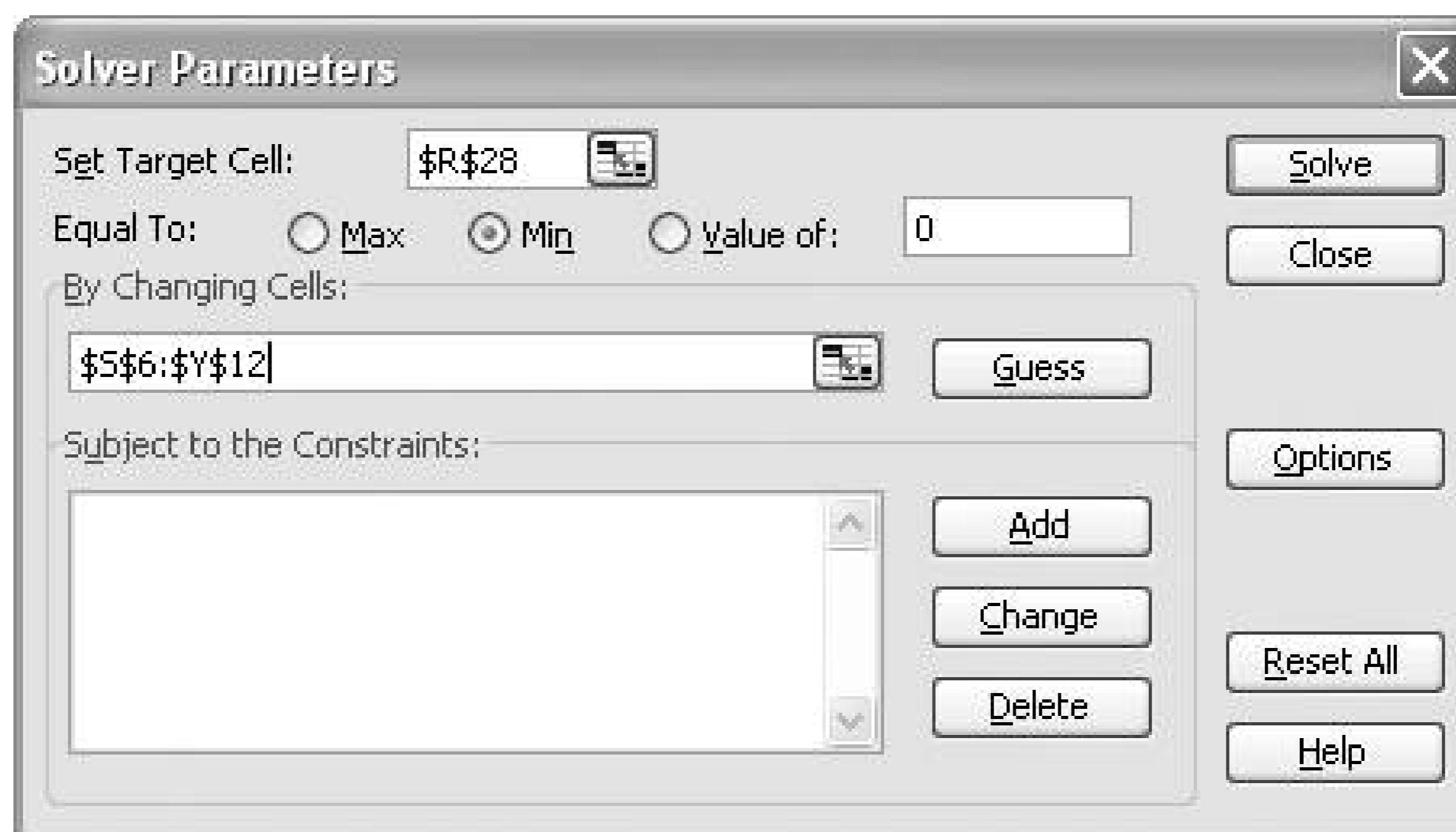


cells. In this case, the shaded cells contain the finite difference equation for each node, squared. The cell formulas are of the following form:  $=(-4*S6+S5+S7+R6+T6)^2$ . This is the cell formula for cell S18. All the other cells contain similar formulas with relative cell references. This form is essentially the square of the residual for each node. Note that these formulas refer to the values for each node contained in the upper table.

Cell R28 contains the formula  $=SUM(S18:Y24)$ . This is simply the sum of all squared residuals contained in the lower table.

Now you can use Solver to minimize the value in cell R28 by changing all the values contained in the shaded cells in the upper table. [Figure 12-4](#) shows the Solver model I used for this example.

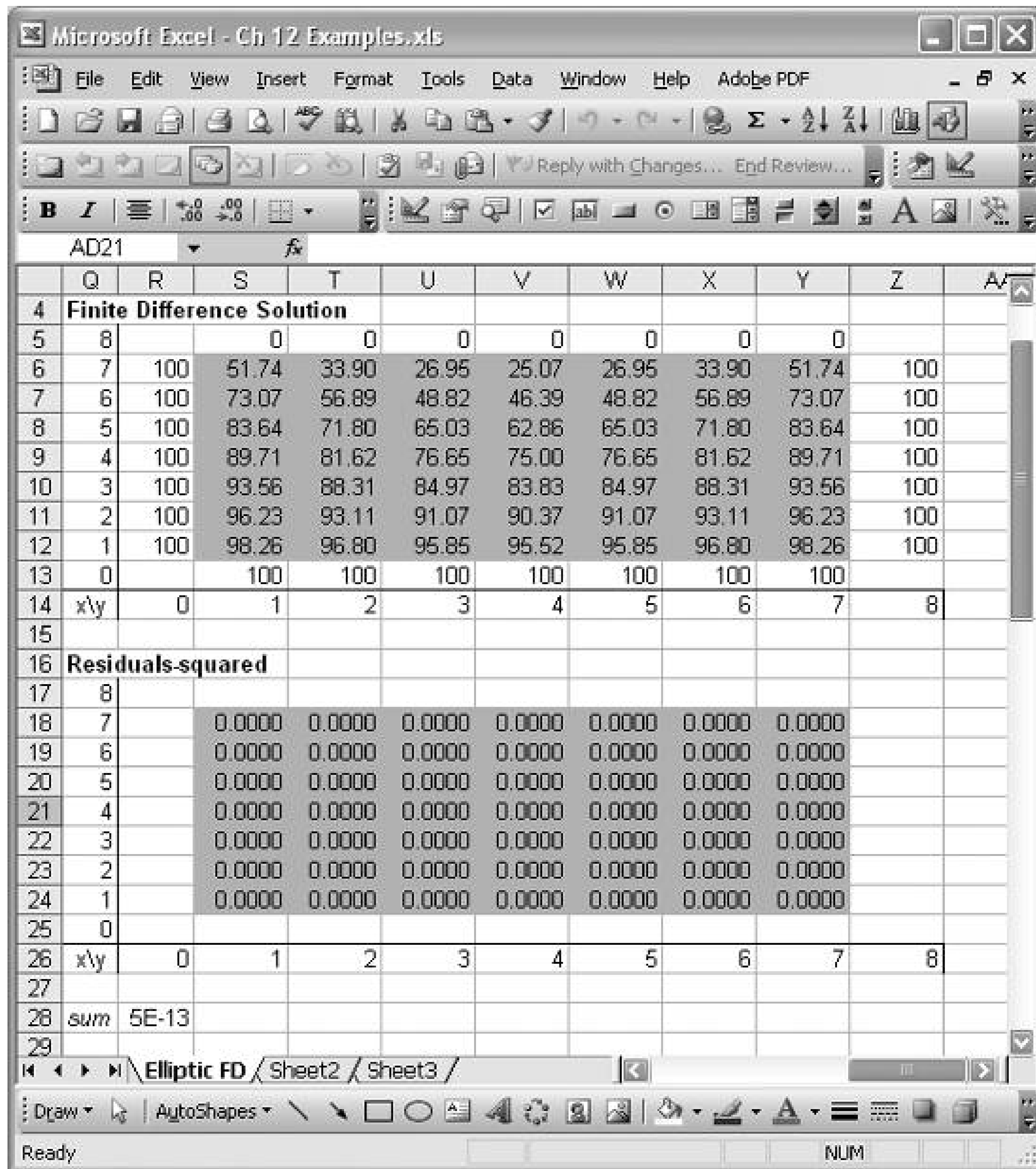
Figure 12-4. Solver model for finite difference solution



You can see that this model aims to minimize the value in cell R28, the sum of squared residuals, by changing all the values contained in cells S6 to Y12. If Solver is successful, cells S6 to Y12 in the upper table in [Figure 12-3](#) will contain a temperature distribution that satisfies the governing equations and boundary conditions.

After you press the Solve button, Solver does indeed find a solution, as shown in [Figure 12-5](#).

Figure 12-5. Solution to finite difference problem

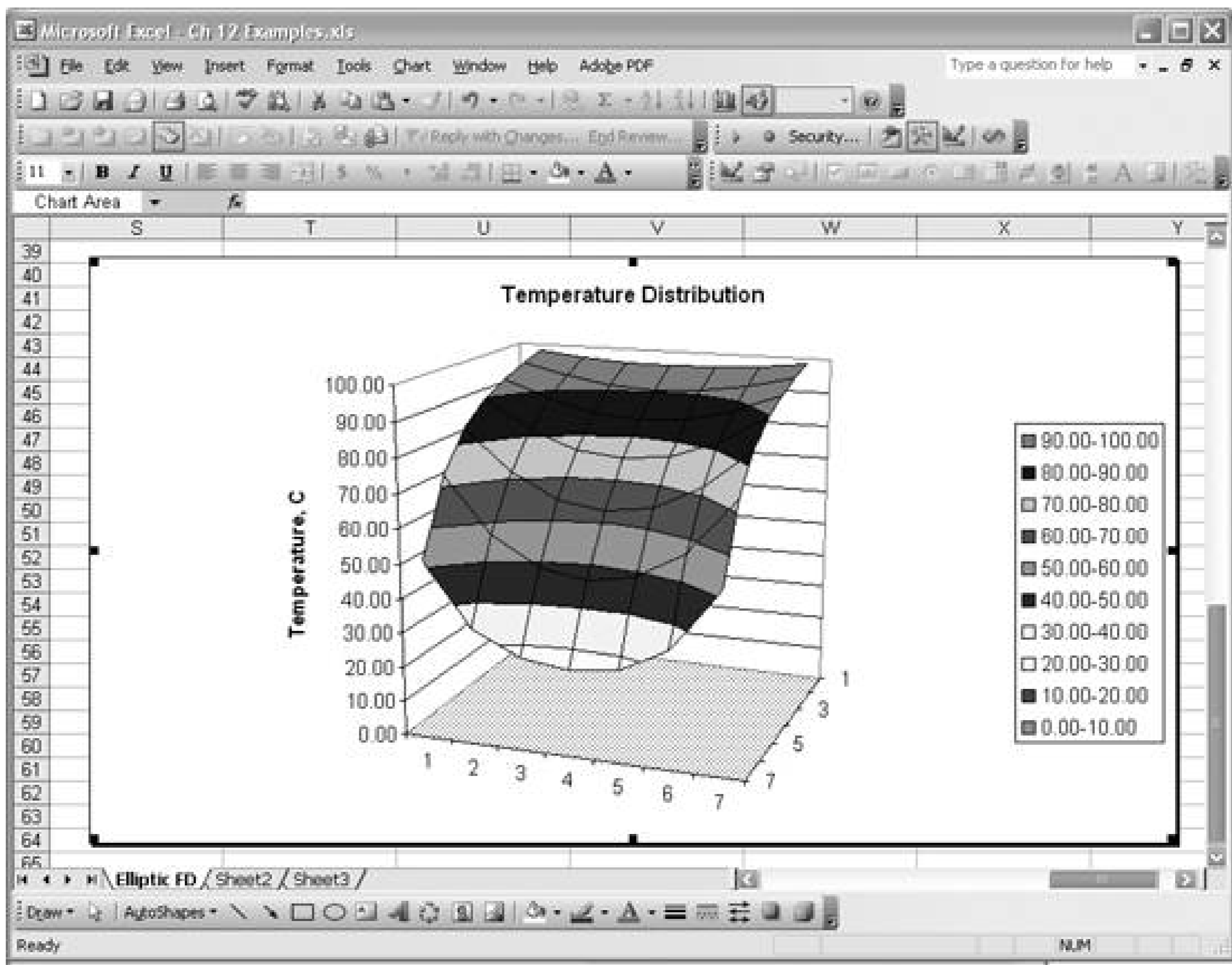


As you can see, the sum of squared residuals is a very small number and the squared residuals for each cell are all 0 (to 14 decimal places). The upper table now contains the solution to the temperature distribution problem.

Just for fun, I plotted these results using Excel's 3-D Surface chart feature (see [Chapter 4](#)). [Figure 12-6](#) shows the resulting plot.

This highlights another advantage of using Excel: you can quickly visualize your results without having to program your own visualizations or use a separate visualization tool.

Figure 12-6. Surface plot of temperature distribution



## Recipe 12.3. Solving Initial Value Problems

### Problem

You're dealing with a parabolic or hyperbolic problem and have developed the finite difference equations for the problem, which includes both discretized spatial and time variables. Now you have to solve these equations.

### Solution

Use Excel and Solver to solve the finite difference equations in a manner similar to that discussed in [Recipe 12.2](#).

### Discussion

In the previous recipe, I showed you how to leverage Solver to solve the finite difference equations, arriving at a steady state solution to an elliptic-type boundary value problem. You can apply the same techniques to solve time-dependent parabolic or hyperbolic types of problems.

Consider the time-dependent one -dimensional heat equation:

This is a parabolic equation, which can be used to model the time-dependent heat conduction in a metal rod, for example, subject to some prescribed initial temperature distribution and boundary conditions at the ends of the rod.

Let's assume we have a 1m rod that's insulated all around except at the ends. The ends of the rod are maintained at a constant 10°C and the initial temperature distribution along the length of the rod (at time  $t = 0$ ) is as shown in [Figure 12-7](#), row 13. Further assume that  $c^2$  equals 1. Now we can use the finite difference method to solve for the temperature distribution along the length of the rod over time.

Erwin Kreyszig solves a similar problem in *Advanced Engineering Mathematics*.<sup>[\*]</sup> Kreyszig uses the *Crank-Nicolson* finite difference method, where the following finite difference equation applies at each computational node in the finite difference grid:

[\*] Erwin Kreyszig, *Advanced Engineering Mathematics*, 6th ed. (New York: Wiley, 1988).

In this method, both spatial and time variables are discretized. You can set up a finite difference grid similar to that shown in [Figure 12-2](#), where instead of the vertical axis representing the  $y$ -dimension, the vertical axis will now represent time. The horizontal axis still represents the  $x$ -dimension, the distance along the rod's length in this case. The finite difference equation shown here assumes that the time step size,  $\Delta t$ , is related to the  $x$  step size,  $\Delta x$ , such that  $\Delta t = \Delta x^2$ .

You can solve this problem in a manner similar to that discussed in [Recipe 12.2](#), where I showed you how to solve a boundary value problem using Solver. [Figure 12-7](#) shows the spreadsheet I set up to solve this initial value problem.

The upper table, *Finite Difference Solution*, initially contains only specified boundary and initial values (nonshaded cells) and initial guesses for the unknown temperature values along the length of the rod (shaded cells). We're going to let Solver alter the values in the shaded cells in the upper table to find a solution.

The shaded cells in the lower table, *Residuals-squared*, contain the finite difference equation shown earlier for each node. Actually, the square of the lefthand side, the residual, is computed; the idea is to minimize these residuals as discussed earlier. For example, the formula in cell H22 is  $= (4 * H8 - I8 - G8 - I9 - G9)^2$ . All the other cell formulas are similar. Notice that these formulas refer to values contained in the upper table.

Cell D29 contains the sum of all these residual cells. We're going to have Solver attempt to minimize the value computed in cell D29. The Solver model I used for this example is shown in [Figure 12-8](#).

Figure 12-7. Spreadsheet for solving initial value problem

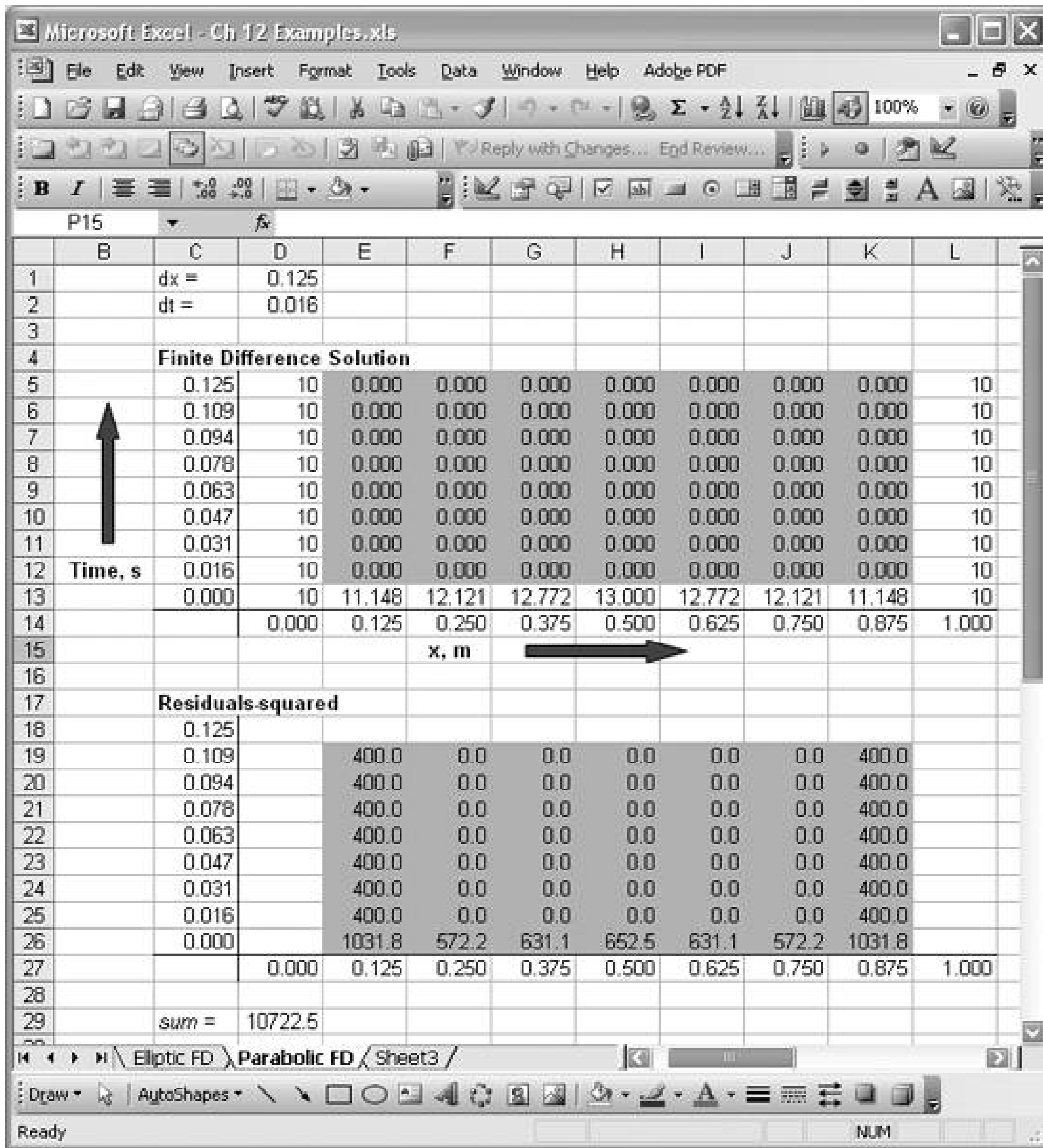
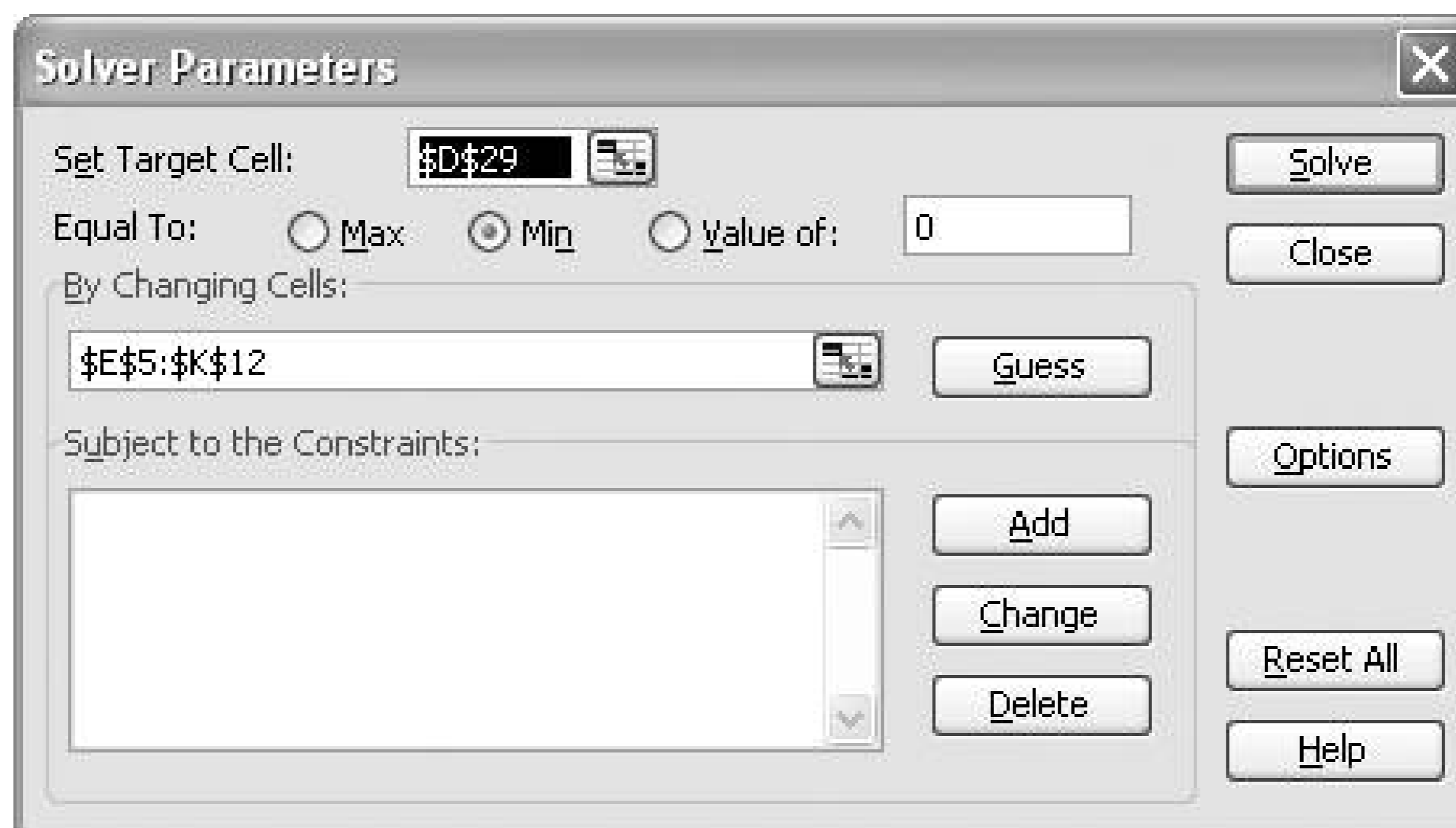
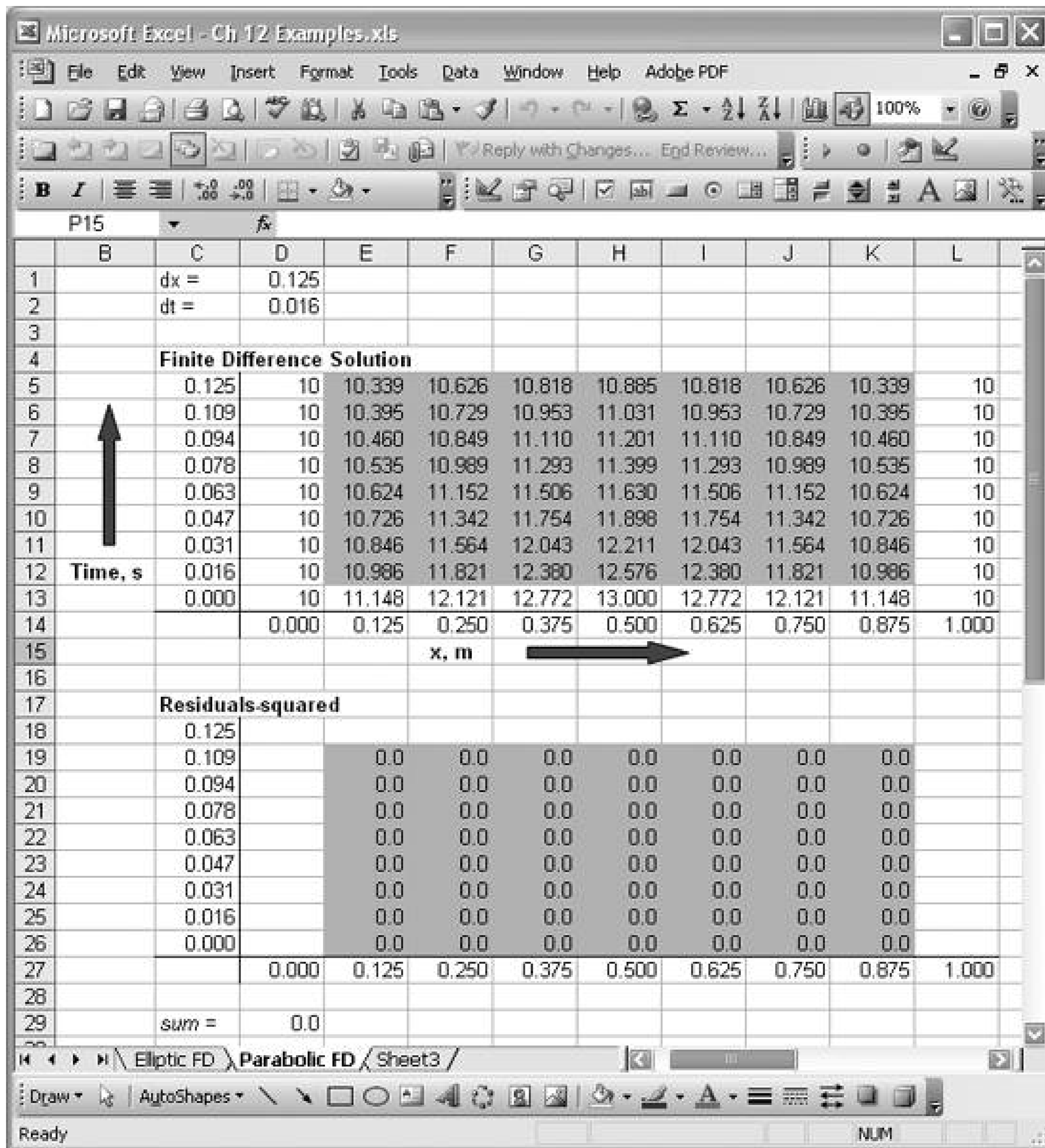


Figure 12-8. Solver model for initial value problem



I set the target cell to D29 so as to minimize the sum of squared residuals. The cells to change are all the shaded cells in the upper table of [Figure 12-7](#). The corresponding cell range is E5 to K12. Pressing the Solve button sets Solver going on its search for a solution. After a few seconds, Solver presents the results shown in [Figure 12-9](#).

Figure 12-9. Final solution to initial value problem

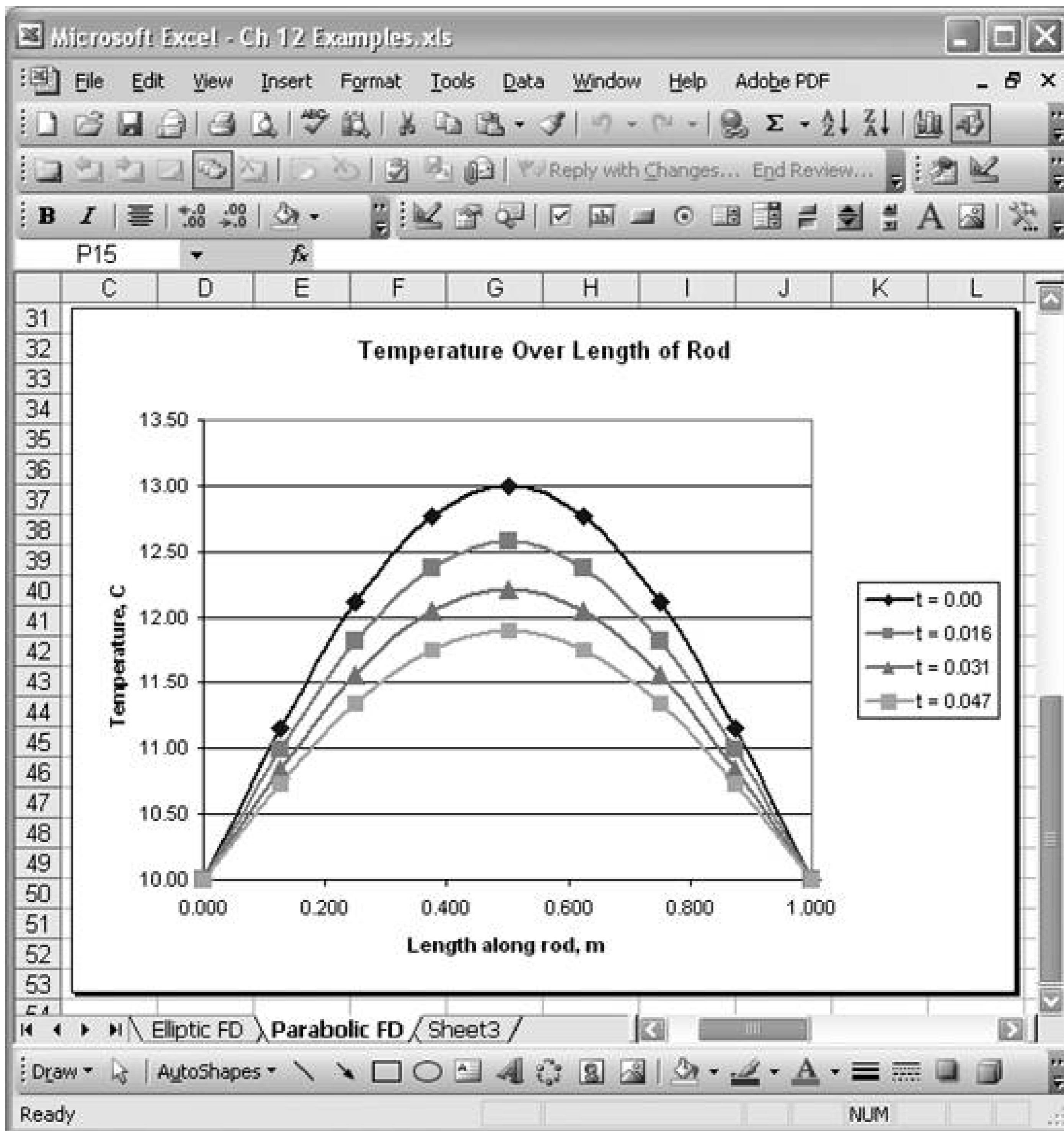


As you can see, all the residuals are 0 and the upper table now contains the solution to this initial value problem. [Figure 12-10](#) shows a plot of these results.

As I mentioned in the previous recipe, the ability to quickly visualize your results in Excel without having to use a separate visualization tool is another compelling reason to use Excel for this sort of problem when possible. Refer to [Chapter 4](#) to learn more about creating charts in Excel.

Figure 12-10. Plot of initial value problem results





## Recipe 12.4. Using Excel to Help Solve Problems Formulated Using the Finite Element Method

### Problem

You've applied the finite element method to a nonlinear problem and must now iteratively find a solution.

### Solution

Use Solver to find a solution to the resulting nonlinear equation or system of equations. See [Chapter 9](#) for recipes showing how to use Solver.

### Discussion

The finite element method is a very popular method for numerically solving differential equations. The method involves discretizing the problem domain and governing equations, resulting in an algebraic system of equations that must be solved to arrive at an approximate solution to the problem. In some cases, the problem is nonlinear, which results in a nonlinear equation, or system of equations, that must be solved. As discussed in [Chapter 9](#), Excel's built-in Solver tool is very adept at finding solutions to such equations.

By way of example, consider a heat conduction problem presented by Kythe and Wei in their *Introduction to Linear and Nonlinear Finite Element Analysis*<sup>[\*]</sup> The governing partial differential equation, with boundary conditions, is as follows:

[\*] Prem Kythe and Dongming Wei, *An Introduction to Linear and Nonlinear Finite Element Analysis* (Boston: Birkhauser, 2004). See pages 244 and 245 for a complete derivation using the Galerkin method.

This equation represents steady state heat conduction in a laterally insulated rod with constant temperature maintained at one end ( $x = 0$ ) and with radiation heat transfer at the other end ( $x = L$ ).  $k$  is the thermal conductivity of the rod material.  $\sigma$  is the Stefan-Boltzman constant.  $T_0$  is the

emissivity constant. Kythe and Wei apply the Galerkin method with linear shape functions to derive a solution for this problem. The resulting nonlinear equation to be solved is:

$$0 = \frac{-\sigma \epsilon}{k} (T_L^4 - T_\infty^4)L + T_0 - T_L$$

$T_L$  represents the unknown temperature at the end of the rod. All other parameters in this equation are given. The solution to this equation was actually assigned as a homework problem in a graduate course I took with Dr. Wei. I used Excel's Solver tool to find a solution to this problem. [Figure 12-11](#) shows the spreadsheet I prepared.

Cells C3 through C10 contain all of the given data for this problem. Cell C11 contains the unknown temperature at the end of the rod that must be found iteratively. Cell C12 contains the righthand side of the nonlinear equation shown earlier. In [Figure 12-11](#), cell C12 is selected so that you can see the cell formula in Excel's formula bar. The formula is  $=-(C9*C10)/C6*(C11^4 - C8^4)*C5+C7-C11$ .

Figure 12-11. Solving for  $T_L$  using Solver

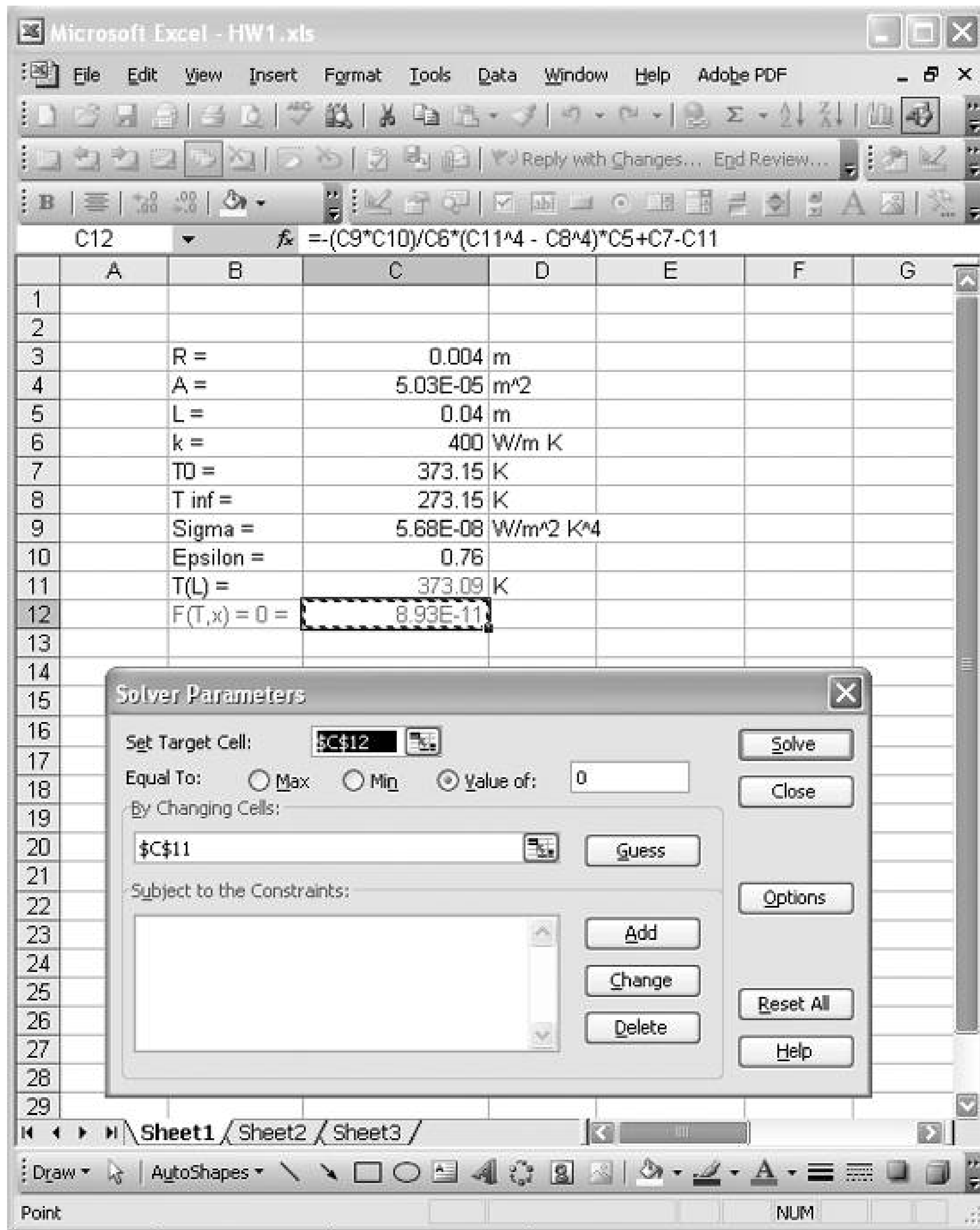


Figure 12-11 also shows the Solver window overlaid on the spreadsheet. The Solver model for this problem is relatively simple. Basically, you want to set the value of cell C12 the one containing the nonlinear equation to 0 by changing the value in cell C11, which contains the temperature at the end of the rod. Setting up this model and pressing the Solve button in Solver yields the solution shown in Figure 12-11. That is, the temperature at the end of the rod is approximately 373.09 K.

## See Also

Some approaches to formulating finite element solutions, for example the *Rayleigh-Ritz* approach, result in a minimization problem to find an approximate solution. In these cases, once the minimization problem has been formulated, you can use Excel's Solver tool. See [Chapter 9](#) for more information on using Solver.

Solutions to many problems formulated using the finite element approach (particularly linear ones) result in an algebraic system of equations that must be solved. Here again, you can use techniques discussed in [Chapter 9](#) to solve these systems.

Some finite element solution formulations require you to solve a system of ordinary differential equations in order to find an approximate solution to the original partial differential equation. Such systems can be solved in Excel. See [Recipe 11.3](#) for an example.



# Chapter 13. Performing Optimization Analyses in Excel

[Section 13.0. Introduction](#)

[Recipe 13.1. Using Excel for Traditional Linear Programming](#)

[Recipe 13.2. Exploring Resource Allocation Optimization Problems](#)

[Recipe 13.3. Getting More Realistic Results with Integer Constraints](#)

[Recipe 13.4. Tackling Troublesome Problems](#)

[Recipe 13.5. Optimizing Engineering Design Problems](#)

[Recipe 13.6. Understanding Solver Reports](#)

[Recipe 13.7. Programming a Genetic Algorithm for Optimization](#)

## 13.0. Introduction

Optimization problems come up frequently. A businessman may try to optimize profit given production performance from several different production facilities. An engineering manager may try to optimize his labor force to meet forecast demands. Engineers often try to optimize design problems to produce optimum performing designs at minimum cost or perhaps minimum weight. The variety of applications is incredible.

By now you're familiar with Solver; we've used it for least-squares curve fitting and for solving nonlinear equations and systems of equations. In reality, Solver is an optimization tool that has been very nicely integrated with Excel. You can use Solver to handle many different types of optimization problems directly within Excel. You can use Solver for *linear programming* types of problems and for nonlinear optimization problems (sometimes called *dynamic programming*). I use Solver very often for engineering design optimization and some business optimization problems. While Solver is very capable, it isn't your only option, and sometimes it's not necessarily the best option.

In this chapter, I will show you how to leverage Excel's built-in features, like Solver, to solve several typical types of optimization problems. Further, I will show you how you can use Excel and VBA to program a nontraditional algorithm (a genetic algorithm) to solve optimization problems.

## Recipe 13.1. Using Excel for Traditional Linear Programming

### Problem

You've formulated an optimization problem in traditional linear programming form and would like to use Excel to solve the problem.

### Solution

Use Solver's linear optimization capabilities.

### Discussion

Linear optimization problems can be written in the form of an objective function to maximize (or minimize) subject to constraints. Constraints can be written in the form of inequalities or equalities. Consider this simple example:

*Objective function*

*Constraints*

If this were a *product mix* type of problem, the variables  $x_1$  and  $x_2$  could represent different products subject to some availability or production constraints, while the objective function could represent total profit given the mix of products produced. The idea is to find the optimum mix of products so as to maximize profit. Linear optimization is not limited to this sort of product mix problem. For example, your problem could consist of trying to maximize the vitamin content of a livestock feed given certain ingredients, subject to their availability and cost. Or your problem could consist of trying to minimize the cost of labor for producing certain products in your laboratory (see the next recipe for a hypothetical example).

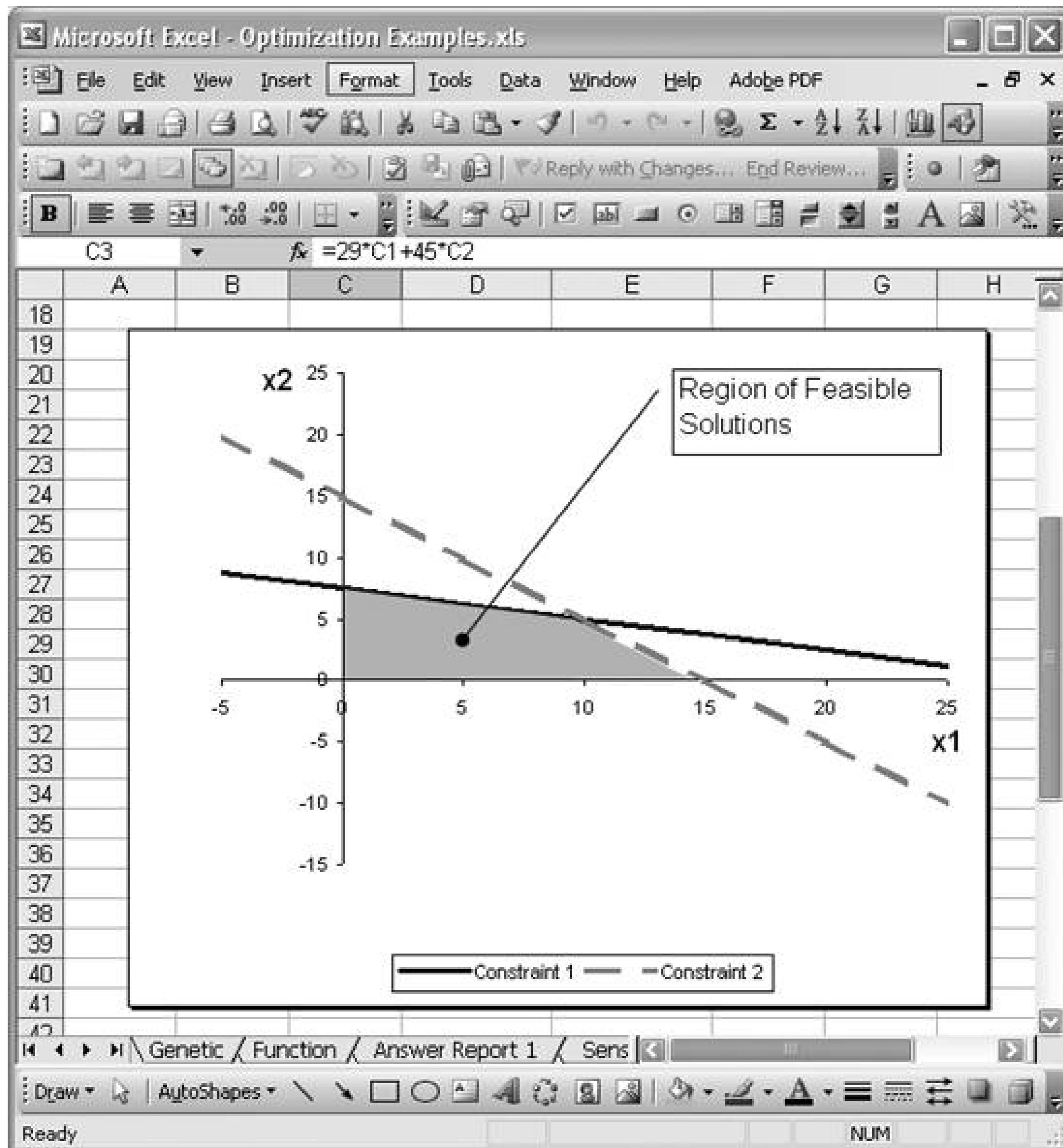


Moreover, the problem need not be limited to two variables. You can have any number of variables and constraints, depending on your problem. I chose two variables for this simple example because we can plot it using Excel's charting feature and gain some insight into the solution. Problems with more variables are much more difficult, if not impossible, to visualize adequately. Such problems require you to exercise great care when searching for a solution (particularly if the problem is nonlinear) or to simplify the problem in ways that will allow you to visualize certain variables while holding others constant.

Getting back to our simple example, [Figure 13-1](#) shows a graph of our problem. I used Excel's charting features (see [Chapter 4](#)) to prepare this graph.

This graph essentially consists of the constraints for the problem within the  $x_1, x_2$  space. The two straight lines labeled Constraint 1 and Constraint 2 represent the first two constraints. The shaded region below these constraints represents the region of feasible values for  $x_1$  and  $x_2$  that are within the given constraints. Notice that this region is bounded by the  $x_1$ - and  $x_2$ -axes because we also have the two greater-than-or-equal-to-zero constraints on these variables.

Figure 13-1. Graph of linear optimization example



I drew the shaded region using Excel's AutoShapes Lines Freeform drawing tool, available from the Drawing toolbar. To draw the shaded area, simply define vertices by pointing and clicking with the mouse. Be sure to select your chart before you start drawing the shaded region, because if you don't, you'll find the resulting shaded region drawn on your spreadsheet, not on your chart. Then when you reposition your chart, you'll find that the shaded area won't move with it.

From this graph, you can deduce that the optimum solution corresponds to  $(x_1, x_2) = (10, 5)$ , which is at the intersection of the two constraint lines. You can use Solver to verify this result.

Figure 13-2 shows a simple spreadsheet I set up to facilitate finding optimum values for  $x_1$  and  $x_2$  so as to maximize the objective function.

Figure 13-2. Linear optimization example

	A	B	C	D	E	F	G	H
1		x1 =	10					
2		x2 =	5					
3		f(x1, x2) =	515					
4								
5			Value	Limit				
6		Constraint 1	60	60				
7		Constraint 2	60	60				

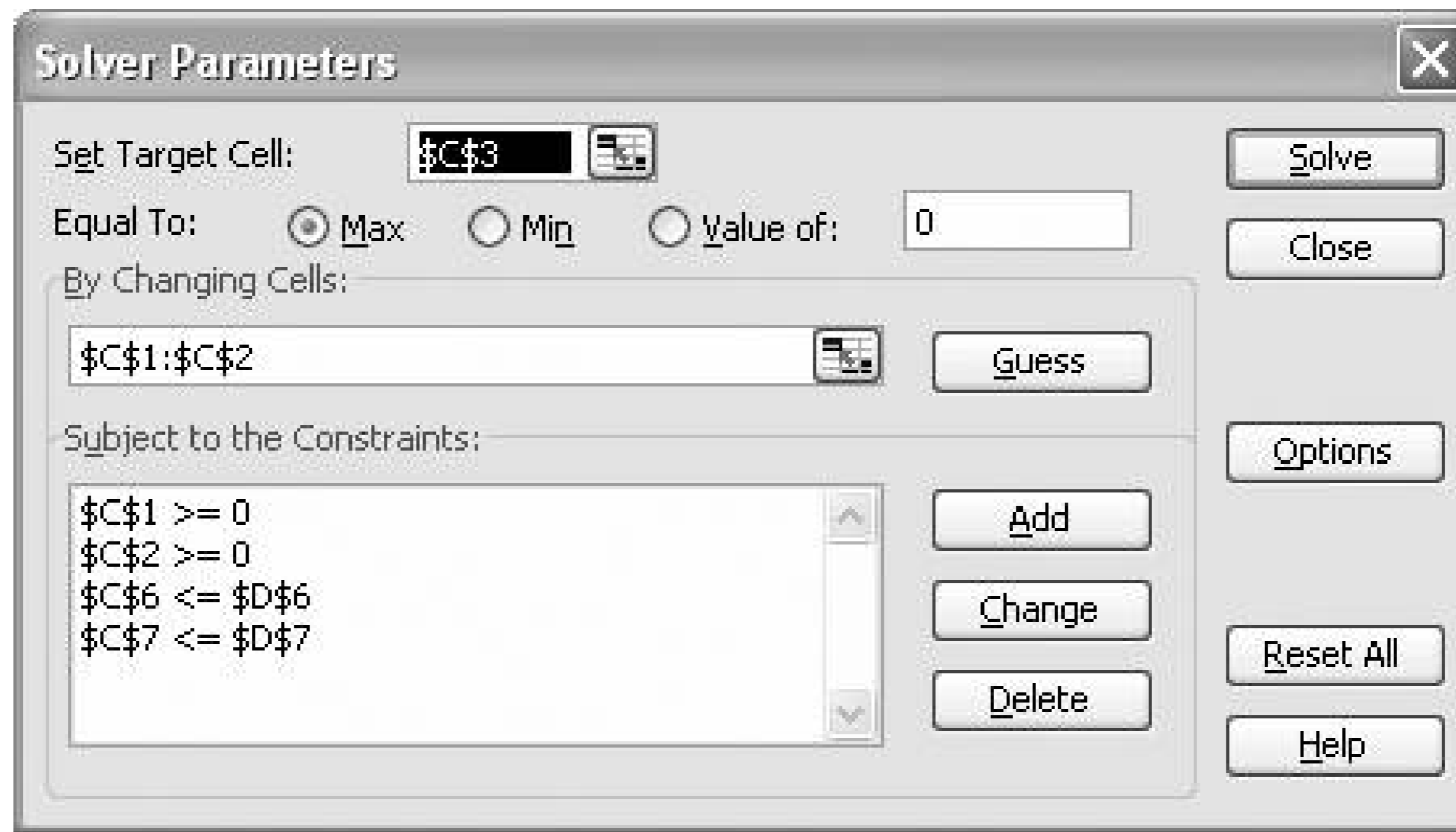
Cell C3 contains the righthand side of the objective function. The formula is  $=29*C1+45*C2$ . The values for  $x_1$  and  $x_2$  shown in Figure 13-2 are indeed the optimum values, and the value of 515 for the objective function is the maximum value. Before calling Solver, cells C1 and C2 contain only initial guesses for the optimum. I had initially set these to 0.

Cells C6 and C7 contain formulas corresponding to Constraint 1 and Constraint 2, respectively. The formula in C6 is  $=2*C1+8*C2$ , and the formula in C7 is  $=4*C1+4*C2$ . These formulas represent the lefthand side of the constraint equations shown earlier. The limiting values (called *righthand side values*) for these constraints are contained in cells D6 and D7.

Figure 13-3 shows the Solver model I used to solve this problem.

The target cell is C3 and I instructed Solver to attempt to maximize its value. The cells to change are C1 and C2, corresponding to  $x_1$  and  $x_2$ , respectively. There are four constraints as well. Refer to Recipe 9.4 to learn how to add constraints in Solver.

Figure 13-3. Solver model for linear optimization



The first two constraints shown in [Figure 13-3](#) correspond to the constraints that  $x_1$  and  $x_2$  must be greater than or equal to 0. The last two constraints correspond to the constraint equations discussed earlier, that is, Constraint 1 and Constraint 2 as plotted in [Figure 13-1](#).

I also set the Solver option Assume Linear Model for this problem, since it is a linear optimization problem. See the introduction to [Chapter 9](#) for a discussion of Solver's options. Pressing the Solve button results in the optimum solution shown earlier in [Figure 13-2](#).

## See Also

As I mentioned in the introduction to [Chapter 9](#), Solver can generate several reports for you upon finding a solution. See [Recipe 13.6](#) for more information.

## Recipe 13.2. Exploring Resource Allocation Optimization Problems

### Problem

You'd like to use Excel to solve a resource allocation problem.

### Solution

You can use Excel and Solver to tackle more complicated linear optimization problems than what you saw in the previous recipe. Resource allocation problems, like finding the optimum product mix to yield maximum profit, are commonly formulated as linear optimization problems, which are easily solved in Excel. The following discussion shows you an example. The techniques discussed here are not limited to resource allocation problems; on the contrary, they apply to many difficult problems so long as you can formulate them in the form of an objective function plus constraints.

### Discussion

Let's assume you're running a laboratory that produces a certain chemical product. You currently employ 10 regular employees, technicians, who make the product. Given input from your sales and marketing group, you've put together a demand schedule for your product over the next six months. Operating budgets are tight, however, and you need to develop an optimum labor schedule for meeting demand for your product at minimum labor cost. I should point out that you know that the demand for your product over the next six months will fluctuate, so at some times you may need to hire new employees and at others you may have to let folks go.

You have the options of having your employees work overtime or hiring contract labor. Although they have a higher hourly rate than regular employees, contractors charge straight time for overtime, whereas you must pay regular employees time and a half for overtime.

Regular employees earn \$15 per hour (\$22.50 per overtime hour), while contract employees earn \$18 per hour (both regular and overtime hours). To complicate matters, regular employees can work only 8 hours overtime per week, due to union rules. Contract employees are willing to work up to 16 hours per week overtime.

Historically, you know that each employee is capable of producing 2 units of product per hour, but this productivity rate drops to about 1.75 units per overtime hour. Further, you have only enough space in your facility to employ a maximum of 20 technicians at any given time. Also, it is company policy to employ no less than 5 regular technicians at any given time.

Finally, your human resources director tells you that it costs \$500 in overhead charges to hire a

regular employee and \$600 to let one go. Similarly, it costs only \$250 to hire a contractor and \$300 to let one go.

Your task now is to come up with a labor schedule for the next six months that will allow you to meet projected demand for your product, while at the same time minimizing labor costs.

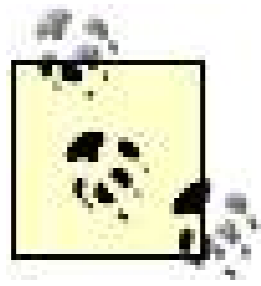
This is a classic sort of linear optimization problem that's ideally suited for solving in Excel. I'll walk you through the process.

The first thing I did toward solving this problem was enter all the data in a spreadsheet. [Figure 13-4](#) shows how I arranged this data.

The cells in column B contain text labels I used to describe the data. The actual data values are contained in column D. The short text labels you see in column C are labels I included to allow me to easily define cell names for each data item. I use these cell names in other cell formulas later.

Figure 13-4. Resource allocation problem data

	B	C	D	E	F
1	Working Hours per month per employee =	whrs	160	hours	
2	Regular Employee Rate =	rrate	\$ 15.00	per hour	
3	Regular Employee Overtime Rate =	orate	\$ 22.50	per hour	
4	Contractor Rate =	crate	\$ 18.00	per hour	
5	Cost to hire regular employee =	hrcost	\$ 500.00		
6	Cost to fire regular employee =	frcost	\$ 600.00		
7	Cost to hire contractor =	hccost	\$ 250.00		
8	Cost to fire contractor =	fccost	\$ 300.00		
9	Max. regular employee overtime hours =	romax	32.00	per month per employee	
10	Max. contractor overtime hours =	comax	64.00	per month per employee	
11	Num. products produced per hour =	n	2.00		
12	Num. products produced per overtime hour =	no	1.75		
13	Max. number of employees (regular + contract) =	emax	20		
14	Min. number of regular employees =	ermin	5		
15					



To quickly define names for a group of cells arranged like those shown in [Figure 13-4](#), select the range of cells containing text label and data value pairs and then select Insert → Name → Create from the main menu bar. This will display a simple Create Names dialog box, allowing you to choose how you want the names created. You can choose to create the names given text labels in the left column, right column, or top or bottom row of the selected cell range. For this example, I selected the cell range C1:D14 and then selected Insert Name → Create. Then I chose the Left Column option from the Create Names dialog box.

After setting up the data, I created a table in the spreadsheet to contain a monthly labor schedule. This part of the spreadsheet is shown in [Figure 13-5](#).

The first row of this table contains the projected demand for your product over the next six months. These are given values. At month 0, the current month, you know you have 10 regular employees working regular 40-hour work weeks. The second to last row, *Production*, calculates the monthly production given the number of regular and contract employees and their overtime schedules each month. The formulas contained in this row look like this:

$=n*(D23+D24)*whrs+no*D26*romax+no*D27*comax$ . Clearly the projected production, given current staffing in month 0, is not going to be enough to meet the forecast demand. So your job is to figure out how many (if any) regular or contract employees to hire each month (along with their overtime schedules) to allow you to meet production targets at minimum labor cost.

Figure 13-5. Resource allocation monthly schedule

		Month						
		0	1	2	3	4	5	6
18	Product Demand		5000	6000	4000	3600	6400	5600
19	Num. Employees hired	0	0.00	0.00	0.00	0.00	0.00	0.00
20	Num. Employees fired	0	0.00	0.00	0.00	0.00	0.00	0.00
21	Num. Contractors hired	0	0.00	0.00	0.00	0.00	0.00	0.00
22	Num. Contractors fired	0	0.00	0.00	0.00	0.00	0.00	0.00
23	Regular Employees	10	10.00	10.00	10.00	10.00	10.00	10.00
24	Contract Employees	0	0.00	0.00	0.00	0.00	0.00	0.00
25	Num. Employees	10	10.00	10.00	10.00	10.00	10.00	10.00
26	Regular Employees Working Overtime	0	0.00	0.00	0.00	0.00	0.00	0.00
27	Contract Employees Working Overtime	0	0.00	0.00	0.00	0.00	0.00	0.00
28	Production		3200	3200	3200	3200	3200	3200
29	Cost		\$ 24,000	\$ 24,000	\$ 24,000	\$ 24,000	\$ 24,000	\$ 24,000
32	Total Cost =		\$144,000					

Labor cost, or **Total Cost**, is the objective function. In this case you want to minimize total cost. Cell C32 contains the total cost for this example. It is simply the sum of monthly costs computed in the last row of the table, the row labeled **Cost**.

Monthly cost is computed using formulas like  $=D23*whrs*rrate + D24*whrs*crate + D26*romax*orate + D27*comax*crate + D19*hrccost + D20*frccost + D21*hccost + D22*fccost$ . There are eight terms in this cost function:

$D23*whrs*rrate$

This term computes the direct labor cost per month, given the total number of direct employees in a given month, along with the number of regular working hours per month and the regular employee labor rate.

$D24*whrs*crate$

This term computes the contract labor cost per month, given the total number of contractors, along with their labor rate and the number of regular working hours per month.

$D26*romax*orate$

This term computes the overtime labor cost for regular employees, given the number of regular employees working overtime (assuming each employee works the maximum 8 hours per week



overtime), along with the overtime labor rate.

$D27 * comax * crate$

This term computes the overtime labor cost for contractors, given their rate and assuming that each contractor working overtime works the maximum 16 hours per week overtime.

$D19 * hrcost$

This term computes the overhead cost of hiring regular employees, given the number of regular employees hired at the beginning of a month.

$D20 * frcost$

The term computes the overhead cost of firing ("letting go" is a nicer way to say this, I suppose) regular employees, given the number of regular employees fired at the beginning of a month.

$D21 * hccost$

This term computes the overhead cost of hiring contract employees, given the number of contract employees hired at the beginning of a month.

$D22 * fccost$

The term computes the overhead cost of firing contract employees, given the number of contract employees fired at the beginning of a month.

Given this cost function, you can readily see the variables in this problem. They include the following: the number of regular employees hired, the number of regular employees let go, the number of contractors hired, the number of contractors let go, the number of regular employees working overtime, and the number of contractors working overtime each month. I highlighted all of the variable cells in the table shown in [Figure 13-5](#) so you can readily see them.

The next step is to use Solver to optimize this problem. I basically told Solver to minimize the total cost in cell C32 by changing all the shaded cells in [Figure 13-5](#), subject to a bunch of constraints. (See [Recipe 9.4](#) for instructions on adding constraints in Solver.) The constraints for this problem are as follows:

*Production meets or exceeds demand*

The production values computed in cells D28 through I28 must be greater than the corresponding projected demand values contained in cells D18 through I18.

*Greater than zero requirement*

Each of the variables (shaded cells in [Figure 13-5](#)) must be greater than or equal to 0.

#### *Maximum employees*

There can be no more than 20 total employees in any given month.

#### *Overtime consistency*

The number of regular employees working overtime must be less than or equal to the total number of regular employees in a given month. Also, the number of contract employees working overtime must be less than or equal to the total number of contract employees in a given month.

#### *Minimum regular employees*

The total number of regular employees must be greater than the specified minimum number of regular employees in any given month.

Refer to the previous recipe and the introduction to [Chapter 9](#) for details on setting up and using Solver. Running Solver for this example quickly yielded the results shown in [Figure 13-6](#).

Figure 13-6. Resource allocation results

As you can see, Solver found a solution satisfying all the constraints. It turns out that the minimum labor costs is \$243,654, which is achieved through a combination of hiring and letting go both regular and contract employees over the six-month period. Some months will also require some contractor overtime. So far so good, but there's one problem: look at the required number of regular employee hires for month 1. Solver says you need 5.62 employees. How do you hire 62% of an employee? Well, either you hire some part-time labor, in which case you'd have to alter your model accordingly, or you require Solver to find an optimum solution using only integer values for the variables. This latter approach requires the use of *integer constraints*. In the next recipe I show you how to apply integer constraints to solve this same example problem.



## Recipe 13.3. Getting More Realistic Results with Integer Constraints

### Problem

You've performed an optimization problem where several variables were real valued, but realistically, certain values should only be integers. For example, in the previous recipe, the number of optimum lab technicians for a given month was 5.62. Obviously hiring 0.62 of a person would be hard (unless you're assuming the number represents part-time employment, in which case you could assume that 0.62 represents 62% of a regular 40-hour work week). Therefore, you want to constrain certain variables to be whole integers like 5 or 6.

### Solution

Use Solver as discussed in the previous recipes, along with integer constraints on certain variables.

### Discussion

Let's pick up where we left off with the resource allocation example from the previous recipe. What you need to do now is add several integer constraints. In fact, you want to require every value contained in cells D19 to I22 and D26 to I27 to assume only integer values. Real numbers will not be allowed.

You can add an integer constraint in Solver just as you would an ordinary constraint (see [Recipe 9.4](#) for a discussion of constraints). After opening the Solver window, click the Add button to open the Add Constraint window. Select the cell you want to constrain and then select "int" from the drop-down list in the middle of the window, as shown in [Figure 13-7](#).

Figure 13-7. Adding an integer constraint

Solver will automatically enter "integer" in the Constraint field. Press Add for each constraint you want to add. A faster approach involves selecting an entire cell range in the Cell Reference field. For example, you could select the range D19:I22 to constrain all of the cells within that range to integer values. This approach avoids the tedium of having to add constraints one at a time for each cell in the range.

After you've added integer constraints, you can rerun Solver for this problem. Be prepared to wait, though. Adding integer constraints results in far greater computation time for this model. Whereas the first example (without integer constraints) converged on a solution in a matter of seconds, this second example (with integer constraints) took several hours to find a solution. In fact, it was taking so long on my computer that I left it running overnight, since I couldn't stay awake waiting for Solver to finish.

When Solver did finish, it yielded the results presented in [Figure 13-8](#).

Figure 13-8. Resource allocation results with integer constraints

		Month						
		0	1	2	3	4	5	6
18	Product Demand		5000	6000	4000	3600	6400	5600
19	Num. Employees hired	0	6.00	0.00	0.00	0.00	6.00	0.00
20	Num. Employees fired	0	0.00	0.00	4.00	0.00	0.00	0.00
21	Num. Contractors hired	0	0.00	3.00	0.00	0.00	2.00	0.00
22	Num. Contractors fired	0	0.00	0.00	3.00	0.00	0.00	2.00
23	Regular Employees	10	16.00	16.00	12.00	12.00	18.00	18.00
24	Contract Employees	0	0.00	3.00	0.00	0.00	2.00	0.00
25	Num. Employees	10	16.00	19.00	12.00	12.00	20.00	18.00
26	Regular Employees Working Overtime	0	0.00	0.00	3.00	0.00	0.00	0.00
27	Contract Employees Working Overtime	0	0.00	3.00	0.00	0.00	0.00	0.00
28	Production		5120	6416	4008	3840	6400	5760
29	Cost		\$ 41,400	\$ 51,246	\$ 34,260	\$ 28,800	\$ 52,460	\$ 43,800
32	Total Cost =		\$251,966					

In this case, the minimum total cost comes to \$251,966. This is a little higher than the earlier results discussed in the previous recipe. The reason is that Solver had to round up the number of employees hired or fired, and so on. Therefore, you end up with a little excess labor capacity. The production numbers reflect this excess capacity, where you can see that production will exceed demand by a little bit each month.

A further refinement of this problem would involve carrying over the excess production from one month to the next and factoring it in the required labor optimization each month. I'll leave that exercise to you.



## Recipe 13.4. Tackling Troublesome Problems

### Problem

You're dealing with a particularly troublesome optimization problem where you can't seem to converge on a global optimum. For example, Solver converges to several different solutions given different initial values. You suspect Solver is converging to local optima rather than a true global optimum solution.

### Solution

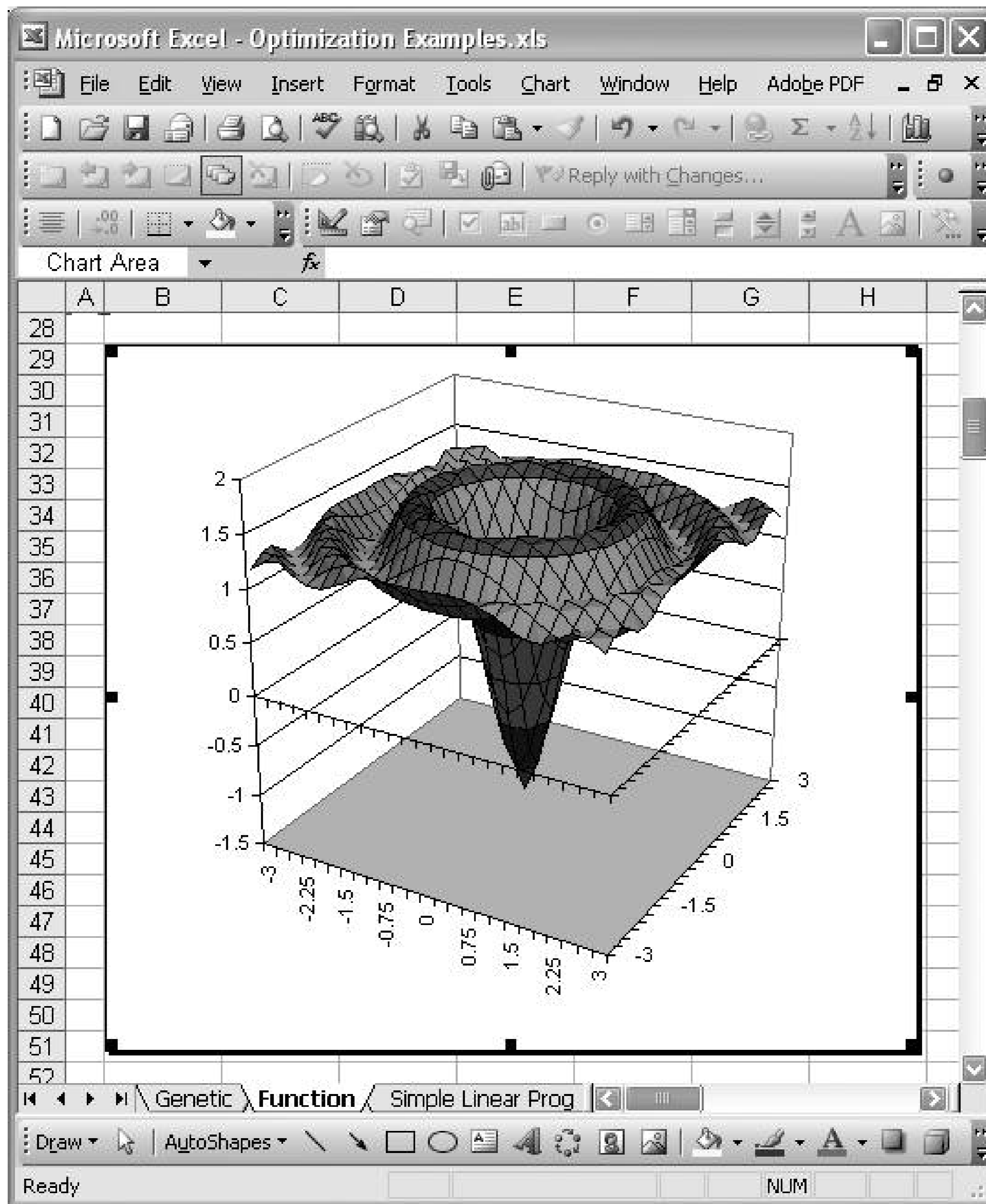
Getting trapped by local maxima or minima is an unfortunate part of optimization. Some problems are just riddled with local extrema over their objective function landscape and, depending on your assumed initial guess, you may end up in one of these locally optimum solutions rather than finding a globally optimum solution. One way around this is to select different initial guesses just as I discussed in Chapter 9 when finding multiple roots of a nonlinear equation. Also, if possible, you should plot the objective function to get a feel for how it behaves. This is not always possible for multidimensional problems, though.

As an alternative, you can use a *Monte Carlo*-inspired approach by repeatedly using Solver for many initial guesses. The following discussion shows you how.

### Discussion

Take a look at the function plotted in Figure 13-9.

Figure 13-9. Troublesome function

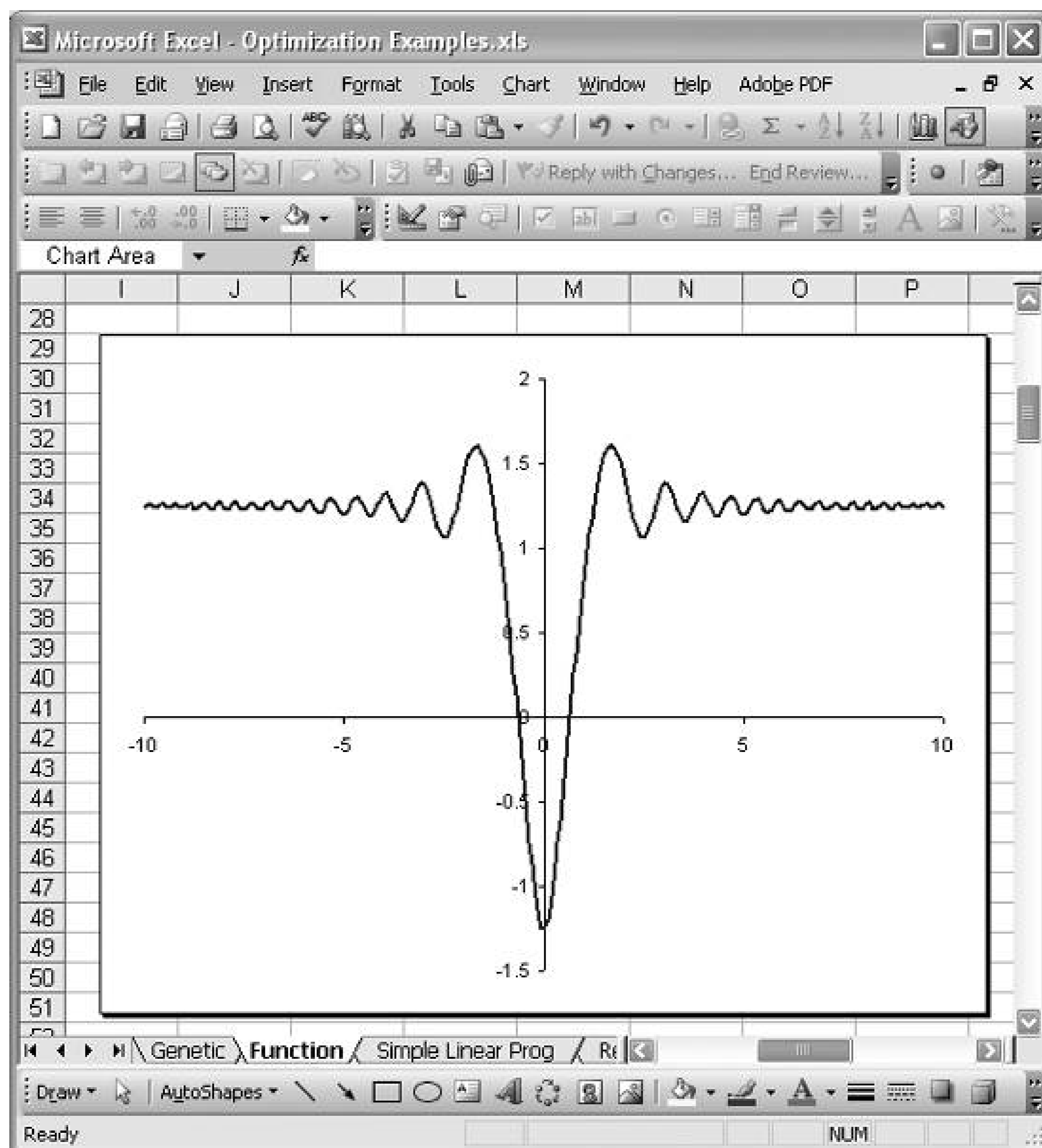


This function wrecks havoc on Solver. The equation for this function is:

Let's look at this function from a different angle. Check out Figure 13-10.

Figure 13-10. Cross-section of troublesome function





The shape of this function provides for an infinite number of local extrema that can trap Solver. It's a veritable mine field of local optima! For example, I ran Solver to minimize this function by varying  $x$  and  $y$  and with an initial  $(x, y) = (5, 8)$ . Solver found the "minimum"  $z = 1.236$  corresponding to  $(x, y) = (4.971, 7.953)$ . Obviously that's wrong, since you can see clearly from the two charts of this function that the minimum  $z$  is  $-1.25$  at  $(x, y) = (0, 0)$ . Trying another initial guess for  $(x, y)$  didn't help matters. An initial guess of  $(x, y) = (10, 3)$  yielded a minimum  $z$  of  $1.238$ ; another initial guess of  $(x, y) = (2, 1)$  yielded a minimum  $z$  of  $1.064$ ; yet another initial guess of  $(x, y) = (1.5, 1.5)$  yielded a minimum  $z$  of  $1.064$ . All of these are wrong!

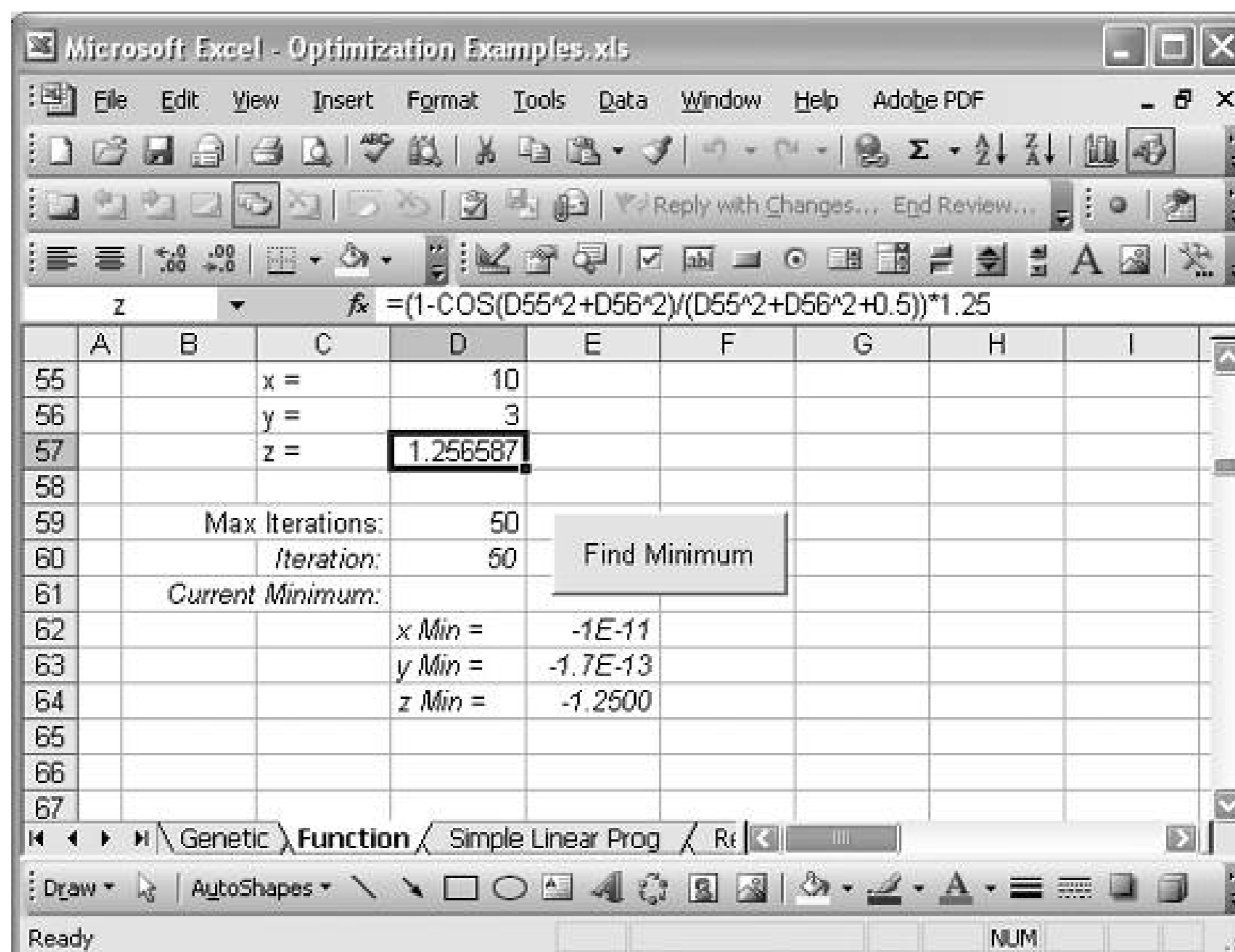
Since you can plot this function, you can see where the global minimum lies and choose much better initial values for  $x$  and  $y$  values close to  $(0, 0)$ . But what if this was a higher-dimensional problem that you could not plot? If you were trying to minimize such a function, you'd very likely get stuck in a local minimum unless you were lucky enough to pick just the right initial guess. Therefore, it would be prudent for you to try many different guesses in an effort to comprehensively cover the possible solution domain, homing in on the global optimum. You can indeed perform such an exercise using Solver manually, but that would get dull rather quickly.

A better approach, inspired by Monte Carlo methods, involves programmatically selecting many random initial guesses and running Solver for each guess. Such sampling of the solution domain

would allow you to quickly evaluate many possible solutions. This approach is easy to program in Excel, since you can call Solver directly from VBA code (see Recipe 9.3). I'll show you how to do it.

Figure 13-11 shows a simple spreadsheet I set up for this problem.

Figure 13-11. Monte Carlo-inspired solution



Cell D57 contains the formula for our objective function:  $=(1 - \text{COS}(D55^2+D56^2) / (D55^2+D56^2+0.5)) * 1.25$ . The formula is also shown in the formula bar in Figure 13-11, since cell D57 is selected in that figure. Cells D55 and D56 contain the initial guesses for  $x$  and  $y$ .

Cell D59 contains a value for the maximum number of iterations that will be used by the VBA subroutine I'll show you shortly. I'll explain this more in a moment. The other cells, formatted with italic text, are status messages generated by the VBA subroutine I'll show you. These status messages will allow you to see how the iterative solution is progressing. The button you see in Figure 13-11 will launch the iterative subroutine when you click it. See Recipe 9.3 to learn how to add this sort of button.

The iterative solution I want to show you involves generating many random initial guesses for  $x$  and  $y$  within the solution domain for this problem. These guesses, or random samples, will then be used as initial guesses for running Solver. Actually, the algorithm I'll show you takes a random sample, runs Solver, and stores the result. Then it picks another random sample, runs Solver again, and compares the new result with the previous one. If the new result, the minimum  $z$ , is lower than the previous one, then we know the previous result was only a local minimum, so we discard it and replace it with the new one. This process repeats over and over until the maximum number of iterations that you specified in cell D59 is reached.

Example 13-1 shows the VBA subroutine I wrote that implements this algorithm.

## Example 13-1. Algorithm to iteratively minimize a function

```

Private Sub FindMinimumButton_Click( )
    ' Declare local variables:
    Dim xMin As Double
    Dim yMin As Double
    Dim zMin As Double
    Dim n As Integer
    Dim Z As Double
    Dim MaxIterations As Integer

    With Worksheets("Function")
        ' Initialize variables:
        n = 0
        xMin = .Range("x")
        yMin = .Range("y")
        zMin = .Range("z")
        MaxIterations = .Range("MaxIterations")

        ' Randomize the random number generator:
        Randomize

        ' Generate random values for x and y:
        .Range("x") = (5 - (-5) + 1) * Rnd + (-5)
        .Range("y") = (5 - (-5) + 1) * Rnd + (-5)

        ' Start solution iterations:
        Do While (n <= MaxIterations)
            .Range("Iteration") = n

            ' Run Solver:
            SolverOK SetCell:=.Range("z"), MaxMinVal:=2, ByChange:=.Range("x", "y")
            SolverSolve UserFinish:=True
            SolverFinish KeepFinal:=1

            Z = .Range("z")

            ' Check result:
            If Z < zMin Then
                xMin = .Range("x")
                yMin = .Range("y")
                zMin = Z

                .Range("x_Min") = xMin
                .Range("y_Min") = yMin
                .Range("z_Min") = zMin
            End If

            ' Generate random values for x and y:

```

```

        .Range("x") = (5 - (-5) + 1) * Rnd + (-5)
        .Range("y") = (5 - (-5) + 1) * Rnd + (-5)
        n = n + 1
    Loop

    .Range("x") = xMin
    .Range("y") = yMin
End With
End Sub

```

I've commented the code so you can generally see what's going on. The first several lines in the subroutine declare local variables in the usual manner. `xMin`, `yMin`, and `zMin` store the current minimum  $x$ ,  $y$ , and  $z$  as the routine iterates over many trial values. `n` is just a counter variable used to track the current iteration. `z` stores the  $z$  value found by Solver in each iteration; it may or may not be the global minimum  $z$ . Finally, `MaxIterations` stores the maximum number of iterations you set in the spreadsheet (cell D59).

Upon entering the `With` statement (see Recipe 9.3) the routine initializes several variables. Basically, the counter `n` is set to 0, while the other variables mentioned earlier are set to values extracted from the worksheet named `Function`.

`Randomize` is then called to initialize VBA's random number generator. Next, two random numbers between the bounds -5 and 5 are generated and stored in cells D55 and D56, named `x` and `y`, respectively. These serve as random initial guesses before we call Solver.

See Recipe 5.8 to learn how to use Excel's and VBA's random number functions. Also, see Recipe 1.14 to learn more about cell names.

Next, a `Do While` loop is entered to iterate `MaxIterations` times, generating random initial guesses and calling Solver each time. Let's look inside the `Do While` statement.

The line `.Range("Iteration") = n` sets the value of cell D60, named `Iteration`, to `n`. This provides feedback to you on the status of the subroutine as it runs. The next several lines invoke Solver as described in Recipe 9.3. In this case, the target cell is set to the cell named `z`, and the cells to change are the ones named `x` and `y`. `MaxMinVal` is set to 2, indicating that we want to minimize the target cell. When Solver is finished, the cell named `z` in the worksheet named `Function`, should contain a minimum of the subject function. However, this may or may not be the global minimum. So, we have to check it against the currently stored minimum, `zMin`.

First, the value from the cell named `z` is extracted and stored in variable `z`. The following `If` statement checks whether or not `z` is less than the stored `zMin`. If it is, then we found a better estimate of the minimum of the subject function, which should be saved in the current minimum variables. The three lines `xMin = .Range("x")`, `yMin = .Range("y")`, and `zMin = z` take care of this task. The next three lines echo the current minimum variables back to the spreadsheet so you can see their status as the routine iterates.

Upon exiting this `If` statement, the routine generates two new random guesses for  $x$  and  $y$  and

increments the iteration counter by 1. The `Do While` loop then repeats until `MaxIterations` is reached.

After the maximum number of iterations has been reached, the minimum values found are echoed back to the spreadsheet.

When the routine is finished, cells E62 through E64 will contain the minimum  $z$ -value and corresponding  $x$ - and  $y$ -values for the subject function. After running this routine many times, I have found that it finds the global minimum in all but a very few cases; in those cases, a randomly selected initial guess that would have converged toward the global minimum was never selected. You can reduce the likelihood of that occurring by cranking up the value for `MaxIterations`.

## See Also

The approach shown in this recipe goes a long way to mitigate getting stuck at local minima or maxima. It's not the only approach, though. Take a look at Recipe 13.7 for a completely different approach.



## Recipe 13.5. Optimizing Engineering Design Problems

### Problem

You're working on an engineering design problem and would like to see how you can leverage Excel to find an optimum design solution.

### Solution

Model your design problem in Excel as an optimization problem and then use Solver to find a solution

### Discussion

It is rather straightforward to model many different engineering design problems in Excel. Many engineers do so on a daily basis. Moreover, once you have a model set up in Excel, it's a small step to use Solver to optimize that model. Take a look at the spreadsheet in [Figure 13-12](#).

Figure 13-12. Engineering design optimization example

Microsoft Excel - Optimization Examples.xls

File Edit View Insert Format Tools Data Window Help Adobe PDF Type a question for help

100%

10 B

D16  $= (D1*D2)*E19+D5*D2*(D20*E20+D21*E21)$

1	Panel Width =	100.00 cm					Max Total Depth =	10.16 cm
2	Panel Length =	150.00 cm					Min Plate Thickness =	0.32 cm
3	Design Pressure =	206.84 kN/m <sup>2</sup>					Min. allowable SM =	143.24 cm <sup>3</sup>
4	Stiffener Spacing =	34.82 cm					Max. Web depth-to-thickness ratio =	52.90
5	Num. Stiffeners =	1.87					Max. Flange width-to-thickness ratio =	17.63
6	Web thk-to-plate thk ratio =	0.69					Max. Web thk-to-plate thk ratio =	2.50
7	Web depth-to-thickness ratio =	24.40					Max. Flange Width =	10.00
8	Flange width-to-thickness ratio =	5.13						
9								
10	Yna =	5.08 cm						
11	Iref =	1710.17 cm <sup>4</sup>						
12	Ira =	727.66 cm <sup>4</sup>						
13	SM top =	143.24 cm <sup>3</sup>						
14	SM bot =	143.24 cm <sup>3</sup>						
15								
16	Panel Material Volume =	13106.854 cm <sup>3</sup>						
17								
18		Width (cm)	Height (cm)	Area (cm <sup>2</sup> )	y (cm)	Ay (cm <sup>3</sup> )	Ay <sup>2</sup> (cm <sup>4</sup> )	I <sub>o</sub> (cm <sup>4</sup> )
19	Plate	34.82	0.46	16.12	0.23	3.73	0.86	0.29
20	Stiffener Web	0.32	7.75	2.46	4.34	10.67	46.28	12.31
21	Stiffener Flange	10.00	1.95	19.49	9.19	179.00	1644.27	6.17
22			10.16	38.07		193.41	1691.41	18.76

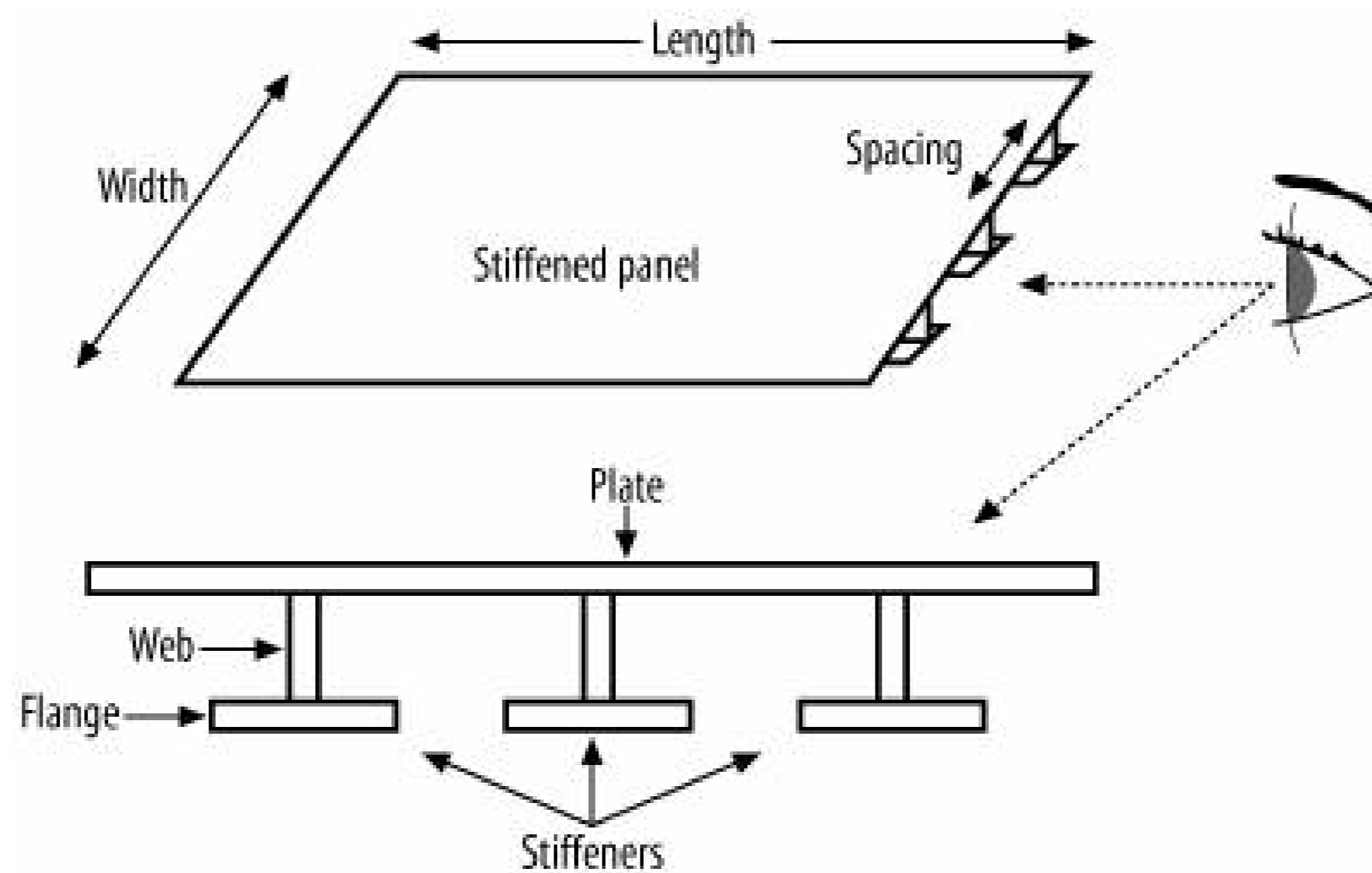
Genetic / Function / Simple Linear Prog / Resource Allocator

Draw AutoShapes

Ready NUM

This spreadsheet represents a typical structural engineering design task. I don't want to go too far describing all the structural engineering background relevant to this problem; but I'll give you an overview so you can get the gist of what we're trying to optimize. Basically, this is a model of a *stiffened panel*. A stiffened panel is comprised of a plate with beams attached to it to provide strength and rigidity. In this case, the panel is comprised of an aluminum plate, 150 cm long and 10 cm wide, supported all around its edges. The plate also has attached to it a number of T-type stiffeners. [Figure 13-13](#) illustrates this sort of panel.

Figure 13-13. Stiffened panel



You can imagine such a panel forming a portion of a wall of an aluminum building or perhaps the hull of an aluminum boat. The latter case is actually the inspiration for this example.

Boats, especially high-speed boats, are sensitive to weight in terms of performance. So a common task facing naval engineers involves sizing panels like this one so that it has enough strength but is also as light as possible. Sure, you could be conservative and use a really thick plate with a lot of stiffening beams, but that would be expensive and heavy.

You have to use some sort of standard criteria to determine whether such a panel is strong enough. There are such *codes* (as in rules and regulations, not to be confused with software code) for many types of structures and in this particular case, you can use the standard codes applicable to aluminum boats. These codes require engineers to calculate certain properties of the panel to compare with acceptable values given in the codes. The spreadsheet shown in [Figure 13-12](#) performs just this sort of calculation and comparison.

Cells D1 through D8 contain basic information on the panel and associated stiffeners. The table toward the bottom, cells C18 through J22, contains more specific information on the geometry of the components making up the stiffened panel. This panel also performs the required engineering calculations needed to determine the properties of the stiffened panel. The results of these calculations are contained in cells D10 through D14.

Finally, cells J1 through J7 contain code-related requirements on the properties of the stiffened panel. These cells also contain some practical engineering limits.

The optimization task is to find an optimum combination of plate thickness, stiffener web size, stiffener flange size, and spacing between stiffeners that will yield a panel meeting the code requirements, while at the same time resulting in the minimum achievable weight. Therefore, in terms of optimization, minimizing weight is the objective. All the other requirements become constraints.

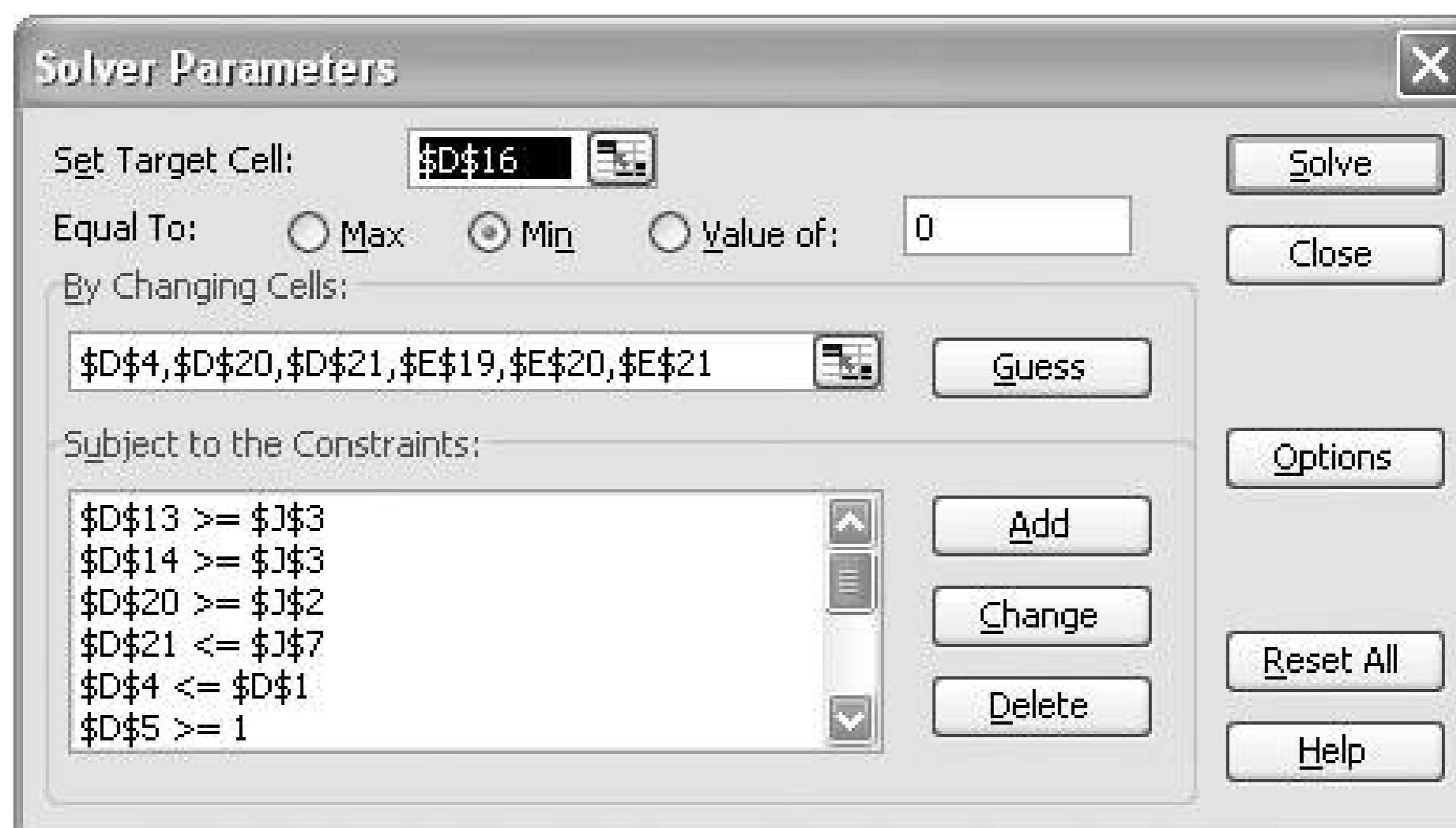
Since the material is homogeneous and uniform in this case, we can minimize weight by minimizing the total volume of metal material used to build the stiffeners and panel. The volume is computed by summing the volumes of the plate and stiffeners. Cell D16 contains the volume calculation, and you can see the formula in the formula bar in [Figure 13-12](#). This is our objective function.



The variables we want to change include the stiffener spacing, plate thickness, stiffener web depth and thickness, and stiffener flange width and thickness.

[Figure 13-14](#) shows the Solver model I set up for this problem.

Figure 13-14. Design optimization Solver model



As expected, cell D16, the objective function, is the target cell, which is set to be minimized. The By Changing Cells textbox includes all of the shaded cells in [Figure 13-12](#) corresponding to the variables we want to vary, as discussed a moment ago.

All the code and design requirements discussed earlier become constraints. (See [Chapter 9](#) for more on setting up Solver models and adding constraints.) For example, the *section modulus* (SM) of the stiffened panel (SM top and SM bot in the spreadsheet) must be greater than or equal to the **Min. allowable SM** specified in cell J3. The corresponding Solver constraints are  $\$D\$13 \geq \$J\$3$  and  $\$D\$14 \geq \$J\$3$ , as shown in [Figure 13-14](#). I added similar constraints for each design requirement listed in cells J1 through J7.

Running Solver resulted in the optimum values shown earlier in [Figure 13-12](#). Initially, I tried 50 cm for stiffener spacing and 1 cm for all other optimization variables. Solver readily found a solution with this initial guess. Just to be sure, I tried several other initial guesses; Solver converged on the same solution, so I'm fairly confident that this solution represents a global optimum in this particular design space.

In summary: if you can model a design problem in Excel, you can optimize it in Excel. The trick is in identifying your objective function and the variables you want to change. For highly nonlinear problems, you also have to be careful that you don't converge to a local minimum. You can use the technique I showed you in [Recipe 13.4](#) to mitigate the risk of not finding a global optimum.

## Recipe 13.6. Understanding Solver Reports

### Problem

You know that Solver can generate several types of reports upon finding a solution to a problem and you'd like to learn more about these.

### Solution

The version of Solver that ships with Excel can generate three reports: an *Answer* report, a *Sensitivity* report, and a *Limits* report. These reports are described in the following discussion.

### Discussion

Solver's three reports are helpful when interpreting the results obtained by Solver. When it finds a solution, you're presented with a dialog box containing a message regarding the nature of the solution found. You can choose which Solver reports, if any, to generate. (See the introduction to [Chapter 9](#) for more information.) Solver will generate the selected reports in separate worksheets that it will create automatically. Each of the three available report types is explained in the following sections.

#### Answer report

[Figure 13-15](#) shows an example Answer report. This is the Answer report for the problem solved in [Recipe 13.1](#).

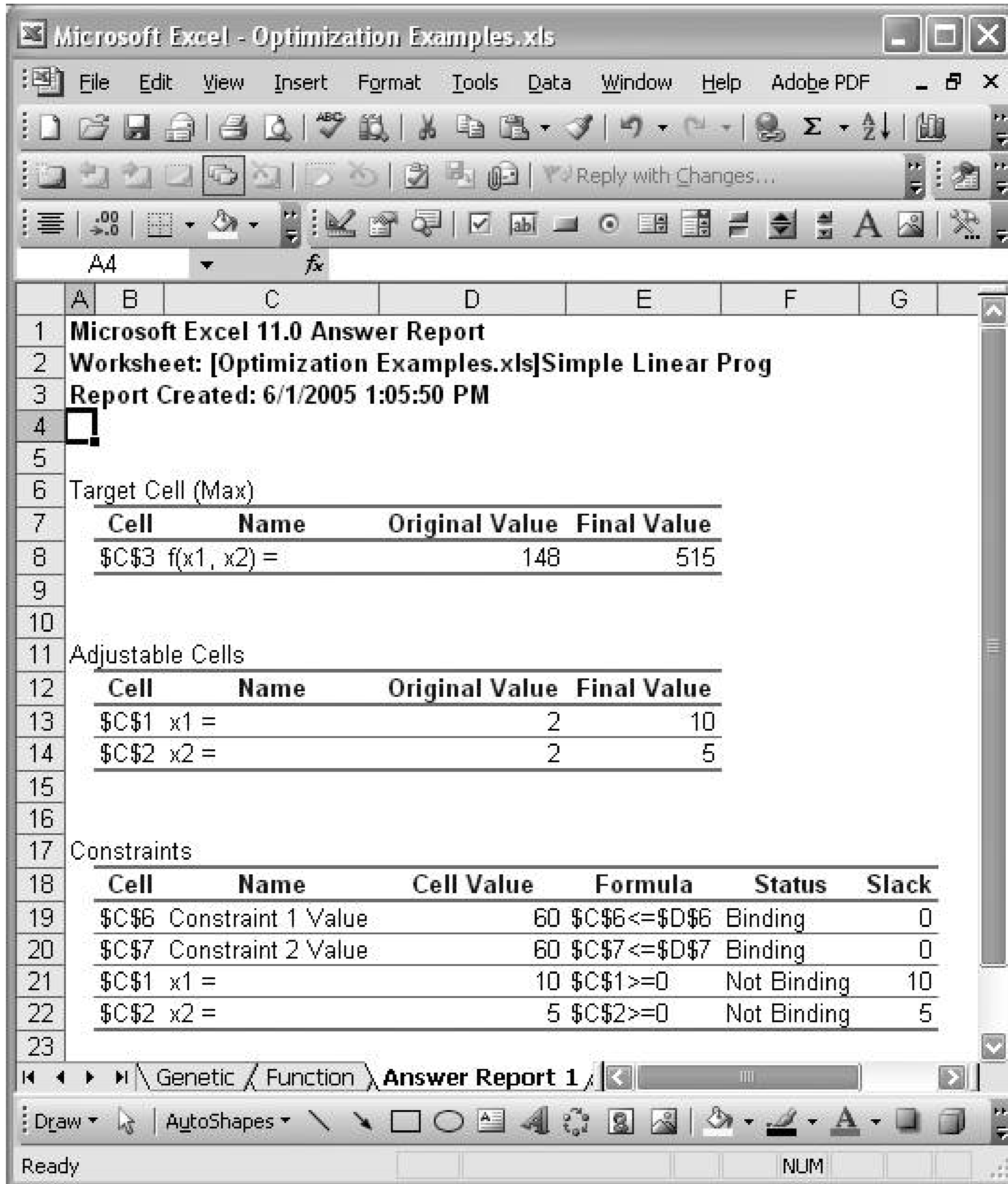
Answer reports show the initial and final values of the objective (Target Cell) and all variables (Adjustable Cells). You can quickly see from these reports how close your initial guess was to the optimal solution.

This report also shows the optimum values for all constraints, including a note indicating which constraints were actually binding. For Not Binding constraints, the amount of slack is also reported. The slack on a constraint tells you how far away a constraint is from becoming a binding constraint. All this information is helpful in determining which constraints govern, or limit, the problem being solved, and how much leeway you have on other constraints.

Notice that these reports contain cell references to the objective, variables, and constraints. They also include a name for these cells. These are not cell names as defined using Excel's cell name feature (discussed in [Recipe 1.14](#)). Instead, these names are automatically generated by Solver using a specific algorithm. If you set your spreadsheet up with this algorithm in mind, the names are

meaningful; otherwise, they could be completely meaningless.

Figure 13-15. Answer report



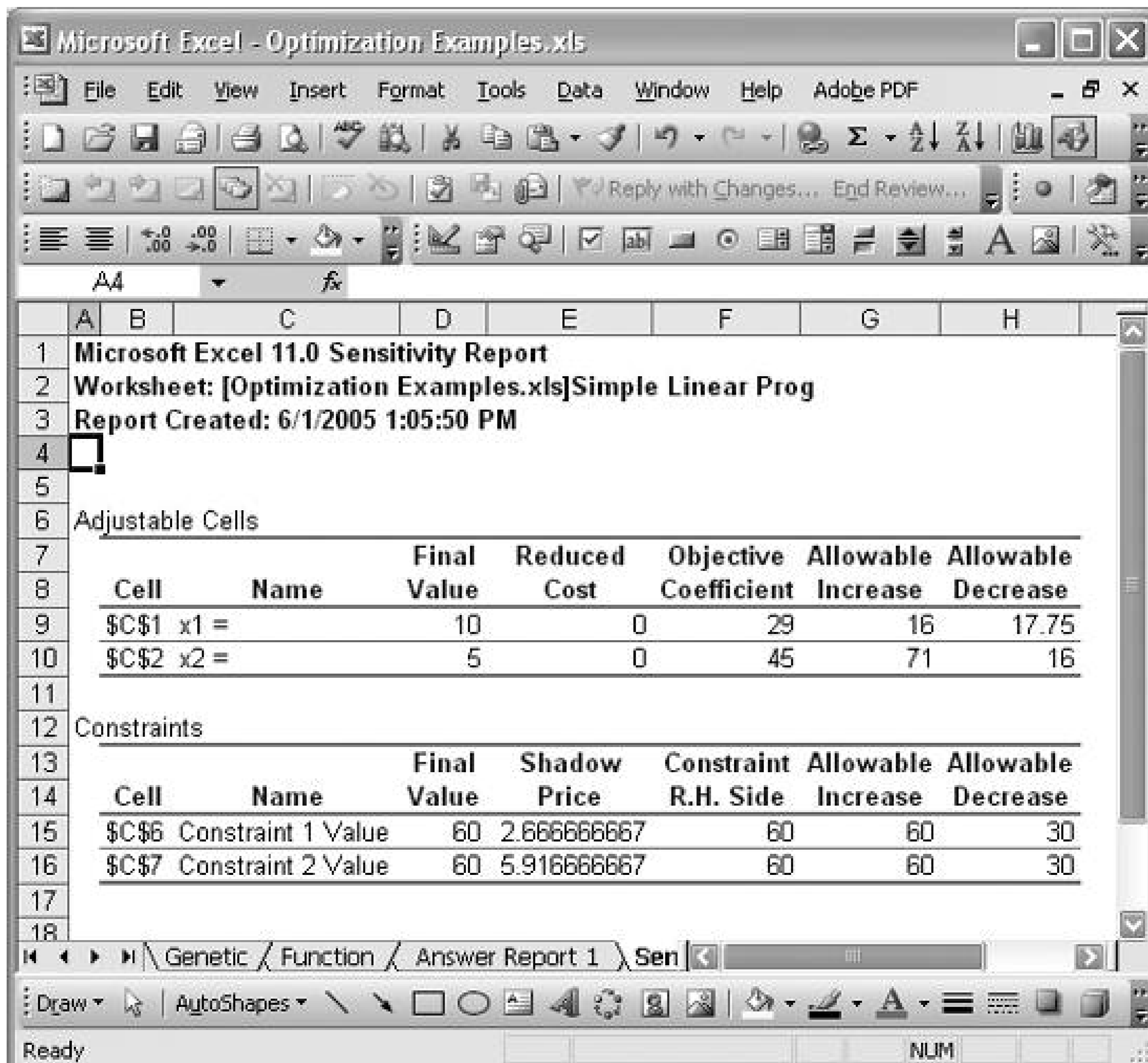
Solver builds names for target, variable, and constraint cells by searching to the left of the given cell for the first cell containing text. At the same time, it looks above the given cell for the first cell containing text. The text in these two cells is combined to create the name shown in Solver reports. Therefore, if you want a cell to have a meaningful label, whether it's a cell containing the objective

function, a variable, or a constraint, you need to make sure that the text cells nearest to the left of and above the given cell contain the desired label. I typically put text labels to the left of my target, variable, or constraints cells, and make sure no cell above these has any text.

### Sensitivity report

There are two versions of the sensitivity report. The one shown in [Figure 13-16](#) corresponds to a linear problem. Solver generates a different sensitivity report for nonlinear problems, as shown in [Figure 13-17](#). These reports tell you how sensitive the solution is to small changes in variables and constraints.

Figure 13-16. Linear sensitivity report

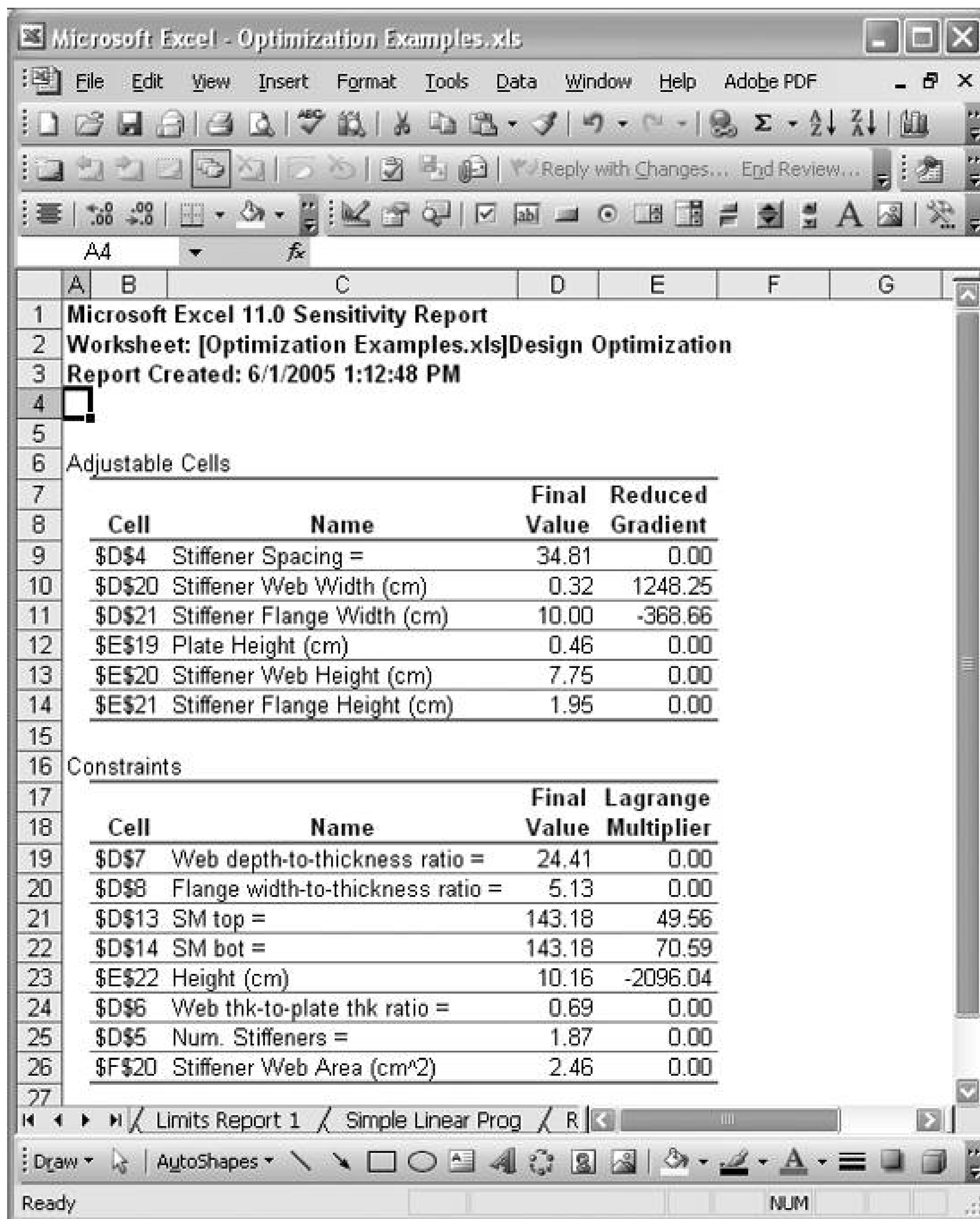


Referring to [Figure 13-16](#), you will see that the linear problem sensitivity report contains two tables. The first table provides information on the sensitivity of the problem's variables, while the second contains information on the sensitivity of the problem's constraints.

Both tables start with a cell reference column, along with a label and then a final value column. The final values are simply the resulting values of the variables or constraints corresponding to the solution found by Solver. The other columns in these tables give you the sensitivity information.

The Reduced Cost applies to variables whose optimum value turns out to be 0. In this case, reduced cost provides an estimate of how much the objective function will change if you force the variable to assume some nonzero value. Variables whose values turn out to be nonzero for the optimum solution will show a reduced cost of 0.

Figure 13-17. Nonlinear sensitivity report



The Objective Coefficient shows the coefficient corresponding to each variable as it appears in the objective function. The Allowable Increase and Allowable Decrease values tell you that so long as these coefficients remain with their current value plus or minus the allowable increase and decrease, respectively, the optimum values for the variables will remain unchanged, although the optimum value for the objective function (target cell) may change.

The Shadow Price for each constraint shown in the Constraints table tells you how much the value of

the objective function will change if the value of the righthand side of the constraint, shown in the Constraint R.H. Side column, changes within the limits specified by the Allowable Increase and Allowable Decrease values.

For nonlinear problems, the sensitivity report is a little different. [Figure 13-17](#) shows the sensitivity report for a nonlinear problem.

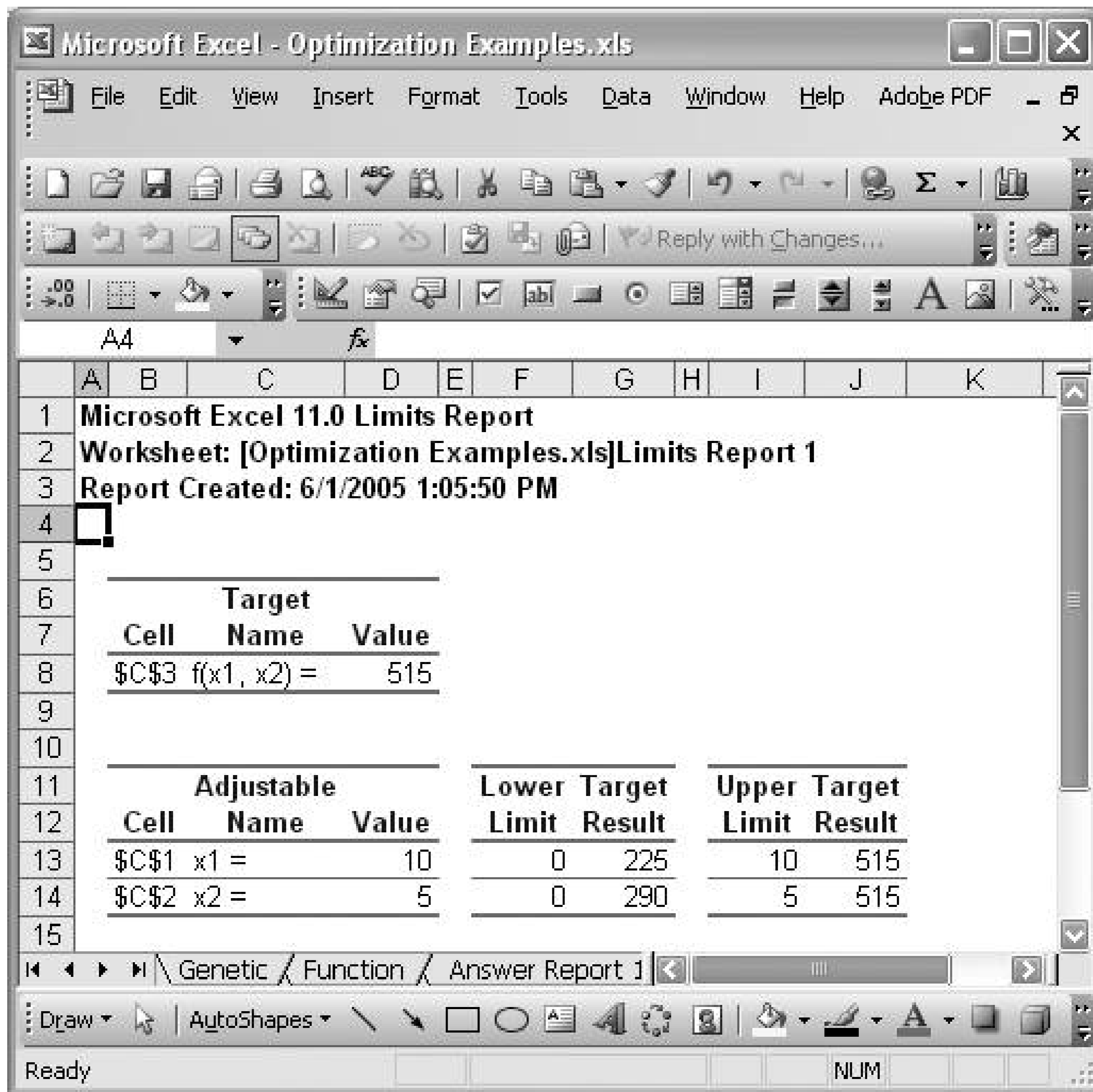
As in the linear case, there are two tables, corresponding to variables (Adjustable Cells) and constraints. In both cases the cell reference, name, and final optimum values for each variable and constraint are shown.

For variables, the Reduced Gradient tells you how much the value of the objective function would change if the value of the variable were increased by 1. For constraints, the Lagrange Multiplier tells you how much the value of the objective function would change if the righthand-side value of the constraint were increased by 1.

## Limits report

Limits reports tell you how the value of the objective function changes as each variable is maximized and minimized, while all other values are held constant and while still satisfying the problem's constraints. [Figure 13-18](#) shows a typical Limits report.

Figure 13-18. Limits report



The Target values show you the value of the objective function corresponding to the lower and upper limits of the variables obtained through a maximization and minimization process as described a moment ago.



## Recipe 13.7. Programming a Genetic Algorithm for Optimization

### Problem

You're an expert at using Solver for optimization but find it still can't handle some troublesome problems, or you'd simply like to use an alternative optimization approach.

### Solution

Genetic algorithms offer an alternative approach and are gaining popularity in optimization problem solving. You can use VBA to program your own genetic algorithm (or any other algorithm you can devise).

### Discussion

This recipe is a little longer than the others and contains a lot of VBA code. If you're not up to speed on using VBA, I recommend that you read [Chapter 2](#) first.

Also, before getting into implementation details, I want to first explain some fundamental ideas behind genetic algorithms in case you aren't already familiar with the concept.

### Genetic Algorithm Fundamentals

Genetic algorithms are inspired by the biological evolutionary process. The basic idea for optimization problems is that solutions to a problem can be "bred" from a population of candidate solutions. Repeated breeding and survival of the fittest creates successive generations of offspring whose characteristics become more and more specialized or, to say it another way, which converge toward an optimum set of characteristics.

The trick with genetic algorithms, or rather one of the many tricks, is coming up with a way to encode the variables of a problem in a meaningful way that can be used to evolve a new set of value for the variables. A set of encoded variables is called a *chromosome*, a term obviously taken directly from the genetic algorithm's biological roots. Chromosomes are comprised of *genes*, and genes represent the encoded variables for the problem.

In addition to devising a suitable structure for chromosomes, you have to devise a way to measure the *fitness* of each chromosome. In other words, you have to quantify how well the solution encoded in each chromosome solves, or optimizes, the problem you're modeling. Fitness in this sense is

directly related to the objective function of the problem.

The next big question that arises when developing a genetic algorithm is how to breed, or combine, chromosomes to generate successive populations in order to evolve a solution.

The way a genetic algorithm works is that you create an initial population of randomly generated chromosomes. The data encoded in these chromosomes will almost certainly not solve your problem well at all. The idea is to pick the fittest ones of this bunch and breed them to create a new population. You decide how chromosomes are bred. Most algorithms use some sort of crossover function, where you simply swap the genes between two parent chromosomes. For example, you might take the first half from the mommy chromosome and the second half from the daddy chromosome. Other practitioners take every other gene from the mommy and daddy. It's really up to you and depends a lot on how you encoded your chromosomes in the first place.

Once a new population has been bred, it replaces the older generation. The new population is then evaluated for fitness that is, how well each new chromosome solves your problem. Next, a new set of fit parents are selected and used to generate a whole new population of offspring. This process repeats over many generations until the bred chromosomes converge on an optimal solution to the problem.

Naturally, there's more to this than meets the eye. One big issue is what to do if you start with an initial, randomly selected population that is just terrible in that it very poorly satisfies the problem you're trying to solve. If you breed with only these chromosomes, you're left with a limited gene pool and almost surely no chance of finding a globally optimum solution.

To introduce new *genetic material*, in genetic algorithm lingo, you need to mutate the genes of offspring chromosomes. These mutations are random and provide opportunities for new characteristics to be introduced into the gene pool. Here again, there are many ways in which you may mutate a chromosome. A simple and common approach is to randomly select a gene and change it to some other randomly selected value. If it's a bad mutation, one that produces poor results or an unfit chromosome, it will be purged through the selection and evolution process. On the other hand, if it's a good mutation, it will be retained and passed along to subsequent generations. This is a key element of genetic algorithms, and we'll talk more about it shortly when I get to some implementation details.

Given this basic description of genetic algorithms, it's clear there's a great deal of leeway available for formulating a genetic algorithm. There is no single genetic algorithm, and you are free to devise one that is well suited to the problem with which you are dealing.

Developing genetic algorithms seems, to me, to consist of a little theory and a lot of voodoo. I like them because the basic concept is simple and you have so much creative leeway in formulating an approach to a given problem. Plus, they work! At the same time, you can treat certain aspects of them in a rigorous theoretical manner. If you're interested in the theory, I encourage you to check out Melanie Mitchell's *Introduction to Genetic Algorithms* (MIT Press). There's absolutely no source code in that book, but it covers the theory and a lot of history behind genetic algorithms.

The genetic algorithm I'll show in this recipe is based on the concepts discussed here, but it's tailored to solve a specific problem. The problem I'll show you how to solve is the same troublesome function discussed in [Recipe 13.4](#). If you have not already read [Recipe 13.4](#), go back and have a look at it now

to become familiar with the function we're dealing with here.

## Chromosome class

The first step in creating our Frankenstein's monster is to create a chromosome class. The encoding for this chromosome is simple since there are only two variables to change,  $x$  and  $y$ .

To create this class, which we'll call `Chromosome`, open the VBA editor window and create a new class module named `Chromosome` using the techniques I showed you in [Recipe 2.14](#) in [Chapter 2](#).

Once you've created the `Chromosome` class module, go ahead and add the properties shown here in [Example 13-2](#).

### Example 13-2. Chromosome properties

```
Public x As Double
Public y As Double
Public z As Double
Public MaxX As Double
Public MinX As Double
Public MaxY As Double
Public MinY As Double
Public MutationRate As Double
Public FitnessScore As Double
```

The first three properties are  $x$ ,  $y$ , and  $z$ . The genes in this chromosome are  $x$  and  $y$ . My encoding here is simple. The genes  $x$  and  $y$  are simply properties of the `Chromosome` class. Historically, many genetic algorithms use bit strings or some other low-level programming device to encode the problem variables. I personally like to use object properties or structures, depending on the language I'm using. I simply find them easier to deal with in code. But that's just my preference.

The next four properties store the  $x$  and  $y$  bounds for the problem. Genes will be randomly selected within the range defined by these upper and lower bounds.

The second to last property, `MutationRate`, is very important. It specifies the probability of a chromosome mutating as described earlier. This value should be tuned to achieve acceptable results. Many researchers use a value of 0.001. I've played with it for this example and find that any mutation rate between 0.001 and 0.01 seems to yield good results. If the mutation rate is too low, then new *genetic material* is introduced at a very slow rate. This means that it could take a long time for the population to converge toward an optimum solution. On the other hand, if you set the mutation rate too high, you may never converge at all, because even fit chromosomes will mutate, possibly destroying their superior fitness.

The final property required for this problem is `FitnessScore`. This property stores a value that is used to rank the chromosomes according to their fitness. I'll discuss how we actually measure fitness

shortly. That's it for `Chromosome` properties. Now I'll show you all the methods required for this class.

The `Class_Initialize` method shown in [Example 13-3](#) is the *constructor*, or initializing subroutine, for the `Chromosome` class. This subroutine is automatically called by VBA when a `Chromosome` object is created. I'm using it here to set the properties to default values.

### Example 13-3. Initialize method

```
Private Sub Class_Initialize( )
    MutationRate = 0.05
    MaxX = 5
    MinX = -5
    MaxY = 5
    MinY = -5

    x = (MaxX - MinX + 1) * Rnd + MinX
    y = (MaxY - MinY + 1) * Rnd + MinY

    EvaluateZ
End Sub
```

I set the mutation rate to 0.05 by default, but as you'll see later, this gets overridden with a value supplied through a spreadsheet cell. The bounds on  $x$  and  $y$  are set to -5 and 5. The next two lines generate random values for  $x$  and  $y$  within the default bounds and store these values in the `x` and `y` properties. Remember, these are the chromosome's genes. Finally, the method `EvaluateZ` is called to calculate  $z$  using the function we're trying to optimize, the one shown in [Recipe 13.4](#). `EvaluateZ` is shown in [Example 13-4](#).

### Example 13-4. EvaluateZ method

```
Public Sub EvaluateZ( )
    Z = (1 - Cos(x * x + y * y) / (x * x + y * y + 0.5)) * 1.25
End Sub
```

`EvaluateZ` simply computes  $z$  given the current values stored in `x` and `y`. I'll show you later how  $z$  is used to evaluate the fitness of the chromosome.

We need a method for `Chromosome` that mutates its genes given the mutation rate stored in the `MutationRate` property. The `Mutate` method shown in [Example 13-5](#) is one I devised for this purpose.

## Example 13-5. Mutate method

```
Public Sub Mutate( )
    Dim num As Double

    num = Rnd

    If num < MutationRate Then
        num = Rnd
        If num < 0.5 Then
            x = (MaxX - MinX + 1) * Rnd + MinX
        Else
            y = (MaxY - MinY + 1) * Rnd + MinY
        End If
    End If
End Sub
```

The first action performed in `Mutate` is the generation of a random number between 0 and 1. I used VBA's `Rnd` function for this purpose and stored the value in the local variable `num`. The subsequent `If` statement compares `num` to the mutation rate. If `num` is less than `MutationRate`, then you need to mutate the chromosome; otherwise, don't do anything with it.

If you do indeed need to mutate the chromosome, then `Mutate` generates another random number. If this number is less than 50, then the `x` gene is mutated; otherwise, the `y` gene is mutated. This scheme gives each gene a 50 percent chance of being mutated.

Mutating either gene is accomplished by simply generating a new random number for the gene between the bounds stored in the `x` or `y` bounds properties.

[Example 13-6](#) shows another required method, the `Mate` method.

## Example 13-6. Mate method

```
Public Sub Mate(mom As Chromosome, dad As Chromosome)
    ' Take average as crossover function
    x = (mom.x + dad.x) / 2
    y = (mom.y + dad.y) / 2

    ' Mutate the result
    Mutate
End Sub
```

As its name implies, this method takes two chromosomes and combines them in such a way as to

yield an offspring chromosome that consists of some combination of its parents' genes.

I played around with two different mating approaches for this example. First, I tried simply taking an `x` from `dad` and a `y` from `mom` to create the offspring. That worked OK, but I thought a better approach might be to simply average the `x` and `y` values, respectively, obtained from both `mom` and `dad`. After many trials, I think the averaging approach is better. It seems to converge faster to an optimum. (I'll leave the reason why for someone else to figure out.)

After the offspring chromosome's genes are computed, `Mate` calls `Mutate` to mutate the offspring.

The final method for this class is `Copy`. It's a utility method that simply copies the properties of a given chromosome into the calling chromosome. [Example 13-7](#) shows the `Copy` method.

### Example 13-7. Copy method

```
Public Sub Copy(dup As Chromosome)
    x = dup.x
    y = dup.y
    z = dup.z
    MaxX = dup.MaxX
    MinX = dup.MinX
    MaxY = dup.MaxY
    MinY = dup.MinY
    MutationRate = dup.MutationRate
    FitnessScore = dup.FitnessScore
End Sub
```

That's it for the `Chromosome` class. In the next section, I show you how to actually use this class.

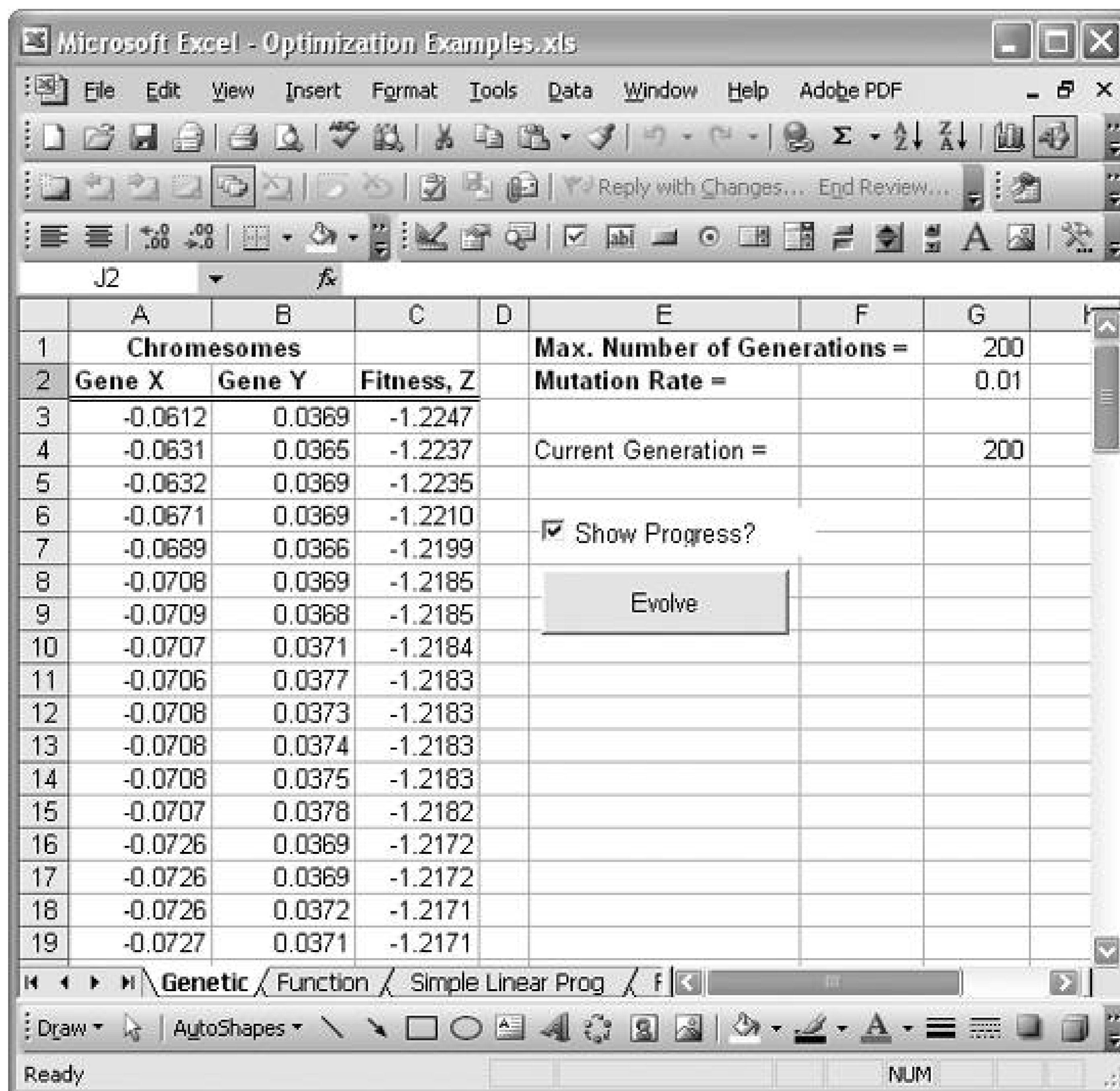
## Genetic optimization

[Figure 13-19](#) shows the spreadsheet I set up for this example.

I set this up so you need only supply two parameters to the genetic algorithm. First, you must supply a limit on the number of generations evolved. Second, you must supply the mutation rate. These two values are contained in cells G1 and G2, respectively.

The `Current Generation` value shown in cell G4 is informational. As the genetic algorithm runs, it will display the current generation in this cell so you can track its progress. Further, the entire population of chromosomes for each generation is echoed back to this spreadsheet starting in cells A1, B1, and C1. This allows you to see how the populations evolve as the genetic algorithm does its thing.

Figure 13-19. Genetic algorithm spreadsheet

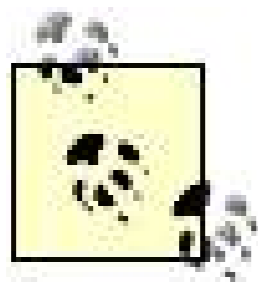


You'll notice the "Show Progress?" checkbox on the spreadsheet just above the Evolve button. This checkbox is another ActiveX control available from the Control Toolbox toolbar (see [Recipe 9.3](#) for more on the Control Toolbox).

You add a checkbox control just as you add a button control, as I've shown in other recipes throughout this book. Simply click the checkbox icon on the Control Toolbar and then click anywhere in your spreadsheet where you'd like the control located. While in design mode, you can right-click on the checkbox control and select Properties from the pop-up menu to display the control's Properties window. As with button controls, you can change the name of the control, the caption displayed on the control, and many other attributes. I changed the caption to "Show Progress?" for this checkbox.

Checkbox controls have a `value` property that can be either `true` or `false`. When this value is `true`, the checkbox is checked. When it's `false`, the checkbox is not checked. When a user clicks the checkbox, checking and unchecking it, the `value` property is changed accordingly. This allows you to programmatically inspect the `value` property to determine the state of the checkbox. I use it here to determine whether to update the spreadsheet with the status of each evolution as the genetic

algorithm runs. Displaying the progress is certainly informative, but it is a lot slower; therefore, if you want the algorithm to run faster, you can uncheck the progress checkbox.



You can also associate code with a checkbox: the code will get executed when the user clicks on the checkbox. This is achieved in a manner similar to the way in which you associate code with a button, as discussed in [Chapter 9](#).

The Evolve button allows you to start the genetic algorithm. I added this button using the same techniques I showed you in [Chapter 9](#). The click event handler for this button is shown in [Example 13-9](#); but, before I explain what it does, I want to go over some global variables required for the genetic algorithm. [Example 13-8](#) shows these global variables.

### Example 13-8. Global variables

```
Const PopulationSize As Integer = 100

Dim MaxGenerations As Integer
Dim MutationRate As Double
Dim CurrentGeneration(1 To PopulationSize) As Chromosome
Dim Offspring(1 To PopulationSize) As Chromosome
Dim SumFitnessScores As Double
```

The first line in [Example 13-8](#) isn't really a variable—it's a constant. The constant `PopulationSize` is an integer value that I set to 100. This means the initial and all subsequent chromosome populations will consist of 100 chromosomes. This is also a tunable value. If you set this too low, then the population won't be diverse enough, and convergence to an optimum solution could be slow or unsuccessful. If it's too high, the algorithm will run more slowly. 100 seemed like a good size for this problem. I tried a population size as low as 10 and as high as 200. In the former case, convergence was poor; in the latter case it was slower and no more accurate than a size of 100. So I stuck with 100.

The next five lines are actual variable declarations. These variables have module-level scope, so any function or subroutine in the same code module has access to them. Each variable is described in the following list:

#### MaxGenerations

`MaxGenerations` is an `integer` variable representing the limit on the number of generations to evolve. This value is extracted from spreadsheet cell G1.

#### MutationRate

`MutationRate` represents the probability of mutation as discussed earlier. Its value is extracted



from spreadsheet cell G2.

### CurrentGeneration

`CurrentGeneration` is an array of `Chromosome` objects. The size of this array is `PopulationSize`. This array represents the entire population of `Chromosomes` comprising the current generation.

### Offspring

`Offspring` is another array of `Chromosome` objects. Its size is also `PopulationSize`. This array is used to store offspring chromosomes bred from the current generation. After all offspring are created, they replace the current generation, becoming the new `CurrentGeneration`.

### SumFitnessScores

`SumFitnessScores` is a `Double` variable. It's used to store the sum of fitness scores for all chromosomes in the current generation. Its value is used in a *roulette wheel*/selection process that I'll show you later.

Now, let's get back to the click event handler for the Evolve button. The handler is shown in [Example 13-9](#).

## Example 13-9. Evolve button's click event handler

```
Private Sub EvolveButton_Click( )
    Dim i As Integer

    Application.ScreenUpdating = ShowProgressCheckBox.Value

    Randomize

    MaxGenerations = Worksheets("Genetic").Range("MaxGenerations")
    MutationRate = Worksheets("Genetic").Range("MutationRate")

    For i = 1 To PopulationSize
        Set CurrentGeneration(i) = New Chromosome
        Set Offspring(i) = New Chromosome

        CurrentGeneration(i).MutationRate = MutationRate
    Next

    DoGeneticAlgorithm

    Application.ScreenUpdating = True
    Application.ActiveWorkbook.Save

End Sub
```

This subroutine is responsible for initializing and kicking off the genetic algorithm. After creating a local counter variable, `i`, the subroutine sets the status of screen updating to the value represented in the "Show Progress?" checkbox. The line `Application.ScreenUpdating = ShowProgressCheckBox.Value` takes care of that task. If the `value` property for the progress checkbox is `true` (indicating it's checked) then screen updating will occur; otherwise, it won't.

The next line of code simply initializes VBA's random number generator with a call to `Randomize`. Next, the values for `MaxGenerations` and `MutationRate` are extracted from cells G1 and G2 in the spreadsheet. I named these two cells `MaxGenerations` and `MutationRate`, respectively (see [Recipe 1.14](#) to learn how to name cells).

The `For` loop that comes next is responsible for creating the two required populations of `Chromosome` objects. Remember, the array declarations for `CurrentGeneration` and `Offspring` don't actually create the `Chromosome` object. You have to use the `Set` and `New` statements as shown in [Example 13-9](#) (and as discussed in [Recipe 2.14](#)) to create memory for the objects, initialize them, and assign them to the arrays.

After the arrays are taken care of, the mutation rate property for the current generation of chromosomes is set to the value provided in cell G2 and stored in the variable `MutationRate`.

After the `For` loop, a call is made to `DoGeneticAlgorithm`, which, as its name implies, executes the genetic algorithm. This subroutine is shown in [Example 13-10](#). Before getting there, let me just say that the last two lines of code in [Example 13-9](#) are responsible for turning screen updating back on and saving the workbook. Now, on to `DoGeneticAlgorithm`.

### Example 13-10. DoGeneticAlgorithm subroutine

```
Public Sub DoGeneticAlgorithm( )
    Dim i As Integer
    Dim Generation As Integer
    Dim mom As Integer
    Dim dad As Integer

    Generation = 0
    EvaluateFitness
    DisplayResults

    Do While (Generation <= MaxGenerations)
        Worksheets("Genetic").Range("Generation") = Generation

        For i = 1 To PopulationSize
            mom = SelectChromosome
            dad = SelectChromosome
            Offspring(i).Mate CurrentGeneration(mom), CurrentGeneration(dad)
        Next
    End While
End Sub
```

```

        For i = 1 To PopulationSize
            CurrentGeneration(i).Copy Offspring(i)
        Next

        Generation = Generation + 1
        EvaluateFitness
        DisplayResults
    Loop

    ' Sort the results:
    Range("A3:C102").Select
    Selection.Sort Key1:=Range("C3"), Order1:=xlAscending, Header:=xlGuess, _
        OrderCustom:=1, MatchCase:=False, Orientation:=xlTopToBottom, _
        DataOption1:=xlSortNormal
End Sub

```

`DoGeneticAlgorithm` requires four local variables. `i` is simply a counter variable used to loop over array elements. `Generation` is a counter variable used to keep track of the number of generations evolved. This value gets echoed back to the spreadsheet in cell G4.`mom` and `dad` are integers used to store indices to array elements in `CurrentGeneration` that represent two chromosomes selected for breeding.

With the local variables taken care of, `DoGeneticAlgorithm` initializes a few things. `Generation` is set to 0, and calls to `EvaluateFitness` and `DisplayResults` are made. `EvaluateFitness` updates the fitness scores for the chromosomes in the current generation, and `DisplayResults` displays the current generation's genes and  $z$ -value in the spreadsheet. We'll come back to the details of these two subroutines later.

`DoGeneticAlgorithm` next enters a `Do While` loop. This loop will repeat until the number of generations evolved reaches the limit stored in `MaxGenerations`. It is within this `Do While` loop that the evolution of a solution takes place.

The first action taken in this loop is simply to display the current generation number in the spreadsheet. Next, a `For` loop is encountered; it is responsible for selecting candidate chromosomes from the current generation and breeding them to create the new offspring population.

The integers `mom` and `dad` are selected via calls to a function called `SelectChromosome`. `SelectChromosome` selects candidates from the current generation, where the probability of selecting any individual chromosome is proportional to the chromosome's fitness score. I'll show you the details later. For now, just know that this is a survival-of-the-fittest selection process.

After two parents are selected, they are mated by calling the offspring `Chromosome`'s `Mate` method. Here's the line of code that calls the `Mate` method:

```
Offspring(i). Mate CurrentGeneration(mom), CurrentGeneration(dad)
```

This line calls the `Mate` method of `Offspring(i)`, which is the  $i$ th `Chromosome` stored in the `Offspring`

array. `Mate` requires two arguments, a mom and a dad `Chromosome`. These arguments are `CurrentGeneration(mom)` and `CurrentGeneration(dad)`. Notice that there are no parentheses around these arguments, unlike when you make a function call or procedure call in most other languages. If you put parentheses around these arguments, VBA will return an error.

When this `For` loop is finished, `Offspring` will contain an entirely new population of chromosomes. The next task is to make this new population the `CurrentGeneration`. This task is achieved in the next `For` loop, which calls the `Copy` method for each chromosome in `CurrentGeneration` where the corresponding chromosome (based on the array index) in the `Offspring` array is passed as the chromosome to copy.

After all the chromosomes have been copied, the new population stored in `CurrentGeneration` is updated. First, the generation counter is incremented by 1. Then a call to `EvaluateFitness` is made to update the fitness scores for the new population. Finally, `DisplayResults` is called to display the current population in the spreadsheet.

At this point, execution goes back to the beginning of the `Do While` loop and repeats until the maximum number of generations has been reached. When that occurs, the last few lines of code in [Example 13-10](#) are executed.

As the comment suggests, these lines sort the population displayed in cells A3 to C102 in the spreadsheet. This population represents the fittest population at the end of the evolution process. Not all of the chromosomes displayed in this list are the most fit and some are more fit than others. So to see which are the fittest, I sort the list to display the chromosomes in order from the fittest down to the least fit. This way you need only look at the uppermost values in the spreadsheet to see the results.

## Harvesting Autogenerated VBA Code

The code in [Example 13-10](#) that takes care of sorting was actually generated by Excel. I didn't know the VBA code to perform a sort; rather than look it up in the help guide, I went into Excel and recorded a macro and harvested the relevant code from the macro code created by Excel. Here's how I did it: first, I went to my spreadsheet in Excel and turned on macro recording (see [Recipe 1.16](#)). Then I selected the cell range I wanted to sort (cells A3 through C102) and sorted it in descending order based on the  $z$ -values contained in column C (see [Recipe 3.9](#)). Next, I turned macro recording off. Then I went to the VBA editor and looked for the new code module created in my VBA project. It was called `Module1` by default. This module contained a few lines of VBA code, corresponding to the macro just recorded. I copied and pasted the sort code from this module to the subroutine shown in [Example 13-10](#). This is a handy trick when you can't figure out the VBA code to perform some action; just let Excel do it for you and copy and paste as needed.

Now let's take a look at the subroutine `EvaluateFitness`. As I mentioned earlier, this subroutine is responsible for assessing the fitness of each chromosome in the current population. `EvaluateFitness` first computes the  $z$ -value for each chromosome and then gives each chromosome a fitness score based on its  $z$ -value relative to the other  $z$ -values in the population. Remember, we're trying to

minimize  $z$  for the objective function; therefore, a lower  $z$ -value is better. So any chromosome whose genes ( $x$  and  $y$ ) yield a lower  $z$ -value is more fit than a chromosome whose genes yield a higher  $z$ -value. We want these fitter chromosomes to reproduce so they'll get a higher fitness score. [Example 13-11](#) shows how I constructed `EvaluateFitness`.

### Example 13-11. EvaluateFitness subroutine

```
Public Sub EvaluateFitness( )
    Dim i As Integer
    Dim MinZ As Double
    Dim MaxZ As Double

    CurrentGeneration(1).EvaluateZ
    MinZ = CurrentGeneration(1).Z
    MaxZ = CurrentGeneration(1).Z

    For i = 2 To PopulationSize
        CurrentGeneration(i).EvaluateZ
        If CurrentGeneration(i).Z > MaxZ Then
            MaxZ = CurrentGeneration(i).Z
        End If
        If CurrentGeneration(i).Z < MinZ Then
            MinZ = CurrentGeneration(i).Z
        End If
    Next

    SumFitnessScores = 0
    For i = 1 To PopulationSize
        CurrentGeneration(i).FitnessScore = 1 - _
            (CurrentGeneration(i).Z - MinZ) / (MaxZ - MinZ)

        SumFitnessScores = SumFitnessScores + CurrentGeneration(i).FitnessScore
    Next
End Sub
```

`EvaluateFitness` first declares a few local variables. Here again,  $i$  is simply a counter variable. `MinZ` and `MaxZ` are used to store the minimum and maximum  $z$ -values found in the current population. They represent the range of  $z$ -values for the population. They are initialized to the  $z$ -value for the first chromosome in the population. Then a `For` loop iterates over all remaining chromosomes in the current population. Within the `For` loop, each chromosome's  $z$ -value is computed and compared to the minimum and maximum  $z$ -values stored in `MinZ` and `MaxZ`. If the chromosome's  $z$ -value is higher or lower than the currently stored range, the range gets updated accordingly. By the time this `For` loop ends, `MinZ` and `MaxZ` will contain the range for the entire population.

Before the subroutine enters the next `For` loop, it sets `SumFitnessScores` to 0. Within the next `For` loop, `SumFitnessScores` will tally all of the fitness scores for every chromosome in the population.

The next `For` loop iterates over the entire current population, computing a fitness score for each chromosome. The formula I came up with for the fitness score is:

$$\text{score} = 1 - \frac{z - z_{\min}}{z_{\max} - z_{\min}}$$

where  $z_{\min}$  and  $z_{\max}$  represent `MinZ` and `MaxZ`, respectively.

This formula gives the chromosome with the highest  $z$ -value a fitness score of 1 and the chromosome with the lowest  $z$ -value a fitness score of 0. All other chromosomes will have a score somewhere between 0 and 1. As I mentioned earlier, `SumFitnessScores` computes a running total of these fitness scores.

That's it for the fitness score subroutine. Now let's turn our attention to the function responsible for selecting the fittest chromosomes, based on their fitness scores. The function I wrote for this task is called `SelectChromosome` and is shown in [Example 13-12](#).

### Example 13-12. SelectChromosome function

```
Public Function SelectChromosome( ) As Integer
    Dim FitnessValue As Double
    Dim FitnessTotal As Double
    Dim i As Integer
    Dim Selection As Integer
    Dim continue As Boolean

    FitnessValue = Rnd * SumFitnessScores
    FitnessTotal = 0
    continue = True

    i = 1
    Do While (i <= PopulationSize) And (continue = True)
        FitnessTotal = FitnessTotal + CurrentGeneration(i).FitnessScore

        If FitnessTotal > FitnessValue Then
            Selection = i
            continue = False
        End If
        i = i + 1
    Loop
    SelectChromosome = Selection
End Function
```

This function selects chromosomes using a standard roulette wheel algorithm. The idea is to select chromosomes with a probability of selection proportional to the chromosomes' fitness scores. Chromosomes with the highest fitness scores have greater probabilities of being selected, while those with lower scores have lower probabilities of being selected.

`SelectChromosome` implements the algorithm as follows: a random number between 0 and 1 is obtained and multiplied by the sum of all fitness scores. The result is stored in the variable `FitnessValue`. Next, a `Do While` loop is entered to iterate over the current population. During each iteration an accumulation variable, `FitnessTotal`, accumulates the fitness scores for each chromosome. When this `FitnessTotal` exceeds the randomly obtained `FitnessValue`, we have a selection. The chromosome whose fitness score tipped `FitnessTotal` over `FitnessValue` is the chromosome to select, and its index gets returned by `SelectChromosome`.

That's pretty much it for the genetic algorithm portion of the code. The remaining function I want to show is `DisplayResults`. As mentioned earlier, it simply populates the spreadsheet (cells A3 through C102) with chromosome data for the current population. [Example 13-13](#) shows the subroutine.

### Example 13-13. DisplayResults subroutine

```
Public Sub DisplayResults( )
    Dim i As Integer
    Dim r As Integer

    r = 3
    For i = 1 To PopulationSize
        With Worksheets("Genetic")
            .Cells(r, 1) = CurrentGeneration(i).x
            .Cells(r, 2) = CurrentGeneration(i).y
            .Cells(r, 3) = CurrentGeneration(i).z
        End With
        r = r + 1
    Next
End Sub
```

As you can see, this function simply iterates over the current population, copying each chromosome's  $x$ -,  $y$ -, and  $z$ -values to the corresponding cells in the spreadsheet.

## Results

[Figure 13-19](#) shows some results of this genetic algorithm, with a mutation rate set at 0.01 and the number of generations limited to just 200. The minimum  $z$ -value obtained in this case is -1.2247. That's not too far off from the true global minimum for the subject function of -1.25. You can, however, achieve better results by increasing the number of allowable generations, giving the solution more time to evolve.

For example, setting the mutation rate to 0.01 and the maximum number of generations to 10,000 yields a minimum  $z$ -value of -1.2498. This run took only a few minutes on a typical desktop computer with "Show Progress?" turned off. I tried many combinations of mutation rate and number of generations. With mutation rates in the range of 0.001 to 0.01 and generation numbers ranging from 200 to 10,000, the algorithm converges to within a few percent, or fractions of a percent, of the true

global optimum every time.

This is a key benefit of such a genetic algorithm: it converges towards the global minimum every time. The algorithm I showed you in [Recipe 13.4](#), where I used Solver in a Monte Carlo fashion, did not always converge to the global solution. It did the vast majority of the time, but there were a very small percentage of cases where it didn't.

Conversely, when the Solver-based algorithm did converge, it did so more accurately in that the exact global minimum at  $(x, y) = (0, 0)$  was found. Another downside of genetic algorithms is that since they incorporate random elements, they don't always yield the same solution each time. For example, one run with the maximum generations set to 10,000 and with a mutation rate set to 0.01 returned a minimum  $z$ -value of -1.2419, while another run using the same settings returned a value of -1.2488.

On the other hand, genetic algorithms are robust and don't require gradient-based computations; plus, they can handle very nonlinear problems in many cases better than a gradient-based approach can. Which sort of method is best suited to a particular problem is very problem-dependent, but at least you now have two powerful techniques at your disposal: genetic algorithms as shown here and Solver-based approaches as discussed earlier.

## See Also

If you have not already done so, read [Recipe 13.4](#) to see the same problem solved using a Solver-based approach.



# Chapter 14. Introduction to Financial Calculations

[Section 14.0. Introduction](#)

[Recipe 14.1. Computing Present Value](#)

[Recipe 14.2. Calculating Future Value](#)

[Recipe 14.3. Figuring Out Required Rate of Return](#)

[Recipe 14.4. Doubling Your Money](#)

[Recipe 14.5. Determining Monthly Payments](#)

[Recipe 14.6. Considering Cash Flow Alternatives](#)

[Recipe 14.7. Achieving a Certain Future Value](#)

[Recipe 14.8. Assessing Net Present Worth](#)

[Recipe 14.9. Estimating Rate of Return](#)

[Recipe 14.10. Solving Inverse Problems](#)

[Recipe 14.11. Figuring a Break-Even Point](#)

## 14.0. Introduction

This chapter is a departure from all the material presented earlier in this book. While this book does focus on scientific and engineering types of calculations, I felt it would be appropriate to at least introduce you to what Excel has to offer in the way of financial calculations. After all, even as scientists and engineers, we do have to worry about money, whether we're preparing bids and proposals, justifying the purchase of new laboratory equipment, or figuring out our retirement benefits.

In practice, you can devise all sorts of financial or economic models in Excel for a huge variety of problems. The large number of financial model templates available over the Internet attests to this fact. To make some common calculations easier, Excel offers several dozen built-in financial formulas. These built-in formulas handle calculations ranging from amortization to depreciation to treasury bill yield calculations, and many others in between. You can find a list of all these functions in Excel's help documents. Look for the help topic "Financial Functions" (from the help Table of Contents, look for the topic "Working with Data" → Function Reference Financial Functions Financial Functions). There you'll find a list of all built-in financial functions with short descriptions and links to more detailed information for each one.

There's little doubt that Excel is used more for these sorts of calculations than for pure scientific and engineering calculations. Further, there are plenty of good books that show you in excruciating detail how to leverage Excel for business and financial calculations. My aim in this chapter is not to try to compete with those sources, but to introduce you, by way of a few examples, to how you can leverage Excel for financial calculations.

[← PREVIOUS](#)

## Recipe 14.1. Computing Present Value

### Problem

You need to determine the present value of a future dollar amount, considering the time value of money.

### Solution

Use the built-in function `PV`.

### Discussion

Consider this example: what is the present value of a \$5,000 payment to occur three years from now if the annual interest rate is 7%? You can use the present value, `PV`, function to find out. The syntax for `PV` is `=PV(rate, nper, pmt, fv, type)`, where `rate` is the interest rate per period, `nper` is the number of payment periods, `pmt` is the payment per period, `fv` is the future value, and `type` is an indicator of the payment type (0 for payments at the end of each pay period or 1 for payments at the beginning of each pay period).

You need to keep the interest rate and pay periods consistent when using this formula. For example, if you're using a 7% annual interest rate, then your pay period is in years. If your payments are each month, then you need to use the monthly interest rate.

To compute the present value for our example, enter the formula `=PV(0.07, 3, 0, 5000)` in a cell. The result comes to -\$4,081.49. Excel generates this as a negative number, implying that this is what you would have to invest now at the given rate to have a future value of \$5,000 three years from now. You can get rid of the negative by taking the absolute value, using the `ABS` function (see [Chapter 7](#)).

### See Also

Take a look at the other recipes in this chapter to learn about other calculations of the time value of money.

[← PREVIOUS](#)

## Recipe 14.2. Calculating Future Value

### Problem

You need to compute the future value of a series of payments, considering the time value of money.

### Solution

Use the `FV` function.

### Discussion

`FV` performs essentially the reverse of `PV`, which was discussed in the previous recipe. Consider this example: what is the future value of \$500 monthly payments over five years at an annual 8% interest rate? You can use `FV` to find an answer. The syntax for `FV` is `=FV(rate, nper, pmt, pv, type)`, where `rate` is the interest rate per period, `nper` is the number of payment periods, `pmt` is the payment per period, `pv` is the present value, and `type` is an indicator of the payment type (0 for payments at the end of each pay period or 1 for payments at the beginning of each pay period).

For our example, the cell formula `=FV(0.08/12, 5*12, -500, 0, 0)` returns a future value of \$36,738.43. The interest rate passed in as the first argument in the call to `FV` is `0.08/12` to compute the monthly rate, since payments are made every month. Further, the number of payments over the five-year period was entered as `5*12`. Also, the \$500 monthly payment was entered as a negative value, indicating it's an out payment.

### See Also

Take a look at the other recipes in this chapter to learn about other calculations of the time value of money.

[← PREVIOUS](#)

## Recipe 14.3. Figuring Out Required Rate of Return

### Problem

You want to figure out the required rate of return for some given present value to grow to a desired future value.

### Solution

Use the `RATE` function.

### Discussion

Let's say you have \$5,000 now and want to know what rate of return would be required for that sum to grow to \$15,000 in 10 years if interest is compounded annually. The `RATE` function can figure this out for you.

The syntax for `RATE` is `=RATE(nper, pmt, pv, fv, type, guess)`, where *nper* is the number of payment periods, *pmt* is the payment per period, *pv* is the present value, *fv* is the future value, *type* is an indicator of the payment type (0 for payments at the end of each pay period or 1 for payments at the beginning of each pay period), and *guess* is an initial guess at the required interest rate.

`RATE` performs an iterative search for the required rate and your initial *guess* helps it converge. *guess* is actually an optional parameter and you can omit it if you'd like.

For our example, the formula `=RATE(10, 0, -5000, 15000, 0, 0.1)` returns a required interest rate of 11.61%. Notice that I entered the present value as a negative value, indicating an out payment. The signs for *pv* and *fv* must be opposite (*fv* can be 0); otherwise, `RATE` returns an error.

### See Also

Take a look at the other recipes in this chapter to learn about other calculations of the time value of money.



## Recipe 14.4. Doubling Your Money

### Problem

You want to figure out how long it will be before an initial investment will double, given an annually compounded interest rate.

### Solution

Use the `NPER` function.

### Discussion

Let's say you invest \$5,000 now at an annual interest rate of 8% and want to know how many years it will be until the investment doubles, assuming interest is compounded annually. You can use `NPER` to figure this out.

The syntax for `NPER` is `=NPER(rate, pmt, pv, fv, type)`, where *rate* is the interest rate per period, *nper* is the number of payment periods, *pmt* is the payment per period, *pv* is the present value, *fv* is the future value, and *type* is an indicator of the payment type (0 for payments at the end of each pay period or 1 for payments at the beginning of each pay period).

The formula for our example is `=NPER(0.08, 0, -5000, 10000, 0)`, which returns a value of 9.01 years.

You can, of course, compute how long before your investment triples, quadruples, and so on, by entering the appropriate future value. In this example, I entered twice the present value. Notice that I change signs between present and future values as well.

[← PREVIOUS](#)

## Recipe 14.5. Determining Monthly Payments

### Problem

You're borrowing money and want to figure out what your monthly payments will be, given the principal amount and interest rate.

### Solution

Use the `PMT` function.

### Discussion

Let's say you're borrowing \$25,000 and the term of the loan is five years. Payments are due at the end of each month and the annual interest rate is 3.9%. What are your required monthly payments?

`PMT` can figure this out for you. The syntax for `PMT` is `=PMT(rate, nper, pv, fv, type)`, where *rate* is the interest rate per period, *nper* is the number of payment periods, *pmt* is the payment per period, *pv* is the present value, *fv* is the future value, and *type* is an indicator of the payment type (0 for payments at the end of each pay period or 1 for payments at the beginning of each pay period).

For this example, the formula `=PMT(0.039/12, 12*5, 25000, 0, 0)` returns -\$459.29, which means you'd have to pay \$459.29 per month to cover principal and interest.

### See Also

Excel includes a loan amortization template that's very handy when investigating loan options. Select `File > New ...` from the main menu bar to open the New Workbook Task Pane on the right side of the window. Under `Templates`, select "On My Computer ..." to open the `Templates` dialog box. Select the `Loan Amortization` template and press `OK` to open the template. Enter your loan parameters and the template will automatically compute monthly payments, along with principal and interest summaries over the term of the loan.

## Recipe 14.6. Considering Cash Flow Alternatives

### Problem

You're considering two different projects that will generate cash flow streams over a period of years, and each project requires some start-up costs. You want to decide which project offers the highest present value (making it the more attractive option).

### Solution

You can set up a cash flow table rather easily using basic spreadsheet functions. Further, you can use the time value of money formulas discussed to *discount* the future values to present dollars, forming a present value basis of comparison.

### Discussion

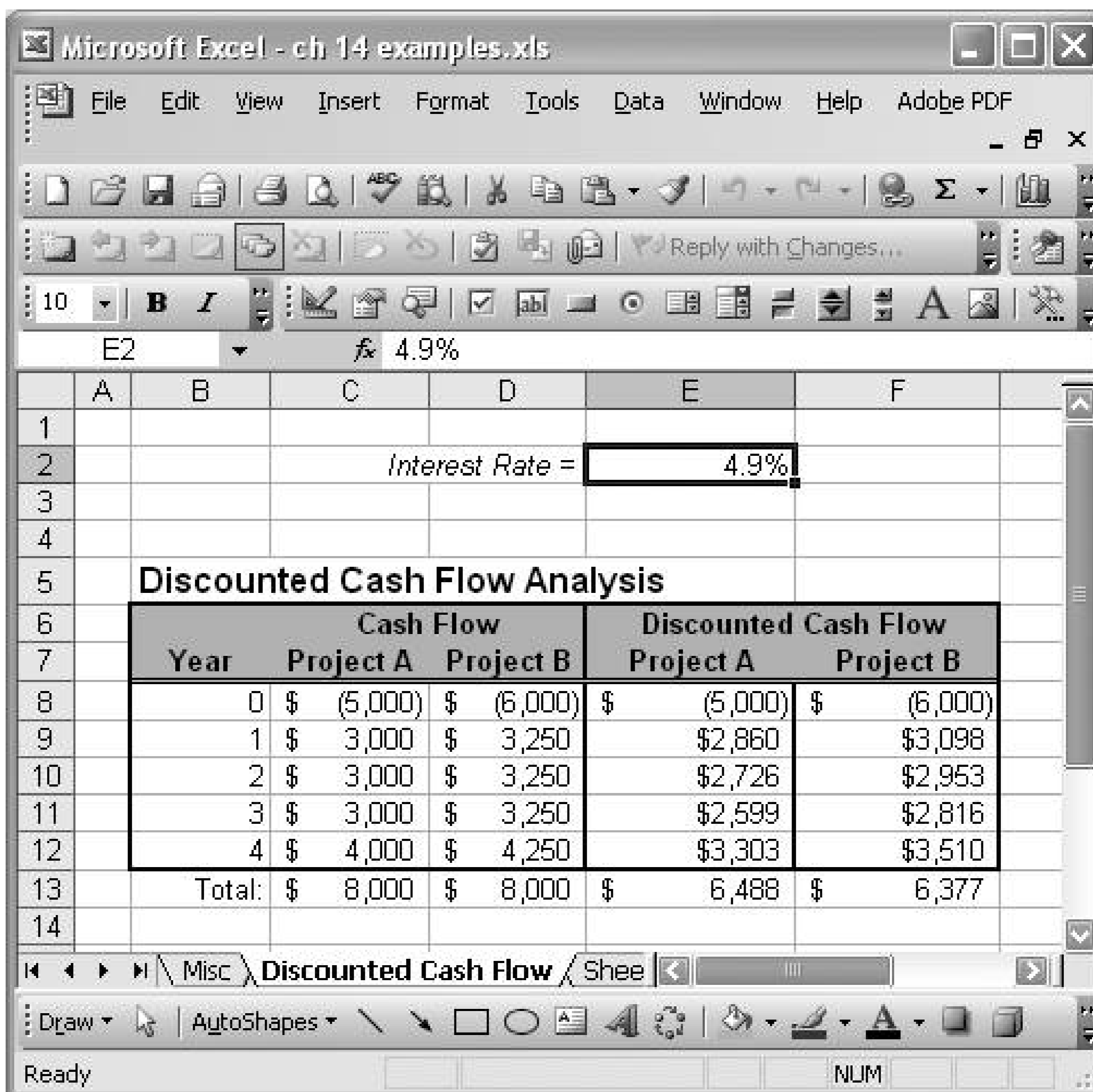
As an example, let's assume that project A requires an initial outlay of \$5,000 during the first year and will generate a positive cash flow of \$3,000 per year for the next three years and \$4,000 during the fifth year. Project B requires an initial outlay of \$6,000 during the first year and will generate a positive cash flow of \$3,250 per year for the next three years and \$4,250 during the fifth year. We'll assume the time value of money (interest rate) is 4.9%.

If you sum the initial outlay and cash flows for each project, you'll find the sum is \$8,000 for each project. Therefore, it would seem they are both equally attractive. However, the story is somewhat different when you consider the time value of money.

What's required is to discount the future cash flow values using the `PV` function (see [Recipe 14.1](#)) to derive cash flow sums for each project in terms of present dollars. [Figure 14-1](#) shows a simple spreadsheet I set up to make these calculations.

Figure 14-1. Discounted cash flow





Cell E2 contains the interest rate (4.9%) to be used as the first argument in calls to **PV**.

Normally, you have to enter percentages in decimal format like 0.049 for 4.9%, 0.1 for 10%, and so on. However, if you format a cell as a percentage (by using **Format Cells ... Category Percentage** or by entering % after a number in a cell), then you can enter percentages as 4.9, 10, and so on. Excel will take care of the 1/100 factor for you.

The table contained in cells B6 to F13 consists of columns showing the year and cash flow for project A and for project B. The first two cash flow columns are not discounted. The values in these columns were entered manually. The last two cash flow columns are discounted.

The cash flow value for year 0 is not discounted, whereas the cash flows for years 1 through 4 are discounted. The formulas in these cells look like **=PV(\$E\$2,\$B9,0,-C9,0)**. The first argument is the interest rate contained in cell E2. The second argument refers to the corresponding year for the particular cash flow value, which is passed in as the fourth argument. I took the negative of the cash flow value so the discounted results come out positive.

The totals in the last row of the table are computed using the `SUM` formula (see [Chapter 7](#)). As it turns out, project A's present value is higher than that of project B. The difference is about 2% and makes project A the more attractive option even though when you look at the nondiscounted cash flows, they both appear to generate the same total cash flow.

## See Also

Excel offers a function called `NPV` that discounts a series of cash flow values in one function call. See [Recipe 14.8](#) for more information.



[← PREVIOUS](#)

## Recipe 14.7. Achieving a Certain Future Value

### Problem

You want to put away a sum of money annually at a given interest rate in order to achieve a specific future value to cover a known upcoming expense. You need to know how much to put away each year.

### Solution

Use the `PMT` function.

### Discussion

By way of example, let's assume you need \$150,000 in 15 years' time and you want to invest a sum of money annually to raise this money. You're confident you can get an annual interest rate on your investment of 7.6%. How much do you invest each year?

You can use the `PMT` function (see [Recipe 14.5](#)) to compute the annual amount to invest. The formula `=PMT(0.076, 15, 0, 150000, 0)` returns a payment value for this example of \$5,698.76 per year.

### See Also

See [Recipe 14.5](#) for more information on the syntax for `PMT`.

## Recipe 14.8. Assessing Net Present Worth

### Problem

You're considering certain equipment upgrades in your lab and you want to perform a net present worth analysis of the projected cash flow streams from these upgrades to determine which option presents the best alternative.

### Solution

You can use the same techniques discussed in [Recipe 14.6](#) to discount the projected cash flow streams in terms of present values. The option with the highest net present value is the best choice.

### Discussion

Consider this example: you're currently running a numerical simulation laboratory that uses an aging supercomputer to run your simulations. You've forecast cash flows for your simulation services over the next five years. Now you're considering purchasing one of two candidate cluster-based systems that will boost your forecast cash flow over the next five years. These new clusters cost money, though, so you're not sure which one, if either, offers the best choice economically speaking. You need to decide whether to purchase system A, system B, or nothing at all.

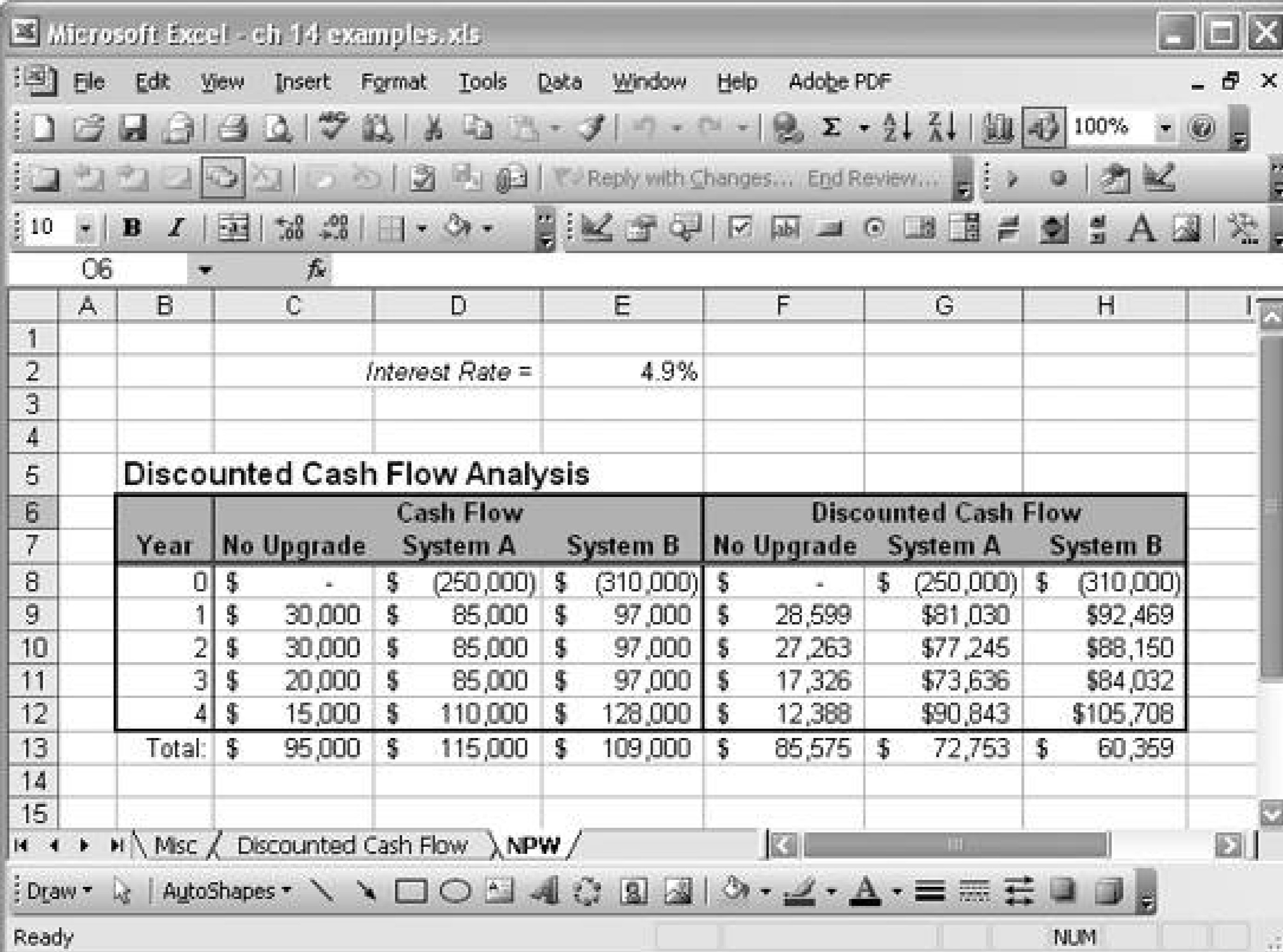
Let's further assume that system A costs \$250,000 and system B, which is somewhat more powerful than system A, costs \$310,000. Also, at the end of the fifth year you can sell either system for salvage at 10% of its original cost.

You've worked out cash flow forecasts for these two options as shown in [Figure 14-2](#).

The **No Upgrade** option means you don't upgrade your system, but instead continue offering services over the next five years using your current system. In this case, your cash flow is projected to decrease somewhat over the five-year period due to anticipated increases in operating and maintenance costs. Forecast cash flows for the system A option are expected to be fairly high, and the cash flow in the fifth year includes the 10% salvage value. Similarly, system B is expected to generate more cash flow due to its higher performance capabilities. The fifth-year cash flow value for system B also includes the 10% salvage value.

Examination of the net cash flow values shown in the **Total** row for the first three cash flow columns in the table in [Figure 14-2](#) gives the impression that upgrading to system A offers the best choice. This is before discounting and figuring out the net present value of each option.

Figure 14-2. Net present worth example



Discounted Cash Flow Analysis							
Year	No Upgrade	System A	System B	No Upgrade	System A	System B	
0	\$ -	\$ (250,000)	\$ (310,000)	\$ -	\$ (250,000)	\$ (310,000)	
1	\$ 30,000	\$ 85,000	\$ 97,000	\$ 28,599	\$ 81,030	\$ 92,469	
2	\$ 30,000	\$ 85,000	\$ 97,000	\$ 27,263	\$ 77,245	\$ 88,150	
3	\$ 20,000	\$ 85,000	\$ 97,000	\$ 17,326	\$ 73,636	\$ 84,032	
4	\$ 15,000	\$ 110,000	\$ 128,000	\$ 12,388	\$ 90,843	\$ 105,708	
Total:	\$ 95,000	\$ 115,000	\$ 109,000	\$ 85,575	\$ 72,753	\$ 60,359	

Discounting these cash flows uses the same formulas shown in [Recipe 14.6](#). The formulas in the last three cash flow columns, the discounted columns, are of the form  $=PV(\$E\$2, \$B9, 0, -C9, 0)$ , where the present value function,  $PV$ , is used to compute the present value of each year's cash flow amount for each option.

Net present values are represented by the totals for the last three discounted cash flow columns. Examining these totals reveals that purchasing system A is not the best option as originally thought. Instead the **No Upgrade** option offers the best alternative at this time. Upon viewing results like this, you might be inclined to hold off on your upgrade until new system prices come down a bit, in which case the net present worth analysis might yield different results. Other factors to consider include fluctuations in interest rates, as well as the declining cash flow of the old system as it ages further.

Excel offers a built-in function called  $NPV$  that computes the net present value of a series of cash flows given a discount rate (interest rate). Its syntax is  $=NPV(rate, values)$ , where  $rate$  is the interest rate and  $values$  is a cell range or series of values separated by commas representing the cash flow series. You can use  $NPV$  to compute net present values for the cash flows shown in the first three columns of the table in [Figure 14-2](#). Doing so will save you the trouble of constructing the discounted cash flow tables and using the  $PV$  function.

For example, to compute the net present value of the **No Upgrade** option, use the formula  $=C8+NPV(E2, C9:C12)$ . Notice that this formula discounts only the cash flow values for the second through fifth years. The first year, year 0, is then added to the discounted cash flow series to arrive

at the net present value. Similarly, the formulas  $=D8+NPV(E2,D9:D12)$  and  $=E8+NPV(E2,E9:E12)$  compute the net present values for the system A and system B options, respectively. These three formulas yield the same net present value results computed using the  $PV$  function.

## See Also

Check out [Recipe 14.9](#) to learn how to compute internal rates of return for the options in this example.



[← PREVIOUS](#)

## Recipe 14.9. Estimating Rate of Return

### Problem

You're evaluating a series of payments and cash flows and want to compute the rate of return on your investment.

### Solution

Use the `IRR` function to compute the *internal rate of return*.

### Discussion

Let's reconsider the example cash flow options discussed in [Recipe 14.6](#). In addition to looking at net present values, you also want to compute the rate of return on each investment. `IRR` will do that for you. The syntax for `IRR` is `=IRR(values, guess)`, where *values* is a cell range containing the investment and cash flow value and *guess* is your initial guess at the rate of return.

`IRR` uses an iterative solution approach; your initial guess is used to help it converge on a solution.

To compute the rates of return for the two projects, project A and project B, enter these two formulas: `=IRR(C8:C12)` and `=IRR(D8:D12)`. The first formula computes the rate of return for project A before discounting, and the second formula computes the rate of return for project B before discounting. These formulas return 50% and 43%, respectively, indicating that project A is the better option.

The formulas `=IRR(E8:E12)` and `=IRR(F8:F12)` compute the rates of return for projects A and B after discounting their cash flows. The resulting rates of return are 43% and 36% for project A and project B, respectively. These results again indicate that project A is the better alternative.

## Recipe 14.10. Solving Inverse Problems

### Problem

You want to solve a problem in reverse. For example, referring to the net present worth example discussed in [Recipe 14.8](#), you might want to find what annual cash flow value would make one upgrade option the more attractive alternative.

### Solution

Use Goal Seek or Solver to iteratively find a solution. See [Chapters 9](#) and [13](#) for an in-depth discussion and examples of using Goal Seek and Solver.

### Discussion

Let's reconsider the example discussed in [Recipe 14.8](#). Earlier it was found that the **No Upgrade** option yielded the highest net present worth. Now the question is what annual cash flow would be required for, say, the system A option to yield a net present worth equal to or greater than the **No Upgrade** option. You can solve this problem with just a few small changes to the spreadsheet shown earlier in [Figure 14-2](#). All you have to do is add a new value in a cell that represents the assumed annual cash flow for system A. Then you need to revise the cash flow values shown in the undiscounted cash flow column for system A to refer to the new assumed value. Finally, you can use Goal Seek to iteratively find the assumed cash flow value while setting the net present worth value for system A to that of the **No Upgrade** option. [Figure 14-3](#) shows the result of this exercise.

Cell E3 contains the new assumed annual cash flow value for system A. After running Goal Seek as discussed a moment ago, the resulting value is \$88,607, and you can see that the net present worth values for the **No Upgrade** and **System A** options in cells F13 and G13, respectively, are the same. Therefore, in order for system A to be the more attractive option, your annual cash flow for that option must exceed \$88,607.

The only changes I made to the original spreadsheet to find this solution are as follows:

1. I added a value in cell E3 to hold the assumed annual cash flow value for system A.
2. I added a text label in cell D3.
3. I changed cells D9, D10, and D11 from hardcoded values to the cell formula `=E3`.
4. I changed the formula in cell D12 from `=85000+0.1*ABS(D8)` to `=E3+0.1*ABS(D8)`.



After making these changes, I ran Goal Seek to set cell G13 to a value of 85,575 by changing the value in cell E3. The result is as shown in [Figure 14-3](#).

Figure 14-3. Net present worth example revisited

The screenshot shows an Excel spreadsheet with the following data:

- Interest Rate = 4.9%
- System A Annual Cash Flow = \$ 88,607.46

**Discounted Cash Flow Analysis**

Year	Cash Flow			Discounted Cash Flow		
	No Upgrade	System A	System B	No Upgrade	System A	System B
0	\$ -	\$ (250,000)	\$ (310,000)	\$ -	\$ (250,000)	\$ (310,000)
1	\$ 30,000	\$ 88,607	\$ 97,000	\$ 28,599	\$84,469	\$92,469
2	\$ 30,000	\$ 88,607	\$ 97,000	\$ 27,263	\$80,523	\$88,150
3	\$ 20,000	\$ 88,607	\$ 97,000	\$ 17,326	\$76,762	\$84,032
4	\$ 15,000	\$ 113,607	\$ 128,000	\$ 12,388	\$93,822	\$105,708
Total:	\$ 95,000	\$ 129,430	\$ 109,000	\$ 85,575	\$ 85,575	\$ 60,359

## See Also

Be sure to read [Chapter 9](#) to learn a lot more about using Solver and Goal Seek.

← PREV

## Recipe 14.11. Figuring a Break-Even Point

### Problem

You want to figure out a break-even point for an investment or business venture.

### Solution

Set up a spreadsheet containing projected expenses and revenue over a period of time and compute cumulative expenses and revenue over that period. Compute the net revenue and find the point at which net revenue goes from negative to positive.

### Discussion

Consider the data shown in [Figure 14-4](#), which shows hypothetical expenses and revenue for a proposed business venture.

An initial, one-time cost of \$97,000 is required in year 0, with operating expenses increasing linearly from \$9,600 to \$14,400 from year 1 through year 10. Revenue is expected to start flowing in year 1 and to increase annually until year 6, at which point it levels off.

Figure 14-4. Break-even example

The screenshot shows an Excel spreadsheet titled "Microsoft Excel - ch 14 examples.xls". The active window displays a "Break-even Analysis" table. The table is structured as follows:

Year	Expenses	Revenue	Cumulative Expenses	Cumulative Revenue	Net Revenue
0	\$ 97,000	\$ -	\$ 97,000	\$ -	\$ (97,000)
1	\$ 9,600	\$ 15,000	\$ 106,600	\$ 15,000	\$ (91,600)
2	\$ 10,560	\$ 18,000	\$ 117,160	\$ 33,000	\$ (84,160)
3	\$ 11,040	\$ 25,000	\$ 128,200	\$ 58,000	\$ (70,200)
4	\$ 11,520	\$ 37,000	\$ 139,720	\$ 95,000	\$ (44,720)
5	\$ 12,000	\$ 40,000	\$ 151,720	\$ 135,000	\$ (16,720)
6	\$ 12,480	\$ 41,000	\$ 164,200	\$ 176,000	\$ 11,800
7	\$ 12,960	\$ 41,000	\$ 177,160	\$ 217,000	\$ 39,840
8	\$ 13,440	\$ 41,000	\$ 190,600	\$ 258,000	\$ 67,400
9	\$ 13,920	\$ 41,000	\$ 204,520	\$ 299,000	\$ 94,480
10	\$ 14,400	\$ 41,000	\$ 218,920	\$ 340,000	\$ 121,080

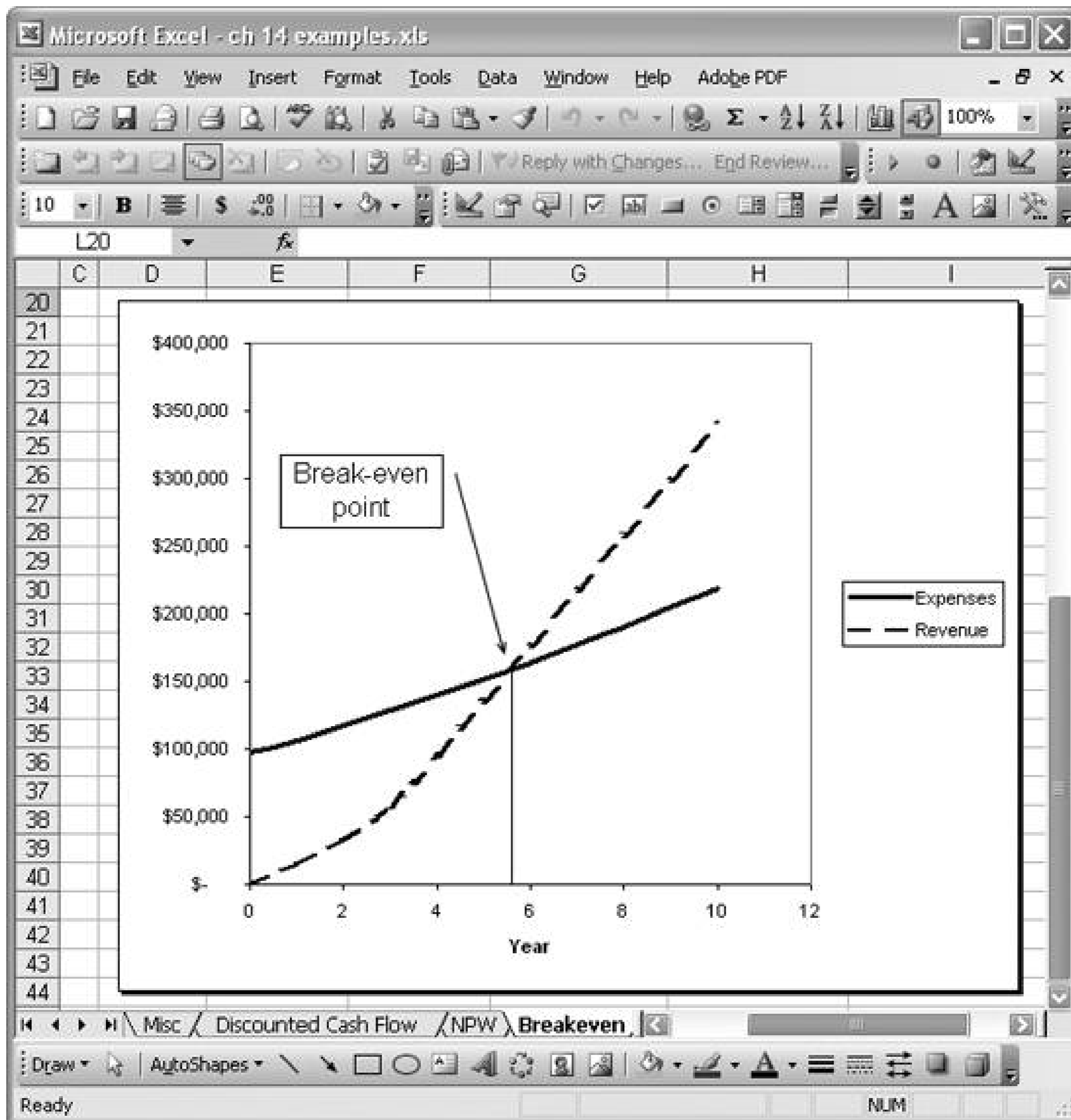
The question is how long it will be before you break even. That is, how many years until your total revenue exceeds your total expenses?

Columns G and H compute cumulative expenses and revenue over the period in question. These are computed by summing all previous expenses and revenue for years prior to and including the current year.

The last column in the table computes the difference between cumulative revenue and cumulative expenses. The break-even point is found when this value changes from negative to positive. In this example, break-even occurs between years 5 and 6. A plot of the cumulative expenses and revenue illustrates the break-even point graphically; see [Figure 14-5](#).

The point where these two curves cross is the break-even point. If you plot the net revenue values over time (the last column in the table shown in [Figure 14-4](#)), you'll see that the break-even point occurs when the net revenue curve crosses the horizontal axis. You can pinpoint the break-even point by interpolating the net revenue curve. Or you can fit a curve through it and use Solver or Goal Seek to find its root. See [Chapter 9](#) to learn how.

Figure 14-5. Break-even plot



## See Also

You can also compute rates of return for this venture by using techniques discussed in [Recipe 14.9](#). Further, you can consider the time value of money using techniques shown in [Recipe 14.8](#). You can also perform a so-called what-if analysis to see how changes in your expense and revenue projections affect your break-even point. Finally, you can use techniques discussed in [Recipe 14.10](#) to figure out required cash flows in order to achieve a break-even point by a specific year.

[← PREV](#)

## About the Author

David M. Bourg is a practicing engineer who provides consulting services through his company, David M. Bourg & Associates, LLC. He is also an adjunct professor of naval architecture at the University of New Orleans. His current research involves the direct numerical simulation of multiphase fluid flow problems. Additionally, David is professionally involved in computer game development and consulting. He is the author of two other O'Reilly books, *AI for Game Developers* and *Physics for Game Developers*.

[← PREV](#)

## Colophon

The animal on the cover of the *Excel Scientific and Engineering Cookbook* is a common genet (*Genetta vulgaris* or *Genetta genetta*), also known as a true or small-spotted genet. (The taxonomy of the genus is controversial, with several similar species and numerous proposed subspecies, reflecting many geographic and habitat-specific variations.) The animal's name is pronounced with a soft "g" and rhymes with "Senate," in contrast to the homographic surname of the French novelist and playwright of the theater of the absurd.

Related to but distinct from true cats, genets are members of the civet or viverrid family (*Viverridae*), which also includes the mongoose. The viverrids are generally regarded as the closest living descendants of the extinct common ancestor of the carnivores. The oldest genet fossils, found in Morocco, date back to the Pliocene. Genets were kept as pets by the ancient Egyptians and as rat catchers in Europe until they were eventually replaced by house cats (depictions of domesticated genets can be seen on European tapestries from the Middle Ages).

Common genets are distributed throughout southern Europe, parts of the Middle East, North Africa, and the savanna south of the Sahara. They prefer low altitudes but live in diverse habitats, including pine forests, olive groves, rocky areas, and scrublands. Genets are considered arboreal, but they also spend a large amount of time on the ground and are partial to streambanks. They are agile climbers, with semiretractable claws and an extremely flexible body that can squeeze through any opening large enough to admit the head. They are primarily solitary; genets of opposite sexes may share overlapping territory, but those of the same sex will not. They are mostly nocturnal, resting during the day in thickets and hollow trees. Genets do most of their hunting on the ground and, like cats, they hunt by stealth, crouching low to the ground and killing prey with a quick bite on the neck.

The genet is an opportunistic carnivore; its main prey are small mammals, but it rounds out its diet according to seasonal availability with birds, reptiles, amphibians, fish, insects, and fruits. Genets use their sense of smell for both hunting and communication; they convey social, mating, and territorial messages via secretions from several scent glands (although these are not as highly developed as those of civets, whose perineal glands have long played a role in the making of perfume and the unusual processing of Indonesian coffee beans called *kopi luwak*). Both sexes use flank and hind-leg rubbing to distribute olfactory cues, in addition to marking with perineal secretions (common among females) and urine (more popular with males).

Genets have five vocalizations: a hiccup (used primarily by mothers to call the litter, and by males during the mating period); a purr (used only during the first week of life); a mew (used by the young while still dependent); a growl (used by the young once their predatory behavior is fully developed); and a click (used by adults in aggressive interactions). In offensive stance, the dorsal fur bristles and the genet arches its back, opens its mouth, and bares its teeth.

The cover image is from Lydekker's Royal Natural History. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[← PREV](#)



# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

- [' \(apostrophe\)](#)
- [^ \(caret operator\)](#)
  - [VBA programming and](#)
- [: \(colon\)](#)
- [\\$ \(dollar sign\)](#)
  - [button](#)
- [= \(equals sign\)](#)
- [# \(hash mark\)](#)
- [\( \) \(parentheses\)](#)
- [% \(percent\) button](#)
- [' \(quote, single\)](#)
- [\\_ \(underscore\)](#)
- [3D surface plots, charting](#)
  - [views, rotating](#)



[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[A1-style cell references](#)

[ABS function](#)

[Abs\(\)](#)

[absolute cell references](#)

[accelerator characters](#)

[Access databases, importing into Excel](#)

[ACOS function](#)

[ACOSH function](#)

[Add Procedure dialog box](#)

[add-in functions, creating](#)

[Alt key](#)

[alternating direction implicit \(ADI\) method](#)

[Analysis ToolPak](#)

[Correlation tool](#)

[exponential smoothing](#)

[Fourier Analysis tool](#)

[Moving Average](#)

[Rank and Percentile tool](#)

[statistical testing tools](#)

[ANOVA \(analysis of variance\) 2nd](#)

[Application object](#)

[area center, computing with numerical integration](#)

[arguments](#)

[arrays](#)

[memory allocation for](#)

[As DataType qualifier](#)

[ASIN function](#)

[ASINH function](#)

[ATAN function](#)

[ATANH function](#)

[Atn\(\)](#)

[AutoFilter](#)

[automation](#)

[macros](#)

[averages](#)

[AVERAGE formula](#)

[average-percentage method](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[band pass filters](#)

[Beep statement](#)

[boundary value problems](#)

[break-even point on investments, calculating](#)

[breakpoints](#)

[built-in financial formulas](#)

[built-in functions](#)

[\[lookup functions\]\(#\)](#)

[button controls, adding to spreadsheets](#)

[ByVal keyword](#)



# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Call keyword](#)

[callout boxes](#)

[caret \(^\) operator](#)

[VBA programming and](#)

[cash flow alternatives, comparing](#)

[CEILING function](#)

[cells](#)

[cell names](#)

[cell references](#)

[A1-style](#)

[absolute cell references](#)

[R1C1-style](#)

[ranges](#)

[relative cell references](#)

[data entry](#)

[data types, setting](#)

[scientific notation](#)

[formulas, entering](#)

[selection operations](#)

[centering data](#)

[charts](#)

[3D surface plots](#)

[views, rotating](#)

[annotating](#)

[axes, customizing](#)

[Scale tab](#)

[Chart Wizard](#)

[Axes tab](#)

[Data Labels tab](#)

[Data Range tab](#)

[Gridlines tab](#)

[Legend tab](#)

[Series tab](#)

[Titles tab](#)

[contour plots](#)

[customizations, saving](#)

[error bars](#)

[formatting](#)

[Ctrl-1 key](#)

[linear curve fitting using](#)

[log or semilog scales, setting](#)

- [multiple axes](#)
  - [dummy series, using](#)
  - [nonlinear curve fitting](#)
  - [Source Data dialog box](#)
  - [styles](#)
  - [trendlines, creating](#)
  - [types](#)
    - [changing on existing charts](#)
    - [combining](#)
  - [Word documents, copying to](#)
  - [XY \(scatter\) charts](#)
- [chromosomes](#)
- [classical methods for solving equations](#)
- [CLEAN function](#)
- [code conformance in engineering problems](#)
- [code formatting](#)
- [coefficients of determination 2nd](#)
- [colon \(:\)](#)
- [comments 2nd](#)
- [COMPLEX function](#)
- [complex numbers](#)
- [Compute Cf subroutine](#)
- [CONFIDENCE function](#)
- [confidence intervals 2nd](#)
- [constants](#)
- [constructors](#)
- [contour plotting](#)
- [conversion between number systems, functions for](#)
- [CONVERT function](#)
- [Convert Text to Columns Wizard](#)
- [CORREL function](#)
- [correlation coefficient](#)
- [Correlation tool](#)
- [COS function](#)
- [Cos\( \)](#)
- [COSH function](#)
- [Coursey, J. S.](#)
- [Crank-Nicolson finite difference method](#)
- [Ctrl key](#)
  - [Ctrl-A](#)
  - [Ctrl-C](#)
  - [Ctrl-F1](#)
  - [Ctrl-V](#)
  - [Ctrl-X](#)
- [curve fitting](#)
  - [assessing fit](#)
    - [coefficient of determination](#)
    - [correlation coefficient](#)
    - [explained and unexplained variation](#)
    - [total variation](#)
  - [confidence intervals, computing](#)
  - [fitting nonlinear curves](#)

[Excel charts, using Solver, using multiple linear regression](#)  
custom functions  
[example](#)  
[unit conversion](#)  
[vector-based calculations](#)  
[cut-and-paste operations](#)



[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[damping factors](#)

[data](#)

[correlating of](#)

[data series, centering](#)

[importing](#)

[refreshing data](#)

[random sampling of](#)

[Data Analysis ToolPak](#)

[Sampling tool](#)

[data entry](#)

[data operations](#)

[cleaning of nonprintable characters](#)

[filtering 2nd](#)

[importing](#)

[Access databases](#)

[delimited text files](#)

[drag-and-drop](#)

[text files](#)

[web pages](#)

[XML files](#)

[parsing](#)

[sorting](#)

[tabular data lookups](#)

[Data Range Properties button](#)

[data types 2nd](#)

[As DataType qualifier](#)

[type-declaration characters](#)

[data validation](#)

[databases](#)

[database functions](#)

[importing data from](#)

[date and time functions](#)

[debugging](#)

[breakpoints](#)

[DEC2HEX function](#)

[decimal places, setting](#)

[DEGREES function](#)

[delimited data](#)

[delimited text files, importing](#)

[derivative of a function](#)

[Descriptive Statistics tool](#)

[deseasonalization of time series](#)

[design mode](#)

[detrending](#)

[difference \(forward, backward, and central\)](#)

[diffusion equations](#)

[Dim statements](#)

[declaring arrays](#)

[discrete Fourier transforms](#)

[division functions](#)

[Do loops](#)

[For loops, versus](#)

[Do While and Do Until loops](#)

[dockable toolbars](#)

[DoEuler VBA code for nonlinear system](#)

[DoEuler1stOrder VBA subroutine](#)

[modified](#)

[double integrals](#)

[Dragoset, R. A.](#)

[Drawing toolbar](#)

[DuChateau, Paul](#)

[dynamic programming](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[E or e](#)

[Edit menu](#)

[engineering functions](#)

[equals sign](#)

[Equation Editor](#)

[Microsoft Word](#)

[equation notes](#)

[equations, documentation in spreadsheets](#)

[equations, solving](#)

[automating solutions with VBA](#)

[classical methods](#)

[Newton's method](#)

[secant method](#)

[linear systems](#)

[matrix inversion, using](#)

[Solver, using](#)

[nonlinear equations, solving iteratively](#)

[nonlinear systems of equations](#)

[roots, finding graphically](#)

[Euler's method](#)

[coupled equations, solving](#)

[implementing](#)

[spreadsheets, using](#)

[VBA, using](#)

[EVEN function](#)

[Excel and VBA functions](#)

[Excel objects](#)

[Application object](#)

[Object Browser](#)

[Range objects](#)

[Save method](#)

[Worksheet objects](#)

[WorksheetFunction objects](#)

[Exit For statement](#)

[Exp\(\)](#)

[exponential functions](#)

[exponential smoothing](#)

[exponents](#)

[External Data Range Properties dialog box](#)

[External Data toolbar 2nd 3rd](#)





[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[F-test](#)

[F2 shortcut key](#)

[fifth-order formula](#)

[Filip.txt](#)

[Filippelli, A.](#)

[financial calculations](#)

[break-even point on investment, calculating](#)

[cash flow alternatives](#)

[future value, calculating](#)

[inverse problems](#)

[monthly payments on debt](#)

[net present worth, calculating](#)

[rate of return, estimating](#)

[required rate of return](#)

[return on interest](#)

[target value, calculating investment required](#)

[financial functions](#)

[finite difference method](#)

[finite element method \(FEM\)](#)

[Fisher, R. A.](#)

[fixed-width data](#)

[For Each loops](#)

[For loops versus Do loops](#)

[For Next loops](#)

[FORECAST function and time series analysis](#)

[forecasting](#)

[Format Axis dialog box](#)

[Format Cells dialog box](#)

[Format Data Series dialog box](#)

[Format menu](#)

[formatting](#)

[custom styles, defining](#)

[style drop-down lists](#)

[spreadsheets](#)

[formulas](#)

[cell references](#)

[cells, entering into](#)

[exponents](#)

[mathematical operators used in](#)

[operator precedence](#)

[Fourier transforms](#)

[FREQUENCY function](#)

[FTEST function](#)

[Function Arguments dialog box](#)

[Function keyword](#)

[functions 2nd](#)

[add-in worksheet functions, creating arguments](#)

[built-in financial formulas](#)

[built-in functions](#)

[lookup functions](#)

[derivative of a function](#)

[documentation](#)

[Excel functions versus VBA functions](#)

[FV function](#)

[Insert Function button](#)

[IRR function](#)

[linear fits with spreadsheet functions](#)

[NPER function](#)

[NPV function](#)

[PMT function 2nd](#)

[PV function 2nd](#)

[SUM function](#)

[future value, calculating](#)

[FV function](#)

[Fylstra, Daniel](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Galerkin method](#)

[Gauss-Seidel iteration](#)

[Gaussian quadratures](#)

[GCD function](#)

[Generalized Reduced Gradient optimization algorithm](#)

[genes](#)

[genetic algorithm](#)

[chromosome class, creating](#)

[chromosomes and genes](#)

[Class\\_Initialize method](#)

[Copy method](#)

[EvaluateFitness subroutine](#)

[EvaluateZ method](#)

[Evolve button](#)

[event handler](#)

[genetic material](#)

[genetic optimization](#)

[global variables](#)

[Mate method](#)

[Mutate method](#)

[MutationRate property](#)

[SelectChromosome function](#)

[global scope](#)

[Goal Seek](#)

[inverse solutions to financial calculations](#)

[nonlinear equations, automating solutions with VBA](#)

[nonlinear equations, solving iteratively](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Hawley, David 2nd](#)

[Hawley, Raina 2nd](#)

[Help](#)

[Excel 2nd](#)

[VBA](#)

[Visual Basic User Interface Help](#)

[HEX2DEC function](#)

[Hildebrand, F. B.](#)

[Histogram tool](#)

[HLOOKUP function](#)

[hyperbolic equations](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[IDE \(integrated development environment\)](#)

[IMABS function](#)

[IMAGINARY function](#)

[IMARGUMENT function](#)

[IMCONJUGATE function](#)

[IMCOS function](#)

[IMDIV function](#)

[IMEXP function](#)

[IMLN function](#)

[IMLOG10 function](#)

[IMLOG2 function](#)

[Immediate window](#)

[IMPOWER function](#)

[IMPRODUCT function 2nd](#)

[IMREAL function 2nd](#)

[IMSIN function](#)

[IMSQRT function](#)

[IMSUB function](#)

[IMSUM function](#)

[INDEX function](#)

[initial value problems 2nd](#)

[Insert Function button](#)

[INT function](#)

[Int\( \)](#)

[integer constraints](#)

[optimization problems and](#)

[internal rate of return](#)

[investments, calculating break-even point on](#)

[IRR function](#)

[iterative methods](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Jacobi's method](#)

[← PREV](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[kernel smoothing](#)

[keyboard shortcuts](#)

[cut, copy, and paste](#)

[Kreyszig, Erwin](#)

[Kythe, Prem 2nd](#)

[← PREV](#)



[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[labor statistics dataset](#)  
[Laplace equation 2nd](#)  
[Lasdon, Leon](#)  
[LCM function](#)  
[Lenz, Evan](#)  
[Liebmann's method](#)  
[linear curve fitting](#)  
    [charts, using](#)  
    spreadsheet functions, using  
        [LINEST function](#)  
[linear optimization programming](#)  
[linear systems, solving](#)  
    [matrix inversion, using](#)  
    [Solver, using](#)  
LINEST function  
    [F-tests](#)  
    [linear curve fitting](#)  
    [multiple linear regression](#)  
[LN function](#)  
[loan amortization template](#)  
[local scope](#)  
[Locals window](#)  
[log charts](#)  
[LOG function](#)  
    [VBA Log versus Excel LOG](#)  
[Log\(\)](#)  
[logarithmic functions](#)  
[logical functions](#)  
[Longley, J. W.](#)  
[LOOKUP function](#)  
[lookup functions](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[macros](#)

[main menu bar](#)

[MATCH function](#)

[mathematical functions 2nd](#)

[ABS function](#)

[complex numbers](#)

[division functions](#)

[exponential functions](#)

[logarithmic functions](#)

[matrix functions](#)

[multiplication functions](#)

[number systems, conversion between](#)

[rounding and truncating numbers](#)

[SIGN function](#)

[spreadsheet functions, using in VBA code](#)

[SQRT function](#)

[summation functions](#)

[trigonometric functions](#)

[vector-based calculations](#)

[matrix functions](#)

[large systems and](#)

[linear systems of equations, solving with](#)

[McRae, Mary](#)

[MDETERM function](#)

[memory allocation for arrays](#)

[menus](#)

[main menu bar](#)

[options](#)

[methods](#)

[Michelson](#)

[Microsoft Word](#)

[copying charts to](#)

[spreadsheets, embedding in](#)

[MINVERSE function](#)

[Mitchell, Melanie](#)

[MMULT function](#)

[MOD function](#)

[moment of inertia](#)

[Monte Carlo integration 2nd](#)

[Monte Carlo method](#)

[monthly payments on debt, calculating](#)

[moving averages, computing](#)

[Analysis ToolPak Moving Average](#)

[spreadsheet formulas for](#)

[trendlines, using](#)

[MROUND function](#)

[multidimensional arrays](#)

[multiple axes charts](#)

[multiple linear regression](#)

[multiple-document interface main window](#)

[multiplication functions](#)

[MutationRate property](#)



[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[net present worth, calculating](#)

[New Web Queries](#)

[Newton's method for solving equations](#)

[Newton-Cotes integration formulas](#)

[Next statement](#)

[nonlinear equations, solving iteratively](#)

[nonlinear systems of equations, solving](#)

[NORMSINV function](#)

[NPER function](#)

[NPV function](#)

[number systems, conversion between](#)

[numerator degrees of freedom](#)

[numerical differentiation](#)

[numerical integration](#)

[computing the center of an area](#)

[computing the second moment of an area](#)

[double integrals](#)

[integrating a definite integral](#)

[trapezoidal rule](#)

[VBA, implementing in](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Object Browser](#)

[Classes list](#)

[object-oriented programming](#)

[objects](#)

[creating](#)

[members](#)

[ODD function](#)

[operator precedence](#)

[optimization analysis](#)

[engineering design problems](#)

[integer constraints, using](#)

[locally optimal versus globally optimal solutions](#)

[resource allocation optimization](#)

[traditional linear programming](#)

[Option Base statement](#)

[ordinary differential equations, solving](#)

[boundary value problems](#)

[coupled equations](#)

[Euler's method](#)

[Runge-Kutta method](#)

[first-order initial value problems](#)

[second-order initial value problems](#)

[solution methods](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[parabolic equations](#)

[parentheses \( \)](#)

[partial differential equations, solving](#)

[finite difference equations](#)

[finite element method](#)

[initial value problems](#)

[Paste and Paste Special](#)

[PEARSON function](#)

[PERCENTILE function](#)

[PERCENTRANK function](#)

[PI function](#)

[PMT function 2nd](#)

[pointers](#)

[Poisson equation](#)

[POWER function](#)

[procedural programs](#)

[PRODUCT function](#)

[product mix problems](#)

[properties](#)

[Public keyword](#)

[PV function 2nd](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[QUOTIENT function](#)

[← PREV](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[R1C1-style cell references](#)

[Radar charts](#)

[types](#)

[RADIANS function](#)

[random number generation](#)

[random sampling](#)

[Range objects](#)

[ranges](#)

[RANK function](#)

[Spearman rank correlation using](#)

[rate of return, estimating](#)

[relative references](#)

[Rem keyword](#)

[required rate of return](#)

[resource optimization allocation](#)

[return on interest, calculating](#)

[Reynolds number](#)

[Roman, Steven](#)

[roots, finding graphically](#)

[ROUND function](#)

[Round\(\)](#)

[rounding functions](#)

[Runge-Kutta method](#)

[coupled equations, solving](#)

[second-order initial value problems, applying to](#)

[VBA code for nonlinear system](#)



[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Sampling tool](#)  
[Save method](#)  
[schemas \(XML\)](#)  
[Schwab, D. J.](#)  
[scientific notation](#)  
[scope qualifiers](#)  
[seasonal indices, estimating](#)  
[secant method for solving equations](#)  
[secondary axes](#)  
[semilog charts](#)  
[Sgn\( \)](#)  
[shooting method](#)  
[shortcut keys](#)  
[SIGN function](#)  
[Simon, Jinjer](#)  
[Simpson's rule](#)  
[SIN function](#)  
[Sin\( \)](#)  
[SINH function](#)  
[small sampling theory](#)  
[smoothing data](#)  
    [exponential smoothing](#)  
    [kernel smoothing](#)  
[Solver](#)  
    [By Changing Cells field](#)  
    [finite difference equations, solving](#)  
    [first-order initial value problems, solving](#)  
    [Guess button](#)  
    [inverse solutions to financial calculations](#)  
    [linear systems, solving](#)  
    [nonlinear curve fitting](#)  
    [nonlinear equations, automating solutions with VBA](#)  
    [optimization analysis](#)  
        [engineering design problems](#)  
        [integer constraints, using](#)  
        [locally optimal versus globally optimal solutions](#)  
        [resource allocation problems](#)  
        [traditional linear programming](#)  
    [options](#)  
    [reports feature](#)  
    [Set Target Cell field](#)

[solutions using the finite element method](#)  
[Solver reports](#)  
[Answer reports](#)  
[Limits reports](#)  
[Sensitivity reports](#)  
[speed of light dataset](#)  
[spreadsheet functions](#)  
[linear curve fitting using](#)  
[spreadsheets](#)  
[Euler's method, implementing](#)  
[formatting](#)  
[procedural programs, compared to](#)  
[Word documents, embedding in](#)  
[Sqr\( \)](#)  
[SQRT function](#)  
[St.Laurent, Simon](#)  
[statistical analysis](#)  
[ANOVA \(analysis of variance\)](#)  
[confidence intervals, calculating](#)  
[data, correlating](#)  
[frequency distributions, plotting](#)  
[FREQUENCY function, using](#)  
[Histogram tool, using](#)  
[random number generation](#)  
[random sampling](#)  
[ranking and percentiles](#)  
[statistical tests](#)  
[summary statistics, computing](#)  
[statistical functions](#)  
[STEYX function](#)  
[stiffened panel model](#)  
[Student's t-distribution](#)  
[style drop-down lists](#)  
[subroutines](#)  
[calling](#)  
[data types](#)  
[names](#)  
[parameters](#)  
[Sub keyword](#)  
[SUM function](#)  
[summation functions](#)  
[SUMPRODUCT function](#)  
[trapezoidal rule, implementation using](#)  
[SUMSQ function](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[table value lookups](#)

[TAN function](#)

[Tan\(\)](#)

[TANH function](#)

[target value, calculating investment required](#)

[task pane](#)

[Taylor series expansion](#)

[testing functions](#)

[inverse functions of](#)

[text and data functions](#)

[text files](#)

[delimited files, importing](#)

[Excel, opening in](#)

[importing](#)

[column-break specification](#)

[dragging and dropping](#)

[Save As...](#)

[Text Import Wizard](#)

[Thurber, R.](#)

[time series analysis](#)

[data series, centering](#)

[deseasonalization](#)

[detrending a time series](#)

[discrete Fourier transforms](#)

[forecasting](#)

[nonlinear trendlines](#)

[plotting time series data](#)

[seasonal indices, estimating](#)

[trendlines, adding](#)

[TINV function](#)

[To keyword](#)

[Toolbars](#)

[dockability](#)

[TRANSPOSE function](#)

[trapezoidal rule](#)

[VBA, implementing in](#)

[function for](#)

[trendline tool](#)

[trendlines](#)

[detrending time series using](#)

[trigonometric functions](#)

[truncating functions](#)

[TTEST function](#)

[two-point boundary value problem](#)

[type-declaration characters](#)



[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[unit conversion](#)

[add-ins, creating](#)

[CONVERT function](#)

[custom functions](#)

[user input](#)

[validating](#)

[← PREV](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[v\\_CrossProduct code](#)

[v\\_Mag code](#)

[variables](#)

[VBA \(Visual Basic for Applications\)](#)

[arrays](#)

[autogenerated code, using](#)

[automating solutions to nonlinear equations](#)

[built-in functions](#)

[code completion](#)

[comments](#)

[constants](#)

[custom functions for vector calculations](#)

[data types](#)

[debugging](#)

[breakpoints](#)

[debugging code](#)

[derivative functions](#)

[Dim statements](#)

[DoEuler subroutine](#)

[DoEuler1stOrder subroutine](#)

[modified](#)

[DoRK2ndOrder subroutine \(Runge-Kutta\)](#)

[DoRunge-Kutta subroutine](#)

[Euler's method, implementing](#)

[functions](#)

[Excel functions and](#)

[y=f\(x\), computing](#)

[Help](#)

[Visual Basic User Interface Help](#)

[Immediate window](#)

[Locals window](#)

[names](#)

[objects, creating](#)

[procedures](#)

[spreadsheet functions, using in code](#)

[statements](#)

[conditional statements](#)

[spanning over multiple lines](#)

[subroutines](#)

[trapezoidal rule, implementing](#)

[function for](#)

[v\\_CrossProduct code](#)

[v\\_Mag code](#)

[variables](#)

[Visual Basic Language Reference](#)

[Watch window](#)

[Wcs function](#)

[VBA Editor](#)

[code modules](#)

[properties](#)

[IDE](#)

[object and procedure lists](#)

[Object Browser](#)

[procedure and full module views](#)

[project panel or explorer](#)

[VBAProject](#)

[vector-based calculations](#)

[VLOOKUP function](#)





# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Waren, Allan](#)

[Watch window](#)

[Watson, John](#)

[wave equations](#)

[Wcs function](#)

web pages

[importing into Excel](#)

[Wei, Dongming 2nd](#)

[weighting factors](#)

[workbook](#)

[Workbook object](#)

[Worksheet object](#)

[WorksheetFunction objects](#)

[accessing Excel functions from within VBA](#)



[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[XML \(Extensible Markup Language\)](#)

[importing files into Excel](#)

[schemas](#)

[XY \(scatter\) charts](#)

[generating nonlinear curve fits](#)

[linear curve fitting](#)

[types](#)

[← PREV](#)

[← PREV](#)

# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Zachmann, David](#)

[ZTEST function](#)

[← PREV](#)