

HARTMUT F. W. HÖFT  
MARGRET H. HÖFT

COMPUTING WITH  
**MATHEMATICA®**

SECOND EDITION



INCLUDES  
CD-ROM

# Computing with *Mathematica*® Second Edition

## LIMITED WARRANTY AND DISCLAIMER OF LIABILITY

ACADEMIC PRESS ("AP") AND ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE ACCOMPANYING CODE ("THE PRODUCT") CANNOT AND DO NOT WARRANT THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE PRODUCT. THE PRODUCT IS SOLD "AS IS" WITHOUT WARRANTY OF ANY KIND (EXCEPT AS HEREAFTER DESCRIBED), EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY OF PERFORMANCE OR ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. AP WARRANTS ONLY THAT THE MAGNETIC DISKETTE(S) ON WHICH THE CODE IS RECORDED IS FREE FROM DEFECTS IN MATERIAL AND FAULTY WORKMANSHIP UNDER THE NORMAL USE AND SERVICE FOR A PERIOD OF NINETY (90) DAYS FROM THE DATE THE PRODUCT IS DELIVERED. THE PURCHASER'S SOLE AND EXCLUSIVE REMEDY IN THE EVENT OF A DEFECT IS EXPRESSLY LIMITED TO EITHER REPLACEMENT OF THE DISKETTE(S) OR REFUND OF THE PURCHASE PRICE, AT AP'S SOLE DISCRETION.

IN NO EVENT, WHETHER AS A RESULT OF BREACH OF CONTRACT, WARRANTY OR TORT (INCLUDING NEGLIGENCE), WILL AP OR ANYONE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE PRODUCT BE LIABLE TO PURCHASER FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT OR ANY MODIFICATIONS THEREOF, OR DUE TO THE CONTENTS OF THE CODE, EVEN IF AP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Any request for replacement of a defective diskette must be postage prepaid and must be accompanied by the original defective diskette, your mailing address and telephone number, and proof of date of purchase and purchase price. Send such requests, stating the nature of the problem, to Academic Press Customer Service, 6277 Sea Harbor Drive, Orlando, FL 32887, 1-800-321-5068. AP shall have no obligation to refund the purchase price or to replace a diskette based on claims of defects in the nature or operation of the product.

Some states do not allow limitation on how long an implied warranty lasts, nor exclusions or limitations of incidental or consequential damage, so the above limitations and exclusions may not apply to you. This Warranty gives you specific legal rights, and you may also have other rights which vary from jurisdiction to jurisdiction.

THE RE-EXPORT OF UNITED STATES ORIGIN SOFTWARE IS SUBJECT TO THE UNITED STATES LAWS UNDER THE EXPORT ADMINISTRATION ACT OF 1969 AS AMENDED. ANY FURTHER SALE OF THE PRODUCT SHALL BE IN COMPLIANCE WITH THE UNITED STATES DEPARTMENT OF COMMERCE ADMINISTRATION REGULATIONS. COMPLIANCE WITH SUCH REGULATIONS IS YOUR RESPONSIBILITY AND NOT THE RESPONSIBILITY OF AP.

# Computing with *Mathematica*® Second Edition

**Hartmut F. W. Höft**  
*Eastern Michigan University*  
*Ypsilanti, MI*

**Margret Höft**  
*University of Michigan-Dearborn*  
*Dearborn, MI*



**ACADEMIC PRESS**

---

An imprint of Elsevier Science

Amsterdam Boston London New York Oxford Paris  
San Diego San Francisco Singapore Sydney Tokyo

Sponsoring Editor	Barbara Holland
Production Editor	Angela Dooley
Editorial Assistant	Tom Singer
Marketing Manager	Anne O'Mara
Cover Design	Dick Hannus
Copyeditor	Adrienne Rebello
Composition	CEPHA
Printer	Edwards Bros.

This book is printed on acid-free paper. ∞

Copyright 2003, 1998, Elsevier Science (USA)

All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

Requests for permission to make copies of any part of the work should be mailed to:  
Permissions Department, Harcourt, Inc., 6277 Sea Harbor Drive, Orlando, Florida 32887-6777.

*Mathematica* is a registered trademark of Wolfram Research, Inc.

## Academic Press

*An imprint of Elsevier Science*

525 B Street, Suite 1900, San Diego, California 92101-4495, USA

<http://www.academicpress.com>

## Academic Press

*An imprint of Elsevier Science*

84 Theobald's Road, London WC1X 8RR, UK

<http://www.academicpress.com>

## Academic Press

*An imprint of Elsevier Science*

200 Wheeler Road, Burlington, Massachusetts 01803, USA

<http://www.academicpressbook.com>

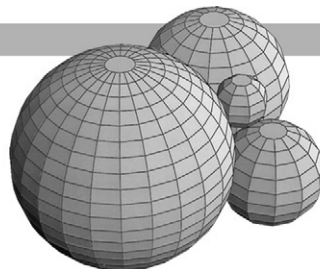
Library of Congress Control Number: 2002107986

International Standard Book Number: 0-12-351666-8

PRINTED IN THE UNITED STATES OF AMERICA

02 03 04 05 06 9 8 7 6 5 4 3 2 1

# Contents



## *Preface*

*xiii*

## *Part I Learning the Basics*

---

*1*

### *Mathematica Basics: An Introduction*

*3*

Introduction	3
Arithmetic	3
Mathematical Functions	5
Symbolic Computations	6
Exact and Approximate Solutions for Equations	7
Graphs of Functions of One Variable	9
Graphs of Parametric Equations	10
Graphs of Surfaces in Three Dimensions	11
Graphs of Contour Curves for Surfaces	11
Matrix Algebra	12
Random Numbers	13
Exercises	14

### *Mathematica Basics: Help*

*19*

Introduction	19
The ? and ?? Operators	19
The Help Browser	20
Interrupting Mathematica Computations	21
Memory Management in Mathematica	21
Exercises	22

## ***Mathematica Basics: Notebooks*** **25**

Introduction	25
Notebook Toolbars	25
Format Preferences	25
Page Break Preferences	26
Cell Groupings	26
Tag Names	27
Links	28
In/Out Names	28
Hyperlinks to Documentation	29
Exercises	29

**Associated Notebook on CD-ROM: Cell Hierarchy**  
 CellHierarchy.nb

## ***Mathematica Basics: Text and Typesetting*** **31**

Introduction	31
The BasicTypesetting and CompleteCharacters Palettes	31
Algebraic Manipulation Palette	33
Basic Input Palette	34
Basic Calculations Palette	34
Exercises	35

## ***Mathematica Basics: Packages*** **37**

Introduction	37
Loading Standard Packages	37
Forgetting to Load a Package	39
Exercises	41

## ***Part II Designing Functions*** **43**

---

### ***Values, Variables, and Assignments*** **45**

Introduction	45
The Immediate Assignment Operator = for Values	46
The Delayed Assignment Operator := for Functions	49
Evaluation Issues for the Immediate and Delayed Assignment Operators	53

Computational Efficiency and the Assignment Operators	56
Exercises	58

## ***Functions***

63

Introduction	63
Functions with a Single Argument	63
Functions with Several Arguments	69
Functions with Structured Arguments	71
Functions with Default Values for Arguments	75
Functions with a Varying Number of Arguments	80
Exercises	81

## ***Recursive Definitions***

87

Introduction	87
Functions with a Single Recursive Argument	87
Functions with Several Recursive Arguments	92
Limitations to Recursive Computations	95
Exercises	99

## ***Substitution Rules and Optional Arguments***

107

Introduction	107
Substitution Rules and the Replacement Mechanism	107
Substitution Rules and Interactive Computations	110
Options for Built-in Functions	111
Defining an Option for a Function	114
Exercises	117

## ***Four Spheres Packing Problem***

119

Introduction	119
Analysis of the Problem	119
Computational Solution of the Problem	120
Graphic Rendering of the Solution	121
Exercises	123

### **Associated Notebook on CD-ROM: Two-dimensional Views of the Spheres**

Views2D.nb

- Cross-sectional View of Two Touching Spheres
- Vertical Projection of Three Touching Spheres



**Associated Folder on CD-ROM: Four Spheres Drawings**

Views2D-Drawings.nb

Evaluation of Views2D.nb

SphereDrawings.nb

Evaluation of “Graphic Rendering of Solution”

***Part III Designing Programs*** **125**

---

***List Processing Functions*** **127**

Introduction 127

Processing Lists with the Map[ ] Function 127

Applications of the Map[ ] Function to Graphics Objects 134

List Manipulation, Element Extraction, and the Fold[ ] Function 140

The Functions Head[ ], Apply[ ], Outer[ ], Depth[ ], and Position[ ] 146

Exercises 153

***Iterations with Loops*** **159**

Introduction 159

Using the Do[ ] Loop: Basics 159

Using the Do[ ] Loop: A Hula Hoop Animation 163

Using the While[ ] Loop: Basics 165

Using the While[ ] Loop: Termination Conditions and the  
Bisection Method 167

Using the For[ ] Loop: Basics 170

Using List Processing Functions as Alternatives for Loops 173

The Collatz Function 175

Exercises 176

***Computations with Modules and Local Variables*** **181**

Introduction 181

Using the Module[ ] Function: Basics 181

Using the Module[ ] Function: Returning Numbers and Lists 184

Using the Module[ ] Function: Rendering Graphics Objects 188

Exercises 191

**Part IV Exploring Advanced Features****197*****Advanced Mathematica: Options*****199**

Introduction	199
Defining a Single Option for a Function	199
Defining Several Options for a Function	204
Using Built-in Graphics Options in User-Defined Functions	208
Defining Options for One Function That Are Options in Several Other Functions	211
Exercises	216

***Advanced Mathematica: Hyperlinks and Buttons*****221**

Introduction	221
Hyperlinks within a Notebook	221
Jumps within a Notebook without Hyperlinks	222
Hyperlinks between Notebooks	222
Hyperlinks to the Help Browser	222
Hyperlinks to Internet Resources	223
Hyperlinks as Buttons	224
Creating a Typesetting Palette	225
Creating a Palette of Expressions	227
Creating a Palette of Characters and Expressions	228
Creating an Evaluation Palette	229
Printing a Palette	231
Exercises	231

**Associated Notebook on CD-ROM: Target of a Hyperlink**

JumpTarget.nb

**Associated Folder on CD-ROM: Sample Palettes**

n-CompleteCharacters.nb

p-Pi.nb and n-Pi.nb

p-Trig.nb and n-Trig.nb

p-Logic.nb and n-Logic.nb

p-Evaluate.nb and n-Evaluate.nb

***Advanced Mathematica: Packages*****233**

Introduction	233
Contexts and Names	233

Initialization Cells	236
The Basic Scheme of a Package	236
Package Files	244
A Package for an Iteration Function	247
A Package for Gram-Schmidt Orthogonalization	250
Loading Packages	254
Exercises	256
<b>Associated Folder on CD-ROM: PackagesSupport</b>	
InitCellA.nb and InitCellA.m	
InitCellM.nb and InitCellM.m	
Template.nb and Template.m	
Thermometer.nb and Thermometer.m	
Iteration.nb and Iteration.m	
GramSchmidt.nb and GramSchmidt.m	

## ***Advanced Mathematica: Files, Data Exchange, and Conversions*** **261**

Introduction	261
Directories and File Paths	262
Import and Export of Data	263
Conversions to Other File Formats	266
Protected Functions	267
Exercises	270

### **Associated Folder on CD-ROM: DataTests**

Survey.xls

Files generated from the notebook:

- GoodAges.dat
- SortedAges.dat
- Std100CSV
- Std100Lines
- Std100List
- Std100Table
- Std100.xls
- SortedAges.xls

### **Associated Folder on CD-ROM: HTMLDemos**

SphereCenters.nb

SphereCenters Folder:

- contains the files generated by Mathematica

SphereDrawings.nb

SphereDrawings Folder:  
 contains the files generated by Mathematica  
 Views2D-Drawings.nb  
 Views2D-Drawings Folder:  
 contains the files generated by Mathematica

## ***Part V Student Projects***

273

### ***Student Projects***

275

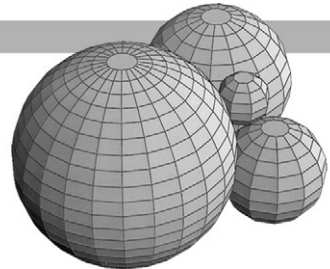
Introduction	277
Arnold's Cat Map and Chaotic Mapping	280
Bouncing Balls	281
Collatz's Function	282
Conway's Challenge Sequence	283
Exponentially Damped Surfaces	284
Finite Automata	285
Fractals and Chaotic Boundary Sets	286
Fractals and Iterated Function Systems	288
Geometric Optics and Lens Systems	289
Groups of Rigid Motions	290
Growth Rates of Functions	291
Harmonic Coupled Oscillations	292
Hidden Patterns	293
Implementation of a Package	294
Interpolation of Curves with Cubic Splines	295
Juggling Balls	296
Leasing a Car	297
Markov Chains and Dynamic Models	298
Moiré Fringes	299
Oscillating Mass System	301
Pell's Equation	302
Public Key Cryptography	303
Rainbows	305
Recurrence Relations	306
Spanning Trees of a Graph	307

**Associated Folder on CD-ROM: ProjectsSupport**  
 Deterministic Finite Automata:

DFAIntro.nb and DFA.m  
Iterated Function Systems  
IFSIntro.nb and IFS.m  
Local Parametric Splines  
LPSIntro.nb and LPS.m  
Public Key Cryptography  
RSAIntro.nb and RSA.m

***Index******309***

# Preface



## Overview

*Computing with Mathematica*, the book and CD-ROM, is for readers who want to learn how to use Mathematica as a tool for solving problems in mathematics, computer science, engineering, and other scientific fields, where skills in mathematics and computing are essential. The content of the book and CD-ROM is structured to support the teaching of an undergraduate course in mathematics or computer science.

This course has been designed for the *junior/senior* level of the mathematics and computer science curricula, and it builds on topics and concepts in the sophomore level core of both fields. Students are expected to have completed at least the calculus sequence and a course in linear algebra. Though not essential, it will be helpful if they have had a course in programming so they are familiar with the basic concepts of computer programming. A course in differential equations is not expected, since it is typically not part of the core requirements in computer science. With these prerequisites, the course is appropriate for junior/senior level mathematics, statistics, computer science, engineering, and science students.

The *goal of the course* is that students learn how to use Mathematica to solve problems arising in their major field of study. The course begins with a basic introduction to Mathematica and does not require previous knowledge of the computer algebra system. Students gain experience with the more sophisticated aspects of the system by studying examples and then doing exercises and solving problems where they use the features presented in the examples. The exercises involve a large variety of topics from mathematics and computer science and are designed to encourage the students to investigate many aspects of each topic. The exercises require knowledge of calculus and linear algebra but not of differential equations.

At the end of the course all students are expected to complete a *project*, which they may choose from their major field of study. The project should be a significant piece of work and the course structure allows about four to five weeks for the completion of the project. The project should be written up as a Mathematica notebook where explanatory text alternates with executable code, where graphics can easily be generated and displayed, and where links may lead to resources in other notebooks and on the Web. A list of suggestions for student projects is included in Part V

of this book and in electronic form on the CD-ROM. Students can choose a project from this list or they can design their own.

The *teaching model* is a laboratory course based on an interactive text. The textbook consists of a sequence of Mathematica notebooks that should be available in electronic form to the students in the class on the servers in the departmental computer labs, the servers in the campus computer labs, and also for use on the students' individual computers. Students should be able to work through the notebooks in the departmental labs as well as in general campus labs, and on their home computers. Extensive use of *electronic communication* and *remote access software* allows the laboratory to be located where a student and a computer running the software interact, and where a connection to the Internet is provided. A course web site based on modern electronic course management systems such as Blackboard or WebCT could support the teaching of this course in a distance learning format.

For a three-credit undergraduate course students should be expected to complete all the exercises in Part I, most of the exercises in Part II, and a selection of the exercises in Part III. The assignments in Part III could reflect the interests of the instructor and the students. Part IV could be optional or a few topics could be chosen from this part. The sequence of notebooks in Parts I, II, and III reflects the dependence of the topics. The notebooks in Part IV are independent of each other. Students might be interested in some of the topics in Part IV because they want to use them for their projects. While working on Part III, the students should begin to work on their projects. They could choose a project from the list in Part V, or they could choose their own.

## Chapter Descriptions

---

In Part I we introduce the basic features of Mathematica, its numerical, symbolic, and graphical capabilities, and its online help features. We also provide a brief introduction to the basic structure and usage of the notebook front-end, the typesetting capabilities, and the use of built-in Mathematica packages.

In Part II we discuss immediate and delayed assignment for functions, functions with structured arguments, and how to define functions with default values for arguments. We explain how to define and use functions with a single and with several recursive arguments. As a naming convention, we consistently start the names of user-defined functions with a lowercase *u* and otherwise use the standard capitalization scheme for Mathematica functions. This chapter also contains an introduction to substitution rules and replacement mechanisms and to the process of defining options in user-defined functions. The chapter ends with a notebook that calculates and renders the picture of the four touching spheres on the cover of the printed text.

In Part III we introduce the basic aspects of programming in Mathematica: the use of list processing functions and the use of loops and modules. This material also is used to introduce animations and computations with large lists of objects. The order of the topics, with list processing functions coming before loops and modules, was chosen to encourage students to think in terms of functional programming rather than procedural programming.

In Part IV we discuss selected advanced topics and demonstrate how we can define Mathematica objects that we have used in built-in functions and notebooks. These objects include option arguments in user-defined functions; hyperlinks, buttons, and palettes; the design of packages; reading from and writing to external data files; creating HTML files from notebooks; and modifying built-in functions.

In Part V we provide suggestions for the student projects. We include the time table, the expectations for the written and oral presentation of a project, and the evaluation criteria. There are 25 student projects.

## *New to this Edition*

---

At the suggestion of the reviewers of the first edition, we changed the order of some topics, added a few new topics, and adapted others. The most significant change from the first edition is that this new edition is based on Mathematica 4.1, which led to the inclusion of material not available in earlier versions. Specifically, in Part I we discuss the basic features of the Help Browser, added a notebook about the features of notebooks, preferences, and hyperlinks, and added a notebook on basic typesetting and the usage of the built-in palettes. At the end of Part II we added a notebook that discusses substitution rules, options in built-in functions, and the definition of optional arguments. These topics were moved from Part IV in the first edition. We also added an applications notebook that uses substitutions interactively to create the basic graphics object for the cover of the printed text. In Part III we added more exercises. Part IV starts with a notebook where we discuss the definition of options in functions in more detail than in Part II. The second notebook in Part IV discusses how hyperlinks, buttons, and palettes are constructed. The packages notebook discusses files, directories, package files, and the initialization mechanism. The last notebook introduces the import and export mechanisms as well as data and notebook conversions.

Mathematica 4.1 allows for two-dimensional typing and editing with the use of palettes. We decided to use this feature in text cells for easier readability but not in input cells. A student using the notebooks is not expected to edit the text in the text cells but is expected to work with the input cells and make changes as needed. Linear editing in input cells seems less cumbersome than two-dimensional editing from the palettes.

## *Course Materials*

---

All materials were developed in Mathematica and exist in electronic form and with identical content for the Macintosh and Windows platforms. A copy of Mathematica 4.1 is desirable for work with the notebooks. If version 4.1 is not available, it is possible to use Mathematica 3.1, but in this case the features that are new in version 4.1 will not be accessible and some of the code and formatting will not work. The materials make no use of special fonts or special functions and will run with the version of Mathematica 4.1 as it is shipped from Wolfram Research. A copy of Mathematica is not included.

**CD-ROM** Seventeen electronic Mathematica notebooks are the core of the course materials, the interactive text for the course. The CD-ROM contains these 17 notebooks for version 4.1 of Mathematica in four folders, Part I–IV. A few of the notebooks require associated notebooks or folders, which can always be found in the same folder as the notebook itself. All notebooks can be used with Windows as well as the Macintosh platforms as long as a copy of either Mathematica 4.1 or higher is running on the computer. The CD-ROM also contains, in the folder



for Part V, the *Project Support Folder*. It includes notebooks for some of the suggested student projects. A copy of Mathematica is not included on the CD-ROM.

**Printed Materials** The text, *Computing with Mathematica*, is a printed version of the 17 notebooks on the CD-ROM, which are the core of the course. It is intended as easily accessible reference material for the student as well as the instructor. The notebooks are printed in unevaluated form. The text also contains, in Part V, a printed version of the student projects.

An *Instructor's Guide for Computing with Mathematica* contains sample solutions to all the exercises in the notebooks. Because of the nature of the materials, solutions are not unique and other (and better) solutions are always possible. The *Instructor's Guide* is available in electronic form but not in printed form. It is not included on the CD-ROM but can be obtained from the publisher by qualified instructors.

## *The Student Project*

---

When we teach this course, the student project is the most important component of the course. All students are expected to complete a project that they may choose from their major field of study. Students can work individually or in teams of two. The course structure allows about four to five weeks for the completion of the project. Students write up their project as a Mathematica notebook. They give a 20-minute oral presentation explaining their project and demonstrating their notebook to the entire class. A list of suggestions for student projects is available in Part V. Each suggested project is introduced with a goal statement for the project, a list of resources for the project, and a list of prerequisites. Students can choose a project from this list, modify a suggested project, or they can design their own. We have found that the most successful projects were those designed by students themselves, because they were able to choose a topic that they really wanted to work on.

## *A Note to the Instructor*

---

The exercises in each notebook introduce a variety of mathematical topics that should be familiar to students with the stated prerequisites for the course. This variety may create a need for some instructor guidance if the students don't quite have the mathematical background that is expected. The instructor may have to provide some of the necessary background information on topics that come up in the exercises. The text makes no attempt to teach the mathematics that may be needed in the exercises. With few exceptions, a standard text on *Calculus and Analytic Geometry*, a standard text on *Linear Algebra*, or a standard text on *Discrete Mathematical Structures* should be all that is needed. The *Instructor's Guide for Computing with Mathematica* contains suggestions for solutions to all the exercises. Exercises are placed at the end of each notebook rather than interspersed in the text as one might expect in an interactive medium. This choice was made to allow for a collection of more challenging exercises based on the material in a whole notebook instead of having only minor interactive exercises based on small parts of a notebook.

## *A Note to Students*

---

This course assumes that you remember what you learned in calculus, in linear algebra, and in high school geometry. When you work on the exercises, occasionally you may have to get out a calculus or linear algebra textbook and revisit some topics that you studied in earlier courses.

## *A Sample Course*

---

At the University of Michigan-Dearborn a semester allows for approximately 14 weeks of classes and one week for final examinations. The following course outline is based on the syllabus that has been used for several semesters at the University of Michigan-Dearborn and, with minor variations, at Eastern Michigan University.

The enrollment in the class is limited to 20 students. Students' performance was assessed through homework, a Mathematica proficiency examination (midterm exam), and the final project.

**Weeks 1 and 2:** Part I; do all exercises in the notebooks of Part I. Get proficient with the use of the campus network, the course management system, electronic communication, and file transfer mechanisms for homework assignments.

**Weeks 3, 4, and 5:** Part II; do all exercises in the notebook Values, Variables, and Assignments. In the notebook Functions do exercises 1–7 and either 8 or 9. From the notebook Recursive Definitions choose a selection of exercises, for example, numbers 1, 2, 3, 5, 8, 9, 12, and 13, or, alternatively, numbers 1, 2, 3, 4, 6, 7, 10, and 11. From Substitution Rules and Optional Arguments do exercises 1 and 2.

**Weeks 6 and 7:** Part III; do at least exercises 1, 2, 3, either 4 or 5, and 6 in List Processing Functions.

**Week 8:** Part III; do a selection of exercises in Iteration with Loops, for example, numbers 1, 2, 3, 5, and one from 6, 7, or 8. Students choose a topic for the final project during this week.

**Week 9:** Proficiency examination. Students also collect materials for the final project and study the necessary mathematics for their final project. Students submit a two-page proposal for the final project with an outline of what they want to accomplish. The proposal also has to contain a detailed time table for completing the project.

**Week 10:** Part III, section 3, Modules and Local Variables; do four of the exercises. Students begin work on the final project.

**Weeks 11 and 12:** Students work on their final projects and study sections from Part IV relevant to their projects.

**Week 13:** Students continue work on the projects, they also start writing their final project report and prepare for the oral presentation of the final project.

**Week 14:** Project presentations. Students continue writing their final project report and prepare for the oral presentation.

The instructor should be prepared to give a lot of individual help and guidance during weeks 9, 11, 12, and 13.

The class is scheduled in a departmental teaching computer lab for three hours per week. During the first week students have to attend class to get organized, to understand the course management system, the campus networks, electronic communication, and file transfer mechanisms for homework assignments. During this time the instructor also gives several short lectures about the mathematics that is needed to do the homework assignments. Later in the semester students do not need to come to the departmental teaching computer lab to work on their assignments. They will typically work on their home computers or in the campus computer labs and come to the departmental teaching computer lab during the scheduled time slots only when they have difficulty with an assignment. Some students choose always to work in the departmental teaching computer lab during the scheduled time. The instructor will schedule several mandatory class days during the semester as needed. The proficiency (midterm) exam is taken in the departmental teaching computer lab, and all students are required to attend when oral presentations of final projects are scheduled.

Homework is submitted electronically, and corrections and homework scores are sent out electronically as well. At the beginning of each week the instructor sends out a Class Update outlining the goals for the week and announcing deadlines. Deadlines for homework are enforced to keep students on schedule.

## *Acknowledgments*

---

The first edition of this book and CD-ROM, published in 1998, was supported by the National Science Foundation with Grant No. DUE 9451360. Kevin Burke, then a faculty member at Siena Heights College, participated in the design of the project and the National Science Foundation grant. Students and faculty at the University of Michigan-Dearborn, Eastern Michigan University, and Siena Heights College who used preliminary versions and the first edition of this material made many excellent suggestions that led to improvements in the text and the CD-ROM. The detailed and thorough reviews that were received from Thomas Cusick, SUNY at Buffalo; Stephen Brick, University of South Alabama; Don Hazlewood, Southwest Texas State; Anthony Peressini, University of Illinois at Urbana; and Bengt Fornberg, University of Colorado led to the rearrangement of some topics and the inclusion of new material. It has been a pleasure to work with the editorial staff at Academic Press. Thanks to all!

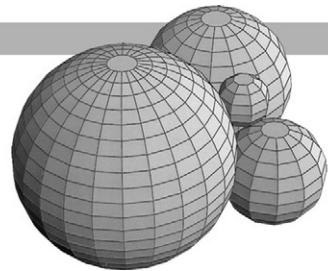
I

---

## *Learning the Basics*

This Page Intentionally Left Blank

# *Mathematica Basics: An Introduction*



## *Introduction*

This is a short introduction of the basic features of Mathematica. It is not in any way meant to be complete but is intended to acquaint you with a few Mathematica commands that are relevant and useful for the study and application of mathematics, science, and engineering.

Mathematica can perform numerical and symbolic computations, it has versatile graphics capabilities, and it is a high-level programming language that allows the user to add individual extensions to the system. To get acquainted with Mathematica, read through the text that follows, enter the input lines, and watch the output appear. In this introductory section we use only very basic features that have been available in earlier versions of Mathematica. A reader or student who already has basic familiarity with Mathematica may use this section as a brief review or move on to the next notebook.

The boldfaced lines below are input lines. To enter them for evaluation, put the cursor anywhere in the input line, then press **SHIFT+RET**; that is, hold the SHIFT key down and press the RETURN key. On a Macintosh, pressing the **ENTER** key on the numeric pad will also work.

## *Arithmetic*

Mathematica can do arithmetic. Enter the input lines and check if the answers make sense. Note how Mathematica assigns numbers to the input and output lines.

**2 + 3**

**2 + 3 + 4 + 5**

**1/3 + 1/2**

**4/16**

**2.5 + 6.33**

**2\*3**

You can omit the `*` for multiplication; blank spaces also denote multiplication in Mathematica.

`2 5`

`2 3 4`

The `^` key is used for exponentiation.

`2 ^ 3`

`67 ^ 2`

`2 ^ 100`

To get the decimal expansion of a number  $k$ , use `N[k]`, and `N[k, 40]` for 40 significant digits.

`N[2/3]`

`N[2/3, 40]`

`N[22/7, 25]`

`N[Pi]`

`N[Pi, 25]`

You can put several evaluations into a single input cell. Complete expressions that are written on different lines in a single input cell are treated as independent expressions. However, an expression may also extend across several lines.

`25 * 5`

`17 + 8`

`25 * 5 - 17 + 8`

`4 * 3`

`-5`

Round parentheses are used to group portions of an algebraic expression.

`(25 - 7) * (25 + 7)`

When an expression is terminated by a semicolon, the result of the computation is not displayed.

`(25 - 7) * (25 + 7);`

## *Mathematical Functions*

---

Mathematica has a large number of built-in mathematical functions. Some of them are given in the following examples. The names of all built-in Mathematica functions begin with capital letters, and the arguments of all Mathematica functions are enclosed in square brackets.

**Sqrt[9]**

**Sqrt[7]**

**N[Sqrt[7]]**

**Sin[Pi]**

**Sin[Pi/4]**

**Cos[1.6]**

**Tan[Pi/4]**

**Exp[ ]** is the exponential function and **Log[ ]** is its inverse, the natural logarithm, not the common logarithm with base 10. **Log[10, x]** is the base 10 logarithm of x.

**Exp[0]**

**Log[1]**

**Log[E^3]**

**Log[10, 100]**

**Abs[ ]** is the absolute value function.

**Abs[-2.3]**

There are several additional numerical functions such as **Round[ ]** and **FractionalPart[ ]** that extract parts of a number.

**Round[2.3]**

**Round[2.8]**

**FractionalPart[-5.123456]**

**FractionalPart[10.0]**

**Round[-12]**



## *Symbolic Computations*

---

Mathematica can work with algebraic formulas that contain symbols such as  $a$ ,  $b$ ,  $x$ , and  $y$ , as well as with numbers. Symbols in such formulas do not have a value. If a value is assigned to a symbol then it is possible to erase the value again with the `Clear[ ]` function.

```
?Clear
```

```
a = 7
```

```
a
```

```
Clear[a]
```

```
a
```

A symbol may have another symbol as its value.

```
a = Pi
```

```
a
```

Mathematica can carry out standard algebraic simplifications.

```
Clear[x, y]
```

```
x^2 - 3x^3 + 2x^2 - 12
```

```
x y - 2x^2y^3 + 6xy + y^3 x^2
```

```
Factor[x^3 + 2x^2 - 1]
```

```
Factor[x^3 + (5x^2)/6 - 2x + 2/3]
```

```
Expand[(1 + x)^2]
```

```
Expand[(x - 3) (x + 5) (x^2 + x + 1)]
```

```
Simplify[x^2 + 2x + 1]
```

Spaces can be used for multiplication, but be careful not to forget the spaces. If you type  $xy$  with no space between the  $x$  and the  $y$ , Mathematica will interpret this as a single symbol with the name  $xy$ , not as the product of  $x$  and  $y$ . However, in the product of a number with a symbol, as in  $2x$  or  $2\cos[x]$ , the space can be omitted.

```
Clear[x, xy]
```

```
xy + xx
```

```
3x^2 + 1/2x
```

```
3x^2 + 1/(2x)
```

Derivatives of functions and of expressions can be taken with `D[ ]`.

```
Clear[n, x, z]
```

```
D[x^4, x]
```

```
D[x^n, x]
```

```
D[5x^3 Sin[x + Pi/4] - 17x Log[x], x]
```

```
D[Sin[x^2], x]
```

```
D[Sin[x^2], z]
```

We also can use the prime symbol,  $f'(x)$  and  $f''(x)$ , to compute the derivatives of a function  $f(x)$ .

```
Sin'[x]
```

```
Sin''[x]
```

Similarly, symbolic computation of indefinite integrals is done with `Integrate[ ]`.

```
Integrate[x^3, x]
```

```
Integrate[Cos[x], x]
```

```
Integrate[x^n, x]
```

## *Exact and Approximate Solutions for Equations*

---

Mathematica can solve polynomial equations in one variable if the degree of the polynomial is less than five. An expression like  $x^2 + 2x - 7 == 0$  represents an equation in Mathematica. Notice the double equality sign. You have to specify that you want to solve for  $x$ . Mathematica will give you the real as well as the complex roots of the equation. All solutions are displayed as replacement rules, for example as,  $x \rightarrow -1$  or as  $x \rightarrow -1$ , and not as values.

```
Clear[x]
```

```
Solve[x^2 - 5x - 6 == 0, x]
```

```
Solve[x^2 + 2x + 1 == 0, x]
```

```
Solve[x^2 + 2x - 7 == 0, x]
```

You can get the numerical values of the solutions by applying the function `N[ ]` to these solutions. `N[%]` refers to the decimal expansion of the previous output. The symbol `%` is the last output and `%%` is the second to last output; you can refer to any previous output, `k`, either by `Out[k]` or by `%k`.

```
N[%]
```

Equations can also be solved symbolically.

```
Clear[a, b, c, x]
```

```
Solve[ax^2 + bx + c == 0, x]
```

When we substitute the solutions for the unknown `x` of this equation into the quadratic expression we get a pair of zeros. The substitution can be done with the `/. operator`.

```
ax^2 + bx + c /. %
```

```
Simplify[%]
```

Here are some examples with complex roots.

```
Solve[x^2 + 1 == 0, x]
```

```
Solve[x^6 == 1, x]
```

```
N[%]
```

You can also use Mathematica to solve sets of linear equations simultaneously. You enter the list of equations and specify the list of variables to solve for. All lists are entered with braces. Here is a list of two linear equations, to be solved for the variables `x` and `y`.

```
Solve[{2x + y == 0, 2x - y == 1}, {x, y}]
```

Here is another example with three linear equations, to be solved for the variables `x`, `y`, and `z`.

```
Solve[{3x + 4y - 5z == 0,  
      2x - y + z == 1,  
      -x - 3y - z == 3}, {x, y, z}]
```

Following is a case where we get infinitely many solutions.

```
Solve[{2x + y == 1, 4x + 2y == 2}, {x, y}]
```

When we choose a replacement value for `y`, then `x` is determined.

```
% /. y -> 12.5
```

The numerical approximation procedure `FindRoot[ ]` can be used to solve transcendental as well as polynomial equations. `FindRoot[lhs == rhs, {x, x0}]` searches for a solution to the equation `lhs == rhs`, starting with the initial estimate `x0` for the unknown `x` (Newton's method). If the right-hand side of the equation is 0, we can omit it.

```
FindRoot[Log[x] == 0, {x, 2}]
```

```
FindRoot[Log[x], {x, 2}]
```

```
FindRoot[Sin[x] == Cos[x], {x, 1}]
```

```
FindRoot[x^6 - 7x + 5, {x, 0}]
```

## *Graphs of Functions of One Variable*

---

An invocation of the form `Plot[f[x], {x, a, b}]` plots  $f$  as a function of  $x$  for  $a \leq x \leq b$ .

An invocation of the form `Plot[{f1[x], f2[x], f3[x], ...}, {x, a, b}]` plots several functions together.

```
Plot[x^2, {x, -2, 2}]
```

A semicolon at the end of the `Plot[ ]` command suppresses just the output cell containing the value `-Graphics-`. The plot is still displayed.

```
Plot[x^2, {x, -2, 2}];
```

```
Plot[(1 - x^2)^(1/2), {x, -1, 1}];
```

```
Plot[x^3 - 2x + 1, {x, -2, 2}];
```

```
Plot[Sin[x], {x, 0, 2Pi}];
```

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2Pi}];
```

`Plot[ ]` has many options. We will give only a few examples of the options here. If you need a different range than the one chosen by Mathematica, you can use the option `PlotRange`. You specify precisely which range you want, as shown in the following examples. The option `AxesOrigin` will set the axes where you want them.

Each option consists of the name of the option, the rule replacement operator, and the value of the option. You can enter the rule replacement operator as the two-character sequence `->` consisting of the minus and the greater-than symbols or as the single-character rule symbol `→` from the `CompleteCharacters` palette. In an input cell the rule replacement operator will be displayed as the symbol `→`.

```
Plot[x^6 - 5x, {x, -5, 5}];
```

```
Plot[x^6 - 5x, {x, -5, 5}, PlotRange → {-20, 50}];
```

```
Plot[x^6 - 5x, {x, -5, 5},  
      PlotRange → {-20, 50}, AxesOrigin → {1, 20}];
```

You can color your graphs with the option `PlotStyle`. `RGBColor[red, green, blue]` specifies that graphical objects are to be displayed, if possible, in the color given. The list of functions is matched with the list of plotting styles. Numbers between 0 and 1 can be entered to vary the

amount of red, green, and blue. Some examples follow:

```
Plot[Sin[x], {x, -6, 6}, PlotStyle → RGBColor[1, 0, 1]];

Plot[Sin[x], {x, -6, 6}, PlotStyle → RGBColor[.4, .3, .6]];

Plot[{Sin[x], Cos[x]}, {x, -6, 6},
     PlotStyle → {RGBColor[0, 0, 1], RGBColor[0, 1, 1]};

Plot[{Sin[x], Sin[2x], Sin[.5x]}, {x, -Pi, Pi},
     PlotStyle → {RGBColor[.4, .3, .6], RGBColor[0, .3, 1],
                  RGBColor[1, 0, .6]}];
```

## *Graphs of Parametric Equations*

---

Functions that are defined by parametric equations can be graphed with `ParametricPlot[ ]`.

`ParametricPlot[{f1[t], f2[t]}, {t, a, b}]` produces a parametric plot with x and y coordinates  $f_1$  and  $f_2$  generated as functions of t. Some examples follow.

```
ParametricPlot[{2Cos[t], 2Sin[t]}, {t, 0, 2Pi}];
```

To make circles look like circles, we can use the option `AspectRatio`.

`ParametricPlot[{f1[t], f2[t]}, {t, a, b}, AspectRatio → Automatic]` produces the same parametric plot as before; the only difference is that the unit distance along the x-axis and along the y-axis are now the same.

```
ParametricPlot[{2Cos[t], 2Sin[t]}, {t, 0, 2Pi},
               AspectRatio → Automatic];

ParametricPlot[{2 + Cos[t], 2 - Sin[t]}, {t, 0, 2Pi}];

ParametricPlot[{2 + Cos[t], 2 - Sin[t]}, {t, 0, 2Pi},
               AspectRatio → 1];

ParametricPlot[{1 + t, t^2}, {t, -5, 5}];

ParametricPlot[{1 + t, t^2}, {t, -5, 5},
               AspectRatio → Automatic];

ParametricPlot[{1 + t, t^2}, {t, -5, 5}, AspectRatio → 1];

ParametricPlot[{1 + t, t^2}, {t, -5, 5}, AspectRatio → 2];

ParametricPlot[{1 + t, t^2}, {t, -5, 5}, AspectRatio → 1/2];

ParametricPlot[{Sin[t], Sin[2t]}, {t, 0, 2Pi},
               PlotStyle → RGBColor[1, 0, 1]];

ParametricPlot[{Sin[t], Sin[2t]}, {t, 0, 2Pi}, AspectRatio → 1,
               AxesOrigin → {0.5, -0.5}, PlotStyle → RGBColor[1, 0, 0]];
```

## *Graphs of Surfaces in Three Dimensions*

---

The function `Plot3D[ ]` produces three-dimensional graphs. `Plot3D[f[x, y], {x, a, b}, {y, c, d}]` graphs the function  $z = f(x, y)$  over the region in the plane where  $a \leq x \leq b$  and  $c \leq y \leq d$ . Many options for `Plot3D[ ]` are similar to the options for `Plot[ ]`, but some are specific to three-dimensional graphing:

`BoxRatios`  $\rightarrow$  `{1, 1, 0.4}` sets the sidelength ratios for the x-y-z-box enclosing the surface

`ViewPoint`  $\rightarrow$  `{1.3, -2.4, 2.0}` sets the point from which to look at the surface

Our first example of a surface does not use any of the options. The default viewpoint, unless otherwise specified, is the point  $(1.3, -2.4, 2.0)$ . The default box ratios are `{1, 1, 0.4}`.

```
Plot3D[y^2 - x^2, {x, -3, 3}, {y, -4, 4}];
```

Mathematica's options can improve the graph of this hyperbolic paraboloid. Enter the next line and make sure you understand the meaning of viewpoint and box ratios.

```
Plot3D[y^2 - x^2, {x, -3, 3}, {y, -4, 4},  
  BoxRatios  $\rightarrow$  {1, 1, 1}, ViewPoint  $\rightarrow$  {3, 2, 1},  
  AxesLabel  $\rightarrow$  {x, y, z}];
```

Selection of a good viewpoint can help to exhibit the characteristic features of a surface. In the following picture we chose the specific viewpoint as well as identical plotting ranges for  $x$  and  $y$  in order to show the saddle points and relative extrema of the surface.

```
Plot3D[x^4 - y^4 - 2x^2 + 2y^2 + 1, {x, -2, 2}, {y, -2, 2},  
  BoxRatios  $\rightarrow$  {1, 1, 1}, ViewPoint  $\rightarrow$  {-1.75, -2.9, 1},  
  PlotPoints  $\rightarrow$  30];
```

Mathematica's 3D Viewpoint Selector can be used to experiment with different viewpoints. It can be found by selecting **Input**  $\triangleright$  **3D ViewPoint Selector...** in the menu bar. To see how it works, open it, choose either Cartesian or spherical coordinates, and rotate the coordinate frame by clicking and dragging the mouse. As you rotate, watch the change in the coordinates of the viewpoint. The viewpoint can be selected by rotation and then pasted into the input line that generates the surface. You will see the details of this process in Exercise 7 at the end of this notebook.

## *Graphs of Contour Curves for Surfaces*

---

An alternate to the graphic representation of a surface with the `Plot3D[ ]` function is a two-dimensional plot of the contour curves of the surface. Such a plot is produced by the function `ContourPlot[ ]`. Its argument specifications are those of the `Plot3D[ ]` function except that the `ContourPlot[ ]` function has a different list of options. We show some contours for the surfaces in the previous section.

Our first example does not use any of the options.

```
ContourPlot[y^2 - x^2, {x, -3, 3}, {y, -2, 2}];
```

Since the plotting range is not a square, the contour curves appear to be nonsymmetric.

We can change the number of the contours of this hyperbolic paraboloid, change their accuracy, and suppress the shading or the contourlines themselves, among other options.

```
ContourPlot[y^2 - x^2, {x, -3, 3}, {y, -3, 3},
  PlotPoints → 10, Contours → 15, ContourLines → True,
  ContourShading → False];
```

```
ContourPlot[y^2 - x^2, {x, -3, 3}, {y, -2, 2},
  PlotPoints → 25, Contours → 30,
  ContourLines → False, ContourShading → True,
  AspectRatio → Automatic];
```

Selection of an appropriate number of PlotPoints and Contours can help to exhibit the characteristic features of a surface.

```
ContourPlot[x^4 - y^4 - 2x^2 + 2y^2 + 1,
  {x, -1.75, 1.75}, {y, -1.75, 1.75},
  PlotPoints → 30, Contours → 30];
```

## Matrix Algebra

---

Matrices can be entered as lists of rows or with the BasicTypesetting palette. How to use the BasicTypesetting palette will be discussed in Exercise 11 at the end of this notebook. Following are two  $3 \times 3$  integer matrices typed as lists of rows. Both have to be entered; that is, the input cell containing them must be evaluated before you can perform matrix operations.

We save the two matrices in the symbols a and b, since we use the matrices in several computations here:

```
Clear[a, b]
a = { {1, 2, 3}, {-1, 4, 2}, {0, 8, -4} }
b = { {2, -7, 4}, {3, 5, -1}, {5, -2, 3} }
```

We can display a matrix in standard rectangular form.

```
MatrixForm[a]
MatrixForm[b]
```

Matrix multiplication is done by putting the period symbol, (.), between the two matrices to be multiplied.

```
a.b
```

```
MatrixForm[a.b]
```

To select a row of a matrix specify the row in double brackets. That is, `a[[2]]` will select the second row of the matrix `a`. To select an element of a matrix specify the row and column in double brackets, as in `a[[1,3]]`, which selects the element in row 1 and column 3.

```
a[[2]]
```

```
a[[1, 3]]
```

`MatrixPower[a, n]` computes the  $n^{\text{th}}$  power of `a`; it evaluates the product of `a` with itself `n` times.

```
MatrixPower[a, 4]
```

Mathematica provides a large number of matrix operations such as the transpose, the inverse, and the row-reduced form of a matrix, as well as functions that compute the determinant and the eigenvalues of a matrix.

```
Transpose[a];  
MatrixForm[%]
```

```
Inverse[a];  
MatrixForm[%]
```

```
MatrixForm[a.Inverse[a]]
```

```
Det[a]
```

```
RowReduce[a];  
MatrixForm[%]
```

```
Det[b]
```

```
RowReduce[b];  
MatrixForm[%]
```

```
Eigenvalues[b]
```

## *Random Numbers*

---

Random number generators are useful to provide numbers and data for tests and experiments. Mathematica's function `Random[ ]` generates uniformly distributed pseudorandom numbers in a specified range. In the example that follows we ask for an integer between 0 and 50 inclusive.

```
Random[Integer, {0, 50}]
```

In the case of complex numbers Mathematica will return a number whose real part is within the specified real range and whose imaginary part is within the specified imaginary range. The symbol `I` or  $i$  represents the imaginary unit  $\sqrt{-1}$ .

```
Random[Complex, {1 + 2I, 2 + 3I}]
```



If no range is specified it is assumed to be between 0 and 1 for the real and imaginary parts, respectively.

**Random[Complex]**

If no arguments are specified a floating point number between 0 and 1 will be generated.

**Random[]**

The random number generator is also useful for generating vectors, matrices, and tables. In the following examples, the first input line shows how to do this. Each time you use this input line to generate a matrix, you should expect a different result since the entries are randomly chosen.

```
Table[Random[Real, {-2.5, 7.5}], {i, 1, 8}]
```

```
Table[Random[Integer, {-10, 10}], {i, 1, 3}, {j, 1, 4}];  
TableForm[%]
```

```
Table[ Random[Integer, {-10, 10}], {i, 1, 2}, {j, 1, 3}];  
MatrixForm[%]
```

The next input line creates a random universe of one thousand data points. The semicolon at the end of the first line suppresses the output of the table of values. Evaluate the next input cell several times and with different ranges for *i*.

```
Table[Random[Real, {-10, 10}], {i, 1, 1000}, {j, 1, 2}];  
ListPlot[%, Axes → None, Frame → True, AspectRatio → 1];  
Length[%]
```

## *Exercises*

---

### *Introduction*

The following exercises reinforce the usage of the basic Mathematica operations that we introduced in this chapter. We encourage you to do more than simply solve the exercises. Experiment with different functions, different arguments, and variations of the Mathematica commands.

**EXERCISE 1:** Computing approximate values.

- Compute the square root of 245663 in the standard precision of Mathematica and also to 50 decimal places.
- Compute the square root of the complex number  $2 + i$ .
- Compute  $\pi$  to 300 places.

**EXERCISE 2:** Solving equations in a single unknown.

- Solve the equation  $x^3 - 4x + 1 = 0$ .
- Solve the equation  $\cos x = x$ .

**EXERCISE 3:** Solving systems of equations in several variables.

(a) Find the solution for the system and check your result.

$$a + b + c = 1$$

$$4a + 2b + c = 0$$

$$9a + 3b + c = 2$$

(b) Find the solution for the system and check your result.

$$5^{\frac{1}{2}}a + 4.1b - 6.0c = 0.0$$

$$7.5a - 2^{\frac{1}{3}}b + 1.25c = 1.5$$

$$1.37a + 3^{\frac{1}{2}}b + 9.75c = -3.2$$

**EXERCISE 4:** Understanding the various kinds of parentheses.

Explain Mathematica's use of the four different kinds of brackets that we have introduced in this section: round parentheses, square brackets, wavy braces, and double square brackets. Give at least two examples for the usage of each type of bracket.

**EXERCISE 5:** Dealing with multiple-line input cells.

Decide how Mathematica interprets each two-line expression and what value or values or error messages are returned when Mathematica evaluates it. Then enter the two-line expression as given into an input cell in order to check your answer.

(a)  $3 + 5 -$   
 $(19 - 7)$

(b)  $3 + 5$   
 $-(19 - 7)$

(c)  $(3 + 5) *$   
 $(19 - 7)$

(d)  $(3 + 5)$   
 $*(19 - 7)$

(e)  $(3 + 5)$   
 $(19 - 7)$

**EXERCISE 6:** Plotting a function of one variable with various options.

(a) Experiment with the option `AspectRatio` until you understand precisely what it does. Don't forget `AspectRatio`  $\rightarrow$  `Automatic` as one possibility.

(b) Experiment with the option `AxesOrigin` until you understand precisely what it does. Don't forget `AxesOrigin`  $\rightarrow$  `Automatic` as one possibility.

(c) Select any other option of the `Plot[]` function and experiment with it.

**EXERCISE 7:** Using the 3D ViewPoint Selector.

Open the 3D ViewPoint Selector from the menu by selecting **Input ► 3D ViewPoint Selector...**

To make changes in the viewpoint, select either Cartesian or spherical coordinates and then select the viewpoint you want by rotating the cube in the 3D Viewpoint Selector or by moving the buttons in the bars for the three coordinates.

To get a new viewpoint into the input line, select the current viewpoint specification in your notebook and then click on the Paste button in the 3D ViewPoint Selector to paste the new viewpoint into the input line you are working with. The new viewpoint will replace the old one. If your graphing command does not contain a viewpoint specification yet, put the insertion cursor where you want the viewpoint specification in your command. Do not forget the separating comma between arguments!

To make changes only in the viewpoint, you can save time by using `Show[ ]` rather than recomputing all the points of the surface. Here is an example: `Show[ %, ViewPoint -> {0.617, -1.087, 3.144}]`.

To reset the viewpoint recall that the default value is `ViewPoint -> {1.3, -2.4, 2.0}`. You can also use the Defaults button in the 3D ViewPoint Selector window to revert to the default viewpoint.

(a) Graph the surface  $f(x, y) = x^4 - 2x^2 + y^2 - 1$  from several different viewpoints.

Two examples of views are provided as a starting point.

```
Plot3D[x^4 - 2x^2 + y^2 - 1, {x, -2, 2}, {y, -2, 2},
  BoxRatios -> {1, 1, 1.5}, ViewPoint -> {-1.75, -2.9, 1.5},
  PlotPoints -> 30];

Show[%, ViewPoint -> {0.617, -1.087, 3.144}];
```

(b) Use the `ContourPlot[ ]` function to obtain an alternate graphical presentation for the function in part (a). Experiment with various options of `ContourPlot[ ]` until you have a display of contours that, in your opinion, shows the features of the surface in an optimal fashion.

**EXERCISE 8:** Graphically exploring a surface.

(a) Use the `Plot3D[ ]` and the `ContourPlot[ ]` functions to get a good idea of the shape of the function

$$f(x, y) = (10(x - y))/(x^2 + y^2 + 1)e^{-((x - y)^2 + (x + y)^2)}.$$

(b) Estimate the locations of the maxima and minima on the surface with the cross-hair tool for the contour graphics output cell. To use the cross-hair tool, click on the graphics output cell to select the graphics. Hold down the COMMAND key (the  $\text{⌘}$  key on a Macintosh keyboard) or the CONTROL key (the  $\text{⌃}$  key on a PC keyboard). This changes the cursor to a cross-hair when it is placed over the selected graphics. The coordinates for the position of the cross-hair are displayed in the StatusArea in the lower left of the Mathematica window.

(c) Describe the symmetries in the shape of the surface.

**EXERCISE 9:** Computing symbolic derivatives.

- (a) Find the derivative of the function  $\sin x + \cos x^2$ .
- (b) Find the derivative of the function  $x \sin x$ .
- (c) Find the derivative of the function  $x^x$ .
- (d) Find the derivative of the inverse tangent  $\tan^{-1}x$ , `ArcTan[x]` in Mathematica; graph the function and its derivative in separate plots.
- (e) Graph the inverse tangent and its derivative in a single plot. Use `ArcTan'[x]` in the `Plot[]` command.

**EXERCISE 10:** Computing symbolic indefinite integrals.

- (a) Find the indefinite integral of  $ax^2 + bx + c$ .
- (b) Find the indefinite integral of  $\frac{1}{x^4-1}$ , then find the derivative of the result.
- (c) Find the indefinite integral of the natural logarithm, `Log[x]`, then find the derivative of the result.

**EXERCISE 11:** Computing definite integrals.

The function `Integrate[]` also supports the computation of definite integrals. You only need to specify the two bounds of integration. We give the Mathematica commands in the linear InputForm, not in the mathematical StandardForm. Here are two examples, one with numeric bounds, the other with symbolic bounds: `Integrate[ x^2, { x, 0, 1}]` and `Integrate[ x^2, { x, s, t}]`.

- (a) Find the definite integral of  $ax^2 + bx + c$  for the interval from 0 to 1.
- (b) Find the definite integral of  $ax^2 + bx + c$  for the symbolic interval from  $s$  to  $t$ .
- (c) Find the definite integral of `Log[x]` for the interval from 1 to  $e$ . The name of  $e$  in Mathematica is `E` or `Exp[1]` or  $e$ .
- (d) Is it reasonable to ask for the computation of the definite integral of `Log[x]` for the interval from  $s$  to  $t$  where  $s$  and  $t$  are symbols representing real numbers? Write down a brief justification for your answer. Then find out what Mathematica does.

**EXERCISE 12:** Creating and editing matrices.

- (a) To create a  $4 \times 4$  matrix, select **Input** ► **Create Table/Matrix/Palette...** from the menu bar. Use the new window to select 4 rows and 4 columns, leaving the **Fill with** and **Fill diagonal** fields unchecked. When you click OK, a  $4 \times 4$  blank matrix will be inserted in the notebook at the place where you left your cursor. Use the `TAB` key to fill in successive entries of the matrix. Create a  $4 \times 4$  matrix of your choice. Assign a name, for instance “ $m$ ”, to your matrix in an input line and enter the input line. Now you can compute `m.m`, `Det[m]`, `Inverse[m]`, and other algebraic expressions. Try it.
- (b) Go back to **Input** ► **Create Table/Matrix/Palette...** in the menu bar and use what you learned in (a) to create tables with or without lines between rows and columns and with or without frames.

(c) Here is a quick and easy way to add rows and columns to matrices and tables: with the cursor in the matrix, press the `CTRL+RET` keys to add a row and the `CTRL+COMMA` keys to add a column. Pressing the `CTRL+SPACE` keys will move the cursor out of the matrix. Practice adding rows and columns to matrices and tables.

**EXERCISE 13:** Using the Random Number Generator.

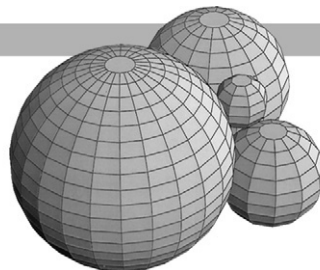
In each part of this exercise invoke the `Random[ ]` function several times.

- (a) Use `Random[ ]` to generate an integer in the range from  $-17$  to  $31$ .
- (b) Use `Random[ ]` to generate a real number in the range from  $-1$  to  $1$ .
- (c) Select some rectangular region in the second quadrant of the complex plane. Use `Random[ ]` to generate a complex number in that region.

**EXERCISE 14:** Computations with a matrix of random integer entries.

- (a) Generate a  $2 \times 3$  matrix of random real numbers between 0 and 1 and display the matrix in rectangular form.
- (b) Generate a  $5 \times 5$  matrix of random integers and then compute its fourth matrix power. Display both matrices in rectangular form.
- (c) Generate a  $2 \times 3$  matrix  $m$  with random integer values in the range  $-50$  to  $+50$ . Then compute the transpose  $t$  of the matrix  $m$ .
- (d) Compute the two matrix products  $m.t$  and  $t.m$  of the matrices  $m$  and  $t$  and compute their determinants.
- (e) Go through the computations of questions (c) and (d) several times, formulate a conjecture from the pattern that you observe in the results, and then prove your conjecture.

# *Mathematica Basics: Help*



## *Introduction*

---

Mathematica's Help Browser allows you to find information about all aspects of Mathematica quickly and efficiently. If you are looking for a function to use, or if you need to look up the specific syntax or usage of a function, you can find what you are looking for in the Help Browser. In fact, the entire book *The Mathematica Book*, by Stephen Wolfram, is available online and the examples in the book are active in the sense that they can be executed and edited.

In what follows, we give some examples for using the online help features.

## *The ? and ?? Operators*

---

To find out about the syntax of the `Plot[]` function, type `?Plot` in an input line and enter it. You should get what is called the usage statement for `Plot[]`. The question mark has to be the first symbol in the input line.

**`?Plot`**

Clicking on the link [More...](#) in the output will open the Help Browser and take you to a page with much more detailed information about the `Plot[]` command. On this page you also find links to the Mathematica book (for instance, Section 1.9.1) with examples on how to use `Plot[]`. Follow the link. The examples are active in the sense that you can change the input lines and produce your own examples. To get back to the `Plot` page in the Help Browser, click the Back button in the upper menu bar of the Help Browser window. At the bottom of the `Plot` page you find "Further Examples," which also contains useful information about `Plot[]`.

The `??` operator gives information beyond the usage statement for a function. It lists the attributes and the options of the function and all default values of the options. The same information is, of course, available from the Help Browser, which can be accessed from the link [More...](#) in the usage statement.

**`??Plot`**

**`??Plot3D`**

**`??FindMinimum`**

If you are not sure if a function exists or what its precise name is, you can use the question mark together with an asterisk (\*). Suppose you want to factor integers but do not remember what kinds of factoring functions exist. The command `?Factor*` will list all functions that begin with the word `Factor`, and provide links to the Help Browser where the specifications of the functions can be found.

### **?Factor\***

Remember that we wanted to factor integers. So we now can click on `FactorInteger` and use the link [More...](#) to get the information we want and to get examples from “Further Examples” at the bottom of the `FactorInteger` page in the Help Browser.

The asterisk symbol can be used to find functions that contain a particular word, not necessarily at the beginning.

### **?\*Plot\***

If you are not sure of the spelling of a function, type the first few letters of the function, such as `Param` for the beginning of `ParametricPlot`, then press the  $\mathbb{X}$  key together with the letter `k` on the Macintosh and `CTRL+[k]` if you are working in Windows. You can then click on your choice in the pop-up menu and have Mathematica complete the function for you. Try it with the next input line and experiment with other functions.

### **Param**

## *The Help Browser*

---

As we have seen before, the Help Browser can be opened by the `?` operator. It can also be opened from the Help menu, which contains several choices for accessing information. Select in the menu bar **Help** ► **Built-in Functions...**, type `Plot` into the text input field, and click the `GoTo` button to the left. This will take you to the same `Plot` page that you visited before. Now select **Help** ► **Mathematica Book...** This takes you to the online Mathematica book, and clicking the items in the slide bar on the upper left can take you to specific places in the book and also to a *Practical Introduction to Mathematica*, which you may want to explore.

If you know the name of a function, but have forgotten its usage or are looking for the names of options of that function, you can type the name of the function in an input cell and with the cursor in that cell select **Help** ► **Find Selected Function...** in the menu bar. This will take you to the page in the Help Browser where the usage of your function is explained and where you have links to the Mathematica book and “Further Examples.” Try it with the `Solve[]` and the `ContourPlot[]` functions.

### **Solve**

### **ContourPlot**

The Help Browser is an excellent source of information about all features of Mathematica. It also gives you access to the Master Index, Getting Started/Demos, and the Standard Packages that you can find under Add-ons. You should spend some time investigating the capabilities and usefulness of the Help Browser.

## *Interrupting Mathematica Computations*

---

To interrupt a Mathematica computation, press the  $\mathbb{M}$  key simultaneously with the PERIOD key if you are working on a Macintosh; press the **[ALT]** key along with the PERIOD key, if you are working in Windows. Try to interrupt the following computation.

```
FactorInteger [135465814365678527298120236367893246578361234567  
432110]
```

It may take quite some time for Mathematica to respond to an attempt to interrupt.

You can also try to stop the evaluation by selecting **Kernel ▸ Abort Evaluation** in the menu bar. This will (usually) stop the computation but will keep the kernel active. To quit the kernel, select **Kernel ▸ Quit Kernel** and specify the kernel you want to quit (probably a choice of Local and Remote). This will (usually) stop the computation by closing the kernel. Your notebook will remain open so that you can save your work. You can restart the kernel by entering any input cell. Sometimes Mathematica seems to ignore all attempts to interrupt and you may have to shut down and start over. In that case you will lose all your work since you last saved your notebook to disk.

## *Memory Management in Mathematica*

---

Some types of computations require large amounts of memory. Rendering large three-dimensional graphs requires resources in the Mathematica front end; computations that store large amounts of data require resources in the Mathematica kernel.

Some general strategies to reduce memory use in the front end are to:

- Close notebooks when they are not needed anymore
- Clear the clipboard of large data
- Save notebooks after large plots and computations

Some general strategies to reduce memory use in the kernel are to:

- Use small settings on the system variable `$HistoryLength`, such as `$HistoryLength = 10`
- Remove variables from the kernel using the built-in function `Remove[ ]`
- Periodically use the built-in function `Share[ ]`

If you work on a PC using a Windows system, then much of the memory is managed in Windows. The physical memory is supplemented by virtual memory. When there is a shortage of disk space, Mathematica might run out of memory or degrade the efficiency of your system.

If you work on a Macintosh, you can control explicitly the size of the front end and the kernel. Find the Mathematica folder on your computer, and double-click on the Executables folder and on the appropriate folder inside it for your system. Now single-click on the MathKernel icon and select **File ▸ Get Info ▸ Memory** from the pop-up menu. At the bottom of the dialog box you can increase or decrease the number for Preferred size (or Current size on some systems).



You can see the actual memory usage in the kernel by displaying the kernel memory monitor. This is a small window containing a gauge for the memory allocated for and actually used in the kernel. The memory window is displayed as follows: Go to the Applications menu in the right-hand corner of the menu bar and switch to the MathKernel as the current application. Now select **File ▸ Show Memory Usage** in the menu bar and position the memory window on the screen to your liking. When you quit the kernel or your current Mathematica session while the memory monitor is shown, then the next time you start the kernel the monitor will be displayed.

You also can adjust the amount of memory to the front end on a Macintosh in the same manner as we just described for the kernel. You need to single-click on the Mathematica icon rather than the MathKernel icon. The front end memory monitor is shown in the lower left-hand corner of the Mathematica notebook window.

## *Exercises*

---

### *Introduction*

All questions in the following exercises are open ended, and you should extend them as much as you like. Whenever you have a question about built-in functions, constants, options, and so on, you should always try to find information in the Help Browser.

**EXERCISE 1:** Using algebraic grouping functions.

- (a) Find out about the Mathematica functions `Apart[ ]` and `Together[ ]`.
- (b) Give some examples where you apply `Apart[ ]` and `Together[ ]`.

**EXERCISE 2:** Using the `Table[ ]` function.

- (a) Find out about the Mathematica functions `Table[ ]` and `TableForm[ ]`.
- (b) Use `Table[ ]` to make a list of the squares of the first 20 positive integers.
- (c) Use `Table[ ]` and `TableForm[ ]` to make a rectangular table of integers so that the numbers in the rows increase as you go to the right and the numbers in the columns increase as you go down.
- (d) Find out about the options of the `TableForm[ ]` function. Apply the `TableHeadings` options to the tables you created in part (c).

**EXERCISE 3:** Hunting for a function name.

- (a) Find all Mathematica functions with `List` somewhere in their name.
- (b) Find out what the function `ListPlot[ ]` does.
- (c) Give several examples that use different options of the `ListPlot[ ]` function.

**EXERCISE 4:** The DisplayFunction option.

Use the features of the Help Browser to find out about DisplayFunction and \$DisplayFunction. You might start by typing DisplayFunction into the text input field in the Help Browser and clicking the GoTo button. Give some examples that use DisplayFunction and \$DisplayFunction.

**EXERCISE 5:** Getting to know Mathematica.

Open the Help Browser, click on Getting Started/Demos, and then on Tour of Mathematica.

- (a) Investigate the content of the notebook Mathematica as a Calculator.
- (b) Investigate the content of the notebook Visualization with Mathematica. Towards the end of this notebook you will find an example of sound output and a link to a Sound Gallery with a variety of sounds. Play some of the sounds by double-clicking on the graphic images. At the very end of the Visualization with Mathematica notebook you will find links to graphical images. Open some of the links.
- (c) Investigate the content of the notebook Handling Data.

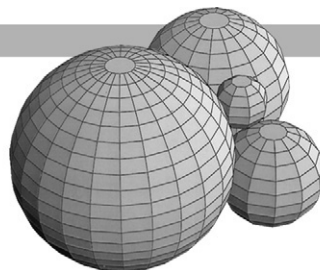
**EXERCISE 6:** Fitting curves to points.

- (a) Find out what the Mathematica function Fit[ ] can do.
- (b) Choose ten points in the plane and then use ListPlot[ ], Fit[ ], and Plot[ ] to get a graph of the least squares line through the points.
- (c) Get a graph of the least squares parabola through the ten points from part (b).
- (d) Show the points and the functions from parts (b) and (c) in one picture.

This Page Intentionally Left Blank

# *Mathematica Basics:*

## *Notebooks*



### *Introduction*

---

This is a short introduction to the features for notebooks in Mathematica. It is in no way meant to be complete, but it is intended to acquaint you with a few of the choices and tools that are available in the menus of the front end of Mathematica. We can use these tools to help us organize the development of a solution for a given problem and the associated computations. Through the cell structure of notebooks as well as the professional typesetting and presentation tools in notebooks we can build within Mathematica the environment we need for a specific problem domain.

We can determine and change the layout of notebooks in the front end with the Option Inspector of Mathematica. Changes to editing options, formatting options, graphics options, and button options are permitted. In Mathematica we can set preferences at the global level, affecting every notebook; at the notebook level, affecting just that one notebook; and at the selection level, affecting only the cells selected in a notebook. The tool for global options is accessible through selecting **Edit ▶ Preferences...** in the menu bar. If the changes should affect only the current selection in a notebook, then we choose **Format ▶ Option Inspector...** .

### *Notebook Toolbars*

---

The first, and very simple, change in the default settings that we have made for the notebooks of this textbook is to add a tool bar and a ruler underneath the title bar of the notebook. Both of these items can be added or removed from the notebook by checking **Format ▶ Show Toolbar** and **Format ▶ Show Ruler** in the menu bar. If we want to see where page breaks will occur when we print the notebook, we can check **Format ▶ Show Page Breaks** in the menu bar. These settings are toggles; that is, the property is toggled on (checked) or it is toggled off (unchecked).

### *Format Preferences*

---

The second change in the default settings that we have made for the notebooks in this textbook deals with automatically italicized words.

Suppose that we selected **Edit ▸ Preferences...** from the menu bar. A new window appears whose default title is **Options for Global Preferences**. When we change the entry in the **Show options values for** from the default **global** to **notebook** the name of the window changes to **Options for BasNote.nb** since now the options apply only to this notebook. In the window we open **Formatting Options ▸ Text Layout Options** and click on the box at the right side of **AutoItalicWords**. In the window that pops up we can change the list of automatically italicized words. We remove a word by selecting it and clicking the **Remove** button. We add a word by clicking the **Add** button and entering the word in the input window that pops up.

Suppose that we selected **Format ▸ Option Inspector...** from the menu bar. A new window appears whose default title is **Options for BasNote.nb** with the **Show options values for** field set as **selection**. We change that value to **notebook** and then proceed as described in the previous paragraph.

We have removed the word *Mathematica* from the list of automatically italicized words. We can change the list, for example, by adding *Mathematica* back to the list. Any word that we want italicized automatically when we type it in a notebook should be added to that list. Use only characters from the standard English alphabet if your copy of *Mathematica* is configured for that language.

## *Page Break Preferences*

---

The third change in the default settings that we have made for the notebooks in this textbook deals with page breaks in the notebooks. We want the page breaks to occur only between cells, and not within cells; that is, an entire cell that fits on one page will always be printed on one page. We use the Option Inspector again for this preference setting.

We open the Option Inspector **Format ▸ Option Inspector...** from the menu bar and set the **Show options values for** field to **notebook** so that the name of the window becomes **Options for BasNote.nb**. Now we select **Cell Options ▸ Page Breaking** and set the property **PageBreakWithin** to **False** by selecting that value from the pop-up menu of the button on the right.

## *Cell Groupings*

---


The fourth change in the default settings that we have made for the notebooks concerns the grouping of cells. The default for this property is automatic grouping, which is set by a check mark in the menu bar **Cell ▸ Cell Grouping ▸ Automatic Grouping**. We have changed this by clicking **Cell ▸ Cell Grouping ▸ Manual Grouping** on the menu bar. This action toggles the check mark between the two properties. The advantage of manual grouping is that we have complete control over how we want to group cells.

A disadvantage of choosing the manual grouping is that any cell newly created at the end of a grouping of cells is put at the outermost level of the cell hierarchy, not inside the grouping of the cell preceding it. We must manually group it with the appropriate group by selecting the sequence of cells that we want grouped together and selecting **Cell ▸ Cell Grouping ▸ Group Cells** on the menu bar.

If a cell is newly created in the interior of a group of cells (that is, it is surrounded by two cells in the grouping) then the new cell is grouped with its predecessor and successor at the same level in the grouping hierarchy.

Other useful tools to structure cells are the divide and merge tools. We can divide a cell into two adjacent cells by putting the cursor at the point where we want to divide the cell and selecting **Cell ▸ Divide Cell** from the menu bar. Both daughter cells are kept at the same grouping level under the manual grouping when the original cell is not the last cell in the grouping level.

We can also merge any number of adjacent cells by selecting the cells at the cell bracket level and selecting **Cell ▸ Merge Cells** from the menu bar. This operation will merge cells at the same or at different levels in the grouping hierarchy.

In some notebooks and palettes that are distributed with Mathematica you will see the icon  in the left margin. It is the GroupOpenCloseIcon, and represents an alternate way of opening and closing groups of cells to the process of double-clicking in the grouping cell brackets in the right margin. We can add those icons to our own notebooks.

To do so, open the Option Inspector from the menu bar **Format ▸ Option Inspector...** and set the **Show options values for** field to **notebook** so that the name of the window becomes **Options for BasNote.nb**. Now we select **Cell Options ▸ Display Option** and set the property **ShowGroupOpenCloseIcon** to **True** by clicking on the associated box at the right margin in the Options Inspector window. Then we close the Option Inspector window. We can see the icons in the notebook when we close all subgroups by selecting first **Edit ▸ Select All** from the menu bar, and then selecting **Cell ▸ Cell Grouping ▸ Close All Subgroups**.

## Tag Names

---

With each cell in Mathematica we can associate a name, called a tag. Tags are needed to identify or to be references for a cell. One important use of a tag is to label a cell as a target for a hyperlink within a notebook or between notebooks. We need to explicitly define each tag. Here is an example of a hyperlink [CellTags](#) that links into the Mathematica Help Browser.

A cell tag is created by first selecting a cell and then choosing **Find ▸ Add/Remove Cell Tags...** in the menu bar. We have added cell tags for every subsection cell in this notebook. We chose to leave the tags hidden. We can display them by clicking on **Find ▸ Show Cell Tags** in the menu bar. It is also possible to assign several tag names to the same cell. We have done this with the “Links” and with the “In/Out Names” subsection cells in this notebook.

We can inspect all cell tags that have been defined in the notebook by selecting **Find ▸ Cell Tags** in the menu bar. We can also easily navigate within a notebook with the help of cell tags. For example, if we want to go to the subsection “Format Preferences” in this notebook, we select **Find ▸ Cell Tags ▸ Format Preferences** in the menu bar and the notebook is scrolled automatically to that subsection.

Another use of a tag is to identify a text cell in which a concept is defined or an input cell in which a function is defined. Once the tag is defined it can be used to build an index for the notebook; see **Find ▸ Make Index...** in the menu bar. Here is a link to the Help Browser for making an index, [Make Index...](#).

## Links

---

We need to be very careful in the control of the cursor when dealing with hyperlinks and buttons that are embedded in a notebook. The shape of the cursor determines the action that is going to occur when we click the mouse. When the cursor is inside a text cell or input cell, for example, it has the shape of the insert cursor. Over the text of a hyperlink the cursor is the bold arrow. As soon as we click on the hyperlink it is executed since a hyperlink always is active. In order to edit a hyperlink, we need to start clear to the side of it, either on the left or on the right. The cursor then has the shape of the insert cursor and we can drag the mouse and select the link for editing as you would select any text segment in Mathematica. Editing of hyperlinks and buttons is explained later in the notebook `BasButton.nb`.

Clicking the hyperlink in this cell will open another notebook with the name of the hyperlink: `CellHierarchy.nb`. The newly opened notebook shows the default hierarchy of cells in Mathematica.

This same action can also be achieved by a button. The button in the next input cell has exactly the same effect as the preceding hyperlink.

**CellHierarchy.nb**

The bracket for the previous input cell contains the letter A in its margin. That indicates an Active cell. Notice that the cell content cannot be edited since the shape of the cursor is the bold arrow anywhere in the cell. The only action that has an effect is to click on the button; that executes its associated action, namely to open the notebook named on the button.

The active status of the input cell was created by checking **Cell ▸ Cell Properties ▸ Cell Active** in the menu bar.

## In/Out Names

---

Each input expression is assigned a number by Mathematica when the expression in an input cell is submitted to the Mathematica kernel for evaluation. This number,  $k$ , is assigned sequentially according to the sequence of computations performed during a session, and the value of the input can be retrieved with the object `In[ k ]`. For more details on the `In[ ]` object, follow the hyperlink to the Help Browser: [In](#). Therefore, one input in a cell can be associated with several input numbers if it is evaluated repeatedly at different times during a session. In a similar fashion the value returned in an output cell as the result of the  $k^{\text{th}}$  input to the Mathematica kernel is stored and can be retrieved at any time during the session with the object `Out[ k ]`. For more details on the `Out[ ]` object follow the hyperlink to the Help Browser: [Out](#).

For example, evaluate the following two input cells in sequence twice.

```
z = Expand[(x + y - xy)^3]
```

```
Plot3D[z, {x, -1, 3}, {y, -1, 3}, PlotPoints → 50,  
PlotRange → {-15, 20}, Mesh → False, BoxRatios → {1, 1, 1/2}];
```

Suppose the first evaluation of the `Expand[ ]` object was `In[49]`, then the plot was evaluated immediately afterwards as `In[50]`, and then the `Expand[ ]` object was evaluated again as `In[51]`.

Now we can retrieve the value of the expanded multinomial with Out[49] as well as with Out[51]. Most likely, the sequence numbers for these evaluations in your current Mathematica session are not 49 and 51, so replace them with the corresponding numbers from your current Mathematica session in the following four input cells.

**Out [49]**

**Out [51]**

When we ask for the value of the Out[50] object for the 3D-graphics, Mathematica returns the designator -SurfaceGraphics-.

**Out [50]**

However, we can recreate the actual graphics by submitting the Out[ ] object to the Show[ ] command.

**Show[Out [50]]**

The In/Out names are shown by default in Mathematica. This property can be turned off by clicking **Kernel** ► **Show In/Out Names** in the menu bar.

## *Hyperlinks to Documentation*

---

There is a simple way to embed hyperlinks to the various parts of the Mathematica documentation into a notebook. Suppose that we want to provide information about the form of the Solve[ ] command and examples for its usage. In the menu bar we select **Help** ► **Master Index...** In the Help Browser window that opens we scroll down and select the letter S and then the Solve name in the second scrolling column. We select the Solve hyperlink (drag the insert cursor over it; do not click on it!) under the heading Built-in Functions in the Help browser window, copy it, and paste it to this notebook.

Here is the copy of the hyperlink: Solve.

If we now click on this hyperlink in this notebook, then the browser window for Solve[ ] opens up with documentation, examples, and further hyperlinks to packages and the Mathematica book. In addition to this link, we can provide specific hyperlinks about solving equations to the Mathematica book from the links in the master index.

For documentation and examples on solving a single equation, go to 1.5.7. For documentation and examples on solving a system of equations, go to 3.4.4.

These two hyperlinks were copied in exactly the same manner from the master index window.

## *Exercises*

---

**EXERCISE 1:** Using cell dingbats for text layout.

Cell dingbats are symbols at the left margin of cells that are used to indicate the level of importance of the text. Most often, different types of cell dingbats are placed at different levels of indentation. An example for the use of cell dingbats is Mathematica's description of how to use



the Help Browser; choose **Help ▸ Help Browser...**, select the Getting Started/Demos button and click on Using the Help Browser. You can see the available Mathematica cell dingbats in the pop-up menu when you choose **Format ▸ Cell Dingbat**.

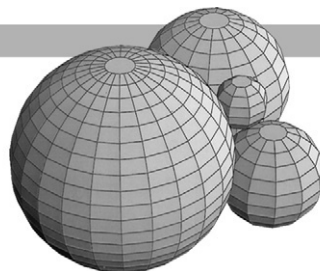
- (a) Open an untitled notebook and set it up with a ruler and tool bar using the Format menu.
- (b) Copy a portion of the content of the notebook Preface.nb, "Preface," and design a layout with cell dingbats and indentations for the text that you copied.
- (c) The section, subsection, and subsubsection types of cells (see the menu **Format ▸ Style**) have cell dingbats and the necessary indentation to accommodate the size of the cell dingbat. Use these three cell types and choose your own indentations for an alternative layout of the text from your solution of part (a).

## EXERCISE 2: Creating an index for a notebook.

For this exercise, make a copy of the notebook BasIntro.nb, "Mathematica Basics: An Introduction," and implement the modifications required for the solutions of this exercise in that copy.

- (a) Enter a tag name for each title section in the notebook. You may use the title for the section itself, an abbreviation for it, or any phrase you think could be a possible index entry for that section.
- (b) Set the Show Cell Tags property in the Find menu so that you can see all tags directly in the notebook. Select **Find ▸ Make Index ...** to create an index. Make sure that the Text Format radio button is clicked before you create the index. Paste the index you created into a new notebook.
- (c) Remove some of the tags and add several new tags for individual cells within sections of the notebook BasIntro.nb, "Mathematica Basics: An Introduction." Create a new index for the notebook based on this new set of tags.
- (d) Apply the text layout features of Mathematica to your notebook from part (b) to make the notebook self-explanatory. The notebook should have a title such as "Index for BasIntro.nb" as well as your name as the author and the date when the notebook was finished.
- (e) Repeat part (b) of the exercise, but save the index into a new Microsoft Word document rather than a Mathematica notebook. Apply the text layout features of Microsoft Word to your document to create a complete index page with a layout as in part (d).

# *Mathematica Basics: Text and Typesetting*



## *Introduction*

---

This is a short introduction to the usage of the palettes that are part of Mathematica, version 4. There are seven palettes that we can access through the menu bar **File ▸ Palettes**. The palettes are organized alphabetically in this submenu. We will not discuss in detail the last two palettes in the menu list. The **InternationalCharacters** palette contains accented and special characters such as the German letter ö and currency symbols such as the Euro, €, the British pound, £, or the Japanese Yen, ¥. The **NotebookLauncher** palette is a list of style names for different notebooks. Clicking on a button in the palette opens a notebook with that style sheet.

For convenient use of the palettes we arrange and size the current notebook in such a fashion that the palette and the notebook are tiled on the screen. That way, all elements of the palette are visible and accessible for clicking. We do not have to click on the appropriate window to bring it to the front. If we need an object from the palette at the current insertion point in the notebook, we just click on the object in the palette and it appears in the notebook at the insertion point.

## *The BasicTypesetting and CompleteCharacters Palettes*

---

We begin the discussion of how to use the various palettes with those palettes that deal with special symbols and typesetting. Two of the palettes in the palettes submenu provide the necessary tools. We open both by selecting in the menu bar **File ▸ Palettes ▸ BasicTypesetting** and **File ▸ Palettes ▸ CompleteCharacters**, and we rearrange the two palettes and this window so that the three windows tile the screen. The **BasicTypesetting** palette contains seven individual arrays of buttons of mathematical symbols. The **CompleteCharacters** palette contains three classes of palettes: letters, letter-like forms, and operators. Some of the objects in the buttons in the last two classes also occur in the **BasicTypesetting** palette; however, there are more symbols in the **CompleteCharacters** palette. Note that the arrangements of symbols into button arrays are different in the two palettes. In this section we will use only the letters from the **CompleteCharacters** palette and the letter-like forms, arrows, operators, and relational symbols from the **BasicTypesetting** palette.

The alphabets in the **CompleteCharacters** palette are Greek, Script, Gothic, and Double-Struck, as well as the Extended Latin special characters. All objects in the palettes are special characters that we will use only in text cells. Only graphics expressions that we build here will be evaluated, and we use the symbols for typesetting purposes only.

Set theoretical notation is pervasive in mathematics. We build the set-theoretic version of De Morgan's law using capital script letters for the sets and the round intersection and union symbols for the set-theoretic operations:

$$\mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) \equiv (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C})$$

All symbols, except the round parentheses on the keyboard, were created by clicking on the appropriate buttons in the two palettes.

Next we show a function,  $f$  (Script f), from the set of real numbers,  $\mathbb{R}$  (Double-Struck R), into the set of integers,  $\mathbb{Z}$ ;  $\mathbb{R} \xrightarrow{f} \mathbb{Z}$  where the function  $f$  is defined as  $f(z) = \lceil z \sin(z) \rceil$ . A graph of this function reveals the step-function quality of the ceiling function  $\lceil \dots \rceil$ , where  $\lceil x \rceil$  is the smallest integer greater than or equal to  $x$ .

We draw the graph first with grid lines and then inside a frame without axes. The ticks on the horizontal lines are set to trigonometric values and on the vertical lines to integer values. Note that we also change the `PlotPoints` option from its default value of 25. This option can be critical for the faithful graphical rendering of a nondifferentiable function. You can see the changes in rendering when you resize the plot and when you change the number of plot points. In particular, it appears as if the function has fractional values for some sizes of the graph and some settings of the `PlotPoints` option.

```
Plot[ $\lceil x \sin[x] \rceil$ , {x, -2 $\pi$ , 2 $\pi$ }, PlotStyle → {Thickness [0.005],
  RGBColor[1, 0, 0]},
  PlotPoints → 333, GridLines → {Range[-2 $\pi$ , 2 $\pi$ ,  $\pi/2$ ],
  Range[-4, 2, 1]}, Ticks → {Range[-2 $\pi$ , 2 $\pi$ ,  $\pi/2$ ], Range[-4, 2, 1]}];
```

Note that we use the standard mathematical symbol for the ceiling function,  $\lceil \dots \rceil$ , as well as the letter  $\pi$  in the `Plot[ ]` command. These forms can be used in input cells as well as in text cells.

```
Plot[ $\lceil x \sin[x] \rceil$ , {x, -2 $\pi$ , 2 $\pi$ },
  PlotStyle → {Thickness[0.01], RGBColor[0, 1, 0]}, PlotPoints → 333,
  Frame → True, Axes → False, FrameTicks → {Range[-2 $\pi$ , 2 $\pi$ ,  $\pi/2$ ],
  Range[-4, 2, 1]}];
```

Next we use some of the arrows to build up a relation diagram. All the symbols are at tab positions on three different lines in the next text cell. The arrows are from the sixth array of buttons on the **BasicTypesetting** palette and the three capital letters are from the Script alphabet in the **CompleteCharacters** palette. We constructed the subscripted letters using the template box  $\blacksquare_{\blacksquare}$  from the second array of buttons in the **BasicTypesetting** palette.

$$\begin{array}{ccc} & & \mathcal{H}_1 \\ & \nearrow & \downarrow \\ \mathcal{L} & \rightarrow & \mathcal{K}_2 \end{array}$$

We can also typeset a function whose value is defined for two cases. As an example we use the delta function, which is 0 everywhere, except at 0 where it equals 1. The middle button in the third array of the **BasicTypesetting** palette provides the pattern for the two cases,  $\left\{ \begin{array}{l} \square \\ \square \end{array} \right.$ . Into each box of the pattern we type the function definition and condition.

$$\Delta(x) = \left\{ \begin{array}{ll} 1, & \text{if } x = 0 \\ 0, & \text{if } x \neq 0 \end{array} \right.$$

We can also typeset functions and expressions whose value is defined by more than two cases. We start with the two cases as before, using the **BasicTypesetting** palette. However, we first make the spanning curly brace expandable to accommodate any number of cases. We achieve this by highlighting the brace in the expression and then selecting **Edit ▸ Expression Input ▸ Spanning Characters ▸ Expand Indefinitely** in the menu bar. We can add a column by pressing the **CTRL** key together with the **COMMA** key and a row by pressing **CTRL+RET**, that is, the **CONTROL** and **RETURN** keys together. We have applied both changes to the patterns in the next input cell by placing the insertion key inside the pattern we want to modify and apply the appropriate keystrokes. All patterns originally were copies of the leftmost pattern of two cases from the palette.

$$\left\{ \begin{array}{l} \square \\ \square \end{array} \right. \quad \left\{ \begin{array}{ll} \square & \square \\ \square & \square \end{array} \right. \quad \left\{ \begin{array}{l} \square \\ \square \\ \square \\ \square \end{array} \right. \quad \left\{ \begin{array}{lllll} \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \end{array} \right.$$

As an example, we write a tax rate function  $\text{Tax}(s)$  based on the year 2000 tax tables, Schedule X, for taxpayers filing with single status. We set up the definition as a pattern of five rows and two columns. The first column contains the dollar amount of tax owed and the second column contains the condition on the salary  $s$ . With this layout the amounts and the conditions automatically are displayed in centered and columnar form.

$$\text{Tax}(s) = \left\{ \begin{array}{ll} 0.15s & \text{if } s \leq \$26,250 \\ 3,937.50 + 0.28(s - 26,250) & \text{if } \$26,250 < s \leq \$63,550 \\ 14,381.50 + 0.31(s - 63,550) & \text{if } \$63,550 < s \leq \$132,600 \\ 35,787.00 + 0.36(s - 132,600) & \text{if } \$132,600 < s \leq \$288,350 \\ 91,857.00 + 0.396(s - 288,350) & \text{if } \$288,350 \leq s \end{array} \right.$$

## *Algebraic Manipulation Palette*

We select **File ▸ Palettes ▸ AlgebraicManipulation** in the menu bar and arrange the notebook and the palette on the screen to our liking. Then we take the expressions in the next input cell, select the expression or the entire cell at the cell bracket, and apply some of the functions in that palette, such as **Expand[■]**, **Factor[■]**, **Apart[■]** and **Together[■]** to see their effect on the expression.

$$(x + 2y)^4$$

$$\frac{(x + 2y)^4}{(x - 2y)^4}$$

Note that the computation is performed “in place;” that is, the original expression is replaced by the result of the computation. Observe also that we can select a portion of an expression, say the numerator in the second input cell, and apply a function to just that part of the expression. A simple way of selecting a portion of a typeset expression is to place the cursor anywhere inside the part of the expression that we want to select, and then quickly and repeatedly click the mouse until we have selected what we want.

Try to manipulate a trigonometric expression with the `TrigExpand[■]` and `TrigReduce[■]` functions in the palette.

`Sin[3 x]`

## Basic Input Palette

---

We get the **BasicInput** palette by selecting it in the menu bar as **File ▶ Palettes ▶ BasicInput** and, as before, we arrange the notebook and the palette on the screen to our liking. We chose this palette next since it has a few commonly used special symbols and Greek letters as well as some two-dimensional input patterns. In each of the following text cells in this section we explain the construction of a typeset form.

In order to create an inequality between alpha and beta, we just click the three buttons in the palette:  $\alpha \leq \beta$ . There we have it.

The Cartesian product of gamma and delta is:  $\Gamma \times \Delta$ . Here we press the spacebar between clicks on the palette.

In order to create a two-dimensional mathematical formula, say the sum of the reciprocals  $1/i$ , for  $i$  from 1 to  $n$ , we first select the sum object from the palette:  $\sum_{\blacksquare}^{\blacksquare} \blacksquare$ . That object has one placeholder, a little square, selected. We can type an  $i$  into that space holder:  $\sum_{i=\blacksquare}^{\blacksquare} \blacksquare$ .

To move from spaceholder to spaceholder cyclically inside the two-dimensional object we use tabs; in order to exit from the object we can use the right arrow key. The completed expression is  $\sum_{i=1}^n \frac{1}{i}$ . The actual sequence of keystrokes from the first highlighted placeholder was  $i$  tab 1 tab  $n$  tab  $\frac{1}{\blacksquare}$  (selection from the palette) 1 tab  $i$ . That fills all the slots. Now we enter the right-arrow key twice to get out of the two-dimensional object and back to the normal text line.

Try this construction process in a text cell as we did earlier, and in an input cell with numeric, not symbolic bounds and evaluate the input cell.

## Basic Calculations Palette

---

We get the palette **BasicCalculations** by selecting **File ▶ Palettes ▶ BasicCalculations** in the menu bar. This palette is a mixture of explanatory text and actual arrays of buttons that are arranged in a hierarchical cell structure. We arrange the palette on the screen to our liking and demonstrate the use of several of the buttons.

We start with the Arithmetic and Numbers section: Select the general root  $\sqrt[n]{\square}$ , then the product  $\square \times \square$  replacing the radicand, then for each factor the power  $\square^\square$ , and finally, fill the placeholders with positive integers. Recall that the TAB is used to move from one placeholder to the next.

$$\sqrt[5]{9^5 \times 10^7}$$

The calculation for the resulting input expression is not done in place; rather the result is computed in an output cell. We apply the numeric converter function `N[ ]` to this result to get an approximate value to 30 decimal digits.

`N[%, 30]`

As the second example we demonstrate integral and derivative buttons. They occur in the Common Operations in the Calculus section of the palette. We take a trigonometric function whose integral is computed using integration by parts.

$$\int x \sin[x] dx$$

$\partial_x \%$

Now we compute the definite integral of the same function over one period of the sine function. Observe that we also used the **BasicTypesetting** palette (in the menu bar **File** ► **Palettes** ► **BasicTypesetting**) in order to create the symbol  $\pi$  in the upper bound of the integral.

$$\int_0^{2\pi} x \sin[x] dx$$

We plot the function.

`Plot[xSin[x], {x, 0, 2\pi}];`

As the last example for the use of the objects in this palette we solve a differential equation. The relevant buttons are in Differential Equations in the Calculus section of the palette. Before you enter a function, highlight `DSolve` and select **Help** ► **Find Selected Function...** in the menu bar for help with the interpretation and usage of the arguments of `DSolve[ ]`.

`Dsolve[ y'[x] - xy[x] == x^3, y[x], x]`

## Exercises

---

### EXERCISE 1: Typesetting algebraic expressions.

Use the **BasicTypesetting** palette to write the following algebraic expressions in the traditional mathematical form. Use subscripts and superscripts where they are suggested from the context in the one-dimensional expression.

(a)  $(5x^2 - 1)/(x^2 + 7)$ .

(b)  $((x - 3)(5 + x))/((1 - x)^2(x + 3)^2)$ .

(c) The square root of  $(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2$ .

Since we have the algebraic expressions in one-dimensional form you can also do this exercise by selecting the expression and choosing **Cell ▶ Convert To** and checking **TraditionalForm**. Alternatively, you might decide to put an expression into an input cell and evaluate it. The output cell will then be in traditional form, but in a layout that Mathematica, not you, determines and with any algebraic simplifications that are possible.

(d) Write a formula for the finite sum of the squares of the whole numbers from  $m$  to  $n$  with  $i$  as an index. Use the capital Greek letter sigma for the sum and the appropriate buttons in the subscript/superscript button array for the bounds of the sum above and below the sigma. Alternatively, you can use the Common Operations in the Calculus section of the BasicCalculations palette. This method uses more sophisticated buttons.

### EXERCISE 2: Typesetting integrals.

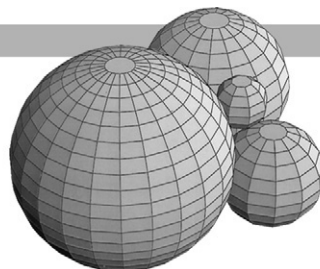
Use the Calculus section in the BasicCalculations palette for the typesetting in this exercise.

- (a) The indefinite integral of the function  $(3x^2 - 2)/(x^4 + 5)$ .
- (b) The definite integral of the function  $(3x^2 - 2)/(x^4 + 5)$  from 0 to 10.

### EXERCISE 3: Typesetting function definitions.

- (a) At the end of the section “BasicTypesetting and CompleteCharacters Palettes” we defined the function  $\text{Tax}(s)$ . Modify the definition of  $\text{Tax}(s)$  by using your individual tax filing status and the tax rate that is in effect for the current tax year. Use two columns for the cases in the definition of the function as we did before.
- (b) Repeat part (a), but use at least three columns for the cases in the definition of the function.
- (c) Take the latest rate information from your local electric company and typeset the function  $\text{ElectricCost}(t)$  that computes the cost of  $t$  kWh’s (kilowatthours).
- (d) Define the following variant of the sawtooth function:  $\text{Sawtooth}(x) = x - \text{Floor}(x)$ , for  $x > 0$ ,  $\text{Sawtooth}(0) = 0$ , and  $\text{Sawtooth}(x) = \text{Ceiling}(x) - x$ , for  $x < 0$ . Use the symbols for the Floor and the Ceiling functions from the BasicTypesetting palette. Note also that two cases would suffice in the definition since at  $x = 0$  all three expressions for  $\text{Sawtooth}(x)$  have value 0.

# *Mathematica Basics: Packages*



## *Introduction*

Packages are Mathematica files containing functions and objects that are not automatically loaded when Mathematica is started. You can find the packages in the Add-ons directory (folder) of the Mathematica directory (folder), and you can also use the Help Browser to get information about packages. In the menu bar select **Help ▸ Add-ons...** and click the Add-ons button and then Standard Packages. This will display subject areas, each of which contains a number of packages. If you click on Calculus, you will find the packages DSolveIntegrals, FourierTransform, Integration, and others.

Locate the Calendar package in the Miscellaneous directory of Standard Packages in the Help Browser. When you click on this package you will open a notebook that begins with a short introduction to the package. The notebook contains a list of functions including DayOfWeek, DaysBetween, EasterSunday, and JewishNewYear. To use the functions in a package, you have to load the package.

## *Loading Standard Packages*

There are several mechanisms for loading packages. You can go to the Help Browser, locate the package you want, click on it, and then enter the first input line that is shown on the screen. In the example of the Calendar package this is the line `<<Miscellaneous`Calendar``. Entering it will load the Calendar package and evaluate all functions and objects defined in the package.

The same result will be achieved if you enter `<<Miscellaneous`Calendar`` in an input cell in the notebook you are working in. The back-quote symbol (`"`"`) on the left-most key in the top row of the keyboard is called the context mark. In this example, it identifies the context Calendar within the context Miscellaneous. Each package has its own context.

Entering `Needs["Miscellaneous`Calendar"]` will also load the Calendar package and evaluate all functions and objects defined in the package. There will be no output in the notebook in which the `Needs[]` command is evaluated.

```
Needs["Miscellaneous`Calendar"]
```



Once we have loaded a package we need to find out the names of the objects that are defined in the package and that we can now use. The following expression returns the list of all the names in the package we just loaded.

```
Names["Miscellaneous`Calendar`*"]
```

We should now be able to use the functions in the Calendar package. Let's try!

```
?EasterSunday
```

```
EasterSunday[2003]
```

```
EasterSunday[2009]
```

```
?JewishNewYear
```

```
JewishNewYear[2003]
```

```
?DayOfWeek
```

```
DayOfWeek[{2003,4,20}]
```

```
DayOfWeek[{2003,5,4}]
```

Next, we want to load the Graphics package from the Graphics directory. Instead of the function Needs[ ] we use the symbol << (two less-than symbols). Again, there will be no output.

```
<<Graphics`Graphics`
```

We can use the ? facility to get a listing of all the names made available in a package.

```
?Graphics`Graphics`*
```

The Graphics package contains functions like PieChart[ ], BarChart[ ], and LogLogPlot[ ], and we can get their usage statements by following the links to the Help Browser or by using the ? operator.

```
?PieChart
```

```
?PieLabels
```

Following is a pie chart representing your school's expenses where 72% of the budget goes to salaries, 12% is spent on instructional support, 10% goes to student services, and 6% supports the physical plant.

```
PieChart[{72,12,10,6},  
  PieLabels → {"Salaries","Instruction","Services","Plant  
  Support"}];
```

We load one more package and use the ? facility to get a listing of all the names in the package.

```
<<DiscreteMath`Permutations`
```

```
?DiscreteMath`Permutations`*
```

An introduction to the Standard Add-on Packages is available in the Help Browser by clicking the **Add-ons** button and then selecting in the scroll windows from left to right **Standard Packages** ► **Introduction** ► **The Standard Add-on Packages** or, for a particular group of packages (for example, the Calculus packages) by clicking the **Add-ons** button and then selecting **Standard Packages** ► **Introduction** ► **Calculus** packages.

If you frequently use many functions from different packages in the same standard package directory, you will find it convenient to load all packages from that directory simultaneously. You can then use any of the functions contained in packages included in the directory. You will not need to load each package separately. For example, entering `<<Graphics`` makes all the functions provided in the 22 different Graphics packages available. Using one of these functions will cause the appropriate package to be loaded if it has not been loaded already.

Information about packages and the content of packages is also available in printed form. For Mathematica 4.1, the book *Mathematica's Standard Add-on Packages* is shipped with the software. The book's index has a complete list of the functions that are contained in the packages. These functions are also contained in the Master Index of the Help Browser.

## *Forgetting to Load a Package*

---

Suppose you use a function whose name and usage are familiar to you, but you forgot that it is defined in a package and is not a built-in function. So you evaluate the function before you load the appropriate package. For example, you may want to use `ImplicitPlot[ ]` from the `ImplicitPlot` package in the `Graphics` folder. If you have not loaded the package, then Mathematica will not recognize the function `ImplicitPlot[ ]` and will leave the expression unevaluated.

```
ImplicitPlot[x^2 + y^2 == 1, {x, -1, 1}]
```

Now you remember that you should have loaded the package first and you attempt to load it.

```
Needs["Graphics`ImplicitPlot`"]
```

Mathematica issues a warning message that the name `ImplicitPlot` appears in several contexts, the context of your original evaluation (global) and the context of the loaded package. Now try to evaluate `ImplicitPlot[ ]` again.

```
ImplicitPlot[x^2 + y^2 == 1, {x, -1, 1}]
```

Your first use of the name `ImplicitPlot` shields the name `ImplicitPlot` from the package, and again, the expression is left unevaluated. The solution is to remove the name `ImplicitPlot` from the current session in Mathematica, then load the package again, and finally, the intended `ImplicitPlot[ ]` from the package will be ready for use.

```
Remove[ImplicitPlot]
```

```
Needs["Graphics`ImplicitPlot`"]
```

```
ImplicitPlot[x^2 + y^2 == 1, {x, -1, 1}];
```

Here is a good place to explain the difference between `Clear[ ]`, which we introduced in the notebook `BasIntro.nb`, and `Remove[ ]`. In a nutshell, `Clear[ ]` eliminates the value of a symbol

whereas `Remove[ ]` eliminates the symbol and its value from the current Mathematica session. The computations in the remainder of this section may be clearer if you terminate your current session and restart Mathematica.

We demonstrate the difference between `Clear[ ]` and `Remove[ ]` with the symbol `Red`. We first demonstrate that there are no global symbols defined since you just started Mathematica and then we give `Red` a numeric value.

```
Names["Global`*"]
```

```
Red = 5.5
```

```
Red
```

```
Names["Global`*"]
```

Now we introduce the context from the `Graphics` package in which the names of colors are defined by their `RGBColor[ ]` values.

```
<<Graphics`Colors`
```

The `Global` context shadows the `Graphics` color `Red`; however, we can access it by using its complete name that specifies the context names.

```
Purple
```

```
Red
```

```
Graphics`Colors`Red
```

When we use `Clear[ ]` for the symbol `Red`, we eliminate the value of the symbol in the `Global` context only. This action still keeps the global symbol and it shadows the color.

```
Clear[Red]
```

```
Red
```

```
Names["Global`*"]
```

Now we eliminate the symbol `Red` from the `Global` context by using `Remove[ ]`. This action makes the `Graphics` color available. The `Global` context again is empty.

```
Remove[Red]
```

```
Red
```

```
Names["Global`*"]
```

Since there is only one object named `Red` in this session, the short and the long name produce the same result.

```
Red
```

```
Graphics`Colors`Red
```

## *Exercises*

---

**EXERCISE 1:** Using functions from packages.

- (a) Use two more functions from the Miscellaneous package.
- (b) Use two more functions from the Graphics package.

**EXERCISE 2:** Loading and using functions from a package.

- (a) Load the VectorAnalysis package from the Calculus packages directory. Pick some points and compute their coordinates in at least three different coordinate systems. Compute also the dot product of two vectors that are given in a non-cartesian coordinate system.
- (b) Load the MatrixManipulation package from the LinearAlgebra packages directory. Create some sample matrices and extract rows and columns from each matrix. Also extract submatrices of different sizes from your sample matrices.
- (c) Load the SurfaceOfRevolution package from the Graphics packages directory. Pick some functions and plot the surface of revolution obtained from them. Include full as well as partial revolutions.
- (d) Load the Permutations package from the DiscreteMath packages directory. Create random permutations of various sizes and compute their cycle decompositions.
- (e) Load the DescriptiveStatistics package from the Statistics packages directory. Create a list of data values and compute the median, various mean values, variance, quartiles, and different quantiles for your data set.

This Page Intentionally Left Blank

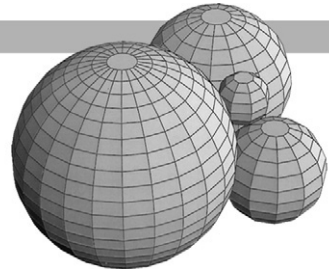
# II

---

## *Designing Functions*

This Page Intentionally Left Blank

# *Values, Variables, and Assignments*



## *Introduction*

There are many different classes of objects in Mathematica. Each object is described by an expression whose value is computed by the Mathematica kernel. We can group the objects according to their use and structure as follows.

### FUNDAMENTAL OBJECTS:

Exact numbers (integers, rational numbers, Gaussian integers)

Approximate numbers (floating point numbers)

Symbolic numbers (special values such as E, Pi, and Infinity)

Text strings (sequences of characters enclosed in the double-quote)

Symbols (names for objects)

Graphics (objects that can be rendered as a picture)

Patterns (objects that represent the structure for a class of objects)

Lists (grouped sequences of objects, which may be lists themselves)

### BUILT-IN FUNCTIONS:

Numerical computations (support evaluations of expressions involving numbers)

Algebraic computations (support evaluations of arbitrary expressions and functions)

Mathematical functions (provide standard and special functions of mathematics)

Lists and matrices (provide general purpose operations for arbitrarily nested lists)

Graphics and sound (support rendering of graphics and sound with many options)

Programming (supports a complete interpreted programming language)

Input and output (provides text formatting and file operations)

System interface (provides communications with other applications programs)



## USER-DEFINED FUNCTIONS:

Functions defined by a user with any of the tools just mentioned, including the programming language within Mathematica.

Throughout the notebooks we will use the following naming convention for user-defined functions: The first letter of the name always is a lowercase `u`, for user-defined. Since all built-in functions start with a capital letter, this convention provides a clear visual distinction between user-defined and built-in functions. Some examples are `uDamping[ ]`, `uHeight[ ]`, `uRandomMatrix[ ]`, `uGetCoordinates[ ]`, or `uBisection[ ]`.

When we perform computations we use explicit values, called constants, such as 15,  $-3.2 + 4.1i$ , and  $\pi$  as well as symbols such as `x`, `alpha`, `PlotRange`, and `Bisection`. The symbols are used in two different ways. Symbols can be names that hold values, for example, of intermediate numeric results or graphs. Symbols also can be names that hold the definitions of functions to be used later, for example, a summation process or the description of a graphics plot. These two distinct uses are supported by two kinds of assignment operations in Mathematica: the `=` symbol for the immediate assignment and the `:=` symbol for the delayed assignment. Usually, the direct assignment is used for values and the delayed assignment is used for function definitions.

The symbols `=` and `:=` are used as assignment operations. They are fundamentally different from the equality symbols `==` and `===`, which are used by Mathematica to test for the equality of values and names, respectively. Use `?==` and then the Help Browser to investigate the usage of `==`.

## *The Immediate Assignment Operator = for Values*

---

In the immediate assignment operation the computation of the value and the assignment action are performed at the time we evaluate the assignment. From that point on the value is associated with the symbol until we make another immediate assignment to the same symbol. Before we make another assignment, we use `Clear[ ]` to erase the previous meaning.

**Example 1:** Assignment of numerical values.

We assign a value to the symbol `x` and evaluate the symbol.

```
x = 15
```

```
x
```

```
?x
```

The next assignment resets the value of the symbol `x`. Recall that the semicolon suppresses the display of the result of the assignment.

```
x = Pi;
```

```
x
```

The function `Clear[ ]` erases all values that are associated with its argument symbols.

```
?Clear
```

```
Clear[x]
```

```
x
```

We can assign values to several symbols in a single input line when we separate them with semicolons.

```
Clear[a,b,c,x]  
a = 1;b = N[Pi];c = x^3/5;
```

```
a
```

```
b
```

```
c
```

**Example 2:** Assignment of structured values.

```
Clear[x,y,a,b,c,d,p]
```

```
x = {1,2,5,-1}
```

```
y = {3,0,4,10}
```

We can use the equality symbol `==` to check if the value of `x` equals the value of `y`.

```
x == y
```

We apply different numerical operations to the two lists of numbers.

```
a = x*y
```

```
b = x.y
```

The equality test fails if we compare different objects, in this case a list to a number.

```
a == b
```

Two more immediate assignments:

```
c = List[x,y]
```

```
d = {y,x}
```

The value of a symbol can be the outcome of a test operation such as an equality comparison. Here the value of `p` is `True`.

```
p = (c == Reverse[d])
```

**Example 3:** Suppression of the printing of large tables.

When we terminate an immediate assignment with the semicolon (;) then the value is computed, but it is not printed out. This is useful when the value is a long list.

```
Clear[y]

y = Table[ {k,k^2,k^-2}, {k,1,1000} ];
```

Since the values of the table have been computed we can select specific values of y with double brackets.

```
y[[1]]

y[[256]]

y[[1000]]

y[[256,2]]

y[[1000,3]]
```

**Example 4:** Assignment of symbolic and graphical values.

```
Clear[a1,a2,a3,b1,b2,b3,x,y,u,v]

x = {a1,a2,a3}

y = {b1,b2,b3}

u = Transpose[{x, y}]

v = TableForm[u]
```

The value of v is a special type of printed output. The value of u is a  $2 \times 3$  matrix. You can perform matrix computations with u, but not with v, since v is not a matrix but an output in table form.

```
Clear[s,t,p1,p2]

p1 = Plot[s^2-1,{s,-3,2}];
```

When we terminate an immediate assignment for a graphical rendering operation such as the Plot[ ] function with the semicolon, then the graph is computed and the picture is rendered, but the output cell Out[ ] is suppressed.

```
p2 = Plot[-t^3 + 4t^2 - 3,{t,-2,3}];
```

We saved the two plots as  $p_1$  and  $p_2$  and can now show them in a single picture. The different x-ranges for the two graphs are adjusted automatically for the combined graph. However, only those portions of the graphs that actually were computed in the earlier steps are rendered in the combined picture.

```
Show[p1, p2];
```

**Example 5:** Simultaneous assignment of values.

An assignment can be made to any symbol or list of symbols as long as the structures on the two sides of the assignment operator are compatible. As a consequence, we can make simultaneous assignments to the parts of a list if the list on the left-hand side of the assignment operator contains only symbols.

```
Clear[x,y,z]
```

```
{x,y,z} = {1.5,-7,3/4}
```

```
x
```

The object on the left-hand side of an assignment may even be a nested list of lists. Furthermore, the values of the objects in the list can be of any type, for example, numbers, strings, or lists.

```
Clear[a,b,c,d,t]
```

```
t = Table[k^5,{k,-3,3}]
```

```
{a,{b,c},d} = {-10.55,{"one",t},Pi}
```

```
c
```

```
c[[2]]
```

```
d
```

## *The Delayed Assignment Operator := for Functions*

---

Recall that an immediate assignment of the form  $v = \text{expr}$  evaluates the expression  $\text{expr}$  and assigns the computed value to the symbol  $v$ . This action occurs when the assignment is first performed; thereafter each evaluation of  $v$  reproduces this value until a different assignment is made to the symbol  $v$ .

In contrast, a delayed assignment of the form  $v := \text{expr}$  takes the expression  $\text{expr}$  as written, leaves it unevaluated, and stores its literal form in the symbol  $v$ . When  $v$  is evaluated after this assignment, the  $\text{expr}$  is invoked, evaluated, and its value returned as the value of  $v$ . If  $v$  is evaluated again then  $\text{expr}$  is invoked again, evaluated again (this time in the computing environment of the second invocation of  $v$ ), and the value of this computation (which may differ from the value computed the first time) returned as the value of  $v$ .

**Example 6:** Immediate and delayed assignment to a symbol.

We demonstrate the differences between the two assignment operations with the `Random[ ]` function. We take the same expression and use it once in an immediate assignment and once in a delayed assignment.

```
Clear[s,t]

s = Random[Integer, {-10,10}]

t := Random[Integer, {-10,10}]
```

Observe that there is an output cell for the symbol `s` because an evaluation takes place, but there is none for the symbol `t`. Evaluate `s` and `t` each several times.

```
s

t
```

The symbol `s` always returns the same value, the one originally computed for it. An evaluation of the symbol `t` invokes the function `Random[ ]`, and the value returned by `Random[ ]` becomes the value returned as the value of `t`. This is some integer in the range from  $-10$  to  $+10$ , and usually is different for each evaluation of `t`.

In the delayed assignment operation the expression on the right-hand side of the assignment is stored unevaluated as the literal value of the symbol on the left-hand side. The right-hand side represents a computational process or a rule on how to compute a value for a symbol.

**Example 7:** Definition of a polynomial of one variable using delayed assignment.

For a function definition the right-hand side represents a computational process or a rule on how to compute a value for the function. Usually, the symbols on the right-hand side are considered to be the variables of the function and are bound to the function. Mathematica uses pattern variables such as `x_` or `t_` or `expr_` to implement this binding mechanism.

Here is the formal definition of the polynomial `uPoly[ ]` with the bound variable `x`.

```
Clear[uPoly]

uPoly[x_] := x^3 - 4x + 1
```

Observe that there is no output cell when the definition is evaluated because no evaluation of the expression on the right-hand side takes place. The only action is to associate the computational rule with the symbol `uPoly`. In the next cell we ask Mathematica what it stores for the symbol `uPoly`.

```
?uPoly
```

We evaluate the function at explicit numeric arguments.

```
uPoly[0]

uPoly[5]

uPoly[-5]
```

```
uPoly[1.1]
```

```
uPoly[-3/4]
```

We can also evaluate the function at a symbolic numeric argument or a complex number.

```
uPoly[E]
```

```
N[%, 30]
```

```
uPoly[2 + I]
```

Furthermore, we can evaluate the polynomial with different symbolic variables such as  $s$  or  $t$  and with symbolic expressions such  $3z$  or  $(a + b^2)$ .

```
Clear[a, b, s, z]
```

```
uPoly[s]
```

```
uPoly[3z]
```

```
uPoly[a + b^2]
```

Note that in all symbolic evaluations of `uPoly[ ]` the resulting polynomial is listed in Mathematica's default: alphabetic ordering of variables and increasing powers of the variables. Finally, we plot the function `uPoly[ ]` as a function of  $t$  and in a coordinate grid.

```
Plot[uPoly[t], {t, -2.5, 2.5}, GridLines → Automatic];
```

**Example 8:** Definition of a transcendental function using delayed assignment.

Here is the formal definition of the function `uDamping[ ]` with the bound variable  $x$ .

```
Clear[uDamping]
```

```
uDamping[x_] := E^(-x/5) Sin[x]
```

We evaluate the function at explicit numeric arguments.

```
uDamping[0]
```

```
uDamping[10]
```

```
N[%, 30]
```

```
uDamping[10.]
```

```
uDamping[-17/112]
```

We can also evaluate the function at symbolic numeric arguments.

```
uDamping[Pi/4]
```

```
N[% , 20]
```

Since Infinity is a symbolic numeric value, we can try to evaluate the function `uDamping[ ]` at Infinity. This amounts to the computation of a limit. Since limits need not exist or since only upper and lower limits might exist, Mathematica returns an expression in terms of the `Interval[ ]` function.

```
uDamping[Infinity]
```

```
?Interval
```

```
uDamping[-Infinity]
```

We can graph the function with `Plot[ ]` just as we would any built-in function. All options of `Plot[ ]` can be used. We can, for instance, draw the damped oscillation together with its exponential envelope in different colors and also with different thicknesses in a single plot. In the latter case, the `PlotStyle` specification is a list of lists of graphics primitives.

```
Plot[uDamping[x], {x, -2Pi, 6Pi}];
```

```
Plot[{-E^(-x/5), uDamping[x], E^(-x/5)}, {x, -2Pi, 6Pi},  
  PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 0, 0], RGBColor[0, 0, 1]},  
  PlotRange → All];
```

```
Plot[{-E^(-x/5), uDamping[x], E^(-x/5)}, {x, -2Pi, 6Pi}, Axes → None,  
  Frame → True, PlotRange → All, PlotStyle → {{RGBColor[1, 0, 0],  
  Thickness[0.008]}, {RGBColor[0, 0, 0], Thickness[0.002]},  
  {RGBColor[0, 0, 1], Thickness[0.008]}}];
```

**Example 9:** Creating patterns of points from the `Sin[ ]` function with a delayed assignment.

We define a function that we use only with integer arguments even though the definition does not restrict us to integer arguments. It is important that we use the numeric approximation function `N[ ]` because Mathematica leaves an exact expression such as `Sin[1]` in its symbolic form.

```
Clear[uPattern]
```

```
uPattern[n_] := N[Sin[n]]
```

Here are some evaluations of the function.

```
uPattern[8]
```

```
uPattern[251]
```

We render the function `uPattern[ ]` with the graphing function `ListPlot[ ]` over several integer ranges.

```
ListPlot[Table[uPattern[n], {n, 1, 250}]]];
```

In the next computations we specify both coordinates for the points. Then the points will be drawn with their correct x-coordinates when the range for a plot starts with a value other than 1.

```
ListPlot[Table[{n,uPattern[n]},{n,0,1000}]];
```

```
ListPlot[Table[{n,uPattern[n]},{n,-1500,1500}]];
```

You may find the rendered graph more effective when you specify `PointSize[ ]` with `PlotStyle[ ]` in a `ListPlot[ ]` command. This is particularly useful when you select the graphics and drag it to a larger size, but want the size of the points to stay small.

```
ListPlot[Table[{n,uPattern[n]},{n,-1500,1500}],
PlotStyle->PointSize[0.003]];
```

## *Evaluation Issues for the Immediate and Delayed Assignment Operators*

---

An immediate assignment of the form  $v = \text{expr}$  evaluates the expression `expr` and assigns the computed value to the symbol `v`. A delayed assignment of the form  $v := \text{expr}$  takes the expression `expr` as written, leaves it unevaluated, and stores its literal form in the symbol `v`. In this section we discuss several computational situations where the type of assignment makes a difference in the outcome of the computation. The difference in behavior can be very subtle because it depends on the computing environment. We demonstrated one such situation with the function `Random[ ]` in Example 6 of the previous section.

**Example 10:** A function variable with a previously assigned numeric value.

First we give the definition of a function with variable `x` using the immediate assignment. Before we evaluate the definition we give the symbol `x` an explicit numeric value.

```
Clear[f,x]
```

```
x = 12
```

```
f[x_] = Sin[x] + 1
```

```
?f
```

We evaluate the function `f` with explicit numbers and symbolic arguments.

```
f[1]
```

```
f[Pi]
```

```
f[x]
```

```
Clear[q]
```

```
f[q]
```



Now we give the definition of a function with variable  $x$  using the delayed assignment. We do not clear the value of  $x$  but keep its value at  $x = 12$ .

```
Clear[g]
g[x_] := Sin[x] + 1
```

```
?g
```

We evaluate the function  $g$  with explicit numbers and symbolic arguments. Note the difference to the evaluation of  $f$ .

```
g[1]
```

```
g[Pi]
```

```
x
```

```
g[x]
```

```
Clear[p]
```

```
g[p]
```

```
g[g[p]]
```

Finally, we look at the definitions of the functions  $f$  and  $g$  stored by Mathematica. We find that  $f$  has been evaluated, although  $g$  has not been evaluated but has been stored in its literal form.

```
?f
```

```
?g
```

As we mentioned earlier, symbols that occur on the right-hand side of an immediate assignment in a function definition are computed in the environment already established. If such symbols already have specific values, then the entire assignment may fail or produce unexpected results. Symbols that occur on the right-hand side of a delayed assignment in a function definition are bound to the pattern variable of the same name on the left-hand side of the assignment. Since no evaluation takes place, no conflict of interpretation can occur at the time of the definition of the function.

**Example 11:** The order of evaluation in the immediate and delayed assignment.

As an example, we use `Factor[ ]` in the definitions of the functions  $f$  and  $g$ . In the immediate assignment the factorization is done immediately and only once. The delayed assignment stores the rule to factor, and therefore factorization takes place each time the function is evaluated.

```
Clear[f,g,x]
```

```
f[x_] = Factor[x^2 - 2x + 1]
```

```
g[x_] := Factor[x^2 - 2x + 1]
```

We see the different return forms when we inspect what Mathematica stores for the two functions.

```
?f
```

```
?g
```

For both functions  $f$  and  $g$  the pattern variable  $x_$  is used to substitute for the symbol  $x$  on the right-hand side. The value returned by  $f$  is obtained by substituting  $x$  into the factored form that is shown by  $?f$ . The value returned by  $g$  is obtained by substituting  $x$  into the original, unfactored form of  $g$ , which is then factored, if possible.

```
Clear[a]
```

```
f[a^2 + 6a + 9]
```

```
g[a^2 + 6a + 9]
```

Evaluation on explicit numbers yields the same result for both functions.

```
f[2]
```

```
g[2]
```

**Example 12:** Computing and plotting the derivative of a function.

Mathematica has several ways to compute the derivatives of functions. Here is an example of the two most common uses for functions of a single variable.

```
Clear[f]
```

```
f[x_] := xSin[x]
```

```
f'[x]
```

```
f''[x]
```

```
?D
```

```
D[f[x],x]
```

```
D[f[x],{x, 2}]
```

These two types of derivative expressions compute, as expected, the same results. However, they behave differently when we want to plot them. We draw  $f$  and its first two derivatives  $f'$  and  $f''$ , which have been evaluated, in a single graph. Numeric values between  $-2\pi$  and  $2\pi$  are substituted for  $x$  in  $f[x]$ ,  $f'[x]$ , and  $f''[x]$ , the function values are computed, and then the function values are plotted.

```
Plot[{f[x], f'[x], f''[x]}, {x, -2Pi, 2Pi},
```

```
PlotStyle → {RGBColor[1,0,0], RGBColor[0,1,0], RGBColor[0,0,1]}];
```

The second plot fails with the two derivatives for the following reason. Mathematica substitutes numeric values between  $-2\pi$  and  $2\pi$  into  $f[x]$ ,  $D[f, x]$ , and  $D[f, \{x, 2\}]$  in order to compute their function values. This happens before the derivative operator  $D[ ]$  is applied to  $f$ , and

the symbolic variable  $x$  has now become a number. For instance, if  $x = 3$ , we have  $D[f, 3]$ , which does not make sense. This is reflected in the first error message: General::"ivar": -6.28318 is not a valid variable.

```
Plot[{f[x], D[f, x], D[f, {x, 2}]}, {x, -2Pi, 2Pi},
PlotStyle -> {RGBColor[1, 0, 0], RGBColor[0, 1, 0], RGBColor[0, 0, 1]}];
```

We can prevent the problem by ensuring that the derivatives have been computed before we attempt to evaluate and plot them. The immediate assignment is helpful in this situation.

```
Clear[d1f, d2f, x]
d1f[x_] = D[f[x], x]
d2f[x_] = D[f[x], {x, 2}]
```

```
Plot[{f[x], d1f[x], d2f[x]}, {x, -2Pi, 2Pi},
PlotStyle -> {RGBColor[1, 0, 0], RGBColor[0, 1, 0], RGBColor[0, 0, 1]}];
```

There is a second method of forcing the computation of the derivative of a function in a `Plot[]` command. The built-in function `Evaluate[]` takes its argument and evaluates it, overriding evaluation restrictions in the surrounding computing environment. For more information consult [Evaluate](#) in the Help Browser and [Section 1.9.1](#) in the Mathematica book as well as [Plot](#) in the Help Browser, and also [A.4.2](#) in the Mathematica book. We evaluate the list of the three functions:

```
Plot[Evaluate[{f[x], D[f[x], x], D[f[x], {x, 2}]}], {x, -2Pi, 2Pi},
PlotStyle -> {RGBColor[1, 0, 0], RGBColor[0, 1, 0], RGBColor[0, 0, 1]}];
```

As a general rule, you should follow this principle: Use the immediate assignment when you want to associate a final value with a symbol. Use the delayed assignment when you want to associate a computation process with a symbol.

## *Computational Efficiency and the Assignment Operators*

---

When we study functions theoretically, we frequently do not care about the precise computational form of their definition. However, when we want to evaluate a function repeatedly, for example, in a table or for a plot, then computational aspects of the expression that defines the function become important.

The following examples demonstrate computational tasks where an immediate assignment either leads to greater efficiency than a delayed assignment or is necessary for applications in other functions.

You should be concerned about the efficiency of your computations. However, you should also weigh the time it takes to design a function against the time you spend actually computing with it. Computational efficiency is of little importance when the function is rarely used.

**Example 13:** Preventing many recomputations of an expression.

We want to make a table of values of the definite integral of a function where we fix the lower bound and vary the upper bound of integration. We define the integration with an immediate assignment and also with a delayed assignment.

Note that we must ensure that  $x$  is a symbol without value whenever we compute either of the two functions. Therefore, we clear the symbol  $x$  and make it a variable in each function. Since we use  $b$  as a symbol in the immediate assignment, we must clear it also.

```
Clear[uImmediate,uDelayed,x,b]
uImmediate[x_, b_] = Integrate[E^x Sin[x],{x,0,b}]
uDelayed[x_, b_] := Integrate[E^x Sin[x],{x,0,b}]
```

When we compute the integral for a specific value of  $b$ , then the results of the two functions may be identical, but they have been computed in different ways.

```
uImmediate[x, Pi]
```

Here,  $\pi$  has simply been substituted for  $b$  in the evaluated expression of `uImmediate[ ]`.

```
uDelayed[x, Pi]
```

Here,  $\pi$  has been substituted for  $b$  in the literal form of `uDelayed[ ]`, then the integration has been performed with  $\pi$  instead of  $b$ . We should expect that the evaluation with  $b = \pi$  of `uDelayed[ ]` takes more time than the evaluation with  $b = \pi$  of `uImmediate[ ]`.

An individual evaluation is much too fast for timing experiments. Therefore, we will compute a table of values. Since we are interested only in the computation time and not in the results of the computations, we suppress the output of the computations with the semicolon symbol inside the invocation of the `Timing[ ]` function.

```
?Timing
```

```
Timing[Table[{b,uImmediate[x,b]},{b,0,Pi,Pi/256}]];
```

```
Timing[Table[{b,uDelayed[x,b]},{b,0,Pi,Pi/256}]];
```

In the first table only the bound needs to be substituted since the integral was computed when we defined `uImmediate[ ]`. In the second table the integral must be computed and evaluated 257 times.

Unless your computer has a very fast processor, you should see that the evaluation of the second table takes more time than the evaluation of the first. If you do not see a difference in computation time, decrease the step size. On a 266 MHz Power PC G3 we recorded computation times of 0.117 seconds for `uImmediate[ ]` and 24.45 seconds for `uDelayed[ ]`. On a 533 MHz PC the computation times were 0.06 seconds and 11.15 seconds, respectively.

**Example 14:** Computational efficiency of two different definitions for one polynomial.

One way to define a polynomial is to explicitly write it down in its standard form with increasing order of exponents. Another way is to collect the coefficients into a list, to collect the powers of the unknowns into a list, and then to compute the dot product of those two vectors. We use these two computational descriptions to define two polynomials and use the `Timing[ ]` function to measure the time required for their evaluation.

```
Clear[uPoly1,uPoly2]
uPoly1[x_] := 1 + 7x - 3x^3 + 2x^4
uPoly2[x_] := {1,7,0,-3,2}.{x^0, x^1, x^2, x^3, x^4}
```

```
?uPoly1
```

```
?uPoly2
```

Here are evaluations of the two polynomials at a numeric argument.

```
uPoly1[5.82]
```

```
uPoly2[5.82]
```

It is instructive to trace the evaluations that Mathematica performs internally during the computation of the values of the two polynomials.

```
?Trace
```

```
uPoly1[4]
Trace[uPoly1[4]]
```

```
uPoly2[4]
Trace[uPoly2[4]]
```

Depending on the speed of your computer's processor, you may want to increase or decrease the step size in the following tables.

```
Timing[Table[uPoly1[k], {k,1.0,5.0,0.0001}]];
```

```
Timing[Table[uPoly2[k], {k,1.0,5.0,0.0001}]];
```

These timing experiments show that the second evaluation takes longer than the first. As in Example 13, if you do not see a difference in computing time, decrease the step size. On a 266 MHz PowerPC G3 we recorded computing times of 4.45 and 5.85 seconds, respectively. On a 533 MHz PC the computing times were 2.31 and 2.69 seconds.

## Exercises

---

**EXERCISE 1:** Computations with a matrix of random integer entries.

(a) Use an immediate assignment and also a delayed assignment to define a square matrix with random integer values in the range from  $-20$  to  $+20$ . If you need to generate lots of matrices to create examples, would you use the immediate or the delayed assignment?

(b) Show by example that the commutative law does not hold for matrix multiplication.

(c) Use part (a) to generate evidence that the inverse of a square matrix exists precisely when the determinant is non-zero. Choose square matrices with random integer values in the range from  $-2$  to  $+2$ .

(d) Generate a  $4 \times 4$  matrix  $m$  and compute its characteristic polynomial  $f(x)$ . Then evaluate the characteristic polynomial at  $m$ ; that is, replace  $x$  by the matrix  $m$  and replace scalar multiplication with matrix multiplication. Computational hint:  $f[x\_] := \text{Det}[m - x * \text{IdentityMatrix}[4]]$ . If the resulting polynomial is  $f(x) = 3x^4 - 2x + 5$  and the  $4 \times 4$  matrix is  $m$ , then you compute  $3 m.m.m.m - 2 m + 5 \text{IdentityMatrix}[4]$ . Repeat this with several different examples. Is there evidence for a theorem? If so, formulate the theorem.

**EXERCISE 2:** Computations with list operation and element extraction functions.

- (a) Make a list  $t$  of 20 numbers, either randomly or according to some rule.
- (b) Use the `Sort[ ]` function to rearrange the elements of  $t$  once into ascending and once into descending order. Use `LessEqual` and `GreaterEqual` as ordering functions.
- (c) Investigate the list extraction functions `First[ ]`, `Last[ ]`, `Rest[ ]`, and `Take[ ]`. Apply these functions to the list  $t$  that you created in part (a).
- (d) Divide the list  $t$  into four disjoint lists by using either various combinations of the functions that you learned about in part (c) or the list function `Partition[ ]`.
- (e) Combine the lists you created in part (d) using the `Join[ ]` function to produce a single list that contains all 20 elements, but is different from the list  $t$ .

**EXERCISE 3:** Plotting functions of one variable.

- (a) Define a polynomial function  $f$  of degree 3.
- (b) Define a polynomial function  $g$  of degree 5.
- (c) Graph each of the functions in separate plots and in a combined plot.
- (d) Experiment with the range specifier `PlotRange` and with `AspectRatio` to improve the graphs.
- (e) Experiment with options such as `GridLines` and `PlotStyle` to improve the graphs. Produce one combined graph using the function `Show[ ]`, which includes options for the individual graphs as well as options for the function `Show[ ]`.

**EXERCISE 4:** Critical points of a polynomial of degree four.

Use either the polynomial `uPoly1[ ]` or the polynomial `uPoly2[ ]` from Example 14.

- (a) Graph the polynomial along with its first and second derivatives.
- (b) Compute the zeros of the first derivative.
- (c) Compute the zeros of the second derivative.
- (d) Determine the coordinates of all critical points and specify whether they are maxima or minima.

**EXERCISE 5:** Relative growth of a family of functions.

- (a) Define several functions of the type  $\frac{x^k}{k^x}$ , for a fixed positive integer value of  $k$ .

- (b) Plot the functions you defined in part (a).
- (c) Use your plots from part (b) to determine whether the functions grow beyond all bounds.
- (d) Give a formal mathematical argument for your answer in part (c).
- (e) Discuss whether it makes sense to use the values 0 or 1 for  $k$  in the expression  $\frac{x^k}{k^x}$ .

**EXERCISE 6:** Analyzing your personal wave function.

The periodic function that you define and then analyze in this exercise is based on your nine-digit Social Security number. For example, let us suppose that your Social Security number is 123-45-6789.

- (a) Pick four different nonzero digits from your Social Security number and arrange them in any order you wish as  $d_1, d_2, d_3$ , and  $d_4$ . Define the trigonometric function  $t(x) = d_1 \sin(d_2 x) + d_3 \cos(d_4 x)$  to be your personal wave function. For example, if we pick the digits 7, 3, 1, and 4 in that order, then our personal wave function will be  $7 \sin(3x) + \cos(4x)$ .
- (b) Plot your personal wave function that you defined in part (a) together with its first derivative in one plot. Choose a plotting range that shows the essential features of the two functions.
- (c) Use `FindRoot[ ]` to find all roots and critical points of your personal wave function in the interval  $0 \leq x \leq \frac{\pi}{2}$ .
- (d) Investigate the function `FactorInteger[ ]` and use it to represent your Social Security number as a product of prime numbers.

**EXERCISE 7:** Immediate and delayed definition of functions.

- (a) Evaluate each of the built-in algebraic action functions `Apart[ ]`, `Together[ ]`, `Expand[ ]`, and `Simplify[ ]` with symbolic algebraic expressions. Find out what each function does and describe it in your own words.
- (b) Use `Expand[ ]` to define a function with the immediate assignment operation that expands the expression  $(x - 2)^3$  for an arbitrary argument  $x$ . Evaluate your function at numeric and symbolic arguments. Use `Trace[ ]` to see how the function is evaluated.
- (c) Use `Expand[ ]` to define a function with the delayed assignment operation that expands the expression  $(x - 2)^3$  for an arbitrary argument  $x$ . Evaluate your function at numeric and symbolic arguments. Use `Trace[ ]` to see how the function is evaluated.
- (d) Compare your results in parts (b) and (c) and relate the results to the evaluation of `Factor[ ]` that was discussed in Example 11.

**EXERCISE 8:** The order of evaluation in immediate and delayed assignments.

Perform the following sequence of definitions in the order given:

```
Clear[f, g, x]
```

```
x = 4
```

```
f [x_] = Factor[x^2 - 2x + 1]
```

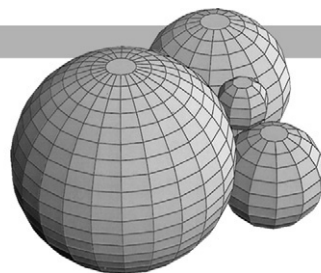
```
g [x_] := Factor[x^2 - 2x + 1]
```

What is the output when you enter these four lines?

- (a) Predict the results of `?f` and `?g` and then evaluate `?f` and `?g`.
- (b) Predict the values of `f[x]` and `g[x]` and then evaluate `f[x]` and `g[x]`.
- (c) Predict the values of `f[7]` and `g[7]` and then evaluate `f[7]` and `g[7]`.
- (d) Predict the result of the sequence of commands `Clear[y]`, `f[y]`, `g[y]`, and then evaluate.



This Page Intentionally Left Blank



## Introduction

There are two essentially different views of a function, the set-theoretical or object view, and the operational or algorithmic view.

The set-theoretical view sees a function  $f : S \rightarrow T$  as a subset of the Cartesian product of  $S$  and  $T$ , which satisfies the function property: for every element  $s$  in  $S$  there is a unique element  $t$  in  $T$  such that  $f(s) = t$ . Therefore, we can represent a function  $f$  as the set of pairs  $G = \{(s, f(s)) \mid s \text{ in } S\}$ . This set  $G$  is usually referred to as the graph of the function  $f$  when both the domain and the range are sets of real numbers. Mathematica computes for each function  $f$  a subset of  $G$  in order to render the graph of  $f$  by joining these points with straight-line segments.

The operational view sees a function as the implementation of an algorithm or rule, with a sequence of input parameters or formal arguments, which returns some output value. We may define a function with one or more arguments depending on the structure of its domain and depending on how we want to express this structure of the domain in terms of the arguments. However, a function always returns a single object whose value and structure are determined by the structure of the range of the function.

Mathematica permits a large amount of freedom in designing the sequence of input parameters for a function: their number, their ordering, and their data types. Similarly, we have a large amount of freedom for the single value that a function returns: it may be a number, an entire list of objects, a graphics object, or another function.

## Functions with a Single Argument

In this section we demonstrate how functions of a single argument are defined and evaluated.

The concept of a function is used very loosely in many areas of mathematics and in application fields. This is done for convenience and generally is of no consequence. However, in the setting of a symbolic algebra system such as Mathematica we need to be precise and unambiguous so that we can perform correct computations. For each of the three parts  $f$ ,  $S$ , and  $T$  of the function  $f : S \rightarrow T$  we need a descriptive element that implements the function computation  $f(s) = t$ . There are two levels of description: the definition and the evaluation or invocation for a function. For example,  $f(x) = -\frac{x}{2} + \frac{7}{2}$  represents the definition of a straight line of slope  $-\frac{1}{2}$  through the point  $(1,3)$ ;  $f(3) = 2$  represents the evaluation of the function  $f$  at the specific  $x$ -value 3 resulting in the value 2. The mathematical notational convention uses the same equality symbol  $=$  to indicate definition as well as evaluation. We are supposed to know from the context whether

a particular = means define or evaluate or even compare. Mathematica must distinguish between these different uses in order to compute unambiguously.

Three items are always part of the definition of a function of one variable: the function name, the name of the variable, and the function expression. In the preceding example, *f* is the function name, *x* is the name of the variable and  $-\frac{x}{2} + \frac{7}{2}$  is the function expression. We could, however, have written  $g(s) = -\frac{s}{2} + \frac{7}{2}$  instead, indicating a variable *s*. The straight lines described by the functions *f* and *g* are the same and their defining expressions are the same except for the name of the variable. In other words, we can use any symbol as a variable as long as the same symbol appears on the left- and right-hand sides of the function definition.

When we define a function in Mathematica we need to explicitly associate the input parameters with the function name on the left-hand side of the function definition. This is done by using pattern names such as *x\_* in the argument list of the function name. Any occurrence of the symbol name *x* on the right-hand side of the function definition is then bound to this pattern. The effect of this association is that the scope of the name *x* is limited to the right-hand side of the function definition.

**Example 1:** A straight-line definition.

```
Clear[uStraight]
uStraight[x_] := -x/2 + 7/2
```

Observe that we include the `Clear[ ]` function in the same cell as the function definition and before the function definition. This has the effect that any time we change the definition of `uStraight[ ]` we first erase the old definition of `uStraight[ ]`. All input expressions in this exercise are evaluations of the function definition given earlier.

```
uStraight[0]

uStraight[3]

uStraight[5.5]
```

Since we did not specify the data type of the argument of the function, any argument value is permitted for which the right-hand side can be successfully evaluated. Such a value could include a complex number, a symbol, and even a list of numbers. Note that a list is a single object, regardless of its internal structure, and can therefore be the argument for a function of one variable.

```
uStraight[5 + I]

uStraight[s]

uStraight[{-2.2, 3.5}]
```

We can apply `uStraight[ ]` to itself or apply any other function, such as the derivative `D[ ]`, to `uStraight[ ]`.

```
uStraight[uStraight[1]]

D[uStraight[x], x]
```

Finally we can plot the function `uStraight[ ]`.

```
Plot[uStraight[x], {x, -2, 4}];
```

If we enter two separate arguments into the function `uStraight[ ]`, no evaluation is carried out since such a function is not defined.

```
uStraight[3, -2]
```

**Example 2:** Using different named pattern variables in a function definition.

We redefine the function `uStraight[ ]` with the pattern variable `s_` and a different function expression on the right-hand side. First we inspect the existing definition from Example 1.

```
?uStraight
```

```
uStraight[s_] := 3s/2 - 5/2
```

```
?uStraight
```

Since we keep the same function name and the same number and (unspecified) data type of argument, the new definition replaces the old definition.

**Example 3:** Conditional evaluation of a function.

Frequently, we want to restrict the data type of the argument to specific objects such as integers rather than permitting any object as we did in Example 1. One way of doing this is to use the condition operator `/;` followed by a test for the arguments. In this example we test whether the argument is an integer. This is accomplished with the function `IntegerQ[ ]`.

```
Clear[uStraight]
```

```
uStraight[s_] := 3s/2 - 5/2
```

```
uStraight[t_] := 9t/2 - 17/2 /; IntegerQ[t]
```

```
?uStraight
```

Observe that there are now two distinct definitions of the function `uStraight[ ]`, and that the definition restricted by a condition is listed first. That means that Mathematica attempts to evaluate it first. Should that evaluation fail with the specific argument provided, the second, more general definition is tried.

```
uStraight[0]
```

```
uStraight[0.]
```

When we use as an argument a symbol that does not have a value, the second definition is evaluated since a symbol is not an integer.

```
Clear[w]
```

```
uStraight[w]
```

Another method to put a condition on the evaluation of a function is to modify the pattern variable, restricting it to the data type that is permissible. This is done by attaching the type of the object, for example, Integer, to the pattern variable.

```
Clear[uStraight]
uStraight[x_Integer] := -4x + 7

?uStraight

uStraight[0]

uStraight[-2]

uStraight[0.]

uStraight[{3,-5}]
```

Since we cleared all previous definitions of uStraight[ ], only this last definition is active. We can now evaluate it only for integer arguments.

**Example 4:** A function of a single argument defined by cases.

Frequently a function is defined by a collection of mutually exclusive cases. Utility rates and income tax schedules are examples. We use the year 2000 tax tables, Schedule X, of the Internal Revenue Service for taxpayers filing with single status. The entire tax structure for this schedule, as listed on page 71 of the 2000 1040 Forms and Instructions booklet, is:

If Amount on Form 1040 Line 39 is Over	But not Over	Enter	+ Taxrate	* the Amount Over
\$0	\$26,250	\$0	15%	\$0
\$26,250	\$63,550	\$3,937.50	28%	\$26,250
\$63,550	\$132,600	\$14,381.50	31%	\$63,550
\$132,600	\$288,350	\$35,787.00	36%	\$132,600
\$288,350	—	\$91,857.00	39.6%	\$288,350

The percentage in the fourth column of the tax rate schedule is called the marginal rate. It is the rate at which an additional earned dollar is taxed, based on your current income. This table gives rise to the following definition of a function by five different cases.

$$\text{Tax}(s) = \begin{cases} 0.15 s & \text{if } s \leq \$26,250 \\ 3,937.50 + 0.28 (s - 26,250) & \text{if } \$26,250 < s \leq \$63,550 \\ 14,381.50 + 0.31 (s - 63,550) & \text{if } \$63,550 < s \leq \$132,600 \\ 35,787.00 + 0.36 (s - 132,600) & \text{if } \$132,600 < s \leq \$288,350 \\ 91,857.00 + 0.396 (s - 288,350) & \text{if } \$288,350 < s \end{cases}$$

To define a function that computes the tax in all these different cases, we use Mathematica's Which[ ] function.

```
?Which
```

```

Clear[uTax]
uTax[s_]:= Which[s ≤ 26250, 0.15 s, 26250 < s ≤ 63550, 3937.50 +
    0.28 (s-26250), 63550 < s ≤ 132600, 14381.50 + 0.31
    (s-63550), 132600 < s ≤ 288350, 35787.00 + 0.36 (s-132600),
    288350 < s, 91857.00 + 0.396(s-288350)]

```

The function `uTax[ ]` now computes the taxes for any taxable income amounts.

```

uTax[3333]

uTax[25000]

uTax[50000]

uTax[90000]

uTax[270000]

uTax[1000000]

```

Graphical representations are useful for visual display of information. We therefore plot the tax function over an income range so that all tax rates are rendered. We also include grid lines positioned at the income levels and at the corresponding tax levels where the marginal tax rate changes.

```

Plot[uTax[x], {x, 0, 325000}, AspectRatio → 1,
    GridLines → {{26250, 63550, 132600, 288350},
    {3937.50, 14381.50, 35787.00, 91857.00}}];

```

For a taxable income of \$100,000 the marginal tax rate is 31% and the actual overall tax rate paid for this income is:

```

Clear[actualrate]
actualrate = uTax[100000]/100000

```

We round this rate to one decimal place since the marginal tax rate percentages in the IRS tax tables contain at most one decimal place. The `Round[ ]` function returns an integer so that we must shift our result first by three digits to the left and after rounding by a single digit to the right.

```

?Round

Round[1000 actualrate]/10.

```

For a taxable income of \$30,000 the marginal tax rate is 28% and the overall tax rate paid for this income is:

```

Clear[actualrate]
actualrate = uTax[30000]/30000

Round[1000 actualrate]/10.

```

**Example 5:** Multiple function definitions for distinct argument ranges.

We do not need the `Which[ ]` function in order to define a function for different cases. Instead, we can give one definition for each range. We define a function `uSlide[ ]` that is defined in four pieces.

```
Clear[uSlide]
uSlide[x_] := -1/(x ≤ -1)
uSlide[x_] := -(1-(x + 1)^2)/( -1 < x ≤ 0)
uSlide[x_] := (1-(x - 1)^2)/( 0 < x ≤ 1)
uSlide[x_] := 1/(1 < x)
```

Mathematica will store the four definitions separately as distinct rules with which to compute values for `uSlide[ ]`.

```
?uSlide
```

Here are a few evaluations and computations with `uSlide[ ]`.

```
uSlide[-0.5]
uSlide[0]
uSlide[0.0]
uSlide[-1]
uSlide[Log[2]]
uSlide[N[Log[2]]]
Plot[uSlide[x], {x, -2, 2}];
```

As an alternative to the four definitions for `uSlide[ ]` we can use `Which[ ]`. In that case we define a single function.

```
Clear[uSCurve]
uSCurve[x_] :=
  Which[x ≤ -1, -1, -1 < x ≤ 0, -Sqrt[1-(x + 1)^2],
    0 < x ≤ 1, Sqrt[1-(x - 1)^2], 1 < x, 1]
?uSCurve
uSCurve[0.6]
uSCurve[-1]
Plot[uSCurve[x], {x, -2, 2}, PlotStyle → RGBColor[0, 1, 0]];
```

We can demonstrate how the two functions `uSlide[ ]` and `uSCurve[ ]` are evaluated by tracing the evaluation steps with `Trace[ ]`.

```
Trace[uSlide[0.5]]
Trace[uSCurve[0.5]]
```

## *Functions with Several Arguments*

---

In this section we introduce a few sample functions with several arguments of a numeric data type. This topic extends the previous section and provides the natural setting for three-dimensional plots and contour plots.

**Example 6:** A function of two numeric arguments returning a numeric value.

We use a function that is logarithmic in the x-direction and trigonometric in the y-direction.

```
Clear[uLogCos1]
uLogCos1[x_, y_] := Log[1 + x] Cos[y]

uLogCos1[0, 4]

uLogCos1[3.5, 0]

uLogCos1[E^2 - 1, Pi]
```

For a negative real number the logarithm returns a complex number.

```
uLogCos1[-2, 2]
```

We plot the surface and the contour defined by the function for several oscillations.

```
Plot3D[uLogCos1[x, y], {x, 0, 10}, {y, -2Pi, 4Pi}, PlotPoints -> 30];

ContourPlot[uLogCos1[x, y], {x, 0, 10},
  {y, -2Pi, 4Pi}, PlotPoints -> 30, ContourLines -> False];
```

The function `uLogCos1[ ]` can be evaluated on symbolic expressions.

```
Clear[a, b]
uLogCos1[a^3 - 1, b + Pi]
```

However, we might want to restrict evaluation only to numeric arguments that are within the domain of the function, for example, to non-negative values for  $x$  or to values of  $x$  greater than  $-1$ . This requires a compound Boolean expression for which we write a separate test function `uTest[ ]`. The property of an object to be either an explicit number or a mathematical constant can be tested with the question function `NumericQ[ ]`. The symbol `&&` is the logical And function.

```
?And
```

```
?NumericQ
```

```
Clear[uTest, uLogCos2]
uTest[x_, y_] := NumericQ[x] && NumericQ[y] && (x >= 0)
uLogCos2[x_, y_] := Log[1 + x] Cos[y] /; uTest[x, y]
```



Here are several evaluations of the test function `uTest[ ]` and of `uLogCos2[ ]`.

```
uTest[4,-5]
uTest[-4,5]
uTest[0.5,Pi]
uTest[0.5,N[Pi]]
```

Whenever the test function `uTest[ ]` fails, the function `uLogCos2[ ]` stays unevaluated and Mathematica returns your unevaluated input expression.

```
uLogCos2[4,-5]
uLogCos2[-4,5]
uLogCos2[0.5,Pi]
uLogCos2[E-1,3Pi]
uLogCos2[0,10]
```

Evaluation of `uLogCos2[ ]` will now fail for symbolic arguments.

```
Clear[a,b]
uLogCos2[1.5,a - b]
```

**Example 7:** A function of two simple arguments that returns a structured value.

There are many occasions when we want to draw a function  $r = f(t)$  that is given in polar coordinates. However, Mathematica renders its graphics in a rectangular coordinate system. Therefore, we must compute the rectangular coordinates first and then plot those. We can compute the two rectangular coordinate functions as functions of the single parameter  $t$  and plot those two functions with `ParametricPlot[ ]` or we can do the plotting with `ListPlot[ ]`.

Here we define the function that translates a radial value  $r$  and an angle  $t$  into a pair of rectangular coordinates. This function returns this coordinate pair as a structured value.

```
Clear[uPolar]
uPolar[r_,t_] := {r Cos[t],r Sin[t]}
```

As a first application we translate the polar equation for an Archimedean spiral,  $r = 2t$ , into the corresponding rectangular coordinate functions and use `ParametricPlot[ ]` to graph the spiral.

```
?ParametricPlot

Clear[s]
uPolar[2s,s]

ParametricPlot[uPolar[2t,t],{t,0,2Pi}];
```

This warning message for `ParametricPlot[ ]` relates to an implementation issue for making calculations in Mathematica as efficient as possible. We are interested in the results of the plot, not

the speed of its computation, and would like to suppress this message. That can be done with the following command:

```
Off[ParametricPlot::"ppcom"]

ParametricPlot[uPolar[2t,t],{t,0,2Pi}];
```

The corresponding command to turn on the messages is:

```
On[ParametricPlot::"ppcom"]

ParametricPlot[uPolar[2t,t], {t,0,2Pi}];

Off[ParametricPlot::"ppcom"]
```

We can also use `uPolar[ ]` to compute a table of points on the Archimedean spiral in rectangular coordinates. This list of points can then be displayed with `ListPlot[ ]`.

```
ListPlot[Table[N[uPolar[2t,t]], {t,0,2Pi,Pi/16}]];
```

As a second application we draw the hyperbolic spiral  $r = \frac{1}{t}$ .

```
uPolar[1/t,t]

ParametricPlot[uPolar[1/t,t], {t,Pi/6,6Pi}];
```

Mathematica did not draw the curve over the entire range for the parameter  $t$  so we force a complete rendering with the option `PlotRange`.

```
ParametricPlot[uPolar[1/t,t],{t,Pi/6,6Pi},PlotRange->All];
```

The function `PolarPlot[ ]` from the package `Graphics.m` combines into a single step our two separate steps, the translation to rectangular coordinates and the graphics rendering. We need to load the graphics package before we can use `PolarPlot[ ]`.

```
Needs["Graphics`Graphics`"]

?PolarPlot

PolarPlot[1/t,{t,Pi/6,6Pi}];
```

When we loaded the function `PolarPlot[ ]`, actually a number of additional packages were needed and automatically loaded by Mathematica. We can find out about those by inspecting the system variable `$Packages`. It returns the list of currently loaded packages.

```
$Packages
```

## *Functions with Structured Arguments*

---

Many of the built-in Mathematica operators have arguments that are lists. Examples are the range specifiers such as  $\{x, x_{\min}, x_{\max}\}$  and  $\{i, i_{\min}, i_{\max}, di\}$  in functions like `Table[ ]`, `ParametricPlot[ ]`, and `NIntegrate[ ]`. We can specify formal parameters that are lists in the same fashion for functions that we define ourselves.

**Example 8:** The structure of some built-in functions.

The `Table[ ]` function has two arguments. The first is any expression, for example a numeric expression or a matrix, and the second is a range specifier, that is, a list consisting of a symbol followed by two or three values.

```
Table[Sin[x], {x, 0, Pi/2, Pi/12}]
```

```
Table[{k, k^2, k^3, k^4}, {k, 10, 20}];
```

```
TableForm[%]
```

The simplest form of the function `ParametricPlot[ ]` has two arguments, each of which is a list. The first is a pair of coordinate functions and the second is a range specifier. We first compute a list of points on the curve at simple fractions of  $\pi$  to help us understand the curve.

```
Table[{Sin[2u], E^u}, {u, 0, Pi, Pi/4}];
```

```
TableForm[N[%]]
```

```
ParametricPlot[{Sin[2u], E^u}, {u, 0, Pi}];
```

The numeric integration function `NIntegrate[ ]` has a similar interface with one argument that is a function expression and one argument that is a range specifier.

```
NIntegrate[Sin[t], {t, 0, Pi}]
```

Here we have four invocations of functions that return different kinds of objects. The first evaluation of `Table[ ]` returns a list of exact symbolic mathematical expressions and the second returns a list of quadruples of integers. `ParametricPlot[ ]` returns a graphics object, and `NIntegrate[ ]` returns an approximate real number.

**Example 9:** Creating a square matrix of random integers.

We have already explained some uses of the random number generator `Random[ ]` and we have created matrices with random entries. We combine the two processes of number generation and list generation into a single function that has a structured argument and that returns a structured value.

To accomplish this task we need to specify two quantities, the size of the square matrix and the range of integers from which the entries of that matrix are chosen. Therefore, the first formal parameter represents a positive whole number, and the second parameter represents a list of two integers that specify a range.

```
Clear[uRandomMatrix]
```

```
uRandomMatrix[size_, {min_, max_}] :=
```

```
Table[Random[Integer, {min, max}], {size}, {size}]
```

We create a few  $3 \times 3$  random matrices over different ranges.

```
uRandomMatrix[3, {0, 1}]
```

```
MatrixQ[%]
```

```
TableForm[%]
```

Observe that our function `uRandomMatrix[ ]` returns the mathematical object “matrix.” This is the object we need for further algebraic manipulation. The function `TableForm[ ]` controls the

printed form of its argument and returns a special formatting value rather than a mathematical object.

```
uRandomMatrix[3,{-10,10}];  
TableForm[%]
```

```
uRandomMatrix[3,{75,225}];  
TableForm[%]
```

When we supply values of an inappropriate type, such as decimal numbers or symbolic values, then `uRandomMatrix[ ]` attempts a computation. It may be successful and return a matrix depending on the behavior of the built-in functions that we used in `uRandomMatrix[ ]`.

```
uRandomMatrix[2.5,{-10,10}];  
TableForm[%]
```

```
uRandomMatrix[3,{0,Pi}]
```

We write another definition of `uRandomMatrix[ ]`, which enforces the restrictions and conditions for its arguments. In this example we use restrictions on the pattern variables as well as the conditional evaluation with the `/;` operator. We enforce the Integer data type condition on the pattern variables size, min, and max, and also that the range from which the entries of the matrix are chosen is nonempty.

First, we erase the old definition of `uRandomMatrix[ ]`.

```
Clear[uRandomMatrix]  
uRandomMatrix[size_Integer,{min_Integer,max_Integer}]:=   
  Table[Random[Integer,{min,max}],{size},{size}]/;(size > 0)&&  
  (min ≤ max)
```

Here are invocations of the new function that will compute a random matrix.

```
uRandomMatrix[2,{-10,10}]
```

```
uRandomMatrix[4,{-1,1}]
```

```
uRandomMatrix[1,{1,1}]
```

Here are invocations of the new function that will fail to compute a random matrix.

```
uRandomMatrix[2.5,{-10,10}]
```

```
uRandomMatrix[4,{-Pi,Pi}]
```

```
uRandomMatrix[5,{10,-2}]
```

```
uRandomMatrix[-5,{-10,10}]
```

**Example 10:** The commutator for square matrices of integers.

The multiplication of matrices is a non-commutative operation; that is, the equation  $AB = BA$  does not hold in general. We can therefore consider the matrix product  $A^{-1}B^{-1}AB$ , called the commutator of  $A$  and  $B$ , as a function of the matrices  $A$  and  $B$ . This function returns a matrix, provided that the inverses  $A^{-1}$  and  $B^{-1}$  exist. However, our definition of `uCommutator[ ]` does not check for this condition.

```
Clear[uCommutator]
uCommutator[A_,B_] := Inverse[A].Inverse[B].A.B
```

Observe that the formal parameters do not show any structure. However, our intention is that the actual arguments will be two square matrices of the same dimension. We use the function `uRandomMatrix[ ]` from the previous example to generate matrices for the commutator function.

Evaluate the sample pairs of random  $4 \times 4$  matrices and their commutator several times.

```
Clear[a,b]
a = uRandomMatrix[4,{0,10}];
MatrixForm[%]
b = uRandomMatrix[4,{0,10}];
MatrixForm[%]

MatrixForm[a.b]

uCommutator[a,b]
MatrixForm[N[%]]
```

Here is a pair of random  $3 \times 3$  matrices. The range of values for the entries in the two matrices is so restricted that it is likely that one of the matrices is not invertible. Evaluate the next four input cells several times in sequence until you obtain at least one singular matrix.

```
Clear[a,b]
a = uRandomMatrix[3,{-1,1}];
MatrixForm[%]
b = uRandomMatrix[3,{-1,1}];
MatrixForm[%]

MatrixForm[a.b]

uCommutator[a,b]

MatrixForm[%]
```

Here is an example where the function `uCommutator[ ]` fails. We define two specific matrices, one of which is not invertible.

```
Clear[a,b]
a = {{1,2},{-4,-8}}
b = {{2,1},{-1,2}}

a.b

uCommutator[a,b]
```

**Example 11:** Composing functions to achieve complex results.

We now use the function `uPolar[ ]` of Example 7 and the built-in function `ParametricPlot[ ]` together to draw graphs for a family of trigonometric functions. Because of the shape of their graphs we call these functions butterfly functions.

```
Clear[uPolar]
uPolar[r_,t_]:= {r Cos[t],r Sin[t]}

Clear[uButterfly]
uButterfly[t_,{n_,k_}]:= Sin[n t]^n + Cos[k t]
```

The two parameters  $n$  and  $k$  specify the period of the sine and the cosine function, respectively. Therefore, we group them in a list separate from the argument  $t$  for the angle. Furthermore, the parameter  $n$  is the exponent for the sine function resulting in a nonlinear combination of the two trigonometric functions. Again we suppress the error message for the `ParametricPlot[ ]` that we encountered earlier in this notebook.

```
Off[ParametricPlot::"ppcom"]

ParametricPlot[uPolar[uButterfly[t,{4,3}],t],{t,0,2Pi}];

ParametricPlot[uPolar[uButterfly[t,{2,2}],t],{t,0,2Pi}];
```

We would like to see many of these butterfly graphs in a single picture. We can achieve this by first calculating a table of graphs, for example, a  $4 \times 4$  table. We suppress rendering the actual graphics with the option `DisplayFunction`. In order to present a clear picture we set the `AspectRatio` to 1 so that a circle is displayed as a circle and we eliminate the axes in each graph.

```
?DisplayFunction

Clear[t,i,k]

t = Table[ParametricPlot[uPolar[uButterfly[t,{i,k}],t],
  {t,0,2Pi}, Axes → False,AspectRatio → 1,
  DisplayFunction → Identity], {i,1,4},{k,1,4}];
```

The function `GraphicsArray[ ]` arranges the graphs in the  $4 \times 4$  table as one single graph that is structured as a  $4 \times 4$  matrix of graphs. Finally, in order to render the graphics we apply the function `Show[ ]`.

```
?GraphicsArray

Show[GraphicsArray[t]];
```

## *Functions with Default Values for Arguments*

---

In many applications some of the parameters of a function have default values. For example, we tend to draw the graph of a trigonometric function over one period or a multiple of the period. Many weather-related data are normalized to sea level and to one atmospheric pressure.

Mathematica permits us to specify such default values for the pattern variables in the definition of functions. In this section we present examples of functions with simple numeric default values as well as with structured default values for some of their variables.

There is one important restriction for all default parameters of a function. The parameters for which defaults are defined must follow all parameters without a default specification. A pattern variable with a default value has the form `variable_:defaultvalue`.

**Example 12:** Defining a plotting function over the default range from 0 to  $2\pi$ .

In Example 11 of the previous section we drew some butterfly curves. The command to draw the butterfly curves was rather lengthy. If we want to study the curves in greater detail, this command has to be invoked over and over again and it will be advantageous to have a function whose invocation involves less text. We need to isolate those aspects of the invocation that vary, and make these the parameters of our new function. Furthermore, we should give this function a name that conveys its purpose and action.

Since we want to study a family of functions that are of the same type, we might decide that it is appropriate for most plots to be over the same interval, for instance from 0 to  $2\pi$ , but at times we might want to change this interval. Mathematica provides a mechanism for just this purpose—the parameter with a default value.

The interface for this function will consist of two structured arguments. The first structured argument contains the pair of period parameters  $n$  and  $k$  for the butterfly. The second structured argument specifies the interval from the starting value  $a$  to the final value  $b$  over which to plot the butterfly. We give these two bounds the default values of 0 and  $2\pi$ , respectively.

The definition requires the functions `uPolar[ ]` and `uButterfly[ ]` from Example 11. Make sure they have been evaluated.

```
?uPolar
```

```
?uButterfly
```

```
Clear[uFlyPlot,n,k,r,t]
```

```
uFlyPlot[{n_,k_},r_: {0,2Pi}]:= ParametricPlot  
  [uPolar[uButterfly[t, {n,k}],t], {t,First[r],Last[r]}]
```

We evaluate the `uFlyPlot[ ]` function with the default values and also with explicit bounds for the parameter.

```
uFlyPlot[{3,4}];
```

```
uFlyPlot[{3,4},{-Pi/2,Pi/2}];
```

```
uFlyPlot[{5,2}];
```

Once we have a picture of the entire butterfly we can study individual portions of it.

```
uFlyPlot[{5,2},{-Pi/8,Pi/8}];
```

Observe that `uFlyPlot[ ]` has only two arguments and that the graphing function `ParametricPlot[ ]` is used inside the definition of `uFlyPlot[ ]`. Therefore, we cannot use any of the options of `ParametricPlot[ ]` with this definition of `uFlyPlot[ ]`.

**Example 13:** Shifting the phase in a trigonometric expression.

In many applications in physics and engineering, oscillating processes, which are described by trigonometric functions, show phase shifts. Examples are the movement of a mass on a spring with an external periodic force acting on the mass and electric circuits built with resistors and capacitors. We define a function `uPhase[ ]` that adds a phase shift to a function. Our intent is to use the `uPhase[ ]` function in the context of trigonometric expressions, but we can use it with any algebraic expression as well.

The first argument of `uPhase[ ]` is the functional expression, `expr`, the second is the variable `x` in that expression, and the third argument, `shift`, is the phase shift with default value 0. In order to include the phase shift in the expression we use the replacement operator (`/.`).

```
Clear[uPhase]
uPhase[expr_, x_, shift_:0] := (expr /. x -> (x + shift))
```

We evaluate the `uPhase[ ]` function with the default value and with an explicit shift. When we use an expression as an argument, Mathematica attempts to simplify it.

```
Clear[x]
uPhase[Cos[x] + Sin[x], x]

uPhase[Cos[x] + 2Sin[x], x, Pi/2]
```

Any variable name can be used in the expression since `x_` is a pattern.

```
Clear[s]
uPhase[Cos[s] + Sin[s], s, Pi/4]
```

When we use a phase shift with an algebraic, non-trigonometric expression we may need to explicitly force Mathematica to simplify the resulting expression.

```
uPhase[(3x^2 - 5x + 1)/(x^2 + 1), x, -1]
Simplify[%]
```

**Example 14:** Designing a function to compute car loans.

Suppose we want to buy a car and must take out a loan for it since we do not have the purchase price in the bank. Obviously, we need to be able to judge our total financial commitment for such a loan. Therefore, we would like to experiment with different interest rates, different loan periods, and different monthly payments that we might be able to afford. We also want to play through the numbers with different values for the initial loan amount.

This suggests that we should design a function, say `uCarLoan[ ]`, of the four variables just mentioned, which returns the remaining debt. For example, `uCarLoan[ p, m, r, k ]` computes the remaining loan amount when:

- `p` is the initial principal (in \$)
- `m` is the monthly payment (in \$)
- `r` is the annual interest rate (in %)
- `k` is the period of the loan (in months)

Since car loan interest rates tend to change slowly and since the duration of car loans frequently is standardized, we specify default values for these two parameters in our implementation.



Now, if  $x_k$  represents the outstanding loan balance at the beginning of the  $k^{\text{th}}$  month then

$$x_{k+1} = x_k(1 + \frac{r}{1200}) - m$$

is the outstanding balance at the beginning of the  $(k + 1)^{\text{st}}$  month.

It is common bank lending practice that the first loan payment is due at the signing of the loan papers. This means that the initial amount on which interest must be paid is the loan amount minus one monthly payment. For example, when we take out a loan of \$10,000 with a monthly payment of \$325 then we have to pay interest on \$9,675 during the first month.

With this in mind we let  $x_0 = p$  be the total loan amount and  $x_1 = p - m$  the loan balance at the beginning of the first month. A proof by induction then yields the following closed formula for  $x_k$  when  $k \geq 1$ :

$$x_k = \left(p - m - \frac{1200}{r}m\right) \left(1 + \frac{r}{1200}\right)^{k-1} + \frac{1200}{r}m$$

We define the `uCarLoan[ ]` function with the default value of the interest rate set at 9.5% and the period of four years for the loan. You should change these numbers if they do not reflect the common values in the current loan market. Observe also that the outstanding principal on a loan for  $k$  months must be zero at the beginning of month  $k + 1$  or, equivalently, equal to the monthly payment at the beginning of month  $k$ .

We set the default for the time period of the loan at 48 months so that the function returns the outstanding balance on the loan at the beginning of the 49<sup>th</sup> month, that is, after 48 payments have been made. That means that we replace  $k$  by  $k + 1$  in the formula for  $x_k$ , earlier.

```
Clear[uCarLoan]
uCarLoan[p_, m_, r_:9.5, k_:48] := (p - m - m 1200/r)
  (1 + r/1200)^k + m 1200/r
```

As a first experiment we assume that the amount of the loan is  $p = \$10,000$  and that the monthly payment is  $m = \$250$ . Then we get as the outstanding loan amount at the end of four years:

```
uCarLoan[10000, 250]
```

It turns out that this was a pretty good guess. We can now experiment with the value of the second argument, the monthly payment, to get the outstanding loan amount as close to zero as possible or below zero. Evaluate this function with different values for the second argument.

If we want to see the values for a four-year loan with different interest rates at 0.5% intervals then we can create the following list:

```
Table[uCarLoan[10000, 250, r], {r, 8.0, 12.0, 0.5}]
```

We can also compute a list for the remaining principal with a fixed interest rate. We vary the loan period from three to five years at three-month intervals. Observe that we must enter the interest rate explicitly, even the default value of 9.5, since we need to specify the fourth parameter in this computation.

```
Table[uCarLoan[10000, 250, 9.5, k], {k, 36, 60, 3}]
```

We can produce a tabular form for this list of outstanding balances at the end of four years that also includes the months as follows:

```
TableForm[Table[{k, uCarLoan[10000, 250, 9.5, k]}, {k, 36, 60, 3}]]
```

The financial picture with an interest rate of 6.0% is the following:

```
TableForm[Table[{k,uCarLoan[10000,250,6.0,k]},{k,36,60,3}]]
```

Rather than experimenting with different numbers and printing out tables as we did earlier, it is much more efficient to solve the equation  $\text{uCarLoan}[p, m, r, k] == 0$  for the monthly payment  $m$  directly. As we can see from the definition of the function  $\text{uCarLoan}[]$  it is a linear function of the variable  $m$ . Therefore, we can solve the equation with the function  $\text{Solve}[]$ . For a four-year loan period we get the following result:

```
Clear[m, pay]  
pay = Solve[uCarLoan[10000, m] == 0, m]
```

If we want to see the outstanding loan balance at each month for the last year of the loan we enter the amount for the monthly payment in the input cell:

```
TableForm[Table[{k,uCarLoan[10000,m,9.5,k]/.pay},{k,37,48}]]
```

For a different interest rate and a three-year loan period we get:

```
Clear[m, pay]  
pay = Solve[uCarLoan[10000,m,6.0,36] == 0, m]  
  
TableForm[Table[{k,uCarLoan[10000,m,6.0,k]/.pay},{k,24,36}]]
```

It is, however, an entirely different matter to solve the equation for the number of months when all other parameters are specified because we then must solve an exponential equation. In Mathematica, we can approximate solutions to such equations with the function  $\text{FindRoot}[]$ .

If we assume a loan amount of \$10,000, a monthly payment of \$300, and an interest rate of 9.5%, we make a guess that it will take about three years, that is 36 months, to pay off the loan.

```
FindRoot[uCarLoan[10000,300,9.5,k] == 0,{k,36}]
```

**Example 15:** Non-standard evaluation in plotting functions.

We want to make a final comment about the evaluation of arguments in a function. We have made the basic assumption that all variable symbols that appear on the right-hand side of the definition of a function have their corresponding pattern variables on the left-hand side of the definition.

In the  $\text{uFlyPlot}[]$  function of Example 12 however, the symbol  $t$  occurs on the right-hand side, but not on the left-hand side. That is, in  $\text{uFlyPlot}[]$  the variable  $t$  apparently is not bound to a formal argument of  $\text{uFlyPlot}[]$ . We do not get into a problem with the variable name  $t$  in this context, since the built-in function  $\text{ParametricPlot}[]$ , like all plotting operators, has a non-standard evaluation mechanism. All its arguments are taken literally, that is, in symbolic form, since they must be evaluated many times for points in the plotting range. Therefore, the variable  $t$  in the definition of  $\text{uFlyPlot}[]$  is also taken literally. This evaluation mechanism is implemented in Mathematica with the attribute  $\text{HoldAll}$ .

```
Attributes[ParametricPlot]
```

All symbols built into Mathematica have the property  $\text{Protected}$ . This is done so that we cannot accidentally change their meaning.

The situation is very different with `Solve[]`. In Example 14 we solve for the duration of a car loan; that is, we use `Solve[]` to compute a value for the symbol `m`. We use `m` as a free parameter in the equation. Since `Solve[]` uses the standard evaluation mechanism of Mathematica we must ensure that `m` is a symbol without a value. We achieve this by invoking `Clear[m]`. Then `Solve[]` returns a substitution rule for the symbol `m`.

```
Attributes[Solve]
```

## *Functions with a Varying Number of Arguments*

---

The mechanism of default values for arguments of functions has limitations, not the least of which is the strict left-to-right order in the assignment of values. Mathematica provides a second mechanism, that of an option argument, which is independent of these limitations including the order of the arguments. We have given some examples of option arguments with the built-in functions `Plot[]` and `ParametricPlot[]` in many previous examples.

We can define option arguments with any of our own function definitions. In this section we demonstrate how options for a built-in function can be carried over to a user-defined function. We discuss the definition of user-defined options in a later chapter.

```
?Options
```

```
Options[Plot]
```

**Example 16:** Designing a function with the plotting options of `ParametricPlot[]`.

In a particular plot drawn with `ParametricPlot[]` we can use one option, several options, or no options at all. If we want to write a plotting function with the same possibilities we must be able to specify that a variable can occur several times as an argument. This variety of occurrences is represented by the pattern of three consecutive underscores following a variable, for example, `x__`. We use `uFlyPlot[]` from Example 12 to demonstrate optional arguments.

There is one basic restriction in the use of default arguments and optional arguments. We may not mix default arguments and optional arguments. Therefore, the two structured parameters of `uFlyPlot[]` will be mandatory parameters in the following definition. We name the parameter that represents the variable sequence of options `opts`.

```
Clear[uButterflyPlot]
uButterflyPlot[{n_, k_},{a_, b_},opts__]:=
ParametricPlot[uPolar[uButterfly[t,{n,k}],t],{t,a,b},opts]
```

Make sure that the two functions `uPolar[]` and `uButterfly[]` from Example 11 have been evaluated.

```
?uPolar
```

```
?uButterfly
```

First, we find all plotting options for `ParametricPlot[]`.

```
Options[ParametricPlot]
```

Here are some plots with varying numbers of plotting options.

```
uButterflyPlot[{3,4}, {0,2Pi}];

uButterflyPlot[{3,4},{0,2Pi},Frame→True,
  Axes→False, PlotStyle→{{Thickness[0.02],
  RGBColor[0,1,0]}}];

uButterflyPlot[{5,2},{0,2Pi},Frame→True, PlotLabel→
  "A butterfly of type {5,2}", PlotStyle→{{Thickness[0.02],
  RGBColor[1,0,0]}}];
```

We draw two halves of a butterfly in different colors and thickness and then show a combined plot.

```
Clear[bf1,bf2]

bf1 = uButterflyPlot[{7,2}, {0,Pi},Frame→True,
  PlotStyle→{{Thickness[0.006],RGBColor[1,0,0]}},
  AspectRatio→1];

bf2 = uButterflyPlot[{7,2},{Pi,2Pi},Frame→True,
  PlotStyle→{{Thickness[0.004],RGBColor[0,1,0]}},
  AspectRatio→1];

Show[bf1,bf2];
```

## Exercises

---

Whenever you design a function for the solution of a problem it is best to use names for the function and its parameters that are suggestive of their usage. Such a naming convention aids the reader and the designer in understanding the function and provides natural documentation of the function and its intended application domain.

For any task in these exercises the definition of a function should always be tested with invocations of the function for a variety of parameter values.

**EXERCISE 1:** A rational function of a single numeric argument.

- Define a rational function `uPolyQuotient[ ]` of the single variable  $x$  where the numerator and the denominator are polynomials whose degrees are between two and five, and where the denominator has at least one real root.
- Evaluate your function `uPolyQuotient[ ]` at several numeric and symbolic arguments.
- Plot your function `uPolyQuotient[ ]` in an appropriate plotting range.
- Repeat the parts (a) through (c) with a rational function `uPolyFraction[ ]` where the denominator has no real roots.

**EXERCISE 2:** A distance function in the plane.

- (a) Given is the circle  $C$  of radius 3 with center  $(-2, 5)$ . Define a function `uDistance[ ]` of two arguments that computes the shortest distance between a point  $P(x, y)$  and the circle  $C$ . For the design of your function you may assume that the point  $P$  is always outside the circle  $C$ .
- (b) Evaluate your function of part (a) for various points outside the circle  $C$ .
- (c) Expand the definition of the function `uDistance[ ]` to allow the computation of the distance for all points outside of as well as inside the circle  $C$ .

**EXERCISE 3:** Varying one coefficient in a polynomial.

- (a) Consider the polynomial  $x^4 + cx^2 + 1$  of degree four with the real coefficient  $c$ , and define the polynomial as a function `uPoly4[ ]` of the two arguments  $x$  and  $c$ .
- (b) Graph the function of part (a) with some fixed values for the coefficient  $c$ , for example  $c = -6, -3, 0, 1, 4$ .
- (c) Draw the five curves of part (b) in different colors in a single plot.
- (d) Graph the function of part (a) as a surface with `Plot3D[ ]`, draw a contour plot, and describe the effect of changing the value of the coefficient  $c$  in the polynomial.

**EXERCISE 4:** Functions using `Floor[ ]`, and `Ceiling[ ]`.

Consider the two functions  $f(x) = \frac{x}{\lfloor x \rfloor}$  and  $g(x) = \frac{x}{\lceil x \rceil}$ .

- (a) Look up the usages for `Floor[ ]` and `Ceiling[ ]` and define the functions  $f$  and  $g$ .
- (b) Graph both functions of part (a) individually. Select the plotting ranges so that no error messages are generated. Hint: Several plots for each function may be necessary.
- (c) Render all individual graphs from part (b) in a single plot.
- (d) Describe the differences between the two functions using your pictures of parts (a) through (c). Find descriptive names for the two functions.
- (e) Explain the local and global maxima and minima for both functions.
- (f) Describe the differences between the graphs produced by Mathematica in parts (a) through (c) and the mathematically correct graphs.

**EXERCISE 5:** A numerical and graphical approximation of  $2\pi$  with regular polygons.

- (a) Compute the circumference of a regular  $n$ -sided polygon inscribed in the unit circle. Write one function that returns the exact value of the circumference and one that returns an approximate numeric value. We expect you to use trigonometric functions.
- (b) Expand the definition of part (a) to allow an arbitrary radius for the circle.
- (c) Write a function that creates a table of circumferences of all regular polygons inscribed in the unit circle up to a given number of vertices. The first entry in the table should be the circumference for the inscribed triangle.

- (d) Improve the format of the output of the function in part (c). Include the number of vertices in the table and print the result in tabular form with headings for the columns.
- (e) Determine by computational experiment how many vertices for an inscribed regular polygon are required to approximate the value of  $2\pi$  to 1, 2, 3, ... decimal places, respectively.
- (f) Use the built-in operator `ListPlot[ ]` to plot the list of values computed by the function of part (c). Use the option `GridLines` to enhance the effectiveness of the plot.

**EXERCISE 6:** Synchronous oscillations of different frequencies.

- (a) Write a function that adds two sine functions of arbitrary frequencies, but that have the same fixed amplitude. Hint: One possibility is a function of three simple parameters.
- (b) Write a function that plots the function defined in part (a) over a specified plotting range. Use a structured parameter for the plotting range specification.
- (c) Write a function that plots the function defined in part (a) over a specified plotting range. However, this time also represent the two frequencies by a structured parameter. In addition, give the parameter for the plotting range a default value.
- (d) Expand the definition of the function in part (b) so that the two sine functions and the superposition function of part (a) are shown in a single plot. Color the graphs so that you can distinguish the various curves.
- (e) Experiment with various combinations of frequencies and plotting ranges. Interesting examples arise when the frequencies are almost identical or when they are integer multiples of each other. Write a short report about your experiments and discuss the curves that you generate.

**EXERCISE 7:** Enforcing restrictions on the arguments of a function.

In Example 10 we defined the `uCommutator[ ]` function for two square matrices. Since the definition of this function requires the computation of the inverses of the two matrices, the evaluation of the `uCommutator[ ]` function is not always successful as we demonstrated in the example. We must require that the two matrices are square, have the same dimension, and have inverses. Only if these conditions are met do we want to evaluate the `uCommutator[ ]` function.

- (a) Implement the `uCommutator[ ]` function so that the restrictions stated earlier are enforced. Use the `/;` condition operator and list all restrictions explicitly.

In Example 6 we defined a function `uTest[ ]`, which tests conditions for the arguments of another function. Use that example as a guide for the next part of this exercise.

- (b) Define a function `uComTest[ ]` for the testing expression of part (a) and use it in an alternate implementation of the `uCommutator[ ]` function.

**EXERCISE 8:** Interest rates for car loans and car leases.

This exercise is based on and extends Example 14. You should use or modify the function `uCarLoan[ ]` of that example whenever you think that it applies to and helps you solve the problem.

- (a) Pick an advertisement from a car dealer in your local newspaper and select a car model that you pretend to buy. Determine a price for the car and select an interest rate. Find the monthly payments to finance the car for three different sizes of the down payment and two different loan periods.
- (b) In Example 14 we described a method to compute the duration of a car loan given the other three quantities. Implement this method as a function and evaluate the function for several different interest rates.
- (c) In Example 14 we described a method to compute the monthly payment of a car loan given the other three quantities. Implement this method as a function and evaluate the function for several different loan amounts.
- (d) Write a function to compute the maximum initial loan amount that you can afford given the other three quantities and evaluate the function for several different monthly payments.
- (e) Describe a method by which you can compute the total amount of interest paid during the term of a car loan. How would you use Mathematica to find the amount? Define a function that implements the computations and evaluate it for several different loan setups.
- (f) Pick an advertisement from a car dealer in your local newspaper and select a car model that you pretend to lease. Determine the price of the car, the purchase price of the car at the end of the lease, the down payment, and the monthly payments to finance the car during the lease period. Compute the effective interest rate that you will be charged. Make the computations for two lease periods.
- (g) In Example 14 we defined the function `uCarLoan[ ]` without any proof for the correctness of the defining expression. Prove by induction that the formulas

$$x_1 = p - m \quad \text{and} \quad x_{k+1} = x_k \left(1 + \frac{r}{1200}\right) - m, \quad \text{for all } k > 0,$$

give rise to the closed form

$$x_k = \left(p - m - m \frac{1200}{r}\right) \left(1 + \frac{r}{1200}\right)^{k-1} + m \frac{1200}{r}, \quad \text{for all } k > 0.$$

- (h) Use the function `RSolve[ ]` from the `RSolve` package in `DiscreteMath` (`discrete/rsolve.m` in Windows) to find the closed form for the recursive definition in part (g). Show that the closed form obtained with `RSolve[ ]` is the same as the one in part (g). Use the platform-independent command `Needs["DiscreteMath`RSolve`"]` or the command `<< DiscreteMath`RSolve`` to load the package.
- (i) Check directly from the recursive formula for the `uCarLoan[ ]` function that the closed formula given in part (g) is correct. Compute in Mathematica, not with pencil and paper.
- (j) Rather than dealing with car loans, consider mortgages for private homes. Change the name of the function and change the default values for the interest and the duration of the loan so that they represent current realistic values for fixed term mortgages.

**EXERCISE 9:** A billing function for an electric utility company.

The amount of money that the electric company charges a customer for every kWh (kilowatthour) used depends on the type of the customer, residential or industrial, on the kind of usage, interruptible or not, and the amount of electricity used during the monthly billing period. You may want to get the rates for your local electric company from the company's web site.

We will use the rates charged by Detroit Edison. The utility uses a nominal month of 30 days and kWh usage per day. For example, suppose that a rate is set at \$0.05 per kWh for usage of at most 10 kWh per day. That allows for up to 300 kWh per month at that rate. If a customer actually uses 250 kWhs during the month, then the energy charge to that customer is  $\$0.05 * 250 = \$12.50$ . In order to simplify calculations you may average daily usage; that is, a customer who used 450 kWh during a month is assumed to have used 10 kWh each day—qualifying for 300 kWh at the \$0.05 rate—and is charged for the excess of 150 kWh for the month at a different rate.

(a) Here are the details for the Residential Customer Rate of Detroit Edison in 2001. All prices are given in fractions of dollars: \$0.0824 per kWh for the first 17 kWh per day; \$0.0965 per kWh for the excess over 17 kWh per day. Translate this information into a function `uResEnergy(w)` of two cases. First create a table similar to the income tax table in Example 4 and then write the function.

(b) In addition to the energy charges there is a base reduction credit of \$0.029574 per kWh that applies to all kWh used during the month. Finally the State of Michigan charges a sales tax of 4% on the utility bill. Include these two additional charges to compute the total utility bill in a function named `uResidentialEnergy[ ]`.

(c) Graph the function `uResidentialEnergy[ ]` across a range that covers the two rates.

(d) For the purposes of the Senior Citizen Rate, Detroit Edison defines a senior citizen to be a person at least 62 years of age and the head of the household. There are some additional restrictions on water heaters and other residential appliances that Detroit Edison may control. We will not take the effect of these restrictions into consideration. The senior citizen rates are: \$0.0537 per kWh for the first 10 kWh per day; \$0.1208 per kWh for the excess over 10 kWh per day. Translate this information into a function `uSenEnergy(w)` of two cases as in part (a). In addition to the energy charges there is a base reduction credit of \$0.023208 per kWh that applies to all kWh used during the month. Finally, the State of Michigan charges a sales tax of 4% on the utility bill. Include these two additional charges to compute the total utility bill in a function named `uSeniorEnergy[ ]`.

(e) Graph the function `uSeniorEnergy[ ]` across a range that covers the two rates.

(f) Determine the break-even point for the residential and senior rates; that is, determine for which levels of electric energy use the Senior Citizen Rate is a better deal for a senior citizen.

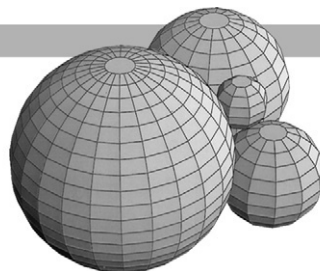
(g) The rate structure for the Small Business General Service rate is quite different from the residential and senior citizen rates. It consists of a monthly service charge of \$9.24, a charge of \$0.0995 per kWh for all kWh used, and a base rate reduction credit of \$0.0033248 per kWh on all kWh. There is also the sales tax of 4%. Compute the total utility bill for this rate in a function named `uSmallBusEnergy[ ]`.

(h) Graph the three functions `uResidentialEnergy[ ]`, `uSeniorEnergy[ ]`, and `uSmallBusEnergy[ ]` in a single plot. Discuss the characteristic features of the three rates.



This Page Intentionally Left Blank

# *Recursive Definitions*



## *Introduction*

One of the powerful tools in logic and in mathematics that is used to define quantities or to prove properties of objects is the method of induction. Its computational counterpart is the method of recursion. In many application domains the basic structures are defined recursively so that computations in such domains are frequently easier and much more natural to formulate recursively than iteratively.

There is a price to be paid for such convenience: speed of computation. But in many instances that price is trivial compared to the programming effort that must be expended in order to implement a non-recursive algorithm. As we shall see, Mathematica provides a mechanism in the definition of a recursive function that improves the efficiency of recursive computations. Recall the typical form of an inductive (recursive) definition of a function  $f$ :

Basis:  $f_k = a$ , for some fixed  $k \geq 0$  and some fixed value  $a$ .

Step:  $f_n =$  an expression in which at least one of  $f_{n-1}, f_{n-2}, \dots$  occurs, for all  $n > k$ .

The simplest form of this recursion principle has  $k = 0$  and  $f_n$  depending only on  $f_{n-1}$ . First we shall discuss recursive functions of one variable. There are more general recursions involving more than one variable. We will show examples for these without specifying the precise general form of these schemes.

Three rules are essential for the successful design of a recursive function:

1. The basis portion is an explicit assignment of some value and usually is written as an immediate assignment for the function value.
2. The step portion of the recursive definition expresses a rule on how to reduce the computation from argument  $n$  to a smaller argument such as  $n - 1$ , and therefore is written as a delayed assignment.
3. Every recursive evaluation must terminate after a finite number of invocations of the step portion with one or more invocations of the basis portion of the recursive definition.

## *Functions with a Single Recursive Argument*

In the following six examples we explain the different aspects of functions with one recursive argument.

**Example 1:** Computing a bank balance with a recursive function.

We start with the computations of the balance in an interest bearing account. We assume a starting balance of \$10,000.00 in the account, no cash deposits into or withdrawals from the account, and a yearly interest rate of 4.5% with the interest credited to the account. If we open the account today, we want to compute the balance in the account in future years. The interest rate sets up a relationship between the current balance  $b$  and next year's balance,  $1.045 * b$ . We can turn this into a recursive formulation as follows.

```
Clear[uBalance]
uBalance[0] = 10000.0;
uBalance[n_] := 1.045 uBalance[n - 1]
```

Here are some sample evaluations of this recursion.

```
uBalance[1]
uBalance[5]
uBalance[10]
uBalance[25]
```

It is not very realistic to evaluate this function with large arguments. In order to compute for instance `uBalance[100]`, we have to compute `uBalance[n]`, for all  $n < 100$  with the recursion.

```
uBalance[50]
uBalance[100]
```

**Example 2:** Saving the results computed in recursive evaluations.

In this example we assume that the function `uBalance[ ]` has been evaluated.

There is a mechanism in Mathematica that implements a more efficient evaluation of recursive expressions as follows. The Mathematica kernel is instructed to store the function results for all intermediate argument values on which the recursive function is evaluated by the following modification of the recursive assignment.

```
Clear[uFastBalance]
uFastBalance[0] = 10000.0;
uFastBalance[n_] := uFastBalance[n] = 1.045 uFastBalance[n - 1]
```

We evaluate `uFastBalance[ ]` on a small argument and inspect the information that Mathematica stores for it. We also inspect `uBalance[ ]` of the previous example.

```
uFastBalance[5]
?uFastBalance
uBalance[5]
?uBalance
```

Remember that Mathematica computes and stores all values with 19-digit accuracy (its default) even though it displays only six digits. We learned already how to control these aspects of numeric computations with the `N[ ]` function.

We evaluate `uFastBalance[ ]` with a larger argument.

```
uFastBalance[100]
```

Even though we did not compute a bank balance for a period of, for instance, 67 years, the balance in the account at that time has already been computed and only needs to be looked up rather than computed again.

```
uFastBalance[67]
```

If we now inspect again the information stored by Mathematica for the function `uFastBalance[ ]`, we get a long list of all values computed so far, together with the recursive definition.

```
?uFastBalance
```

**Example 3:** Non-recursive parameters in a recursive function.

We extend the example of an interest bearing account in another direction. The function `uFastBalance[ ]` will be much more useful if we can change the amount for the starting capital and also the actual interest rate. Neither of these two quantities is part of the recursion. They act as fixed parameters in the problem. We can include them as additional parameters; then only the first parameter, representing the number of years, is recursive while the other two are fixed.

We use the name `uFlexBalance[ ]` for the extended function. In the following definition, we think of the second parameter `s` as the starting capital in dollars. The third parameter `r` is the interest rate in percent.

Observe that now the value of the function at the argument 0 is a rule, rather than an explicit value. Therefore its computation will be implemented with a delayed assignment.

```
Clear[uFlexBalance]
uFlexBalance[0, s_, r_] := s;
uFlexBalance[n_, s_, r_] := uFlexBalance[n, s, r] = (1 + r/100)
uFlexBalance[n - 1, s, r]
```

Sample evaluations of this recursion follow.

```
uFlexBalance[10, 5000, 9.0]
```

```
uFlexBalance[3, 1000, 16.5]
```

```
uFlexBalance[12, 100000, 7.5]
```

At this point it is instructive to inspect the information that Mathematica stores for the function `uFlexBalance[ ]`. Be sure that you understand the meaning of the output of the next command.

```
?uFlexBalance
```

When the two parameters  $r$  and  $s$  are integers or rational numbers, then our definition of `uFlexBalance[ ]` will return an integer or a rational number.

```
uFlexBalance[10, 5000, 9]
```

```
uFlexBalance[3, 1000, 33/2]
```

```
uFlexBalance[12, 100000, 7]
```

The entire recursive calling sequence down to the terminating case  $n = 0$  of `uFlexBalance[ ]` must be computed in each of the three computations since the arguments for the interest are different and therefore the return values are different.

```
?uFlexBalance
```

**Example 4:** Recursion with depth larger than one.

The bank account example uses a recursive call only to the previous argument value. An important example of a recursive function that uses the previous two arguments is the Fibonacci sequence. One possible definition for this sequence is:

$$f_0 = 0, \quad f_1 = 1, \quad \text{and} \quad f_n = f_{n-1} + f_{n-2}, \quad \text{for all } n > 1$$

Observe that since the recursive definition uses the second to last argument in its evaluation we need two starting values in order to terminate the recursion.

```
Clear[uFib]  
uFib[0] = 0;  
uFib[1] = 1;  
uFib[n_] := uFib[n] = uFib[n - 1] + uFib[n - 2]
```

Now we calculate a few Fibonacci numbers.

```
uFib[3]
```

```
uFib[10]
```

```
uFib[16]
```

```
uFib[100]
```

```
TableForm[Table[{n, uFib[n]}, {n, 0, 20}]]
```

**Example 5:** A general Fibonacci sequence.

We extend the Fibonacci sequence from the previous example by allowing arbitrary values for the two starting values:  $f_0 = a$ ,  $f_1 = b$ . This generalization only introduces additional non-recursive parameters for the function. Furthermore, we give these parameters the default values 0 and 1, respectively.

An invocation of the form `uFib[12]` could refer to the function of Example 4 or to the function of three arguments that we define next. There is no harm in such multiple definitions using

the same function name as long as ambiguous references return the same result. In order to avoid any confusion in this example we clear the previous definition of `uFib[ ]`. This action also clears all values computed for `uFib[ ]`.

```
Clear[uFib]
uFib[0, a_ : 0, b_ : 1] := a;
uFib[1, a_ : 0, b_ : 1] := b;
uFib[n_, a_ : 0, b_ : 1] := uFib[n, a, b] = uFib[n - 1, a, b] +
uFib[n - 2, a, b]
```

Some examples follow.

```
uFib[15, 4, -1]
```

```
Table[uFib[n, 4, -1], {n, 0, 12}]
```

We calculate a table of values for the Fibonacci sequence with a variety of starting values. Make sure that the output makes sense to you!

```
Table[{n, uFib[n], uFib[n, 1, 3], uFib[n, 2, 2]}, {n, 0, 8}]
```

Next, we print the table in a tabular format and add a heading line. We control the spacing of the rows and columns in the table with the option `TableSpacing`.

```
TableForm[%, TableSpacing → {1, 3}, TableAlignments → Center,
TableHeadings → {{}, {"n", "uFib[n]", "uFib[n, 1, 3]",
"uFib[n, 2, 2]"}}]
```

**Example 6:** A nonlinear recursion.

In all examples so far, the expressions on the right-hand side of the recursive definition have been linear in terms of the recursive function values. Here we give an example function `uSquareRec[ ]`, where the expression is a quadratic expression. We compute only a few values and observe how quickly this function grows.

```
Clear[uSquareRec]
uSquareRec[1] = 1;
uSquareRec[n_] := uSquareRec[n] = uSquareRec[n - 1]^2 + 3

uSquareRec[7]

uSquareRec[10]
```

The following expression uses the two built-in functions `Length[ ]` and `IntegerDigits[ ]` and computes the number of digits in this large number `uSquareRec[10]`.

```
?IntegerDigits

Length[IntegerDigits[uSquareRec[7]]]

Length[IntegerDigits[uSquareRec[10]]]
```

Since we computed all values `uSquareRec[0]` through `uSquareRec[10]` and saved them, we can easily compute the length of each of these numbers without having to recompute the numbers recursively.

```
Table[Length[IntegerDigits[uSquareRec[k]]], {k, 1, 10}]
```

```
uSquareRec[14]
```

This is a pretty large number.

```
Length[IntegerDigits[uSquareRec[14]]]
```

## *Functions with Several Recursive Arguments*

---

In the previous section we have considered various kinds of functions that were recursive in a single argument. In this section we discuss some examples of functions that are recursive in two arguments.

**Example 7:** Binomial coefficients.

An expression such as  $x + y$  consisting of the sum of two terms is called a binomial. When a binomial is raised to some integer power  $(x + y)^n$  then we can expand the expression and write the result as a polynomial sorted in increasing order of the exponents of  $y$ . Actually, Mathematica has the operator that performs this expansion, `Expand[ ]`, built in.

```
Clear[x, y]
```

```
Expand[(x + y)^3]
```

```
Expand[(x + y)^10]
```

We are interested in finding a recursive formula for the coefficients of the terms in these polynomial expansions. The coefficients are called the binomial coefficients and are denoted by  $C(n, k)$  for the term  $y^k$  in the expansion with exponent  $n$ . We will define these numbers with the function `uPascal[ ]` of two recursive arguments in honor of the Frenchman Blaise Pascal, 1623–1662, who was the first to define these numbers and study their properties.

A recursive definition of the numbers `uPascal[n, k]` proceeds in the following five steps. The first four cases form the basis of the recursion and the last case specifies the recursive step.

1.  $(x + y)^0 = 1x^0y^0$  consists of a single term whose coefficient equals 1.
2. The coefficient of the term  $y^0$  must be 1.
3. The coefficient of the term  $y^n$  must be 1.
4. When  $n < k$ , there is no term in the binomial expansion so we define the value of the binomial coefficient to be zero.
5. For  $0 < k < n$  we can compute the coefficient of the term  $x^{n-k}y^k$  from two pieces: multiply the term  $x^{(n-1)-k}y^k$  by  $x$  and multiply the term  $x^{(n-1)-(k-1)}y^{k-1}$  by  $y$ . Therefore, the two corresponding coefficients need to be added. Note:  $(x + y)^n = (x + y)^{n-1}(x + y)$ .

These five properties give rise to the following recursive definition.

```
Clear[uPascal]
uPascal[0, 0] = 1;
uPascal[n_, 0] = 1;
uPascal[n_, n_] = 1;
uPascal[n_, k_] := 0;/; n < k
uPascal[n_, k_] := uPascal[n, k] = uPascal[n - 1, k] +
  uPascal[n - 1, k - 1]
```

Observe that the first three terminating conditions are implemented with immediate assignments. However, the fourth terminating case must use a delayed assignment since we need to evaluate the comparison  $n < k$  for the specific values of  $n$  and  $k$  at the time when we compute a value for  $\text{uPascal}[n, k]$ .

We compute some binomial coefficients for small  $n$  and  $k$ . Are we getting correct results? Check!

```
uPascal[2, 1]
```

```
uPascal[2, 2]
```

```
uPascal[2, 3]
```

```
uPascal[3, 2]
```

```
uPascal[4, 2]
```

We calculate lists and tables of binomial coefficients, once with the recursive function `uPascal[ ]` and once with the built-in operator `Binomial[ ]`, which computes the same values as our recursively defined function `uPascal[ ]`.

```
TableForm[Table[uPascal[10, k], {k, 0, 10}]]
```

```
TableForm[Table[Binomial[10, k], {k, 0, 10}]]
```

```
TableForm[Table[uPascal[n, k], {n, 0, 10}, {k, 0, 10}]]
```

The triangle we see as the result of the last computation is Pascal's triangle. Many interesting properties of the binomial coefficients are hidden in this triangle, and Blaise Pascal already observed and proved many of these properties. One well-known property has to do with the sum of the entries in each row.

```
Sum[uPascal[3, k], {k, 0, 3}]
```

```
Sum[uPascal[6, k], {k, 0, 6}]
```

We collect the values of the row sums in a table. What is the sum of the numbers in row  $n$ ?

```
TableForm[Table[{n, Sum[uPascal[n, k], {k, 0, n}]}, {n, 0, 10}]]
```



**Example 8:** Stirling numbers of the second kind.

A partition of a set  $S$  is a collection  $C$  of subsets of the set  $S$  such that any two sets in  $C$  are disjoint and such that the union of all sets in  $C$  covers the set  $S$ . When the set  $S$  itself is non-empty then we usually assume that all subsets in the collection  $C$  are also non-empty.

When the set  $S$  contains  $n$  elements, we define a  $k$ -partition of  $S$  to be a partition of the set  $S$  into  $k$  non-empty subsets. Thus, a  $k$ -partition specifies the number of subsets in the partition, but not the size of the individual subsets. For example, if  $S = \{1, 2, 3, 4, 5, 6, 7\}$  then  $C = \{\{2, 5\}, \{3, 4, 6, 7\}, \{1\}\}$  is a 3-partition of the set  $S$ .

We will denote the function that computes the number of  $k$ -partitions of an  $n$ -element set by `uStirling[n, k]`. These numbers are called Stirling Numbers of the Second Kind after the Englishman James Stirling, 1692–1770.

A recursive definition of the numbers `uStirling[n, k]` proceeds in the following four steps. The first three specify the basis values and the last one determines the recursive step.

1. There is a single 1-partition of a 1-element set. It consists of a single 1-element set.
2. There is a single  $n$ -partition of an  $n$ -element set. It consists of  $n$  1-element sets.
3. When  $n < k$ , there is no  $k$ -partition since every set in the partition must have at least one element. In this case we define the function `uStirling[n, k]` to have value zero.
4. The recursive step: There are two ways in which  $k$ -partitions for an  $n$ -element set can be generated from partitions for an  $(n-1)$ -element set.
  - (a) We can add an element to one set in a  $k$ -partition on an  $(n-1)$ -element set. This can be done in  $k$  ways since there are  $k$  subsets to choose from.
  - (b) We can add the singleton set consisting of the  $n^{\text{th}}$  element to a  $(k-1)$ -partition on an  $(n-1)$ -element set to make a new  $k$ -partition. Therefore there is one  $k$ -partition on the  $n$ -element set for every  $(k-1)$ -partition on the  $(n-1)$ -element set.

This gives rise to the following recursive definition.

```
Clear[uStirling]
uStirling[n_, 1] = 1;
uStirling[n_, n_] = 1;
uStirling[n_, k_] := 0 /; n < k
uStirling[n_, k_] := uStirling[n, k] = k uStirling[n - 1, k] +
  uStirling[n - 1, k - 1]
```

We compute some Stirling numbers for small  $n$  and  $k$ . Are we getting correct results? Check!

```
uStirling[2, 1]
uStirling[2, 2]
uStirling[2, 3]
uStirling[3, 2]
uStirling[4, 2]
```

Calculating tables of Stirling numbers:

```
TableForm[Table[uStirling[10, k], {k, 1, 10}]]
```

```
TableForm[Table[uStirling[n, k], {n, 1, 8}, {k, 1, 8}]]
```

Note that `StirlingS2[ ]` is a built-in operator in Mathematica that computes the same numbers as our recursively defined function `uStirling[ ]`.

```
TableForm[Table[StirlingS2[n, k], {n, 1, 8}, {k, 1, 8}]]
```

## *Limitations to Recursive Computations*

---

We controlled the recursive computations in the previous examples in this notebook by using relatively small arguments. It is easy to exhaust the default set in Mathematica or the resources of a computer by invoking a recursive computation without estimating the cost of the computation. In this section we will demonstrate these dangers inherent in recursive computations. We will also show an example of a recursive schema that gives rise to a function of extraordinary growth.

**Example 9:** Manipulating the depth of recursive evaluations.

Every recursive function that we have defined in the preceding examples, we evaluated at small argument values only. There is a reason for this restraint. Imagine that you want to calculate the binomial coefficient `uPascal[2000, 457]`. Since the recursive step decreases the first argument as well as the second argument by one, Mathematica needs to store `uPascal[1999, 457]`, `uPascal[2000, 456]`, `uPascal[1998, 457]`, `uPascal[1999, 456]`, `uPascal[2000, 455]`, and so on, until the basis expressions `uPascal[0,0]`, `uPascal[n,0]`, and `uPascal[n,n]` are encountered. At that point values can be assigned. Mathematica refers to the number of computations required to get to a basis case of the recursive definition as the recursive depth of the computation. Once those assignments have been made, the intermediate recursive expressions can be evaluated, and eventually a value can be returned for the expression that we wanted to compute in the first place, `uPascal[2000, 457]`.

Since this computational process can require a large amount of memory, Mathematica limits the depth of recursion. The built-in global variable `$RecursionLimit` serves this purpose. All symbols that start with a dollar sign, `$`, are global variables and have preassigned values that we can change whenever necessary.

```
$RecursionLimit
```

```
?$RecursionLimit
```

We can find the names of all system variables; there is a large number.

```
?$*
```

For purposes of demonstrating the effects of `$RecursionLimit`, we define the doubling function  $n \rightarrow 2n$  recursively.

```
Clear[uDouble]
uDouble[0] = 0;
uDouble[n_] := uDouble[n] = uDouble[n - 1] + 2

uDouble[256]
```

During the evaluation of the last expression, Mathematica unravels the recursive definition of the function `uDouble[ ]` from 256 down. It keeps the unevaluated intermediate expressions and stops after 255 steps. Unfortunately, that 255<sup>th</sup> step only gets down to argument 1, but there is no explicit value for it since the basis value is 0.

We inspect this problem with the next command. There is a long list of unevaluated expressions.

```
?uDouble
```

All these intermediate computations were saved as symbolic unevaluated expressions rather than numbers. Therefore we cannot compute the function `uDouble[ ]` on a smaller argument.

```
uDouble[175]
```

At this point in the Mathematica session, we can proceed with computations using `uDouble[ ]` only by clearing the definition of `uDouble[ ]` and starting over with the computation. Evaluate the previous definition of the input cell for `uDouble[ ]` and then compute the sequence of invocations with the following smaller arguments.

```
uDouble[250]
```

```
uDouble[500]
```

An alternate mechanism to control the recursive computation is to increase the value of the systems variable `$RecursionLimit` in order to make it large enough so that our intended computations can succeed. We can set it to any positive number and also to infinity. You should exercise caution with the infinity value since it allows computations of unbounded recursive depth, also called infinite recursive descent. The only way to recover from an infinite recursive descent is to abort the calculation or, if that fails, to quit the Mathematica kernel.

We start the computations over again by setting the recursion depth to 1000 and by clearing `uDouble[ ]`.

```
$RecursionLimit = 1000

Clear[uDouble]
uDouble[0] = 0;
uDouble[n_] := uDouble[n] = uDouble[n - 1] + 2
```

Now again only the recursive definition of `uDouble[ ]` is stored by Mathematica. We compute `uDouble[ ]` for a large argument.

```
?uDouble
```

```
uDouble[800]
```

```
uDouble[1798]
```

You should test that using an argument that is bigger by at least 999 than the largest for which `uDouble[ ]` was computed previously, again produces failure.

```
uDouble[2797]
```

One way to avoid the problems of large recursive depth is to compute the recursive function in steps; that is, instead of invoking the function once, we compute a table of values. In order to demonstrate this method we reset the value of `$RecursionLimit` to its default, clear all previously saved values from `uDouble[ ]`, and evaluate `uDouble[ ]` starting with the argument 100 and computing in steps of 100. For this method to work, the recursive definition, which saves all intermediate values of the recursive function, must be used.

```
$RecursionLimit = 256
```

```
Clear[uDouble]
```

```
uDouble[0] = 0;
```

```
uDouble[n_] := uDouble[n] = uDouble[n - 1] + 2
```

```
Table[uDouble[k], {k, 100, 2000, 100}]
```

Since all intermediate values were saved we can obtain any value of `uDouble[k]`, for  $0 \leq k \leq 2000$ , with a look-up rather than a computation.

```
uDouble[683]
```

```
uDouble[1829]
```

**Example 10:** The recursive definition of the Ackermann function.

We conclude this notebook with a function that is defined by a recursive scheme much more general than the ones we have encountered so far. In this definition we use a nested recursive invocation in the recursive definition of the function. As an example we use a function defined by the German logician Wilhelm Ackermann, who died in 1963. The Ackermann function is important since it can be shown that it is impossible to define it with those simpler recursive schemes, called primitive recursive schemes, that we have used in the previous examples in this notebook. It also is an example of a function that grows extremely fast; in fact, it grows faster than any primitive recursive function.

The Ackermann function has two non-negative integer arguments. In order to minimize the amount of typing and for the clarity of the definition we deviate from our naming convention for functions and call the Ackermann function `A[ ]`.

```
Clear[A]
```

```
A[0, 0] = 1;
```

```
A[0, k_] := A[0, k] = k + 1
```

```
A[n_, 0] := A[n, 0] = A[n - 1, 1]
```

```
A[n_, k_] := A[n, k] = A[n - 1, A[n, k - 1]]
```

The nesting in the second argument of the recursive definition of the function makes this an unusual recursive scheme. Observe also that the recursion depends on both arguments  $n$  and  $k$ .

We compute a few values for  $n = 0, 1$  and for reasonably small values of  $k$ . Are the numbers what you expected? Check by hand calculation.

```
A[0, 5]
A[0, 10]

A[1, 1]
A[1, 2]

A[1, 10]
```

We compute a few values for  $n = 2$  and for reasonably small values of  $k$ . Make sure you understand the output.

```
A[2, 2]
A[2, 3]
A[2, 4]
A[2, 10]

TableForm[Table[{k, A[2, k]}, {k, 0, 12}]]
```

Look now at the values that Mathematica stores for  $A[ ]$  after the previous calculations.

```
?A
```

You will see that  $A[0,0] \dots A[0,26]$ ,  $A[1,0] \dots A[1,25]$ , and  $A[2,0] \dots A[2,12]$  have all been computed through the recursive definition. That is, in order to compute the value of the Ackermann function at level  $n = 2$  and  $k = 12$ , we must compute 66 values of the function.

**Example 11:** Evaluating the Ackermann function at levels 3 and 4.

To avoid any failure of computation, re-evaluate the input cell containing the definition of the Ackermann function and then set the value of the global variable `$RecursionLimit` to infinity.

```
Clear[A]
A[0, 0] = 1;
A[0, k_] := A[0, k] = k + 1
A[n_, 0] := A[n, 0] = A[n - 1, 1]
A[n_, k_] := A[n, k] = A[n - 1, A[n, k - 1]]

$RecursionLimit = Infinity
```

We compute a few values of  $A[ ]$  for  $n = 3$  and for small values of  $k$  in order to understand the amount of time required for each computation. Be sure to evaluate them in the order of increasing second arguments, otherwise the values returned by `Timing[ ]` may not make sense because Mathematica may perform a table look-up rather than a recursive computation.

```
A[3, 1]
A[3, 2]
```

**A[3, 3]****A[3, 4]****Timing[A[3, 5]]****A[3, 6]****Timing[A[3, 7]]**

If Mathematica on your computer evaluates the last command in a fraction of a second, re-evaluate the input cell with a larger second argument.

When we ask for the same value of the Ackermann function again, only a look-up is required. This consumes a minimal amount of time.

**Timing[A[3, 7]]**

We can compute the first value at level four,  $A[4, 0]$ .

**A[4, 0]**

We do not recommend any further computations at level four. Computing  $A[4, 1]$  already requires the computation of  $A[3, 13]$ , since  $A[4, 1] = A[3, A[4, 0]] = A[3, A[3, 1]] = A[3, 13]$ . The evaluation of this latter expression may exceed the resources of your computer and crash it!

It can be shown that the function  $A[4, k]$  grows faster than the exponential function; sometimes this function is referred to as super-exponentiation.

## *Exercises*

---

This set of exercises contains problems from a large variety of topics: Calculus, Combinatorics, Business, Linear Algebra, Computer Science, and Physics. Recursive definition and evaluation of functions are useful computational tools. They represent an alternative to iterative computations that usually are implemented with loops. However, they sometimes also require you to think about a problem in a particular way since you must develop the recursive step of a function  $f$  by expressing  $f$  in terms of itself.

### **EXERCISE 1:** Methods of compounding interest payments.

In Examples 1 through 3 we defined the functions `uBalance[ ]`, `uFastBalance[ ]`, and `uFlexBalance[ ]`, which compute interest for an account on a yearly basis. We extend those examples in several ways in this exercise. Choose one of the three functions for the following questions.

- Instead of using the numeric value computed with many digits, round the balance amount to full pennies. Be sure to perform this rounding for each interest payment.
- Investigate the effect of truncating rather than rounding to the nearest penny.
- Many interest bearing accounts compound the interest either monthly or daily. Change the computations in the function of your choice from parts (a) or (b) to these interest calculation methods. Use 12 monthly periods of equal length and a year of 365 days for purposes of your interest calculations.

**EXERCISE 2:** Harmonic sequences.

The harmonic sequence is the sequence  $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots, \frac{1}{n}, \dots$  of reciprocals of the natural numbers.

The alternating harmonic sequence is the sequence  $\frac{1}{1}, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{4}, \dots, \frac{(-1)^{n+1}}{n}, \dots$

(a) A partial sum of the first  $k$  terms of the harmonic sequence is the following expression:  $s_k = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k}$ . Write a recursive function for the partial sums  $s_k$ .

(b) Compute a table of values for  $s_k$  and plot the values.

(c) Interpret the graph that you created in part (b). Present arguments on whether the sequence of partial sums  $s_k$  converges or diverges.

(d) Make a table of values for the expression  $f_k = s_k - \ln k$  and draw its graph. Present a formal argument that  $f_k$  converges to a constant. It is called Euler's constant gamma and its name in Mathematica is EulerGamma.

(e) A partial sum of the first  $k$  terms of the alternating harmonic sequence is the following expression:  $t_k = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{(-1)^{(k+1)}}{k}$ . Write a recursive function for the partial sums  $t_k$ .

(f) Compute a table of values for  $t_k$  and plot the values.

(g) Interpret the graph that you created in part (f). Present arguments on whether the sequence of partial sums  $t_k$  converges or diverges.

**EXERCISE 3:** The Fibonacci numbers and computational efficiency.

In Example 2 we introduced the mechanism with which Mathematica saves all intermediate values in recursive evaluations and in Example 4 we defined the Fibonacci numbers.

(a) Define a recursive function `uFib1[ ]` that implements the Fibonacci numbers and that does not save the intermediate values.

(b) Define a recursive function `uFib2[ ]` that implements the Fibonacci numbers and that saves the intermediate values.

(c) Use an appropriate value for `$RecursionLimit` and the `Timing[ ]` function to compare the computational efficiency of the two definitions when computing a table of Fibonacci numbers.

**EXERCISE 4:** The effects of the basis case and of the recursive expression.

In this exercise we use a recursive expression that is a sum of two terms, similar to the Fibonacci sequence of Example 4. The expressions will use nested invocations similar to the Ackermann function. However, the terms in the recursive expression are chosen in such a way that the resulting function value is close to the argument value in size.

(a) Given is the following recursive definition:

$f(1) = 1, f(2) = 2$ , and  $f(n) = f(f(n-1)) + f(n - f(n-1))$ , for  $n \geq 3$ .

Implement  $f$  as the recursive function `uDiagonal[ ]`. Verify experimentally that `uDiagonal[ ]` is the identity function. Give a formal proof by induction that  $f(k) = k$ , for all  $k \geq 1$ .

(b) We change the definition of part (a) by changing the value of the basis case  $n = 2$ :

$g(1) = 1$ ,  $g(2) = 1$ , and  $g(n) = g(g(n-1)) + g(n - g(n-1))$ , for  $n \geq 3$ .

Implement  $g$  as the recursive function `uConway[ ]`—it was first proposed by J. H. Conway—and compute a table of values for it. Describe any number patterns that you can discover.

(c) Define the function  $c(n) = 2g(n) - n$ . Compute a table of values for it and describe any number patterns that you can discover. You can find an extensive discussion of the sequences  $g(n)$  and  $c(n)$  in the article by Colin L. Mallows, *Conway's Challenge Sequence*, The American Mathematical Monthly, Volume 98, Number 1, January 1991, p. 5–20.

(d) We change the definition of part (b) by changing the recursive expression:

$q(1) = 1$ ,  $q(2) = 1$ , and  $q(n) = q(n - q(n-1)) + q(n - q(n-2))$ , for  $n \geq 3$ .

Implement  $q$  as the recursive function `uHofstadter[ ]`—it was proposed by Douglas R. Hofstadter as a modification of Conway's challenge sequence—and compute a table of values for it. Describe any number patterns that you can discover. You can find some discussions of this sequence in the book by Douglas R. Hofstadter, *Gödel, Escher, Bach: an Eternal Golden Braid*, Basic Books, Inc. New York, 1979, pages 137–138. This sequence appears to be quite erratic and is, in particular, nonmonotonic.

### EXERCISE 5: Derangements of objects.

Consider the set of natural numbers  $\{1, 2, 3, \dots, n\}$  with each number in its natural position. A derangement of the  $n$  numbers is any arrangement in which none of the  $n$  numbers is in its natural position. For example,  $\{3, 1, 2, 5, 4\}$  is a derangement of a 5-element set. A recursive definition for the number of derangements of an  $n$ -element set is  $d_1 = 0$ ,  $d_2 = 1$  and  $d_n = (n-1)(d_{n-1} + d_{n-2})$ , for  $n > 2$ .

(a) Define the function `uDerangement[ ]` that implements the previous recursive definition, create a table of its values, and check by hand that for small  $n$  the values count the number of derangements.

(b) Justify in your own words, for example with an inductive proof, the recursive definition given above for the derangements. Only citing examples does not constitute a justification.

(c) Verify experimentally that the identity  $d_n - n d_{n-1} = (-1)^n$  holds for all  $n > 1$ .

(d) Give a recursive definition for the function  $g_n$ ,  $n \geq 0$ , that computes the partial sums of the alternating reciprocals of the first  $n$  factorials:  $g_n = \frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \frac{1}{4!} - \dots + \frac{(-1)^n}{n!}$ .

(e) Experimentally verify that  $\frac{d_n}{n!} = g_n$ , for all  $n \geq 0$ .

(f) Graph the values for the function  $g_n$  and find an approximate value for the limit of the sequence  $g_n$  as  $n$  approaches infinity.

(g) Use your result of part (f) and your knowledge of power series to compute the precise value of this limit. (Hint: MacLaurin series for  $e^x$ .)

### EXERCISE 6: Determinants of some tridiagonal matrices.

In a square matrix  $M = (m_{ij})$  the index pairs  $(k, k)$  describe the diagonal, the index pairs  $(k+1, k)$  describe the subdiagonal, and the index pairs  $(k, k+1)$  describe the superdiagonal of the matrix. A square matrix  $M$  is called tridiagonal if all entries in  $M$  not on the subdiagonal, not on the



diagonal, and not on the superdiagonal are zero. Equivalently we can say that a square matrix  $M$  is tridiagonal if  $m_{ij} = 0$  holds for all  $i$  and  $j$  so that  $|i - j| > 1$ .

- (a) Suppose that all entries on the three diagonals have the constant value  $x$ , that is,  $m_{ij} = x$  holds when  $|i - j| \leq 1$ . Find a recursive formula for the determinant of  $M$ .
- (b) Compute the determinants for all such tridiagonal matrices up to size 10.
- (c) Develop a closed form—that is, a non-recursive expression—for the determinant from your answer in part (b).

**EXERCISE 7:** A generalization of Exercise 6.

Suppose that the entries of matrix  $A$  on each of the three diagonals are constant, but may be different on different diagonals; that is,  $a_{ij} = a$  for  $i - j = 1$ ,  $a_{ij} = b$  for  $i - j = 0$  and  $a_{ij} = c$  for  $i - j = -1$  and  $a_{ij} = 0$  otherwise.

- (a) Find a recursive formula for the determinant of  $A$ .
- (b) Compute the determinants for all such tridiagonal matrices up to size 10.
- (c) Develop a closed form for the determinant from your answer in part (b).

**EXERCISE 8:** Computations based on the Stirling numbers of the second kind.

In Example 8 we computed the first eight rows of the triangle of the Stirling numbers of the second kind.

- (a) Write a function `uChoices[ ]` of the two arguments  $n$  and  $k$  that computes the sum of the first  $k$  entries in the  $n^{\text{th}}$  row of that triangle.
- (b) One can reinterpret the meaning of the function `uChoices[ ]` as follows: `uChoices[n,k]` computes the number of ways in which  $n$  objects can be placed into  $k$  containers where some containers may be left empty. Give first some specific examples for small values of  $n$  and  $k$ , and then present a general argument.

**EXERCISE 9:** The Stirling numbers of the first kind.

Consider the products of the form  $x(x-1)(x-2)\dots(x-n+1)$  for natural numbers  $n > 0$ . When we expand these products we obtain a polynomial of degree  $n$  in the variable  $x$ :

$$p(n, x) = s(n, 0)x^0 + s(n, 1)x^1 + s(n, 2)x^2 + \dots + s(n, n)x^n$$

The coefficients  $s(n, k)$  are called the Stirling numbers of the first kind.

From the definition of the polynomial  $p(n, x)$  and the coefficients  $s(n, k)$ , the following assignments are apparent for all  $n \geq 0$ :

$$s(n, n) = 1; s(n, 0) = 0; s(n, k) = 0 \text{ for } k < 0; s(n, k) = 0 \text{ for } n < k.$$

- (a) Observe that the identity  $p(n+1, x) = p(n, x) * (x - n)$  holds for all  $x$  and all  $n > 0$ . Use it to derive a recursive formula for the Stirling numbers of the first kind.
- (b) Compute a table of numbers for the Stirling numbers of the first kind.

**EXERCISE 10:** The growth of the Ackermann function.

The basis condition of the Ackermann function is given by the expression  $A[0, k] := k + 1$ , for all  $k \geq 0$ . It is an expression in a single variable that we can write as a function  $f_0(k) = k + 1$ . Observe that  $f_0$  is the successor function for the natural numbers.

- (a) Put  $f_1(k) = A[1, k]$  and  $f_2(k) = A[2, k]$ , for all  $k \geq 0$ . Compute tables of values for the two functions  $f_1$  and  $f_2$ . Describe in words what these functions  $f_1$  and  $f_2$  compute and find a closed form expression for each function.
- (b) Prove your claims of part (a) by induction.
- (c) Do the same as in part (a) for the function  $f_3(k) = A[3, k]$ , for all  $k \geq 0$ .
- (d) Prove your claims of part (c) by induction.
- (e) Use the closed form expression that you found for  $f_3$  in part (c) and use the recursive formula for the Ackermann function to find a closed form expression for the function  $f_4(k) = A[4, k]$ , for all  $k \geq 0$ . Prove your claims by induction.

**EXERCISE 11:** The recursive evaluation of the Ackermann function.

In this exercise you explore the sequence of recursive invocations that is required for the evaluation of the Ackermann function. The main tool is the built-in `Trace[ ]` function. A copy of the recursive definition from Example 11 follows. In questions (a) through (d) make sure that you reevaluate this input cell before you start your computations. Then you will have a clean slate.

```
$RecursionLimit = Infinity;
Clear[A]
A[0, 0] = 1;
A[0, k_] := A[0, k] = k + 1
A[n_, 0] := A[n, 0] = A[n - 1, 1]
A[n_, k_] := A[n, k] = A[n - 1, A[n, k - 1]]
```

- (a) Trace `A[0,10]` and explain the output.
- (b) Trace `A[1,3]` and explain the output. Use the recursive definition of the Ackermann function to compute `A[1,3]` by hand.
- (c) Trace `A[2,1]` and explain the output. Without clearing `A[ ]` trace the evaluations of `A[0,4]`, `A[1,3]`, `A[2,1]`, and `A[2,2]`. Explain the output for each trace.
- (d) Use the `Trace[ ]` function to find the largest values  $k_0$ ,  $k_1$ , and  $k_2$  for which `A[0,  $k_0$ ]`, `A[1,  $k_1$ ]`, and `A[2,  $k_2$ ]` must be computed in the evaluation of `A[3,6]`.

**EXERCISE 12:** A recursive description of radioactive decay.

The decay of radioactive material can be modeled with a recursive function that is similar in its structure to the function `uBalance[ ]` of Example 1. It is customary to describe the process of decay with the half-life  $h$  of the radioactive material—that is, the time it takes for half of the material to decay. From  $h$  we can compute the rate of decay per time unit, denoted by  $r$ .

If  $p(0)$  is the initial amount of radioactive material at time  $t = 0$  and  $p(1)$  the amount at  $t = 1$ , then

$$\begin{aligned} p(1) &= p(0) - r p(0) = (1 - r)p(0) \\ p(2) &= p(1) - r p(1) = (1 - r)p(1) = (1 - r)^2 p(0), \text{ and at } t = h \\ p(h) &= \frac{1}{2} p(0) = (1 - r)^h p(0), \text{ or } \frac{1}{2} = (1 - r)^h, \text{ that is, } 1 - r = 2^{-\frac{1}{h}} \end{aligned}$$

The constant  $c = 1 - r$  allows us to describe the radioactive decay recursively as  $p(t) = c p(t - 1)$ .

(a) The Strontium isotope with atomic weight 95 of Strontium 38, which contains 38 protons and 57 neutrons, decays under beta-decay with a half-life of 25 seconds. Compute the constant  $c$  and use its value to set up a recursive function `uStrontium[ ]` that computes the amount of Strontium at time  $t$ . Use  $p(0) = 100$  gr as the initial amount of radioactive material.

(b) Compute the amount of Strontium left after 10, 15, 20, ..., 100 seconds and display the information in a table.

(c) When you try to find the amount of Strontium left after five or ten minutes your computation may fail to produce an answer because the value of `$RecursionLimit` was exceeded during the computation. Reset the system variable `$RecursionLimit` to various values and repeat your computations of `uStrontium[ ]` with these values of `$RecursionLimit`.

(d) When the argument value in a recursive function is too large for Mathematica to complete the evaluation, then the expression that Mathematica returns contains the function `Hold[ ]`, for example an expression of the form `Hold[uStrontium[Hold[47 - 1] - 1]]`. Use the online help feature and the Mathematica book to explore the role of the function `Hold[ ]`.

(e) How long does it take until 99% of the original amount of Strontium have decayed? State your answer in whole seconds.

**EXERCISE 13:** Radioactive decay.

In Exercise 12 we considered the radioactive isotope of Strontium 38 with atomic weight 95. The half-life of radioactive materials ranges from fractions of microseconds to billions of years. Most college physics textbooks list a table of elements, their isotopes, and the half-life for the radioactive ones.

We took the data from Frank J. Blatt, *Modern Physics*, McGraw-Hill, New York, 1992, and listed several radioactive elements together with their atomic number, their atomic weight, and their half-life.

	Element	Atomic Number	Atomic Weight	Half Life
I	Iodine	53	118	14.3 minutes
Cs	Cesium	55	134	2.065 years
U	Uranium	92	238	4.46 billion years
U	Uranium	92	239	23.54 minutes
Pu	Putonium	94	239	24110 years

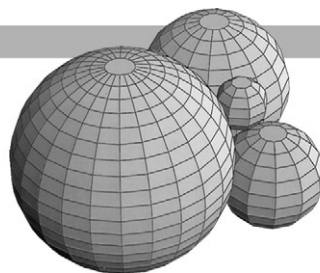
(a) For each element in the list define a recursive function, just as in the previous exercise. Implement each function in terms of time units appropriate for the half-life of the material.

Document this aspect of each function clearly! In all cases use 100 gr as the starting amount of radioactive material. Compute a table of the amounts of radioactive materials left as a function of time.

(b) How long does it take until 99% of the original amount of the radioactive material have decayed? State your answer as a whole number, that is, in whole minutes, whole years, whole multiples of billions of years, and so on.

This Page Intentionally Left Blank

# *Substitution Rules and Optional Arguments*



## *Introduction*

Throughout the notebooks we have used two kinds of assignments, immediate and delayed, and occasionally, substitution rules. The basic difference between an assignment and a rule is that an assignment changes the value of a symbol permanently, whereas a rule computes a value without changing the symbol; we can use a rule to substitute a value temporarily for a symbol.

We have encountered this temporary nature of substitution with the option `AspectRatio` → `Automatic` for the `Plot[ ]` function. We are not assigning a value to the symbol `AspectRatio`; instead, we use the value `Automatic` in place of it in the current plot. In Example 16 of the notebook `FuncDef.nb`, “Functions,” we applied the substitution rules of graphics options to optional arguments in a user-defined function.

Here is another common situation. When we solve for the unknowns in an equation or set of equations, we want to compute values that satisfy the equations and that we then associate with the unknowns. We frequently want to leave the unknowns as symbolic objects because we need to be able to refer to them symbolically in other expressions. For example, in extremal value problems we take the root of a derivative of a function and substitute its value into the second derivative of that function.

In this notebook we demonstrate how:

- The substitution mechanism affects expressions
- Substitution can be used interactively in computations
- Substitution applies to optional arguments in built-in functions
- Optional arguments are implemented for user-defined functions

## *Substitution Rules and the Replacement Mechanism*

In this section we show in detail how rules are used for substitutions in expressions. We give examples of substitutions with a single rule, a list of rules, a list of lists of rules, and repeated applications of the same rule.

**?ReplaceAll**

The symbol /. is an abbreviation for ReplaceAll[ ].

**?/.**

Throughout this section we use the polynomial expression  $x + y^2$  of the two variables  $x$  and  $y$  to demonstrate different invocations of the replacement function ReplaceAll[ ].

We also use the symbols  $x$ ,  $y$ , and  $z$  in the substitution rules. We clear the variables here. Nowhere in this section will we assign values to these symbols.

```
Clear[x, y, z]
```

**Example 1:** Substitution of numeric values into an expression.

First we perform substitutions with a single rule. There are three different ways of expressing a single substitution.

```
(x + y^2) /. x -> 3
```

```
(x + y^2) /. {x -> 3}
```

```
ReplaceAll[x + y^2, x -> 3]
```

When the symbol on the left-hand side of the rule does not appear in the expression at all, then the substitution behaves like the identity function.

```
(x + y^2) /. {z -> 5}
```

Note that although the symbols  $x$ ,  $y$ , and  $z$  are not affected by this substitution mechanism, they still are symbols.

```
x
```

Now we perform substitutions with a list of rules.

```
(x + y^2) /. {x -> 3, y -> 2}
```

```
(x + y^2) /. {y -> 2, x -> 3}
```

```
(x + y^2) /. {x -> 3, z -> 5, y -> 2}
```

All rules in the list are applied to the original expression. We can think of the replacement as a simultaneous substitution of the symbols mentioned on the left-hand side of the rules in the list with the values on the right-hand side of the rules in the list.

**Example 2:** Nested lists of rules.

Now we use a list of lists of rules to substitute into the expression. First, each sublist contains a single rule.

```
(x + y^2) /. {{x -> 3}, {y -> 2}}
```

```
(x + y^2) /. {{y -> 2}, {x -> 3}}
```

We get a list of expressions in which each element is the result of applying each list of rules to the expression.

Next, we evaluate all substitutions from Example 1 in a single statement and obtain a list of expressions.

```
(x + y^2) /. {{x -> 3}, {y -> 2}, {z -> 5}, {y -> 2, x -> 3}}
```

**Example 3:** Substitution of symbolic values into an expression.

We perform a replacement with a single list of rules that contain symbols or symbolic expressions.

```
Clear[a, b]
```

```
(x + y^2) /. {x -> a}
```

```
(x + y^2) /. {y -> b}
```

```
(x + y^2) /. {x -> (a - b), y -> (a + b)}
```

The rule  $x \rightarrow a - b$  has the same effect as  $x \rightarrow (a - b)$  since the algebraic operations have higher precedence than the rule operation  $\rightarrow$ .

```
(x + y^2) /. {x -> a - b, y -> a + b}
```

Now, we perform a replacement with nested lists of rules that contain functions and symbolic expressions.

```
(x + y^2) /. {{x -> Sin[2t], y -> Cos[t]}, {x -> Log[w],  
y -> 1 / w^3}}
```

```
(x + y^2) /. {{x -> (a - b), y -> (a + b)}, {x -> (a - b)^3,  
y -> (a + b)^2}}
```

**Example 4:** Successive replacement of symbolic values in an expression.

In Examples 1, 2, and 3 we have demonstrated the simultaneous application of rules to an expression. We can also apply rules sequentially, substituting one into the result of the previous substitution.

However, the order of replacement becomes important in the sequential replacement when we replace one of the variables in an expression by another variable that also occurs in the expression. To see the details of these substitutions we employ the `Trace[ ]` function.

```
(x + y^2) /. x -> 5y^2 /. y -> x^2
```

```
?Trace
```

In the following trace you can see how the second substitution  $y \rightarrow x^2$  is applied to the result of the first substitution  $x \rightarrow 5y^2$  into the expression  $x + y^2$ .

```
TableForm[Trace[(x + y^2) /. x -> 5y^2 /. y -> x^2],  
TableDepth -> 1]
```



Now we switch the order of the sequential substitutions.

```
(x + y^2) /. y -> x^2 /. x -> 5y^2
```

```
TableForm[Trace[(x + y^2) /. y -> x^2 /. x -> 5y^2],
  TableDepth -> 1]
```

In contrast, when a single list of rules is given, then all rules in the list are applied only to the original expression, and a list of expressions is returned, not a single expression as shown previously.

```
(x + y^2) /. {y -> x^2, x -> 5y^2}
```

```
(x + y^2) /. {x -> 5y^2, y -> x^2}
```

## *Substitution Rules and Interactive Computations*

---

In this section we use an extremal value problem from first semester calculus to demonstrate the interactive use of substitution rules.

**Example 5:** Finding extremal values for a function of a single variable.

The following four computational steps are usually performed to find the local minima and maxima of a differentiable function.

1. Graph the function and its first derivative with `Plot[ ]`.
2. Find the roots of the derivative of the function with `Solve[ ]`, `NSolve[ ]`, or `FindRoot[ ]`.
3. Substitute the roots into the second derivative with `ReplaceAll[ ]` to determine the kind of extremum.
4. Substitute the roots into the function with `ReplaceAll[ ]` to find the value of the extremum.

Once we have computed the roots, given as sets of rules, we only need to apply the rules to substitute the values of the roots into the appropriate function.

We demonstrate this process with a classic example from first semester calculus. A rectangle is to be fitted into a semicircle of radius 10. Find the dimensions of the largest possible inscribed rectangle.

We assume that the origin is the center of the circle and denote half of the horizontal side of the inscribed rectangle with  $s$ . Then the height of the inscribed rectangle is  $\sqrt{100 - s^2}$ . Here is a plot for the problem with a  $12 \times 16$  inscribed rectangle.

```
Show[Graphics[{RGBColor[0, 0, 1], Circle[{0, 0}, 10, {0, Pi}],
  RGBColor[1, 0, 0],
  Line[{{-6, 0}, {6, 0}, {6, 8}, {-6, 8}, {-6, 0}}],
  RGBColor[0, 1, 0],
  Line[{{0, 0}, {6, 8}}]}], AspectRatio -> Automatic, Axes -> True];
```

We define the function `uHeight[ ]` for the height and `uArea[ ]` for the area of the inscribed rectangle.

```
Clear[uHeight,uArea]
uHeight[s_] := Sqrt[100 - s^2]
uArea[s_] := 2 s uHeight[s]
```

We plot `uArea[ ]` in red and its derivative `uArea'[ ]` in blue to see the approximate locations of their roots. Since the derivative is not defined at  $s = 10$ , we plot the two functions in the interval  $[0, 9]$ . Then we solve for the precise value of the roots of the derivative.

```
Plot[{uArea[s], uArea'[s]}, {s, 0, 9},
PlotStyle -> {RGBColor[1, 0, 0], RGBColor[0, 0, 1]};

Clear[sides]
sides = Solve[uArea'[s] == 0, s]
N[sides]
```

We test the value of the second derivative of `uArea[ ]` and confirm the location of the maximum.

```
uArea''[s] /. sides
```

Finally, we calculate the precise as well as the approximate values for the dimensions of the largest rectangle.

```
uHeight[s] /. sides
N[%]

uArea[s] /. sides
N[%]
```

Observe that we could have selected the single positive root for the derivative and done the subsequent calculations with that one root rather than with the two roots of the equation.

```
uArea''[s] /. sides[[2]]

uHeight[s] /. sides[[2]]
N[%]

uArea[s] /. sides[[2]]
```

## *Options for Built-in Functions*

---

Options for built-in functions such as `Plot[ ]` and `Factor[ ]` are lists of substitution rules for symbols that can be used to modify the behavior of the functions. When no options are specified in an invocation, then the preassigned rules are used, for example,  $\text{AspectRatio} \rightarrow \frac{1}{\text{GoldenRatio}}$  for `Plot[ ]` or  $\text{Modulus} \rightarrow 0$  for `Factor[ ]`. When we use an option, we must be able to override the preassigned substitution. Mathematica deals with this problem by always evaluating user-specified options first.

**Example 6:** The order of evaluation of substitution rules for one symbol.

We apply two rules in succession to the single symbol  $x$ . Once we replace the symbol  $x$  in an expression by some value not containing  $x$ , then any further substitution of  $x$  has no effect. Only the first substitution has an effect in the following two examples.

```
Clear[x, y]

(x + y^2) /. {x → 5}
% /. {x → 10}

(x + y^2) /. {x → 10}
% /. {x → 5}

x
```

Note that  $x$  is still a symbol without a value assigned to it. We perform two successive substitutions for the option `AspectRatio`.

```
AspectRatio /. AspectRatio → 2 /. AspectRatio → Automatic

AspectRatio /. AspectRatio → Automatic /. AspectRatio → 2
```

**Example 7:** The option `AspectRatio` in `Plot[ ]`.

When we apply the option `AspectRatio` in `Plot[ ]`, we get the same kind of behavior.

```
??Plot

Plot[Sin[x], {x, -Pi, Pi}];

Plot[Sin[x], {x, -Pi, Pi}, AspectRatio → 2];
```

We can inspect the values of the plotting options with the built-in function `InputForm[ ]`. The list of options is at the nested level `{1,2}` in the `Graphics[ ]` object that represents the plot. We specify values for three options of `Plot[ ]`; find them in the list of options.

```
InputForm[
  Plot[Sin[x], {x, -Pi, Pi},
    PlotRange → {-1, 1}, DisplayFunction → Identity,
    AspectRatio → 2]]][[1,2]]
```

Now we name the option `AspectRatio` twice with different values in the same invocation of `Plot[ ]`. Only the first specification takes effect.

```
InputForm[
  Plot[Sin[x], {x, -Pi, Pi}, DisplayFunction → Identity,
    AspectRatio → 2, AspectRatio → 1]]][[1, 2]]
```

Note that it is necessary to place the closing bracket (`)` of `InputForm[ ]` into the last line of the input cell. If the closing bracket were in the previous line and the last line consisted only of

[[1, 2]], the input cell would contain the two expressions `InputForm[ ... ]` and `[[1, 2]]`, not the single expression `InputForm[ ... ][[1, 2]]`.

**Example 8:** Factorization of polynomials over different domains.

Computations with polynomials that have integer coefficients are frequently done over the set of integers. However, sometimes computations must be done over the Gaussian integers (complex numbers with integer real and imaginary parts) or modulo some prime number. The function `Factor[ ]` provides some options through which we can specify the number system for such a polynomial factorization.

**?Factor**

**Options[Factor]**

To demonstrate the different options, we use the polynomial  $1 + x + x^2 + x^3$ ; it factors partially over the integers and completely, like any polynomial with real coefficients, over the complex numbers.

**Clear[x]**

**Factor[1 + x + x^2 + x^3]**

**Factor[1 + x + x^2 + x^3, GaussianIntegers → True]**

Now we factor the polynomial modulo some small prime numbers.

**Factor[1 + x + x^2 + x^3, Modulus → 2]**

**Factor[1 + x + x^2 + x^3, Modulus → 3]**

**Factor[1 + x + x^2 + x^3, Modulus → 5]**

**Factor[1 + x + x^2 + x^3, Modulus → 13]**

When we expand a factorization, then the computations for the expansion are done again over the integers since the option `Modulus` has only local effect and the default setting for the option is `Modulus → 0`. However, we can restore the original polynomial by reducing the expansion with `PolynomialMod[ ]`.

**?PolynomialMod**

**Factor[1 + x + x^2 + x^3, Modulus → 5]**

**Expand[%]**

**PolynomialMod[%, 5]**

**Factor[1 + x + x^2 + x^3, Modulus → 13]**

**Expand[%]**

**PolynomialMod[%, 13]**

We can map the factorization function over the list of the first 10 primes and print a table for the results.

```
TableForm[Map[{#, Factor[1 + x + x^2 + x^3, Modulus → #]}&,
  Map[Prime, Range[20]]],
  TableHeadings → {{}, {"Modulus", "Factorization"}}]
```

## Defining an Option for a Function

In the three previous sections we have used substitution rules to evaluate expressions and as actual arguments for options in built-in functions. When we define an option for a function we must consider two aspects: An option has the syntactic form of a rule, as in  $\text{symbol} \rightarrow \text{value}$ , and an option always has a default value. When we invoke a function that has an option defined for it, then we can replace the default value of an option by naming the option with a new value.

**Example 9:** Computing the central angle, in degrees, subtended by a circular arc.

Consider a circular arc of length  $c$  on the circle of radius  $R$ . The central angle  $\alpha$  subtended by the arc is given by the relation  $c = \alpha R$ . If we want to measure the angle in degrees rather than radians, then the formula for  $\alpha$  is:  $\alpha = \frac{c}{R} \frac{180}{\pi}$ . We define a function `uAngle[ ]` that has the length of the arc  $c$  as its parameter. The radius  $R$  will be specified with the option `uRadius` whose default value is 1. The option is defined as follows:

```
Options[uAngle] = {uRadius → 1}
```

We are now able to invoke the function `uAngle[ ]` in two ways: `uAngle[c]` or `uAngle[c, uRadius → v]` where  $c$  and  $v$  are numbers. Therefore, we need to make provisions in the definition of the function `uAngle[ ]` to substitute the value of the radius in two different ways: with an option `uRadius → v`, or with a substitution of the default. In the first case we use the option as an application of the rule  $\alpha /. \{uRadius \rightarrow v\}$  or, if `opts` is the name of the parameter for the option, we write  $\alpha /. \{opts\}$ . In the second case we use the default case with  $\alpha /. Options[uAngle]$ . Since we need to ensure that an optional argument is applied before the default option, the following order of replacement is necessary:  $\alpha /. \{opts\} /. Options[uAngle]$ .

There is one other piece of information we need before we can define the function `uAngle[ ]`. In all the previous function definitions we used one underscore (`_`), which means “exactly one argument.” When we define an option for a function we must provide for the possibility to invoke no option or a single option. We can use the pattern mechanism of three underscores (`___`), which represents the more general case of “zero or more arguments.”

```
uAngle[c_, opts___] := (c/uRadius) (180/N[Pi]) /. {opts} /.
  Options[uAngle]
```

We compute the angles for several arcs and radii.

```
uAngle[1]
```

```
uAngle[2]
```

```
uAngle[1, uRadius → 2]
```

In order to keep this definition of a function with an option as simple as possible, we did not include any error checking. Therefore, we can use any numeric value in the option, whether it makes sense or not.

```
uAngle[1, uRadius → -1]
```

```
uAngle[1, uRadius → 0]
```

```
uAngle[-4, uRadius → 3]
```

We can also use symbolic values.

```
Clear[arc, radius]
```

```
uAngle[arc, uRadius → radius]
```

When we use an option that is not defined for the function `uAngle[ ]`, such as `Modulus`, then the default value of the option `uRadius` applies.

```
uAngle[1, Modulus → 5]
```

We can even try to evaluate the function `uAngle[ ]` with more than one option. This is possible because we used the pattern of three underscores in the definition, and because we enclosed the parameter opts in the body of the definition of the function `uAngle[ ]` in list braces. However, since there is only the option `uRadius` defined for `uAngle[ ]`, it is the only option that can have any effect during a function invocation. All other options are ignored.

```
uAngle[1, Modulus → 13, uRadius → 5, AspectRatio → 2]
```

**Example 10:** A cube root function with an option to compute the real-valued cube root.

All numeric computations in Mathematica are performed over the set of complex numbers. This is a reasonable strategy because meaningful computations will return a numeric, but possibly complex, value. One example is the cube root of  $-1$ .

```
(-1)^(1/3)
```

```
N[%]
```

For negative numbers, Mathematica chooses one value as the result of the computation of the cube root, namely the principal root. This is common mathematical practice. One way to compute all possible values of the cube root is to solve a polynomial equation.

```
Solve[x^3 == -1, x]
```

```
N[%]
```

We now have three distinct and correct answers for the cube root of  $-1$ . Since Mathematica picks one of the complex roots—the principal root rather than the real root—computations that

expect real numbers will fail for powers of negative real numbers. The following plot shows this failure.

```
Plot[x^(1/3), {x, -4, 4}];
```

Observe that Mathematica returns a real value for the cube root when the base is a positive real number even though there are again three distinct cube roots. Again, this is common mathematical practice, and Mathematica conforms to the standard conventions in mathematics.

We now define a function `uCubeRoot[ ]` that computes the cube root of a number. We also define an option `uCompute` with which we can force real numbers as answers whenever that is possible. The default will be complex values, just as is Mathematica's computing convention.

To determine whether a computation with complex numbers is required, we must be able to determine that an object is complex. This can be done by inspecting the head of a number using the function `Head[ ]` and comparing it to the Mathematica symbol `Complex`. In this case, however, we compare names, not values. This is done with the comparison operator `===` for identity testing and its negation `!=`.

```
Head[1 + 3I]
```

```
Head[1 + 3I] === Complex
```

```
Head[2]
```

```
Head[2] === Complex
```

```
Head[2] != Complex
```

A comparison of values rather than symbols, using `==`, does not succeed.

```
Head[2] == Complex
```

We implement the computation of the cube root as follows:

1. The argument must be a number.
2. If the number is not complex and the option specifies a real number as result, evaluate  $\sqrt[3]{x}$  as `(signum(x) * |x|1/3)`; otherwise use the default computation for  $x^{1/3}$  in Mathematica.

Before we define the function, we establish the default value for the option `uCompute`.

```
Clear[uCubeRoot, uCompute]
```

```
Options[uCubeRoot] = {uCompute -> Complex}
```

```
uCubeRoot[x_, opts___] :=
```

```
  If[Head[x] != Complex && (uCompute /. {opts} /.
```

```
    Options[uCubeRoot]) === Real,
```

```
    Sign[x] Abs[x]^(1/3), x^(1/3)] /; NumberQ[x]
```

Now we compute some cube roots.

```
uCubeRoot [-1]

uCubeRoot [-1, uCompute → Real]

uCubeRoot [1 + I]

uCubeRoot [1. + I]
```

The principal cube root of a complex number is a complex number. Therefore, the test in `uCubeRoot[ ]` forces the computation of the principal complex root even if the option specifies otherwise.

```
uCubeRoot [1 + I, uCompute → Real]
```

Now we can plot the graph of the cube root for positive and negative real numbers. Note that we use the option `uCompute` inside the function `uCubeRoot[ ]` and the graphics option `PlotStyle` for `Plot[ ]`.

```
Plot[uCubeRoot[x, uCompute → Real], {x, -4, 4},
     PlotStyle → RGBColor[0,0,1]];
```

## Exercises

---

**EXERCISE 1:** Substituting values into functions.

Apply rules and the substitution mechanism to compute the local maxima and minima of a function  $f(x)$ . First find all solutions  $z$  for the equation  $f'(x) = 0$ , then compute the sign of  $f''(x)$  at  $z$ , and finally evaluate  $f(z)$ . Always plot the function before you start the computations for an exercise.

- (a) Find all extremal values of the polynomial  $f(x) = 0.4x^3 - 7.1x^2 + 10.0x - 3.5$ .
- (b) Find one minimum of the function  $f(x) = x + 10 \sin x$  in the first quadrant.
- (c) Consider the following polynomial  $p(x)$  of degree 13:

$$p(x) = x - \frac{2x^3}{3} + \frac{3x^5}{10} - \frac{53x^7}{315} + \frac{1489x^9}{22680} - \frac{4541x^{11}}{207900} + \frac{100319x^{13}}{13899600}$$

This polynomial is a good approximation for the function  $\sin(x \cos x)$  in the interval  $[-1, 1]$ . Find the local minima and maxima for the polynomial in the interval  $[-1, 1]$ .

**EXERCISE 2:** A single option for the base of an exponential function.

There are many situations where the exponential with base 2, that is  $2^x$ , needs to be computed. Examples abound, for instance, in the study of the time complexity of algorithms.

- (a) Define the function `uExp[ ]` that computes the exponential with base 2 as default base. For example, `uExp[7]` computes  $2^7$ .



(b) Define an option `uBase` for the function `uExp[ ]` that changes the base of exponentiation. Give a new definition of `uExp[ ]`. For example, the invocation `uExp[4, uBase → 5]` computes the exponential  $5^4$ .

(c) Evaluate your function from part (b) for a variety of arguments, numeric as well as symbolic.

**EXERCISE 3:** Successive replacement of a symbol with an expression containing that symbol.

Here are computations where the exponent is doubled or halved at each step.

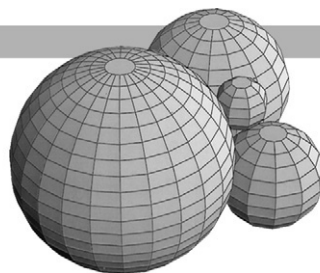
(a) Apply the rule  $x \rightarrow (1 + x)^2$  to the symbol  $x$  once, twice, and three times. Expand the results.

(b) Apply the rule  $x \rightarrow \sqrt{x}$  to the symbol  $x$  once, twice, and so on.

(c) Apply the rule  $x \rightarrow \frac{1}{1+x}$  to the symbol  $x$  once, twice, and so on. Evaluate each result at  $x \rightarrow 1$ .

Compare your results with the Fibonacci function `uFib[ ]` discussed in Example 4 of the notebook `Recurse.nb`, “Recursive Definitions.” Find a connection between the successive substitutions and the Fibonacci numbers.

# *Four Spheres Packing Problem*



## *Introduction*

In this notebook we solve one specific problem: the four spheres packing problem.

Take four spheres  $S_1$ ,  $S_2$ ,  $S_3$ , and  $S_4$ , whose radii are 1, 2, 3, and 4, respectively. Place the three larger spheres on the  $xy$ -plane so that each sphere touches the other two. Then place the smallest sphere  $S_1$  into the hollow formed by the other three spheres. Determine the coordinates of the center of sphere  $S_1$ .

The ultimate goal of the computations is to draw all four spheres in a plot. The computations in this notebook were used to produce the picture elements of the cover of this book.

## *Analysis of the Problem*

The two basic observations that lead to a solution of the problem are the following:

1. The distance between the centers of two touching spheres is the sum of their radii.
2. The difference of the heights of the centers of two touching spheres that rest on the  $xy$ -plane is the difference of their radii.

Consider a pair of touching spheres resting on the  $xy$ -plane. The line segment connecting the two centers is the hypotenuse of a triangle with one leg parallel to the vertical  $z$ -axis and the other leg in a plane parallel to the  $xy$ -plane. Since the two previous observations give us the length of the hypotenuse and the length of the vertical leg, we can find the length of the other leg with the Pythagorean theorem.

If you want to see a visualization of a two-dimensional cross-section of two touching spheres that rest on the horizontal plane, click on the hyperlink [cross-section](#).

If we place the spheres so that their centers are in the  $yz$ -plane, that is, the  $x$ -coordinates of both centers are zero, then we can determine the  $y$ -coordinate of the smaller sphere from the  $y$ -coordinate of the larger sphere and the length of the horizontal leg of the triangle.

Our computational strategy for finding the coordinates of the center of the sphere  $S_1$  will be to start by placing the largest sphere  $S_4$  so that all coordinates of its center are known, namely  $x = y = 0$  and  $z = 4$ . Then we place the sphere  $S_3$  with its center at  $x = 0$ ,  $z = 3$  and an unknown  $y$ -coordinate that can be determined from the triangle. For  $S_2$ , touching the other two, we have to determine the  $x$ - and  $y$ -coordinates of the center, and finally, for  $S_1$ , touching the

other three spheres, all three coordinates of the center are unknown. This strategy demands that we solve systems of one, two, and three quadratic equations, respectively, for the associated right triangles.

## *Computational Solution of the Problem*

---

We imagine that the centers of the four spheres are in the first octant in space. Therefore, we always choose the positive solution for the quadratic equations for the coordinates of the centers of the spheres.

The Sphere of Radius 4

We position  $S_4$  with its center on the z-axis at  $(0, 0, 4)$ .

The Sphere of Radius 3

We position  $S_3$  with its center along the y-axis at  $(0, y_3, 3)$ . From the triangle we get the equation for the unknown  $y_3$ :

$$y_3^2 + (4 - 3)^2 = (4 + 3)^2$$

```
Clear[y3,t3]
t3 = Solve[y3^2 + (4 - 3)^2 == (4 + 3)^2, y3]
N[t3]
```

We must select the second solution since it is the solution in the first octant.

```
y3 = y3/.t3[[2]]
```

The Sphere of Radius 2

If you want to see a visualization of a projection onto the xy-plane of the three touching spheres  $S_4$ ,  $S_3$ , and  $S_2$ , click on the hyperlink [2D-Projection](#).

The sphere  $S_2$  touches both larger ones. Therefore, its center  $(x_2, y_2, 2)$  is inside the first octant. For this sphere there are right triangles to each of the centers of  $S_3$  and  $S_4$ . The equations for the unknown horizontal leg in each right triangle are:

$$(x_2^2 + y_2^2) + (4 - 2)^2 = (4 + 2)^2$$

$$(x_2^2 + (y_3 - y_2)^2) + (3 - 2)^2 = (3 + 2)^2$$

```
Clear[x2,y2,t2]
t2 = Solve[{x2^2 + y2^2 + (4 - 2)^2 == (4 + 2)^2,
  x2^2 + (y3 - y2)^2 + (3-2)^2 == (3 + 2)^2}, {x2,y2}]
N[t2]
```

Again, we must select the second solution since it is the solution in the first octant.

```
{x2,y2} = {x2,y2}/.t2[[2]]
```

### The Sphere of Radius 1

This is the last step in the computation and the point of the problem. The sphere  $S_1$  is dropped onto the see-through hollow created by the three larger spheres. All coordinates  $(x_1, y_1, z_1)$  of the center of  $S_1$  are unknown since this sphere does not rest on the  $xy$ -plane. It is even conceivable that the hole created by the three larger touching spheres is too large, and  $S_1$  will fall through it. In that case, the system of equations will have no solution. The respective equations for the unknown horizontal legs in the three right triangles are:

$$\begin{aligned}(x_1^2 + y_1^2) + (4 - z_1)^2 &= (4 + 1)^2 \\(x_1^2 + (y_3 - y_1)^2) + (3 - z_1)^2 &= (3 + 1)^2 \\((x_2 - x_1)^2 + (y_2 - y_1)^2) + (2 - z_1)^2 &= (2 + 1)^2\end{aligned}$$

```
Clear[x1,y1,z1,t1]
t1 = Solve[{x1^2 + y1^2 + (4 - z1)^2 == (4 + 1)^2,
  x1^2 + (y3 - y1)^2 + (3 - z1)^2 == (3 + 1)^2,
  (x2 - x1)^2 + (y2 - y1)^2 + (2 - z1)^2 == (2 + 1)^2},{x1,y1,z1}]

N[t1]
```

Again, we must select the second solution. It has the larger coordinates and therefore represents the solution where the smallest sphere rests above the hole. The first solution represents the solution where the smallest sphere is glued below the hole.

```
{x1,y1,z1} = Simplify[{x1,y1,z1}/.t1[[2]]]
```

## Graphic Rendering of the Solution

---

**Example 1:** The graphics object `Sphere[ ]`.

For the actual graphical rendering of spheres we need the `Sphere[ ]` object from the `Shape` context of the `Graphics` package. Since `Sphere[ ]` is a three-dimensional graphics object it is drawn with the `Show[ Graphics3D[...]` command combination.

```
<<Graphics`Shapes`
```

```
?Sphere
```

The `Sphere[ ]` function takes zero, one, or three arguments. No arguments render a default sphere.

```
Show[Graphics3D[Sphere[]];
```

The single argument specifies the radius of the sphere and a default number of surface elements for the sphere.

```
Show[Graphics3D[Sphere[2]]];
```

The second argument,  $n$ , specifies the number of longitudes, and the third argument,  $m$ , specifies that  $m-1$  latitudes are to be drawn. This means that an even number as the third argument draws the equator. Furthermore,  $n * (m-2)$  surface elements are drawn, together with two flat polygons, representing the two polar caps.

```
Show[Graphics3D[Sphere[2, 24, 8]], Axes -> True];
```

The graphics package also contains a number of geometric operations such as translation and rotation. Since the `Sphere[ ]` object has the origin as its center, we must translate each of the four spheres to their respective center point, as computed in the previous section.

```
Names["Graphics`Shapes`*"]
```

```
?TranslateShape
```

```
Show[Graphics3D[TranslateShape[Sphere[2, 24, 8], 10, -20, 30]],  
Axes -> True];
```

Now we are ready to render the four spheres as they are used on the cover of this textbook. We first render them with the borders of the surface elements drawn and then without.

**Example 2:** Drawing the four packed spheres.

In the previous section we computed the coordinates of the centers of the four spheres,  $S_4, S_3, S_2, S_1$ . To minimize writing in the graphing commands, we use  $S_4, S_3, S_2, S_1$  as short names for the coordinates of the centers of the spheres. We copy the exact results from the previous section into the next input cell.

```
Clear[S4, S3, S2, S1]
S4 = {0, 0, 4};
S3 = {0, 4  $\sqrt{3}$ , 3};
S2 = {  $\sqrt{\frac{47}{3}}$ ,  $\frac{7}{\sqrt{3}}$ , 2 };
S1 = {  $\frac{1}{648} (101 \sqrt{141} + 17 \sqrt{645})$ ,  $\frac{1499 + \sqrt{10105}}{216 \sqrt{3}}$ ,  $\frac{1}{54} (149 + \sqrt{10105})$  };
```

We experimented with the longitudinal and latitudinal numbers so that the surface elements appear to have roughly the same size on all four spheres. Furthermore, we want the sphere of radius one to look like a sphere, and we want to draw the equator on each sphere. This experimentation has led to the numbers 24, 16, 12, and 8 for the respective longitude numbers of the four spheres. The coordinate triples  $S_4, S_3, S_2, S_1$  of the centers of the packed spheres are the translation vectors that we specify in `TranslateShape[ ]` in order to position the four spheres.

We draw the four packed spheres first with the edges of the surface elements drawn.

```
Show[Graphics3D[{  
  TranslateShape[Sphere[4, 24, 24], S4],  
  TranslateShape[Sphere[3, 16, 16], S3],
```

```

    TranslateShape[Sphere[2, 12, 12], S2],
    TranslateShape[Sphere[1, 8, 8], S1]
  },
  Boxed → False
];

```

Now we draw the four packed spheres without the edges of the surface elements drawn.

**?EdgeForm**

```

Show[Graphics3D[{EdgeForm[],
  TranslateShape[Sphere[4, 24, 24], S4],
  TranslateShape[Sphere[3, 16, 16], S3],
  TranslateShape[Sphere[2, 12, 12], S2],
  TranslateShape[Sphere[1, 8, 8], S1]
}],
  Boxed -> False
];

```

The final plot shows the four packed spheres with the default arguments for the second and third arguments of the Sphere[] objects.

```

Show[Graphics3D[{EdgeForm[],
  TranslateShape[Sphere[4], S4],
  TranslateShape[Sphere[3], S3],
  TranslateShape[Sphere[2], S2],
  TranslateShape[Sphere[1], S1]
}],
  Boxed -> False
];

```

## Exercises

---

**EXERCISE 1:** A special case of the Four Spheres Packing Problem.

In this exercise we consider the Four Spheres Packing Problem when the spheres  $S_4$ ,  $S_3$ , and  $S_2$  have the same radius  $r$  and the sphere  $S_1$  has radius  $s$ .

- Solve the packing problem when  $r = s = 1$ .
- Solve the packing problem when  $r = s$  is an arbitrary positive whole number.
- What is the largest value of the radius  $s$  at which the sphere  $S_1$  drops through the hole that is formed by the other three spheres when their radius is  $r = 1$ ?

**EXERCISE 2:** Computing quantities for the smallest sphere  $S_1$  in the Four Spheres Packing Problem.

In this exercise we assume that the spheres  $S_4$ ,  $S_3$ ,  $S_2$ , and  $S_1$  have radii of 4, 3, 2, and 1, respectively.

- (a) Compute the coordinates of the lowest point for the smallest sphere  $S_1$ .
- (b) Compute the coordinates of all six points of tangency between pairs of the four spheres.

**EXERCISE 3:** A Four Circles Packing Problem.

In this exercise we assume that three circles  $C_4$ ,  $C_3$ , and  $C_2$  in the  $xy$ -plane touch each other.

- (a) Suppose that the circles  $C_4$ ,  $C_3$ , and  $C_2$  all have radius 1. Compute the radius and the center of the circle  $C_1$  that fits between the given three circles so that it touches all three.
- (b) Suppose that the circles  $C_4$ ,  $C_3$ , and  $C_2$  have radii 4, 3, and 2, respectively. Compute the radius and the center of the circle  $C_1$  that fits between the given three circles so that it touches all three.
- (c) Suppose that the circles  $C_4$  and  $C_3$  have radius 1, and that the radius  $R$  of  $C_2$  is arbitrary. Put the centers of the circles  $C_4$  and  $C_3$  on the  $y$ -axis and the center of the circle  $C_2$  on the  $x$ -axis. Compute the radius and the center of the circle  $C_1$  that fits between the given three circles so that it touches all three.
- (d) What happens with circle  $C_1$  when the radius  $R$  of  $C_2$  increases towards infinity?
- (e) Use your work in the earlier parts of this exercise and a geometric argument to verify that  $\lim_{R \rightarrow \infty} (\sqrt{R(R+2)} - R) = 1$ .

**EXERCISE 4:** A problem of right triangles and circles.

In the notebook Views2D.nb, “Two-dimensional Views of Spheres,” a right triangle is constructed that connects the centers of three circles  $C_a$ ,  $C_b$ , and  $C_c$ , which touch each other and have radii  $a$ ,  $b$ , and  $c$ , respectively.

- (a) Under what conditions for  $a$ ,  $b$ , and  $c$  is this situation possible? There is at least one whole number solution,  $a = 3$ ,  $b = 2$ , and  $c = 1$  for the radii; this solution produces the 3-4-5 Pythagorean triangle.
- (b) Determine whether there are infinitely many triples of whole numbers  $a$ ,  $b$ , and  $c$  satisfying the condition.

# III

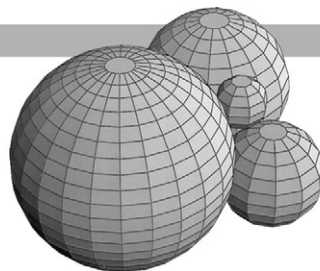
---

## *Designing Programs*



This Page Intentionally Left Blank

# List Processing Functions



## Introduction

One of the fundamental data structures available in Mathematica is the list. The elements of a list can be numbers, symbols, strings, graphics objects, functions, or lists in any combination that a problem may demand. In the previous notebooks, we encountered many computations that processed lists or created lists. In this notebook, we introduce several built-in functions that manipulate lists. Our goal is to show how to use these list functions and to demonstrate, through their use, functional programming in action.

We will start with the `Map[]` function as the fundamental list processing tool and give examples for its use. Then we discuss other list construction, list extraction, and list manipulation functions. There are two sections in the Help Browser that contain all the list processing functions:

1. Choose **Help** ► **Help Browser...** in the menu bar and click the following selections: **Built-in Functions** ► **Lists and Matrices** ► any topic.
2. Choose **Help** ► **Help Browser...** in the menu bar and click the following selections: **Programming** ► **Functional Programming** ► any topic.

## Processing Lists with the `Map[]` Function

The basic and prototypical list processing function is the `Map[]` function, which has two arguments: a function of a single parameter and a list. `Map[]` evaluates the function on every element of the list and returns the list of values computed for the objects in the list. For the computation process to succeed, the objects in the list must be legal arguments for the function.

### ??Map

The `Map[]` function performs two essential iteration control mechanisms:

1. It steps through a list and applies the function to its elements.
2. It constructs the list of resulting values.

This computational control and action is applied only to the top level of the list by the `Map[]` function. It can be applied to deeper levels of the list by giving a level specification in the optional third argument of `Map[]`.

**Example 1:** Constructing a table of function values for a list of numbers.

We use, as our first demonstration the list of the first 10 natural numbers starting with 1 and compute their natural logarithm. The `Range[ ]` function generates a list that we can use as the argument for the `Map[ ]` function.

```
?Range
```

```
Range[1, 10]
```

```
Range[-2, 3, 0.25]
```

```
Clear[a]
```

```
a = Map[Log, Range[1, 10]]
```

In this computation we get the symbolic logarithmic expressions. If we are interested in the numeric values of the natural logarithms we need to apply the function `N[ ]` to the values. There are two ways to do this.

We apply the numeric evaluation to the list of function values.

```
N[a]
```

We change the values in the list from infinite precision to their numeric representation as decimal numbers.

```
Range[1., 10.]
```

```
Map[Log, Range[1., 10.]]
```

We can change the size of the list, and the `Map[ ]` function applies its iteration control over the new list.

```
Range[0.5, 8.0, 0.25]
```

```
Map[Log, Range[0.5, 8.0, 0.25]]
```

**Example 2:** Computing with heterogeneous lists.

A list that contains objects of different data types is called heterogeneous. Here is an example of such a list containing numbers, symbols, strings, and function applications.

```
Clear[h, k]
```

```
h = {9.71, k, "notebook", Pi, 237, Point[{2, 5}]}
```

We can map a function over such lists. If an element in the list is a legal argument for the function, the function is evaluated; otherwise, an unevaluated expression is returned.

```
?NumberQ
```

```
Map[NumberQ, h]
```

```
?NumericQ
```

```
Map[NumericQ, h]
```

```
?Positive
```

```
Map[Positive, h]
```

Note that the first two functions `NumberQ[ ]` and `NumericQ[ ]` return a value for any object so that `Map[ ]` returns a list of Boolean values. On the other hand, `Positive[ ]` stays unevaluated on non-numeric arguments. Even in this case, the `Map[ ]` function succeeds and returns a list.

We can wrap list braces around every element in the list with the `Map[ ]` function. Note that this is different from evaluating `List[ ]` on a list in that it wraps list braces around its argument (see also [List](#)).

```
Map[List, h]
```

```
List[h]
```

Some functions such as `Print[ ]` and `Show[ ]` are used primarily for their control actions, also called side effects, such as writing or drawing on the screen. The value they return is of secondary importance. The function `Print[ ]` returns the value `Null`, and `Show[ ]` returns the value `-Graphics-`. When we use `Map[ ]` with such functions we get the side effect for every element in the list and a list of their values, for example `Null`, as the result of the mapping.

```
Map[Print, h]
```

**Example 3:** Using an anonymous function.

Sometimes we want to evaluate an expression on every element of a list without having to specify a function name for the expression. We might, for example, want to compute the list of squares of the whole numbers between 5 and 15. Computing the square is a function of a single argument and we can present the squaring expression as a function to `Map[ ]`.

Naturally, we can give the squaring operation a name by defining a function in the usual way as we have been doing all along.

```
Clear[uSquare]  
uSquare[x_] := x^2
```

```
?uSquare
```

```
Map[uSquare, Range[5, 15]]
```

An alternative to explicitly naming a function is to take the expression for the squaring operation and wrap the function maker `Function[ ]` around the expression.

```
?Function
```

```
Function[x, x^2]
```

```
Function[x, x^2][9]
```

The entire expression represents a function and can be given as an argument to `Map[ ]`.

```
Map[Function[x, x^2], Range[5, 15]]
```

In this last evaluation we still are forced to name the variable, `x`, in the function expression. However, the name of the variable is the least important piece in the expression. The computational rule, the squaring, is what counts. Therefore, we write the squaring expression in terms of an anonymous argument `#` and, to make it a function, we apply `Function[ ]` to it.

```
Function[#^2][9]
```

```
Map[Function[#^2], Range[5, 15]]
```

Enclosing the squaring expression with the `&`-symbol also turns the expression into a function of the anonymous argument. This has the same effect as using `Function[ ]`.

```
(#^2) &[9]
```

```
Map[(#^2) &, Range[5, 15]]
```

The use of anonymous functions is particularly useful when we deal with computational expressions in which several functions are nested. In Example 1 we needed two steps to compute the numeric value of the natural logarithm of a list of numbers. We can express the computation of the numeric values of the natural logarithm in terms of an anonymous function. We apply `N[ ]` to the results of `Log[ ]`, that is, we use the function `N[ Log[ ] ]`.

```
Map[N[Log[#]] &, Range[1, 10]]
```

If we want the list of numbers between 9 and 16 together with their squares and the approximate values of their square roots, then we just need to describe each of the computations with an anonymous argument, form the list of the three computational expressions, and turn the entire list of three expressions into an anonymous function.

```
Map[{#, #^2, N[Sqrt[#]]} &, Range[9, 16]]
```

Despite its complicated appearance, the expression `{#, #^2, N[Sqrt[#]]}&` still is a function of a single, anonymous argument. That argument is invoked three times inside the expression, and we can think of the function as a three-dimensional, vector-valued function.

Here is the result of the last computation in tabular form with column headings but without row headings.

```
TableForm[%, TableHeadings → {{}, {"Number", "Square", "Root"}}]
```

**Example 4:** Creating a list of random numbers.

In many previous examples we have used the random number generator `Random[ ]`. We show its use with the `Map[ ]` function. Observe that the next invocation evaluates an expression that yields a number.

```
Random[Integer, {1, 10}]
```

Since the first argument of the `Map[ ]` function must be a function we transform the expression `Random[Integer, {1, 10}]` into a function by using the function symbol `&`. The resulting function `Random[Integer, {1, 10}]&` is special also in that it contains no variable designation `#`; it is an

anonymous function with zero parameters. The `Map[ ]` function still applies a value from its second argument, `Range[15]`, but that value has no effect on the random number generator function `Random[ ]`.

```
Map[Random[Integer, {1, 10}]] &, Range[15]]
```

We could have generated the previous list with the `Table[ ]` function as shown next. Observe that the `Table[ ]` function takes an expression as its first argument, not a function, and that it takes an iterator as its second argument, not a list of arguments.

```
Table[Random[Integer, {1, 10}]] , {15}]
```

In order to decide which implementation is more advantageous, we will evaluate the two expressions on large amounts of data and time the computation. Note again the semicolon inside the `Timing[ ]` function. It prevents the listing of the 500,000 numbers on the screen. In this case we are not interested in the numbers; we are interested only in the time it takes to generate them.

Before you evaluate the next two input cells make sure that you have enough memory allocated for the Mathematica kernel. See the notebook `BasHelp.nb` for details on adjusting the allocation to the front end and the kernel.

If you are working on a Macintosh, display the memory gauge for the kernel of Mathematica and watch it while you evaluate the two input cells. The `Range[ ]` function actually constructs a list of five hundred thousand numbers to which the `Map[ ]` function is then applied. This requires a sizable amount of memory and time. The `Table[ ]` function uses a single memory location for that purpose as a counter.

```
Timing[Map[Random[Integer, {1, 10}]] &, Range[500000]]];
```

```
Timing[Table[Random[Integer, {1, 10}]] , {500000}];]
```

**Example 5:** Graphing a list of vertical line segments from a list of points.

Suppose we have a list of points in the plane. For simplicity, we use points of the sine curve in equidistant horizontal spacing.

```
Clear[pointlist]  
pointlist = Map[{#, Sin[#]} &, Range[0, Pi, Pi/8]]
```

In order to draw the vertical segments from the points to the x-axis, we need to create a second list of points with constant y-coordinate zero.

```
Clear[baselist]  
baselist = Map[{First[#], 0} &, pointlist]
```

Now we need to combine these two lists, first point with first point, second with second, and so on, in order to obtain the pairs of endpoints for the line segments. We achieve this by forming a list of the two lists, `pointlist` and `baselist`, and transposing the result.

```
Transpose[{pointlist, baselist}]
```

```
Length[%]
```

Observe that this list contains an element (at the top level) for each pair of points representing the line segments. The final step is to wrap the `Line[ ]` function around each pair of points so that we get graphics objects. This is accomplished by yet another application of the `Map[ ]` function.

```
Show[Graphics[Map[Line, Transpose[{pointlist, baselist}]]]]];
```

We combine all steps into one function that has a list of points as its single argument. This function, `uVerticals[ ]`, also draws the line segments connecting the points on the curve together with the verticals to the x-axis from each point.

```
Clear[uVerticals]  
uVerticals[p1_] :=  
  Show[Graphics[{Line[p1], Map[Line, Transpose[{p1, Map[{First[#], 0}  
    &, p1]}]]]]]
```

Here are several invocations of this function with lists of points from various trigonometric curves. The last one is drawn according to a logarithmic horizontal scale.

```
uVerticals[pointlist];  
  
Map[{#, Sin[#]} &, Range[0, 2Pi, Pi/16]];  
uVerticals[%];  
  
Map[{#, Sin[#]^2} &, Range[0, 2Pi, Pi/24]]; uVerticals[%];  
  
Map[{Log[1 + #], Sin[#]} &, Range[0, 4Pi, Pi/16]];  
uVerticals[%];
```

**Example 6:** Computations at the top level of nested lists.

Many mathematical applications use nested lists: matrices are lists of vectors, polygons are lists of coordinates in the plane or in three dimensions, and circles in the plane are described by the center and the radius, that is, lists consisting of a coordinate pair and a non-negative number. The list processing function `Map[ ]` acts on the top level of nested lists.

We use a variant of the function `uRandomMatrix[ ]` from Example 9 in the notebook `FuncDef.nb`, “Functions,” to create matrices quickly. Here we change the first parameter in the function interface to a structured parameter that specifies the number of rows as well as the number of columns. We add a parameter that specifies the type of the entries in the matrix, such as `Integer` or `Real`, as the second parameter. The third parameter specifies the range for the random numbers. The three parameters are arranged in the function interface according to the questions: What size does the matrix have? What types of numbers are its entries? From which interval are the entries chosen?

```
Clear[uRandomMatrix]  
uRandomMatrix[{rows_, columns_}, type_, {min_, max_}] :=  
  Table[Random[type, {min, max}], {rows}, {columns}]
```

If the argument list in a `Map[ ]` function is a matrix then the mapped function acts on entire row vectors, that is, lists of numbers. Here is an example that computes the largest entry in each row vector.

**?Max**

Evaluate the input cell several times until you get a negative maximum value for at least one row vector.

```
Clear[a]
a = uRandomMatrix[{8, 5}, Integer, {-10, 10}];
TableForm[a]
Map[Max, a]
```

We can define the length of a vector as the entry with largest absolute value in the vector. This length is called the maximum norm. A small change in the `Map[ ]` expression computes the maximum norms for the rows in a matrix.

```
Clear[b]
b = uRandomMatrix[{8, 5}, Integer, {-10, 10}];
TableForm[b]
Map[Max[Abs[#]] &, b]
```

Here is a geometric example that uses the usual Euclidean norm in which length, or distance, is computed with the Pythagorean theorem. Suppose we are given a list of points in the plane and we want to compute the distance from the origin for the point farthest away. We accomplish this by computing the list of distances for all points and then finding the maximum among them.

Since `d` is a list of numbers we apply `Max[ ]` directly to `d`, the `Map[ ]` function is not needed in the computation. Evaluate the next input cell repeatedly until one of the corner points of the  $20 \times 20$  square from which the random points are chosen appears in the list. The resulting distance will be  $10\sqrt{2}$  in that case.

```
Clear[p, d]
p = uRandomMatrix[{15, 2}, Integer, {-10, 10}]
d = Map[((First[#]^2 + Last[#]^2)^(1/2)) &, p]
Max[d]
N[%]
```

We conclude with an example where the `Map[ ]` function is used twice in a single expression. Our objective is to compute the definite integral from 0 to  $\pi$  for each of the functions  $\sin(t x)$  over the range of frequencies  $t$  from 1 to 16. We accomplish this by creating the list of sine functions with the `Map[ ]` function and then mapping the `Integrate[ ]` function across the list of sine functions.

```
Map[Sin[# x] &, Range[16]]

Map[Integrate[#, {x, 0, Pi}] &, Map[Sin[# x] &, Range[16]]]
```



We can improve the presentation of the result of the last computation by putting it into tabular form with a heading for the columns.

```
TableForm[Map[{#, Integrate[#, {x, 0, Pi}]} &, Map[Sin[# x] &,
  Range[16]]], TableHeadings → {{}, {"Function", "Integral\n"}}]
```

## *Applications of the Map[ ] Function to Graphics Objects*

---

In many situations we can use the Table[ ] function or the Map[ ] function to create lists of numbers or lists of graphics primitives. The first function needs a specifier for a range to create a list, and the second requires a list as an argument. Therefore, we frequently can use these two functions in tandem, one acting on the result of the other.

**Example 7:** Graphing a list of random integer lattice points in the first quadrant.

Our objective in this example is to demonstrate how Map[ ] can be used to create graphics objects. Toward this end, we draw a list of points within a specified square in the first quadrant.

First, we define a function to compute a list of coordinates in the first quadrant. This function will have two arguments: the first specifies the number of points and the second the coordinate range for the points. We use this function definition to introduce a framework for documenting the interface and computations that implement the function.

DEFINITION :: uCoordinatePairs[n, { $k_0$ ,  $k_1$ }]

INPUT :: A positive integer n and an integer range specification { $k_0$ ,  $k_1$ } for the coordinates.

1. Invoke the random number generator twice in the range { $k_0$ ,  $k_1$ }.
2. Form the coordinate pair from the values in step 1.
3. Perform the previous two steps n times.

OUTPUT :: A list of n random integer lattice points in the square from ( $k_0$ ,  $k_0$ ) to ( $k_1$ ,  $k_1$ ).

IMPLEMENTATION :: uCoordinatePairs[ ] :

```
Clear[uCoordinatePairs]
uCoordinatePairs[n_Integer, {k0_Integer, k1_Integer}] :=
  Table[{Random[Integer, {k0, k1}], Random[Integer, {k0, k1}]}],
  {n}] /; (n > 0 && k0 < k1)
```

First, we compute some sample collections of coordinates.

```
uCoordinatePairs[-10, {-5, 5}]
uCoordinatePairs[-10, {0, 1}]
```

```
uCoordinatePairs[5, {10, 0}]
uCoordinatePairs[5, {10, 25.5}]
```

Next, we transform the coordinates to graphics primitives. Then we plot them and specify additional graphics primitives to control the form of the points.

Here is the graph of a first set of 20 points.

```
Clear[c1, p1]
c1 = uCoordinatePairs[20, {-5, 5}]
p1 = Map[Point, c1]

Show[Graphics[p1], AspectRatio → Automatic];
```

Here is the graph of a second set of 20 points.

```
Clear[c2, p2]
c2 = uCoordinatePairs[20, {-5, 5}]
p2 = Map[Point, c2]

Show[Graphics[p2], AspectRatio → Automatic];
```

Next, we show the two lists of points together in a single plot, each with a different set of graphics attributes.

```
Clear[s1, s2]
s1 = Graphics[{PointSize[0.03], RGBColor[0, 0, 1], p1}];
s2 = Graphics[{PointSize[0.05], RGBColor[1, 0, 1], p2}];
Show[s1, s2, AspectRatio → Automatic];
```

Finally, we specify the two lists of points in a different order so that we can determine whether some of the wider points covered up some of the smaller points.

```
Show[s2, s1, AspectRatio → Automatic];
```

**Example 8:** Graphing a list of circles of different colors in the first quadrant.

We expand the previous example by drawing random circles instead of points and by giving each of the circles an individual color in a random fashion. We use the Mathematica graphics primitive `Circle[ ]` to create the circles. We use the integer grid of the  $10 \times 10$  square in the first quadrant as the locations for the centers of the circles and we allow values between 0.0 and 1.0 for the radii of the circles.

**?Circle**

In order not to have to write so much code we define two functions with the short names `uRI[ ]` and `uRR[ ]`. Each function has zero arguments. The function `uRI[ ]` returns a random integer in the range from 0 to 10, and the function `uRR[ ]` returns a random real number in the range from 0.0 to 1.0.

```
Clear[uRI, uRR]
uRI[ ] := Random[Integer, {0, 10}]
uRR[ ] := Random[Real, {0.0, 1.0}]
```

Now we are ready to create a drawing of 20 circles. This time, we perform all the computations in a single step. That is, we create a random color and a random circle simultaneously as a pair of graphics primitives.

```
Clear[c]
c = Map[{RGBColor[uRR[]], uRR[], uRR[]], Circle[{uRI[], uRI[]],
uRR[]}] &, Range[20]];
```

If you want to see the coordinates of individual circles (such as the first or the last circle) or the entire result of the computation, evaluate some of the following input cells.

```
First[c]

c[[20]]

c
```

Finally, we render the circles in a frame and make sure that circles are drawn as circles and not as ellipses.

```
Show[Graphics[c, AspectRatio → Automatic, Frame → True]];
```

We conclude this example by defining a function that combines all these steps. We only need a single parameter for the number of circles to be drawn since all other properties of the drawing are determined.

DEFINITION :: `uRandomCircles[ n ]`

REQUIREMENTS :: Needs the two functions `uRI[ ]` and `uRR[ ]`.

INPUT :: A positive integer `n` for the number of circles.

1. Generate a random color specification for each circle using `uRR[ ]`.
2. Generate a random circle on the integer lattice in the square from (0,0) to (10,10), using `uRI[ ]`, and having a radius between 0.0 and 1.0, using `uRR[ ]`.
3. Form the pair consisting of the graphics primitives created in steps 1 and 2.
4. Generate a list of `n` objects according to step 3, using `Map[ ]`.
5. Render the list of graphics primitives computed in step 4 in a square frame of fixed size extending from  $-1$  to  $11$  in the  $x$ - and  $y$ -directions.

OUTPUT :: A rendered graphics object containing `n` colored circles.

IMPLEMENTATION :: `uRandomCircles[ ]`

```
Clear[uRandomCircles]
uRandomCircles[n_Integer] := Show[Graphics[
```

```

Map[{RGBColor[uRR[], uRR[], uRR[]], Circle[{uRI[],uRI[]],
  uRR[]}&, Range[n]],
  AspectRatio → Automatic, Frame → True,
  PlotRange → {{-1, 11}, {-1, 11}}]] /; (n > 0)

```

We specify the plotting range in the function, so that the frame in the rendered graph stays the same independently of the locations of the circles that are generated. Evaluate this function several times so that you see the random distribution, sizes, and colors of the circles in the square.

```
uRandomCircles[20];
```

If the drawing contains short line segments or small squares rather than circles, enlarge the frame. Then you will see that the short line segments or small squares are an artifact of the graphics rendering in a small frame and that the objects really are circles.

```
uRandomCircles[40];
```

If we draw many more circles on the 121-point integer grid, we can expect several circles with identical centers.

```
uRandomCircles[100];
```

**Example 9:** Graphing a color transition for a list of line segments.

This is an extension of Example 5, where we drew vertical lines from points on a curve to the x-axis. In this example we compute the vertical line segments and color them from blue to yellow in a linear progression. Since we want to do this for arbitrary lists of points, the coloring function must be based on the length of the list. We define three functions for the computational steps that support the definition of the final function `uShowSegments[ ]`.

The first function computes the line segments from a list of points.

DEFINITION :: `uVerticalLines[pts]`

INPUT :: A list `pts` of coordinate pairs for points in the plane.

1. For each point in `pts`, compute the vertical projection point on the x-axis.
2. Generate the list of all projections from the list `pts`.
3. Form the matrix of two rows: `pts` and the list of step 2.
4. Form the list of all pairs of points by transposing the matrix of step 3.
5. Map the graphics primitive `Line` across the list of pairs computed in step 4.

OUTPUT :: A list of graphics primitives for the vertical lines to the x-axis for each point in `pts`.

IMPLEMENTATION :: uVerticalLines[ ]

```
Clear[uVerticalLines]
uVerticalLines[pts_] := Map[Line, Transpose[{pts, Map[{First[#], 0}
&, pts]}]]
```

We use the data set from Example 5 in the evaluations of this example.

```
Clear[pointlist]
pointlist = Map[{#, Sin[#]} &, Range[0, Pi, Pi/8]]

uVerticalLines[pointlist]
```

The second function computes the linear scale for the colors of the line segments. The input for this function is the list of points.

DEFINITION :: uLinearScale[pts]

INPUT :: A list pts of the coordinate pairs for at least two points in the plane.

1. Determine the number of points in the list pts.
2. Subdivide the interval [0, 1] into as many values as there are points in the list pts.

OUTPUT :: An equidistant subdivision of [0, 1].

IMPLEMENTATION :: uLinearScale[ ]:

```
Clear[uLinearScale]
uLinearScale[pts_] :=
Map[(# - 1) / (Length[pts] - 1) &, Range[Length[pts]]] /;
(Length[pts] > 1)
```

Here is the subdivision for the data set pointlist that we just evaluated.

```
uLinearScale[pointlist]
```

The third function maps RGBColor[ ] across the list of the linearly spaced values between 0 and 1 in such a way that we start with the color blue, RGBColor[0,0,1], and end with yellow, RGBColor[1,1,0]. Therefore, we need to vary the first two color parameters from 0 to 1 and the third parameter from 1 to 0.

DEFINITION :: uBlueYellow[pts]

REQUIREMENTS :: Needs the function uLinearScale[ ].

INPUT :: A list pts of the coordinate pairs for at least two points in the plane.

1. Compute the subdivision from the list pts using the function uLinearScale[ ].
2. Map the values in the subdivision into the appropriate arguments of RGBColor[ ].

OUTPUT :: The list of linearly increasing colors from blue to yellow for the point in the list pts.

IMPLEMENTATION :: `uBlueYellow[ ]`

```
Clear[uBlueYellow]
uBlueYellow[pts_] := Map[RGBColor[#, #, 1 - #] &,
  uLinearScale[pts]]
```

Here is the list of colors for line segments of the data set pointlist.

```
uBlueYellow[pointlist]
```

Finally, we combine each of these colors with the appropriate line segment. We use the `Transpose[ ]` function again as we did for the function `uVerticalPairs[ ]` in Example 5. Furthermore, we draw a polygon through the points.

DEFINITION :: `uShowSegments[pts]`

REQUIREMENTS :: Needs the functions `uVerticalLines[ ]` and `uBlueYellow[ ]`.

INPUT :: A list pts of the coordinate pairs for at least two points in the plane.

1. Compute the list of vertical line segments using the function `uVerticalLines[ ]`.
2. Compute a color for each vertical line segment using the function `uBlueYellow[ ]`.
3. Combine both lists as a list of pairs using `Transpose[ ]`.
4. Add the list of line segments, drawn in red, connecting the points in the list pts.

OUTPUT :: Render the combined graphics computed in steps 1 through 4 for the list pts.

IMPLEMENTATION :: `uShowSegments[ ]`:

```
Clear[uShowSegments]
uShowSegments[pts_] :=
  Show[Graphics[{Transpose[{uBlueYellow[pts], uVerticalLines[pts]}],
    Thickness[0.008], RGBColor[1, 0, 0], Line[pts]}]]
```

We draw the color transitions for several sets of points.

```
uShowSegments[pointlist];

uShowSegments[Map[{#, Sin[#]} &, Range[0, 2Pi, Pi/16]]];

uShowSegments[Map[{#, Sin[#]} &, Range[0, 2Pi, Pi/256]]];
```

## *List Manipulation, Element Extraction, and the Fold[ ] Function*

---

In this section we discuss several of the list operations and a number of the element extraction functions that are built into Mathematica. In many applications it is useful to be able to extract single elements from a list or sublists starting at a specified point in a list. There are many functions in Mathematica that accomplish this and we present several of them here.

**Example 10:** The basic actions of the FoldList[ ] and Fold[ ] functions.

The FoldList[ ] function has three arguments: a function  $f[ , ]$  of two parameters, a starting value  $s$  with which to begin the computations, and a list  $L$  to which the function of two parameters is applied. The FoldList[ ] function builds a list  $M$  of values from the starting value  $s$  as follows. It applies  $f$  to the value computed for the previous element of the list  $M$  with the current element in list  $L$  and it appends the computed value to list  $M$ . This process continues until the end of the list  $L$  is reached. Suppose that list  $L = \{u_1, \dots, u_k\}$ . Then the sequence of computations is  $s, f[s, u_1], f[f[s, u_1], u_2], f[f[f[s, u_1], u_2], u_3], \dots$

```
?FoldList
```

```
Clear[x, a, b]  
FoldList[Plus, x, {b, 12, a}]
```

```
FoldList[Times, 1, Range[6]]  
Length[%]
```

```
FoldList[List, {0}, Range[5]]  
Length[%]
```

This action is similar to that of the Map[ ] function since it treats each element of a list separately. However, the list  $M$  that is returned by FoldList[ ] has one more element than the input list  $L$ .

The Fold[ ] function computes the same value as the expression Last[ FoldList[ ] ].

```
?Fold
```

```
Fold[Plus, 0, Range[100]]
```

```
Last[FoldList[Plus, 0, Range[100]]]
```

```
Clear[x, a, b, c]  
Fold[Plus, x, {b, 12, a}]
```

```
Fold[Times, 1, Range[6]]
```

**Example 11:** Computing the mean of a list of test scores.

When we want to compute the arithmetic mean of a list of numbers we first need to add up all elements in the list. We can do this in a left-to-right scan as follows: we start with the value zero and we accumulate the sum as we step through the elements in the list until we get to the end of the list. The `Fold[ ]` function can implement this action.

Here are 20 randomly created test scores in the range from 0 to 100 points, and their total sum.

```
Clear[scores, total]
scores = Table[Random[Integer, {0, 100}], {20}]
total = Fold[Plus, 0, scores]
```

To compute the mean we divide the accumulated total by the number of scores, that is, by the length of the list of test scores.

```
total/Length[scores]
```

If we are interested only in a whole number as the average, then we use the `Round[ ]`, `Floor[ ]`, or `Ceiling[ ]` function; otherwise we use `N[ ]` on the quotient expression.

```
?Round
```

```
Round[total/Length[scores]]
```

```
N[total/Length[scores]]
```

We combine these computations in the function `uArithmeticMean[ ]`. We require that the single parameter of this function is a list of non-negative whole numbers, that is, a vector.

```
?VectorQ
```

```
Clear[uArithmeticMean]
uArithmeticMean[L_] :=
  N[Fold[Plus, 0, L] / Length[L]] /; VectorQ[L, (IntegerQ[#] &&
    NonNegative[#]) &]
```

We supply a list of test scores and compute their average with `uArithmeticMean[ ]`.

```
Clear[test1]
test1 = {49, 100, 0, 95, 84, 75, 67, 43, 98, 87, 83, 91, 100, 46, 12, 89};
uArithmeticMean[test1]
```

In order to easily build lots of examples of lists of scores we write a function `uRandomScores[ ]` that takes a count for the scores and an integer range for the scores as its two parameters. The function `Random[ ]` expects a range specifier. We name the range parameter, `r`, in the function interface, but do not specify its pieces by name. It is important, however, that we provide the correct type of range, consisting of two numbers. We do this with the underscore symbol and the pattern `{_,_}`. Even though the variables in the pattern are anonymous, we can specify their data type, for example, as `Integer` in the form `{_Integer, _Integer}`.



```
Clear[uRandomScores]
uRandomScores[k_, r: {_Integer, _Integer}] :=
  Table[Random[Integer, r], {k}]
```

Here are two examples that use the function `uRandomScores[ ]`. Check the results by hand.

```
Clear[test2]
test2 = uRandomScores[5, {70, 95}]
uArithmeticMean[test2]
```

Evaluate the next input cell several times until at least one of the scores is zero.

```
Clear[test3]
test3 = uRandomScores[6, {0, 25}]
uArithmeticMean[test3]
```

**Example 12:** Creating a list from the digits of  $\pi$  and extracting values from the list.

In this extended example we manipulate the digits of the number  $\pi$  in various ways:

1. We create lists of the digits of  $\pi$  and extract elements from these lists.
2. We manipulate the entire list of digits.
3. We compute frequency distributions for the digits of  $\pi$ .

We first must compute  $\pi$  to an appropriate level of accuracy and then build the list of its digits. Since  $\pi$  is a real number the first step is to multiply  $\pi$  with the appropriate power of 10 and to convert the result to the integer data type with the `Floor[ ]` function. From this integer we get the list of digits with the `IntegerDigits[ ]` function.

```
?IntegerDigits

IntegerDigits[1349173]

IntegerDigits[Floor[134.9173]]
```

We put the two functions together in the definition of `uDigitsOfPi[ ]`, whose single parameter `k` is the number of digits to be computed for  $\pi$ .

```
Clear[uDigitsOfPi]
uDigitsOfPi[k_] := IntegerDigits[Floor[N[Pi*10^(k - 1), k]]]
```

We create `list10` and `list100`, the lists of the first 10 and 100 digits of  $\pi$ , respectively.

```
Clear[list10]
list10 = uDigitsOfPi[10]

Clear[list100]
list100 = uDigitsOfPi[100]
```

Now we extract individual digits from list100.

```
First[list100]
```

```
Last[list100]
```

```
list100[[72]]
```

Here we extract blocks of digits from list100 using three of the built-in functions that manipulate entire lists. We can specify an arbitrary subrange and ranges from either end of the list.

```
?Take
```

```
Take[list100, {15, 20}]
```

```
Take[list100, 10]
```

```
Take[list100, -10]
```

Here we remove blocks of digits from list100 and keep the remainder list.

```
?Drop
```

```
Drop[list100, 95]
```

```
Drop[list10, {2}]
```

```
?Rest
```

```
Rest[list10]
```

Now we show three built-in functions that manipulate entire lists of the digits of  $\pi$ .

We compute a list of fifty digits and rearrange the elements in the list in various ways.

```
Clear[list50]
```

```
list50 = uDigitsOfPi[50]
```

For example, we can sort the list and eliminate duplicate digits.

```
?Sort
```

```
Sort[list50]
```

```
?Union
```

```
Union[{9, 4, 7, 3, 6}, {6, 1, 9, 4, 3}]
```

```
Union[list50]
```

We can count the number of occurrences of a particular digit.

```
?Count
```

```
Count[list50, 0]
```

```
Count[list50, 3]
```

By mapping the Count[ ] function across the list of digits we get the distribution of the digits in this list.

```
Map[Count[list50, #] &, Range[0, 9]]
```

We implement this last computation as the function uCountOfPi[ ], whose single argument specifies the number of digits of  $\pi$  that we want to compute, and which returns the distribution of the ten digits. Note that any actual argument of this function must be of type Integer.

```
Clear[uCountOfPi]
uCountOfPi[k_Integer] := Map[Count[uDigitsOfPi[k], #] &,
  Range[0, 9]]
```

Here are some distributions for the digits of  $\pi$ .

```
uCountOfPi[100]
```

```
uCountOfPi[200]
```

The definition of uCountOfPi[ ] presents serious problems when we want to perform an extensive study of the digits of  $\pi$ . We recompute the list of digits of  $\pi$  every time we invoke the function uCountOfPi[ ] and we always start with the first digit of  $\pi$ .

The best way to accommodate these concerns is to compute a large list of digits of  $\pi$  and save it. Then we can use the function Take[ ] to extract the block of digits that we are interested in.

We give an alternate definition for uCountOfPi[ ] where the single argument is a list rather than an integer. Observe that we do not use Clear[ ] to erase the previous definition of the function uCountOfPi[ ] because we want to have both forms of invocation available.

```
uCountOfPi[L_List] := Map[Count[L, #] &, Range[0, 9]]
```

Now we have two different implementations associated with the same function name. This is referred to as overloading. Mathematica will choose the appropriate implementation according to the type of the argument that we supply to the function uCountOfPi[ ]. This is possible since we restricted the pattern variable in two different ways in the two function definitions.

```
?uCountOfPi
```

```
Clear[list40]
list40 = uDigitsOfPi[40]
```

```
uCountOfPi[list40]
```

```
uCountOfPi[40]
```

Next we construct a long list of digits and time the evaluation for the two implementations.

```
Clear[list10000]
Timing[list10000 = uDigitsOfPi[10000];]
```

```
Timing[uCountOfPi[list10000]]
```

```
Timing[uCountOfPi[10000]]
```

In this last expression we computed the list of 1000 digits 10 times! For each of the two functions, Count[ ] scans through the list of digits 10 times.

As another example for the list extraction functions we compute the distribution of the 10 digits in each 1000-digit interval for the first 10,000 digits of  $\pi$ . This we can do by taking successive 1000-digit blocks from list10000 and mapping the function uCountOfPi[ ] across the list of these blocks. Observe that the base counter starts at zero and that the 1000-digit blocks are specified by the offsets 1 and 1000. Recall that the first element in a list is at position 1.

```
Map[uCountOfPi[Take[list10000, {# + 1, # + 1000}]] &,
  Range[0, 9000, 1000]];
TableForm[%, TableSpacing -> {2,1}, TableHeadings ->
  {Range[0, 9000, 1000], Range[0, 9]}]
```

**Example 13:** Computing the length of vectors of any dimension.

In Example 6 we computed distances in the plane. With the Fold[ ] function we have the tool to compute the length of a vector with an arbitrary number of coordinates. The Euclidean length is the square root of the sum of the squares of the coordinates. We use an anonymous function for the squaring and adding and let Fold[ ] scan across the vector.

In order to see the individual steps we first show an evaluation of the sum of the squares with FoldList[ ]. The symbols #1 and #2 are the two anonymous parameters of the function with two arguments that we fold over the input list.

```
FoldList[(#1 + #2^2) &, 0, {3, -6, 4, 2, 10}]

N[Sqrt[Fold[(#1 + #2^2) &, 0, {3, -6, 4, 2, 10}]]]
```

We implement this computation of the length of a vector as a function uEuclidean[ ]. We modify the pattern variable v\_ for the vector by questioning whether v is a vector. This is done by preceding the function name VectorQ with the ? symbol.

```
Clear[uEuclidean]
uEuclidean[v_?VectorQ] := N[Sqrt[Fold[(#1 + #2^2) &, 0, v]]]

uEuclidean[{8, -6}]

uEuclidean[{1, -2, 3, -4, 5, -6, 7}]

uEuclidean[{1, Pi, E, Sqrt[2]}]

Clear[a, b, c, d]
uEuclidean[{a, b, c, d}]
```

## *The Functions `Head[ ]`, `Apply[ ]`, `Outer[ ]`, `Depth[ ]`, and `Position[ ]`*

---

In this section we introduce additional built-in list functions that are helpful in extracting information from lists of numbers. We apply these to the following problems:

1. Determining the type of an object or expression.
2. Computing a value from a list such as an average.
3. Finding elements with a given property in a list.
4. Finding the position of certain elements in a list.
5. Extracting objects from deeper nesting levels of a list.

**Example 14:** The head of an expression.

Every object in Mathematica, whether it is symbolic or numeric, a single term or a compound expression, a user-defined function or a built-in operator, is associated with a head. The head classifies the object. As with all objects in Mathematica that we have encountered so far, the head is itself an object that we can inspect with a Mathematica function, `Head[ ]`. We can even change the head if we want to—for example, a list into a sum. A closely related function is `FullForm[ ]`, which displays the full extent of an expression as it is stored in Mathematica.

**?Head**

**?FullForm**

We first show the heads for a list of symbols.

```
Clear[x, y, z]
{x, y, z}
Head[{x, y, z}]
FullForm[{x, y, z}]
```

The positions of the elements in a list start with position one. However, the position zero actually is defined and stores the head of the list.

```
{x, y, z}[[1]]
{x, y, z}[[0]]
```

Next we compute the heads of the elements in a heterogeneous list.

```
Clear[a, b]
Map[Head, {3, "x-axis", 8.31, -8/31, Pi, a^2, {a, b}}]
```

We can think of the head of an expression as the outermost function that is applied to the pieces of the expression.

```
Plus[x, y, z]
FullForm[Plus[x, y, z]]
```

Just as in a list of objects, the pieces in any expression, including its head, have positions in that expression.

```
Head[Plus[x, y, z]]
```

```
Plus[x, y, z][[1]]
```

```
Plus[x, y, z][[0]]
```

**Example 15:** The arithmetic mean of a list of numbers revisited.

We revisit Example 11 of the previous section where we computed the arithmetic mean of a list of numbers with the `Fold[ ]` function. Here, we manipulate the head of the list of numbers by changing the head from `List` to `Plus` with the effect that the list of numbers becomes the sum of the numbers. Mathematica provides the function `Apply[ ]` to make this change and to evaluate the new head on the resulting expression. Clearly, the new head must be consistent with the existing objects in the list.

```
?Apply
```

```
Apply[Plus,{x, y, z}]
```

```
FullForm[%]
```

```
Apply[Plus, {3, -17, 38}]
```

```
FullForm[%]
```

We now compute the average of a list of numbers using `Apply[ ]`.

```
Clear[d4]
```

```
d4 = {12, 34, 25, 19}
```

```
Apply[Plus, d4]/Length[d4]
```

```
N[%]
```

What is the average of the first ten powers of two? Make an estimate before you evaluate the next input cell. What is the average of the first five or first twenty powers of two? This computation is done by an expression that fits easily into one line. Such expressions are referred to as one-liners.

```
Apply[Plus, Map[(2^#) &, Range[10]]]/10
```

**Example 16:** Finding minimal and maximal entries in a matrix.

Let us consider an  $m \times n$  matrix  $A$  of real numbers. An entry in a matrix  $A$  is called a saddle point for  $A$  if it is the smallest element in its row and the largest element in its column.

The saddle point is used in game theory to determine whether the players in a two-person zero-sum matrix-game have optimal strategies. The phrase zero-sum refers to the fact that on each move in the game one player's gain equals the other player's loss. The phrase matrix-game refers to the fact that each person has a fixed finite number of possible moves and that the game can be modeled by a matrix whose entries are the payoff for each pairing of moves.

We can find the smallest element in a row of matrix  $A$  by evaluating the `Min[ ]` function on the row. Mapping the `Min[ ]` function across the rows of the matrix gives us the list of minimal elements of all the rows.

```

Clear[A]
A = {{5, 6, 4, 4}, {0, 7, 1, 2}, {7, 5, 2, 7}};

Min[First[A]]

Map[Min, A]

```

Observe that a minimal value can occur several times in a single row.

We can find the maximum value in each column of the matrix A by transposing the matrix and then mapping the `Max[]` function across the rows of the transpose.

```
Map[Max, Transpose[A]]
```

We collect the two computations for the extremal values into two functions, each with a matrix as the single parameter.

```

Clear[uRowMins, uColMaxs]
uRowMins[m_] := Map[Min, m]
uColMaxs[m_] := Map[Max, Transpose[m]]

```

For the matrix A these two functions produce the following lists:

```

uRowMins[A]
uColMaxs[A]

```

Once we have the two lists of row minima and column maxima we can find a saddle point value by matching every row minimum with every column maximum. This can be done with the `Outer[]` function. First we show a few sample invocations of the `Outer[]` function.

```

?Outer

Clear[x, y, z]
Outer[Plus, {x, y, z}, {1, 2, 3, 4}];
TableForm[%]

Outer[Equal, {1, 2, 3, 4, 5}, {5, 1, 4}];
TableForm[%]

```

This last form we use to find the saddle points of the matrix A.

```

Clear[S]
S = Outer[Equal, uRowMins[A], uColMaxs[A]];
TableForm[S]

```

Every entry whose value is `True` in this Boolean valued matrix S represents a saddle point of the matrix A. We need to get the index values for the positions of the saddle points.

```

?Position

Position[A, 7]

Clear[p]
p = Position[S, True]

```

So far we have computed the position of the single saddle point of A. We also might want to compute the unique value at the saddle point. To do this we can pick the first position pair from the position list p and compute the value of that entry for the matrix A.

```
First[p]
```

To extract a value from the matrix A we must change the list returned by First[ ] indicating a position into a sequence of row and column indices. This is accomplished by applying Sequence[ ] to the list.

```
?Sequence
```

```
Apply[Sequence, First[p]]
```

```
A[[Apply[Sequence, First[p]]]]
```

**Example 17:** Manipulating highly nested and unevenly nested lists.

We will demonstrate two additional list manipulating functions, the Depth[ ] function and the Map[ ] function, acting on deeper levels of a list. First, we show the action of these functions on a regular list, namely a  $2 \times 3 \times 2$  matrix. After that, we construct a short sample list whose elements are lists that are nested to various levels.

```
Clear[t232]  
t232 = Table[i + j + k, {i, 1, 2}, {j, 1, 3}, {k, 1, 2}]  
MatrixForm[t232]
```

We use the Map[ ] function at a specific level of the list. There are two possible level specifications for the Map[ ] function: the first causes action on levels 1 through some specified level k and the other acts on just that level k. We take the functions Head[ ], Point[ ], and Line[ ] to show this behavior of Map[ ].

```
?Map
```

```
Map[Head, t232, {3}]
```

```
Map[Head, t232, {2}]
```

```
Map[Head, t232, {1}]
```

In the next expressions, we get two lines and a  $2 \times 3$  matrix of Point objects.

```
Map[Line, t232, {1}]
```

```
Map[Point, t232, {2}]
```



We now create a short list of six (top-level) elements where each element has a different nesting level. For that purpose we evaluate the `FoldList[ ]` function on a flat list of symbols as our starting point and with the function `uOrderedPair[ ]`. This function will form a pair of its two arguments and enclose the second component in an additional layer of list braces.

```
Clear[uOrderedPair]
uOrderedPair[u_, v_] := {u, v}

Clear[sample, a, b, c, d, e]
sample = FoldList[uOrderedPair, {a}, {b, c, d, e}]

Length[sample]
```

We can see the individual elements of the list `sample` much better by using the function `TableForm[ ]`. However, since the elements in the list `sample` have various nesting levels we control the stripping of the list parentheses for display purposes with the option `TableDepth` in `TableForm[ ]`.

```
??TableDepth

TableForm[sample, TableDepth → 1]

TableForm[sample, TableDepth → 2]
```

The nesting level of the entire list `sample` and the nesting level of each of its elements can be determined with `Depth[ ]`.

```
?Depth

Depth[sample]

Map[Depth, sample]
```

Now we map the `Depth[ ]` function at deeper levels of the list.

```
Map[Depth, sample, {1}]

Map[Depth, sample, {2}]
```

Finally, we show how we can extract individual values from various depths of the list `sample`.

```
TableForm[sample, TableDepth → 1]

sample[[1]]

sample[[4]]

sample[[4, 1]]

sample[[4, 1, 1]]
```

```
sample[[4, 1, 1, 1, 1]]
```

```
sample[[5, 1, 1, 1, 1, 1]]
```

We needed five additional indices 1, 1, 1, 1, and 1 to get to the value most deeply nested in the list `sample[[5]]`. This corresponds to the fact that the depth of the fifth element in the list `sample` is six.

```
Depth[sample[[5]]]
```

**Example 18:** Obtaining the points that Mathematica computes in the `Plot[ ]` function.

The graphing functions in Mathematica incorporate a sophisticated and adaptive method to compute points on a graph so that the rendered curve appears smooth for a smooth function. We can get the coordinates of these points with the `InputForm[ ]` function.

```
?InputForm
```

As an example we choose the graph of one period of the sine function since its curvature changes rapidly enough around the values  $\frac{\pi}{2}$  and  $\frac{3\pi}{2}$  so that we can see Mathematica's adaptive computations.

```
Clear[gp]
```

```
gp = InputForm[Plot[Sin[x], {x, 0, 2Pi}]
```

We investigate this list with the `Depth[ ]` function globally and at several levels of the `Graphics[ ]` object.

```
Depth[gp]
```

```
Map[Depth, gp, {3}]
```

```
Map[Depth, gp, {5}]
```

We see that the list of coordinates of the points on the graph is the first argument and is nested five levels deep: `Graphics[ {{ Line[...], {...} ]}`.

```
gp[[1, 1, 1, 1, 1]]
```

We get an individual coordinate pair by going down into the list one additional level. Here is the third point in the list.

```
gp[[1, 1, 1, 1, 1, 3]]
```

We can access the list of options in the graphics command in a similar way—we just need to extract the second component at level 1.

```
gp[[1, 2]]
```

It is now simple to write a function that gets the coordinates for the 2D-graph of a function. We will define a function `uGetCoords[ ]` whose first parameter is a functional expression and whose second parameter specifies a range of arguments for the functional expression, just as in the `Plot[ ]` function. Note that the first parameter in the range specification must be a symbol, the variable in the functional expression. Since we do not want to see the graph of the function we use the `Identity` value for the `DisplayFunction` option.

```

Clear[uGetCoords]
uGetCoords[expr_, r: {_Symbol, _, _}] :=
  InputForm[Plot[expr, r, DisplayFunction →
    Identity]][[1, 1, 1, 1, 1]]

```

We evaluate this function for a few simple functions.

```

Clear[sc]
sc = uGetCoords[Sin[x], {x, 0, Pi}]
ListPlot[sc, PlotStyle → PointSize[0.01]];

```

The density of points is largest around the local maximum of the sine function.

```

ListPlot[uGetCoords[3t + 5, {t, -2, 3}], PlotStyle →
  PointSize[0.02]];

```

For a straight line, an equidistant subdivision is sufficient.

We can visualize the computations behind the graphics rendering of Mathematica when we take the standard `Plot[ ]` function to draw the graph of the `ArcTan[ ]` function and when we take the `uGetCoords[ ]` function to provide us with the points that Mathematica calculates for that plot. We then render these points with `ListPlot[ ]` and finally draw them on top of the graph for `ArcTan[ ]`.

```

Clear[at, p1, p2]
at = uGetCoords[ArcTan[x], {x, -20, 20}];
p1 = Plot[ArcTan[x], {x, -20, 20},
  PlotStyle → {{RGBColor[0, 1, 0], Thickness[0.015]}},
  DisplayFunction → Identity];
p2 = ListPlot[at, PlotStyle → {PointSize[0.012], RGBColor[1, 0, 0]},
  DisplayFunction → Identity];
Show[p1, p2, DisplayFunction → $DisplayFunction];

```

We get a different view of the graphics when we draw the vertical lines from the points that Mathematica computes for a plot to the x-axis as we did in Example 9. Here we incorporate an alternate implementation for the computations performed by the function `uVerticalLines[ ]` from that example.

```

Show[Graphics[Map[Line[{{First[#], 0}, #}] &, uGetCoords[Sin[x]^2,
  {x, 0, Pi/2}]]]];

```

Here is another example with an oscillating function that has a varying period.

```

Show[Graphics[Map[Line[{{First[#], 0}, #}] &, uGetCoords[Sin[x^2],
  {x, 0, Sqrt[2Pi]}]]]];

```

## *Exercises*

---

These exercises ask you to create lists, to compute values from existing lists, or to manipulate lists. Think about the computations in terms of functions that apply to entire lists, such as the list constructors `Map[ ]` and `FoldList[ ]` that step through lists. Complicated computations are achieved by a hierarchy of function applications, that is, by evaluating a function on the results of another function.

### **EXERCISE 1:** Creating lists with the `Map[ ]` and `Table[ ]` functions.

Choose at least three significantly different mathematical functions, each having a single parameter.

- (a) Construct a table of values for each of the functions where the argument values occur at regular intervals.
- (b) Wherever you used `Map[ ]` in part (a), now use `Table[ ]`, and vice versa.
- (c) Construct a table of values for each of the functions where the argument values are chosen at random.
- (d) For each of the functions, generate a list of values for the function and for its first and second derivatives.
- (e) Choose some number as a fixed lower bound for an interval of integration and create a list `L` of numbers where you treat the numbers in the list `L` as the upper bound of integration. Compute the values of the definite integrals over these bounds for each of the functions you chose for the earlier parts of this exercise.

### **EXERCISE 2:** Creating a picture of randomly placed ellipses.

Circles and ellipses can be drawn with the graphics primitive `Circle[ ]`. Filled circles and ellipses can be drawn with the graphics primitive `Disk[ ]`.

- (a) Change the function `uRandomCircles[ ]` in Example 8 to draw filled circles.
- (b) Extend Example 8 to ellipses instead of circles.
- (c) Extend part (b) to filled ellipses instead of filled circles.
- (d) In Example 8 the centers of the circles were limited to a  $10 \times 10$  square integer lattice in the first quadrant by the function `uRI[ ]`. Change the definition of the function `uRandomCircles[ ]` in Example 8 so that you can specify the rectangular range in which to draw the circles as a second argument for the function.
- (e) Extend the definition of your function from part (d) to ellipses and filled circles.
- (f) Include among the graphics specifications of your functions in parts (b) and (d) the `Thickness[ ]` primitive so that different circles and ellipses are drawn in various widths.

### **EXERCISE 3:** Using functional programming in computations with matrices.

The objective in this exercise is to use the list processing functions to full advantage. In many cases the resulting function will just be a composition of several functions.

- (a) Write a function that returns the diagonal of a square matrix using the `Map[ ]`, the `Table[ ]`, or the `Array[ ]` function.
- (b) Write a function `uMatrixTrace[ ]` that computes the trace of a square matrix. Recall that the trace of a matrix is the sum of its diagonal elements.
- (c) In linear algebra, the length or norm of a vector usually refers to the standard Euclidean distance. Write a function that computes the Euclidean length of the diagonal of a square matrix.
- (d) Write a function `uRangeQ[ ]` that determines whether a real number  $x$  is in a given interval  $a \leq x \leq b$ . Use `uRangeQ[ ]` to write a function `uMatrixRangeQ[ ]` that tests whether each entry in an  $m \times n$  matrix is in a given interval  $a \leq x \leq b$ . Your function `uMatrixRangeQ[ ]` should have a matrix  $A$  and a range  $\{a, b\}$  as its two arguments, and should return one of the Boolean values, `True` or `False`. Hint: The built-in function `Flatten[ ]` may be useful for the implementation of `uMatrixRangeQ[ ]`.

#### EXERCISE 4: Manipulations with a list of digits.

All the questions in this exercise refer to lists of digits of  $\pi$ . You may use any of the functions that we defined in Example 12 in your answers.

- (a) Write a function `uDigitsOfPi[ ]` that takes a single range argument  $\{m, n\}$  and returns the list of digits of  $\pi$  starting with the  $m^{\text{th}}$  digit and ending with the  $n^{\text{th}}$  digit.
- (b) Investigate the built-in function `Select[ ]` and use it to return the list of all even digits in a list of digits.
- (c) Write a function `uToInteger[ ]` that takes a list of digits and returns the number containing exactly these digits. For example, `uToInteger[{2,5,3,7,4,3,2}]` returns 2537432. Hint: Consider how to combine an integer and a digit to a number with one more digit.
- (d) Given a list  $L$  of  $n$  digits and a divisor  $k$  of  $n$ , write a function `uToIntegerList[ ]` that takes the list  $L$  and the number  $k$  as its two arguments and returns  $\frac{n}{k}$   $k$ -digit numbers. Test for the two actual input values that  $k$  is a divisor of  $n$ . Hint: One way to implement the function is to cut up the original list  $L$  into a list of  $\frac{n}{k}$  sublists and then to apply `uToInteger[ ]` to the sublists.

#### EXERCISE 5: Patterns in the sum of the first $n$ squares.

- (a) Define two functions that compute the sum of the squares of the first  $n$  integers in two different ways. Use the `Map[ ]` and `Apply[ ]` functions for one implementation and the dot product for the other.
- (b) Estimate the largest value of  $n$  for which each function in part (a) can be computed on your computer and then perform the computation. If you are working on a Macintosh, indicate the size of memory that you allocated to the Mathematica kernel. Beware that the kernel may terminate if you overestimate.

- (c) It is well known that  $1 + 4 + 9 + \dots + n^2 = \frac{n}{6}(n+1)(2n+1)$ . Use this formula to write a more efficient function that computes the sum of the squares of the first  $n$  integers. Evaluate this function on arguments that are orders of magnitude larger than the upper bound that you established in part (b).
- (d) Use the function from part (c) to create a table of values for  $n = 10^i$  where  $1 \leq i \leq 6$ . The digits of the sums seem to follow a fixed pattern. Predict the sum for  $n = 10^7$  and for  $n = 10^8$ . Check to verify you were right.
- (e) Describe the pattern of digits for  $n = 10^i$ ,  $1 \leq i \leq m$ , and give reasons why this pattern occurs.
- (f) Use the pattern of digits for  $n = 10^i$  to build a list of the digits in the pattern. Implement this list construction as a function of the single argument  $i$  that returns the digits of the sum of the squares of the first  $10^i$  numbers without actually computing the sum.

**EXERCISE 6:** Using the points of a plot in Mathematica to approximate the definite integral.

In this exercise we will use the function `uGetCoords[ ]` from Example 18 to give us points on a graph that we then use to approximate the definite integral over the specified range. We will use Riemann sums and trapezoids for the approximations.

```
Clear[uGetCoords]
uGetCoords[expr_, r: {_Symbol, _, _}] :=
  InputForm[Plot[expr, r, DisplayFunction -> Identity]]
[[1, 1, 1, 1, 1]]
```

- (a) Write a function `uMapList[ ]` that takes two lists of equal length, a list  $\{f_1, f_2, \dots, f_n\}$  of functions and a list  $\{v_1, v_2, \dots, v_n\}$  of values, and that returns the list  $\{f_1(v_1), f_2(v_2), \dots, f_n(v_n)\}$ . For example, the invocation `uMapList[{Sqrt, Log[10, #]&, Point}, {49, 1000, {1, 2}}]` should produce the list `{7, 3, Point[{1, 2}]}`.
- (b) Given is a list of numbers  $\{x_1, x_2, \dots, x_n\}$  that you may think of as the points of a subdivision of the interval  $x_1 = a \leq b = x_n$ . Write a function `uWidth[ ]` that returns the list of the lengths of the subdivision, that is, the  $(n-1)$ -element list  $\{x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}\}$ .
- (c) Given is a list of pairs of numbers such as the list of coordinates that the function `uGetCoords[ ]` from Example 18 produces,  $\{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\}\}$ . Write a function `uLeftHeight[ ]` that takes such a list as its input and returns the list  $\{\{x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}\}, \{y_1, y_2, \dots, y_{n-1}\}\}$  of widths of the subintervals defined by the  $x$ -coordinates of the points and the heights at the left endpoints of the subintervals. The two functions `uMapList[ ]` and `uWidth[ ]` may be useful in the implementation of `uLeftHeight[ ]`.
- (d) If you take the pair of lists that you computed in part (c) and apply the dot product to the two lists, you will produce an approximation of the definite integral with the left Riemann sums. Write a function `uLeftRiemann[ ]` that implements these computations and that has two parameters: the first is some real-valued function expression and the second specifies an interval. Therefore, its interface is as in `Plot[ ]`. Choose several functions and intervals for them and

approximate the definite integrals with `uLeftRiemann[ ]` as well as with `NIntegrate[ ]`. Compare and explain your results.

(e) Write a function `uRightHeight[ ]` that takes the list  $\{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\}\}$  as its input and returns the list  $\{x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}\}, \{y_2, y_3, \dots, y_n\}$ .

(f) Write the function `uRightRiemann[ ]` in the manner of part (d). Use your function of part (e).

(g) Write a function `uTrapezoidHeight[ ]` that takes the list  $\{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\}\}$  as input and that has the list  $\{x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}\}, \{\frac{y_1 + y_2}{2}, \frac{y_2 + y_3}{2}, \dots, \frac{y_{n-1} + y_n}{2}\}$  as its return value.

(h) Write the function `uTrapezoid[ ]` in the manner of part (d). Use your function of part (g).

### EXERCISE 7: Plotting colored points in three-dimensional space.

We want to use the three coordinates of a point in space to color the point. A point on the x-axis is red, on the y-axis green, and on the z-axis blue. Since the arguments for `RGBColor` must be between 0 and 1 we use the coordinates of the point and the distance of the point from the origin in computing the color values for the point. We copy the function `Euclidean[ ]` from Example 13.

```
Clear[uEuclidean]
uEuclidean[v_?VectorQ] := N[Sqrt[Fold[(#1 + #2^2) &, 0, v]]]
```

(a) Write a function `uRandomPoints[ ]` that takes two arguments. The first is the number of points to be generated. The second is a range argument  $\{a, b\}$  for the coordinates of the points. Therefore, all points will be inside the cube from  $(a, a, a)$  to  $(b, b, b)$ .

(b) Write a function `uPointColor[ ]` that takes a point  $(x, y, z)$  as its argument and returns the color value `RGBColor[ $\frac{|x|}{d}, \frac{|y|}{d}, \frac{|z|}{d}$ ]`, where  $d = \sqrt{x^2 + y^2 + z^2}$ . Hint: Use the `Apply[ ]` and `uEuclidean[ ]` functions.

(c) Given is a list  $L = \{p_1, p_2, \dots, p_k\}$  of  $k$  random points. Compute a color for each point according to part (b). Write a function `uColoredPoints[ ]` that takes the list  $L$  as its input and returns the list of pairs  $\{uPointColor[p], Point[p]\}$ .

(d) Write a function `uPlotColoredPoints[L]` that takes a list of points, specified as coordinate triples, and plots the points in the colors computed in part (c). Experiment with additional graphics primitives for `Graphics3D[ ]` and the options of `Show[ ]` to get an optimal graphics rendering.

(e) Write a function `uPlotColoredPoints[k, {a, b}]`, overloading the function of part (d), that has the same interface as `uRandomPoints[ ]` of part (a) and plots the  $k$  points in the colors computed in part (c). Experiment with additional graphics primitives for `Graphics3D[ ]` and the options of `Show[ ]` to get an optimal graphics rendering.

### EXERCISE 8: Numerical experimentation, symbolic computations, limits, and formal proofs.

Here is a detailed formulation of the question that we want to address in this exercise. We fix an interval of consecutive integers, say  $\{1, \dots, n\}$ , and randomly choose numbers (with replacement) from that interval  $k$  times, and collect those  $k$  numbers. In this selection process we are

likely to choose some numbers several times and other numbers not at all. Each of the different numbers chosen is called a hit. If there are  $s$  hits we call the ratio  $\frac{s}{n}$  the hit-ratio. The numbers not selected from  $\{1, \dots, n\}$  are called misses. If there are  $s$  hits then there are  $n-s$  misses.

- (a) Write a function `uChoose[n, k]` that chooses randomly  $k$  times from the interval  $\{1, \dots, n\}$ .
- (b) Use the `Union[ ]` and `Complement[ ]` functions to compute sets of hits and misses. Experiment with various values of  $n$  and  $k$ .
- (c) Combine your computations of parts (a) and (b) into a function `uHits[n, k]` that returns a list of the hits for  $k$  random choices from the interval  $\{1, \dots, n\}$ .
- (d) Write a function `uHitRatio[n, k]` that returns the hit-ratio for  $k$  random choices from the interval  $\{1, \dots, n\}$ . Evaluate `uHitRatio[ ]` on a wide variety of numbers. Do you see any patterns?
- (e) Consider the problem now from a theoretical perspective. Find a recursive formula for the expected number of hits for  $k$  random choices from the interval  $\{1, \dots, n\}$ . Implement your formula as the recursive function `uExpectedHits[n, k]`.
- (f) Note that the value of the expression  $\frac{\text{uExpectedHits}[n, k]}{n}$  is the theoretical value for the experimental value `uHitRatio[n, k]`. Implement the expression as the function `uHitsProbability[n, k]`.
- (g) Compute a  $10 \times 10$  table of values of `uHitsProbability[n, k]` where  $n, k \in \{10, 20, \dots, 100\}$ .
- (h) Find a closed form expression for `uHitsProbability[n, k]`. Hint: It may be easier to compute the expression  $1 - \text{uHitsProbability}[n, k]$ . Use a symbolic value  $n$  for its argument. Make a table for some specific values of  $k$ .
- (i) Verify the closed form by induction using the recursive definition for `uHitsProbability[ ]`. Use Mathematica's symbolic computation capability to prove the correctness of the closed form.
- (j) Investigate the function `uHitsProbability[n, n]`. Use the built-in function `Limit[ ]` to determine the asymptotic behavior of the function `uHitsProbability[n, n]` as  $n \rightarrow \infty$ .

#### EXERCISE 9: Computations with the rows and columns of a matrix.

In this exercise you will use the function `uRandomMatrix[ ]` from Example 6 and the function `uEuclidean` from Example 13. We copy the two definitions.

```
Clear[uRandomMatrix]
uRandomMatrix[{rows_, columns_}, type_, {min_, max_}] :=
  Table[Random[type, {min, max}], {rows}, {columns}]

Clear[uEuclidean]
uEuclidean[v_?VectorQ] := N[Sqrt[Fold[(#1 + #2^2) &, 0, v]]]
```

The norm of a vector is the length of a vector according to a specified rule of computation. The maximum norm defines the largest coordinate of the vector as its length. The Euclidean norm defines the Euclidean length of the vector as its length.

- (a) Write a single function to compute the maximum norm for all rows of a matrix. The matrix should be the only input data.



- (b) Write a single function to compute the Euclidean norm for all rows of a matrix. The matrix should be the only input data.
- (c) Write a single function to compute the maximum norm for all columns of a matrix. The matrix should be the only input data.
- (d) Write a single function to compute the Euclidean norm for all columns of a matrix. The matrix should be the only input data.

### EXERCISE 10: Linear transition between two color specifications.

This exercise is an extension of Example 9. Suppose you are given two color specifications in terms of the triples of numbers between 0.0 and 1.0, as required for the `RGBColor[ ]`. Compute a linear transition between the two colors according to a specified number of intermediate steps.

- (a) Given two numbers  $a$  and  $b$ , compute  $n \geq 2$  equidistant values between  $a$  and  $b$ , including the bounds  $a$  and  $b$ . Define this computation as a function `uSteps[n,{a,b}]` that returns the list of  $n$  values.
- (b) Let  $c_1 = \{c_{11}, c_{12}, c_{13}\}$  and  $c_2 = \{c_{21}, c_{22}, c_{23}\}$  be two triples of color specifications, that is,  $0.0 \leq c_{ij} \leq 1.0$ , for all  $1 \leq i \leq 2$  and  $1 \leq j \leq 3$ . Write a function `uColorTransition[n,c1,c2]` that returns the list of  $n$  triples for a linear color transition from color  $c_1$  to color  $c_2$ . Use your function `uSteps[ ]` from part (a).
- (c) Use your function `uColorTransition[n,c1,c2]` from part (b) to define a function `uColorSteps[n,c1,c2]` that creates a list of  $n$  `RGBColor[ ]` specifications.
- (d) Define a function `uColorPlot[pts,c1,c2]` that creates a list of pairs from the list of points  $pts$  and the two color triples  $c_1$  and  $c_2$ . Use `Show[Graphics3D[...]]` in your function. Plot several combinations of lists of points and color transitions.

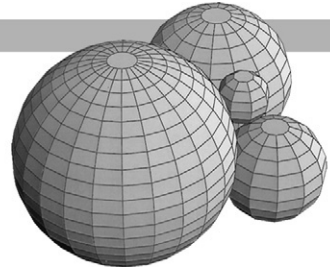
### EXERCISE 11: Properties of Mathematica's plotting algorithm.

This exercise is based on Example 18. We copy the function `uGetCoords[ ]` from that example.

```
Clear[uGetCoords]
uGetCoords[expr_, r : {_Symbol, _, _}] := InputForm[Plot[expr, r,
DisplayFunction->Identity]] [[1, 1, 1, 1, 1]]
```

- (a) Write a function `uXCoords[ ]` that has the same interface as `uGetCoords[ ]`, but returns the list of x-coordinates of the points on the graph. Apply `uXCoords[ ]` to three functions and intervals for plotting.
- (b) Write a function `uMinStep[ ]` that has a list of numbers, sorted in ascending order, as its only argument and computes the minimum distance between two adjacent numbers in the list.
- (c) Combine your functions from parts (a) and (b) to define a function that has the same interface as `uGetCoords[ ]` and returns the minimum distance between the x-coordinates of the points on the graph. Name this function also `uMinStep[ ]`. There can be no conflict since the two functions have different interfaces.
- (d) Discuss the efficiency of computations in the two versions of `uMinStep[ ]` of parts (b) and (c).

# *Iterations with Loops*



## *Introduction*

Many computing tasks consist of an iteration, or looping process, that is, a sequence of repeated evaluations of some set of actions. The list processing functions, such as `Map[ ]` and `Fold[ ]`, discussed in the notebook `ListFcts.nb`, “List Processing Functions,” are examples of functions that control iteration tasks. In addition, Mathematica supports three looping constructs in its programming language, the `Do[ ]`, `While[ ]`, and `For[ ]` functions.

Most programming languages incorporate several loop constructs. Unfortunately, the precise structure of the loops is different in different languages. All looping functions require counters, increments, or Boolean expressions to control the iteration of the loop. We will demonstrate the structure of each of the three types of loops in Mathematica with several examples. In this notebook we show how the loop constructs can be used interactively. We will use the `Print[ ]` function extensively to document the behavior of various loop constructs.

The heart of Mathematica’s programming language is functional and provides many operations, including iterations, as we saw in the notebook `ListFcts.nb`, “List Processing Functions.” Since it is frequently more natural to express a computation for a problem in terms of these functional operations, we discussed them before the more conventional `Do[ ]`, `While[ ]`, and `For[ ]` loops of object-oriented programming languages such as C++ and Java.

We can use any of the iteration functions to solve a computational problem. The particular choice of iteration function depends on the problem and the way we think about it. Usually there are many different paths to the solution of a problem and its implementation.

## *Using the Do[ ] Loop: Basics*

The first type of loop that we discuss is the `Do[ ]` function. It corresponds to the action: “Do this task so many times,” and is controlled by an iterator such as we have seen in the `Table[ ]` function. This iterator generates the values of an arithmetic sequence for the loop control variable that is determined by the initial value, the final value, and the step value of the iterator.

The syntax of the `Do[ ]` loop follows the syntax of the `Table[ ]` function: One or more action statements, a separating comma, and a controlling iterator specification. The action statements inside a loop are also called its body.

**?Do**

**Example 1:** Different forms of the iterator in a `Do[ ]` loop.

Loops with a single iteration variable.

```
Do[Print[i],{i, 6}]
```

```
Do[Print[i, " : ", i^2], {i, 3, 21, 4}]
```

A loop with two iterators.

```
Do[Print["i: ", i, " j: ", j, " 100i + j: ", 100i + j], {i, 4, 7}, {j, 0, 2}]
```

In the common procedural programming languages such as Pascal, C, or Ada, and in the object-oriented languages such as C++ and Java, the double iterator does not exist and must be implemented with a pair of nested loops. We can imitate this nesting of loops in Mathematica also.

```
Do[
  Do[
    Print["i: ", i, " j: ", j, " 100i + j: ", 100i + j], {j, 0, 2}
  ],
  {i, 4, 7}
]
```

Note that the entire inner `Do[ ]` loop is a single action statement inside the outer `Do[ ]` loop. You should also note that the counter `i` as the first iterator in the single `Do[ ]` is the outer iterator in the nested `Do[ ]`. The outer loop is acted upon first!

In order to visualize and exhibit the nested structure in this loop, we use the following layout convention:

```
Do[
  Action statement(s),
  Iterator
]
```

Type (do not copy and paste) this nested loop into another notebook and observe how Mathematica lines up items at the correct level of indentation when you enter a `RET` to start a new line.

As an alternative, we can type the entire loop without any `RET`s, but then the structure of the statement is difficult to understand.

```
Do[Do[Print["i: ", i, " j: ", j, " 100i + j: ", 100i + j],
  {j, 0, 2}], {i, 4, 7}]
```

**Example 2:** The evaluation mechanism for the `Do[ ]` iterator.

The iteration variable (loop counter) is local to the `Do[ ]` loop and, therefore, is independent of any value that you may have assigned to the symbol before you evaluate the loop.

```

i = 17
Do[Print[i], {i, +3, -4, -2}]
i

```

**Example 3:** Use of the semicolon in the body of a loop.

Frequently we need to perform several tasks inside a loop, for example, make an assignment and also print a value. Sequences of computations are separated by semicolons:

Do[ first task; second task; ... ; last task, iterator specification ]

To show this syntax we use the doubly nested loop of Example 1. We compute the sums of the values generated by the inner loop and print the value of the sum in the outer loop. This time we use a different expression and different iterator ranges.

```

Clear[s]
Do[s = 0;
  Do[s = s + 10 i + j;
    Print["i: ", i, " j: ", j, " 10i + j: ", 10i + j],
    {j, 2, 3}
  ];
  Print["The sum equals: ", s];
  Print["-----"],
  {i, -2, 1}
]

```

**Example 4:** The Do[ ] loop and the Null symbol.

In the loops of Examples 1 and 2 we used the Print[ ] function to see the effects of the loop control. You may have observed that the answers for each invocation of Print[ ] are printed in a cell, a Print cell, but that there is no output cell. The following statement does not return anything visible.

```
Do[i^2, {i, 5, 10}]
```

The iteration variable  $i$  is assigned six different values and  $i^2$  is evaluated six times. Nothing is done with the six computed values and they are discarded. However, a value is returned that we can inspect with FullForm[ ].

```
FullForm[%]
```

The value Null is returned as value of the Do[ ] function. This is reasonable since Do[ ] is a control structure that causes other statements to be evaluated.

```
?Null
```

The value of Null is Null.

```
Null
```

**Example 5:** The do-nothing or empty Do[ ] loop.

It is possible, and sometimes desirable, to write a Do[ ] loop whose body is evaluated zero times. Note that the expressions in the iterator always are evaluated when the loop is entered.

```
Clear[a, b]
a = 4; b = 7;
Print["a = ", a, " b = ", b]
Do[Print[i], {i, a, b}]
Print["a = ", a, " b = ", b]
```

When we switch the values of a and b we define an empty interval for the iterator and nothing is printed.

```
Clear[a, b]
a = 7; b = 4;
Print["a = ", a, " b = ", b]
Do[Print[i], {i, a, b}]
Print["a = ", a, " b = ", b]
```

**Example 6:** The trace of a square matrix.

The trace of a square matrix is the sum of its diagonal entries. We can compute the trace with a simple loop using one iteration variable that steps through the diagonal positions.

In order to easily generate square matrices we use a variant of the `uRandomMatrix[ ]` function that we introduced in Example 9 of the notebook `FunctDef.nb`, “Functions.” Here we add a parameter that specifies the number type for the entries in the matrix, such as `Integer` or `Real`. We arrange the three parameters according to the questions: What is the size of the matrix? What kind of numbers are the entries? What is the range for the values of the entries?

```
Clear[uRandomMatrix]
uRandomMatrix[size_Integer, type_, {min_, max_}]:=
Table[Random[type, {min, max}], {size}, {size}]
```

For the computation of the trace of a matrix, we first create a matrix and use its number of rows as the final value for the loop iterator. Observe that this involves a call to a function inside the loop iterator. The assignment `x+ = y` is a short form for the assignment `x = x + y`: “to x add y.”

```
Clear[t, m]
t = 0;
m = uRandomMatrix[4, Integer, {-5, 5}];
MatrixForm[m]
Do[t + = m[[i, i]], {i, 1, Length[m]}]
t
```

Here is another listing of the same computation with a matrix of real entries over a different range.

```

Clear[t, m]
t = 0;
m = uRandomMatrix[3, Real, {-2.5, 2.5}];
TableForm[m, TableAlignments → Right]
Do[t += m[[i, i]], {i, 1, Length[m]}]
t

```

Evaluate these two input cells with the two loops several times and check the answers.

## *Using the Do[ ] Loop: A Hula Hoop Animation*

In this and the next example we want to simulate a person who is gyrating a hula hoop around the waist. We model this situation by rotating a circle, representing the hoop, around a fixed disk, representing the person's waist. We will draw a sequence of images with the circle touching the disk at different and discrete positions on the rim of the disk. We use the animation tool by choosing **Cell ► Animate Selected Graphics** in the menu bar to create the appearance of continuous movement.

**Example 7:** Animation as iteration, some graphics preliminaries.

Circles and disks can be drawn with built-in functions.

**?Disk**

**?Circle**

We enliven our drawing by painting the disk red and the circle blue. To get circles rather than ellipses we also must specify the aspect ratio of the plot. We also draw a coordinate system so that we can see the absolute positions of the circle and the disk in the plane. Note that the graphics objects are drawn over the axes.

First we draw a disk of radius 1 inside and touching a circle of radius 2.

```

Show[Graphics[{RGBColor[1, 0, 0], Disk[{0, 0}, 1]}],
Graphics[{RGBColor[0, 0, 1], Circle[{0, -1}, 2]}],
AspectRatio → Automatic, Axes → True];

```

Here is a drawing of the disk touching the circle at the angle  $-\frac{3\pi}{4}$  with respect to the positive horizontal axis. Note the coordinate shift of  $\pi$  for the center of the circle since the center of the disk is the origin of the coordinate system, and therefore, the center of the circle is opposite the point of tangency of the circle with the disk.

```

Clear[x0, y0]
x0 = Cos[Pi/4];
y0 = Sin[Pi/4];
Show[Graphics[{RGBColor[1, 0, 0], Disk[{0, 0}, 1]}],
Graphics[{RGBColor[0, 0, 1], Circle[{x0, y0}, 2]}],
Graphics[{PointSize[0.02], RGBColor[0, 0, 1], Point[{x0, y0}]}],
AspectRatio → Automatic, Axes → True];

```

Following is a drawing of the disk and the circle together with the center of the circle and the two radii used in the computations for the coordinates of the center of the circle.

```
Clear[x0, y0, x1, y1]
x0 = Cos[Pi/4]; x1 = Cos[5Pi/4];
y0 = Sin[Pi/4]; y1 = Sin[5Pi/4];
Show[Graphics[{RGBColor[1, 0, 0], Disk[{0, 0}, 1]}],
Graphics[{RGBColor[0, 0, 1], Circle[{x0, y0}, 2]}],
Graphics[{RGBColor[0, 1, 0], Line[{x1, y1}, {0, 0}]}],
Graphics[{RGBColor[0, 1, 0], Line[{0, 0}, {1, 0}]}],
Graphics[{PointSize[0.02], RGBColor[0, 0, 1], Point[{x0, y0}]}],
AspectRatio → Automatic, Axes → True];
```

**Example 8:** Animation as iteration, an implementation with a `Do[ ]` loop.

We think of the disk as the fixed object and of the circle as the moving object. Therefore, all our coordinate calculations are relative to the center of the disk, not the center of the circle. That means that the center of the circle, as viewed from the center of the disk, initially is at the angle  $\pi$  with respect to the positive horizontal axis.

Now we have all the pieces to produce an animation for the rolling circle. Observe that the body of the loop consists of two statements, first the computation for the coordinates of the center of the circle, and then the graphic rendering of the disk and the circle. We choose a step size of  $\frac{\pi}{6}$  for the animation of a roll of the circle once around the disk. This results in 13 graphs.

Evaluate the next input cell. To animate the circle, select all the graphics output cells created during the evaluation of the `Do[ ]` loop and double-click on the grouping bracket to close the group of plots. The first plot of the group will be displayed on the screen. Now choose the animation tool **Cell ▸ Animate Selected Graphics** in the menu bar. Experiment with the three animation directions and with various animation speeds for the circle in the lower left-hand corner of the window. A click with the mouse anywhere will stop the animation.

```
Clear[x, y, t]
Do[{x, y} = {Cos[t + Pi], Sin[t + Pi]};
Show[Graphics[{RGBColor[1, 0, 0], Disk[{0, 0}, 1]}],
Graphics[{RGBColor[0, 0, 1], Circle[{x, y}, 2]}],
AspectRatio → Automatic], {t, 0, 2Pi, Pi/6}]
```

Since the circle encloses the disk it determines the plotting range, and the circle seems to be fixed while the disk seems to roll inside the circle. We draw the `Graphics[ ]` objects directly with the `Show[ ]` function and do not include coordinate axes. Therefore, the 13 graphs do not contain a fixed reference point at the same screen location. To get the illusion of the rolling circle, we must fix the disk; that is, we must fix the position of the origin of the coordinate system in each of the plots. We accomplish this with the `PlotRange` option.

```
Clear[x, y, t]
Do[{x, y} = {Cos[t + Pi], Sin[t + Pi]};
```

```
Show[Graphics[{RGBColor[1, 0, 0], Disk[{0, 0}, 1]}],
Graphics[{RGBColor[0, 0, 1], Circle[{x, y}, 2]}],
AspectRatio → Automatic,
PlotRange → {{-3, 3}, {-3, 3}}, {t, 0, 2Pi, Pi/6}]
```

If your computer has enough memory you may refine the animation by reducing the size of the drawing step to  $\frac{\pi}{10}$  or even smaller angular values. Select the graphics output cells, animate again, and you should get a smoother animation.

## *Using the While[ ] Loop: Basics*

---

The second type of loop that we discuss is the While[ ] function. It corresponds to the action: “Do this task as long as a given condition holds true.” The While[ ] loop is used for problems where the number of evaluations of a process is controlled by a property rather than by a simple counter. The controlling property is a Boolean expression, that is, an expression that evaluates to True or False. In many cases this Boolean expression is a comparison.

### **?While**

Note that in the While[ ] function the test condition is given first, followed by a comma, and followed by the action statements. This is the reverse of the Do[ ] loop syntax.

**Example 9:** A single comparison controlling a While[ ] loop.

We start with a simple and self-explanatory example of the While[ ] loop, which uses a single comparison of a variable. The assignment  $i^* = 2$  is a short form for the assignment  $i = i \cdot 2$ : “multiply  $i$  by 2.”

```
Clear[i, j]
i = 7; j = 1;
While[i < 10^3,
  Print["i=", i, " :: j=", j];
  i *= 2;
  j += 1
]
```

Observe that both variables  $i$  and  $j$  must have values before they can be used in the While[ ] loop. Usually, their values are changed by the evaluation of the loop, and the last value computed during the execution of the While[ ] loop is retained after the loop is finished.

```
i
j
```

Explain the values of  $i$  and  $j$  with respect to the values printed by the While[ ] loop.

Note also that semicolons must be used to separate statements in the body of the loop, just as in a Do[ ] loop.



When we change the comparison in the loop to  $i \geq 10^3$  this expression evaluates to False for the initial values of  $i$  and  $j$ . Therefore, the body of the While[ ] loop is not executed at all and the values of the symbols  $i$  and  $j$  remain unchanged.

```
Clear[i, j]
i = 7; j = 1;
While[i ≥ 10^3,
  Print["i = ", i, " :: j = ", j];
  i *= 2; j += 1]

i
j
```

**Example 10:** Simulating a sequence of bets playing Roulette.

Roulette is a game of chance. A wheel with small indented sections numbered from 0 through 36 is set into a bowl. The wheel can be spun around and while it is rotating a metal ball is thrown into the bowl. Bets are placed on the position at which the ball comes to rest when the wheel stops spinning.

Suppose we bet on the number 17 consistently; that is, we keep on betting 17 until the ball stops on 17. How many times do we have to bet until the ball rests on 17?

We solve this with a While[ ] loop using the random number generator Random[ ] as the controlling feature; after all, a true roulette wheel behaves in random fashion. We keep a counter  $n$  for the number of rolls of the roulette ball and inspect the value of the random number generator that represents the final position of the roulette ball.

It is important to initialize the counter  $n$  to 1 since the While[ ] loop tests the terminating condition before it executes its body.

```
Clear[n]
n = 1;
While[Random[Integer, {0, 36}] ≠ 17, n += 1];
n
```

Evaluate this loop several times.

**Example 11:** Running an experiment playing Roulette.

Once we have evaluated the loop in Example 10 a few times it becomes obvious that this manual repetition should be automated by another loop. Therefore, we put the three statements, initialization, the loop, and the final value of the counter inside another loop. This outer loop will be a Do[ ] loop since we can specify explicitly how many times we execute the inner While[ ] loop.

```
Clear[n]
Do[n = 1;
  While[Random[Integer, {0, 36}] ≠ 17, n += 1];
  Print[n], {10}]
```

To see how many times we have to spin the roulette wheel we print the counter  $n$  each time through the `Do[ ]` loop. Therefore, we need to reset the counter  $n$  each time.

Rather than printing the counter values we can also compile a list of them. We start with the empty list and append to it the value computed in the current iteration step.

```
?Append

Clear[n, L]
L = {};
Do[n = 1;
  While[Random[Integer, {0, 36}]  $\neq$  17, n += 1];
  L = Append[L, n], {10}]
L
```

To see the content of the list  $L$  we evaluate it; we do not need to `Print[ ]` it.

## *Using the While[ ] Loop: Termination Conditions and the Bisection Method*

---

It is possible that the Boolean expression that controls the `While[ ]` loop never changes its truth value. In this case, the loop is not executed at all when the condition initially evaluates to `False`, as in the second loop of Example 9, or it is executed forever when the condition initially evaluates to `True`. The latter situation can be avoided by specifying an upper bound for the number of times that a `While[ ]` loop executes. In this section we present two examples that incorporate such bounds.

**Example 12:** Enforcing the termination of an iteration process.

Imagine that we stand at the origin of the complex plane and jump to some point according to a given rule. As our rule we choose the quadratic function  $f(z) = z^2 + c$ , where  $c$  is some constant complex number and  $z$  is a complex variable. The initial jump, always is from 0 to  $c$ , the next jump to  $c^2 + c$ , and so on. We will stop jumping when we are far enough away from the origin, say 2 units or 10 units or 100 units. This iteration process is related to fractals and, in particular, to the set of complex numbers known as the Mandelbrot set.

There is one minor point in the computations. We want all quantities to be complex numbers, including zero.

```
Head[0]

Head[0. + 0.I]

Clear[z, c];
z = 0.0 + 0.0 I;
c = 0.3 + 0.6 I;
While[Abs[z^2 + c]  $\leq$  2, z = z^2 + c; Print[z]]
```

Because of our particular choice of the initial jump the loop terminated. We want to be sure that the While[ ] loop terminates for every value of the initial jump  $c$ . For that purpose we include a counter  $n$  that will permit at most 100 jumps in the plane. We also want to see whether the While[ ] loop terminates because we jumped too far away (normal termination) or because the counter reached our preset bound of 100 (forced termination).

```
Clear[z, c, n]
z = 0.0 + 0.0I;
c = 0.3 + 0.6I;
n = 0;
While[Abs[z^2 + c] < 2 && n < 100,
  z = z^2 + c;
  Print[z];
  n += 1
]
n
```

Observe that we need to use Print[ ] to see all the values of  $z$ , since  $z$  is computed inside the While[ ] function. We can just mention  $n$  to get its value since it appears at the top level of evaluation of Mathematica.

Evaluate this loop with several values for  $c$ . Find some  $c$  where  $n$  is zero, where  $n$  is small, and where the loop would evaluate forever, if it were not for the bound on the counter  $n$ .

Rather than printing the points in the sequence of jumps we store their values in a list. In contrast to the initialization in Example 11 we initialize the list  $L$  with  $0.0 + 0.0i$ , since that is the starting point for the sequence of jumps. Therefore,  $n$  needs to be initialized with 1.

```
Clear[z, c, n, L]
z = 0.0 + 0.0 I;
c = 0.3 + 0.6 I;
n = 1;
L = {z};
While[Abs[z^2 + c] < 2 && n < 100,
  z = z^2 + c;
  L = Append[L, z];
  n += 1
]
n
L
```

**Example 13:** Approximating a root of a function with the Bisection method.

Another important iteration process is that of approximating roots of functions. The number of iterations required to find a root depends on the complexity of the function, the initial distance from the root, and the precision required for the approximation. Therefore, a While[ ] loop seems to be an appropriate control structure for this problem. We shall use the Bisection method to approximate roots because its algorithm is simple. However, it imposes restrictions that do not apply to many other, more sophisticated, approximations.

Suppose that  $y = f(x)$  is a continuous real-valued function on the interval  $[a, b]$  and that  $f(a) \neq 0$  and  $f(b) \neq 0$  have opposite signs. Then we know by the Intermediate Value Theorem that the function  $f$  has a root  $r$  in the interval  $[a, b]$ , that is  $f(r) = 0$  and  $a < r < b$ . A simple and reasonably fast method to approximate this root  $r$  is the Bisection method. It proceeds by repeatedly cutting the interval  $[a, b]$  in half. We will terminate the approximation to the root when the absolute value of  $f(r)$  is zero to within 0.001 or when we have tried 30 approximation steps. The computation proceeds as follows:

1. Initialize  $a$  and  $b$ . We think of  $[a, b]$  as the current interval.
2. Initialize  $n$  to 0 and compute the midpoint  $r = \frac{a+b}{2}$  of the current interval.
3. When  $|f(r)| < 0.001$  or  $n \geq 30$ , the approximation stops and we display the result.
4. If  $f(a)$  and  $f(r)$  have the same sign we put  $a = r$  (move left bound) and continue with step 6.
5. If  $f(b)$  and  $f(r)$  have the same sign we put  $b = r$  (move right bound) and continue with step 6.
6. Compute the new midpoint  $r = \frac{a+b}{2}$ , increment  $n$ , and loop back to step 3.

In the next four input cells we define a function  $f$ , plot  $f$ , define interval bounds  $a$  and  $b$ , and then apply the Bisection method. Find all roots of  $f$  as indicated by the plot.

```
Clear[f]
f[x_] := x^3 - 3x^2 + 1

Plot[f[x], {x, -3, 3}];

Clear[a, b]
a = 0;
b = 1;

Clear[r, n]
n = 0;
r = N[(a + b) / 2];
While[Abs[N[f[r]]] ≥ 0.001 && n < 30, If[N[f[a] * f[r]] > 0,
  a = r, b = r];
  r = N[(a + b)/2];
  n = n + 1
]
n
{r, f[r]}
```

Observe that we do not test for the requirement  $f(a) * f(b) < 0$  before we begin the Bisection loop. Even if that condition is not true, for example with  $a = 1$  and  $b = 2$ , the loop terminates because of the bound on the counter  $n$ . However, a value for  $r$  is returned, but it is not a root.

Use the Bisection method to approximate the roots for the functions defined next. For each function, first evaluate the function definition. Then plot the function in an appropriate range

to find the general location of all roots. Then initialize the interval  $[a, b]$  and evaluate the cell containing the Bisection method for each root.

```
Clear[f]
f[x_] := 12x^5 - 12x^4 - 40x^3 - 20x^2 + 1

Clear[f]
f[x_] := (E^x - 2)/(x^2 + 1)
```

The Bisection method as we set it up previously can perform rather poorly when the slope of the function around a root is small. For the following polynomial function use a nonsymmetric interval around zero, for example  $a = -0.5$  and  $b = 1.0$ . What happens with a symmetric interval?

```
Clear[f]
f[x_] := x^9
```

## Using the For[ ] Loop: Basics

---

Mathematica implements a third type of loop, the For[ ] function. This is a hybrid of the Do[ ] and While[ ] loops and it is very general and versatile. In principle, all iterations can be written in terms of the For[ ] loop. The For[ ] loop construct is similar to the for-statement in the programming languages C++ and Java.

### ?For

The For[ ] function has four parameters that are separated by commas: the initialization, the termination test, the continuation expression, and the loop body. The initialization is evaluated once when the loop is entered. The remaining three parameters are evaluated repeatedly until the termination test evaluates to False. The For[ ] loop executes in the following cycle:

```
initialization →
termination test (True) → loop body → continuation expression →
termination test (True) → loop body → continuation expression →
... →
termination test (False)
```

In many applications the termination test is a simple comparison and the continuation expression is an increment or decrement.

**Example 14:** Computing approximate real number values for the sine function with For[ ].

We print a table of the values of the sine function using a For[ ] loop.

```
Clear[t]
For[t = 0.0, t ≤ N[Pi/2], t += N[Pi/12], Print[t, " :: ", Sin[t]]]
t
```

We get the same sequence of values also with the `Do[ ]` and the `While[ ]` loops. Observe the variations in syntax and make sure you understand where and why to place a comma and a semicolon.

```
Clear[t]
Do[Print[t, " :: ", Sin[t]], {t, 0.0, N[Pi/2], N[Pi/12]}]
t

Clear[t]
t = 0.0;
While[t ≤ N[Pi/2], Print[t, " :: ", Sin[t]]; t += N[Pi/12]]
t
```

Make sure you understand the value of the variable `t` after each of the iterations has terminated.

The three loops do not construct any lists. They compute sequences of values and print them. If we want a list of values then we should use the list constructors `Table[ ]` and `Map[ ]`.

```
Clear[t]
TableForm[Table[{t, Sin[t]}, {t, 0.0, N[Pi/2], N[Pi/12]}]]
t

TableForm[Map[{#, Sin[#]} &, Range[0.0, N[Pi/2], N[Pi/12]]]]
```

Note that the initialization, termination test, and continuation expression for the loop control variable `t` are done with approximate real numbers. We can do exact arithmetic in loops, using symbolic numbers like  $\pi$ , in the same manner as approximate arithmetic.

```
Clear[t]
For[t = 0, t ≤ Pi/2, t += Pi/12, Print[t, " :: ", Sin[t]]]
t
```

**Example 15:** Escaping from a circular region in the plane with a sequence of jumps.

In Example 12 we computed a sequence of jumps in the plane with a `While[ ]` loop. We can express that entire computation with a single `For[ ]` loop statement as follows:

```
Clear[z, c, n]
For[z = 0; c = 0.3 + 0.6I; n = 0,
  Abs[z^2 + c] < 2 && n < 100,
  z = z^2 + c; n++,
  Print[z]
]
{z, n}
```

Observe that we initialize three variables, separated by semicolons, that we use a compound test expression, and that we change two variables, one by evaluating a polynomial expression and the other by using the special incrementing operator `++`. The effect of `++` is to add one to its argument: `n++` means the assignment `n = n + 1`.

Evaluate this `For[ ]` loop for several initial values of `c`.

**Example 16:** Random increments and random termination in a `For[ ]` loop.

For the first computation in this example, we start at  $-\pi$  and add to it successively random numbers between 0 and 1. As soon as the accumulation reaches or exceeds 0, we terminate the loop, and return the accumulated value  $v$ , as well as the number of times,  $n$ , we invoked the random number generator.

```
Clear[v, n]
For[v = -N[Pi]; n = 0, v ≤ 0, v += Random[ ]; n++, Print[v]]
v
n
```

The body of the loop consists of the `Print[ ]` statement for the intermediate accumulated values. Since we are interested only in the final accumulated non-negative value, we can drop the `Print[ ]` statement together with the preceding comma and have an empty loop body.

```
Clear[v, n]
For[v = -N[Pi]; n = 0, v ≤ 0, v += Random[ ]; n++]
v
n
```

We can use the basic structure of this loop to measure the average excess value of the final value of  $v$  beyond 0.0. For this purpose we enclose the `For[ ]` loop in a `Do[ ]` loop where we add the excess value  $v$  from the `For[ ]` loop into variable  $s$ . We execute the `Do[ ]` loop 1000 times. Observe that for the action of the `Do[ ]` loop we do not need an explicit counter. Therefore, its iteration control consists of the specification `{1000}`.

```
Clear[s, v]
s = 0.0;
Do[
  For[v = -N[Pi], v ≤ 0, v += Random[ ], Null]; s += v,
  {1000}
]
s / 1000
```

Evaluate this doubly nested loop several times with starting value  $-\pi$  and starting values other than  $-\pi$ . Estimate the average excess accumulation.

It is good practice to leave a space holder, for example the symbol `Null` as we did earlier, when the body of the loop is empty. This will remind us that we really mean “nothing to evaluate.”

**Example 17:** Changing several control variables inside a `For[ ]` loop.

In the notebook `Assign.nb`, “Values, Variables and Assignments,” we introduced simultaneous assignments to symbols. We can use those in the initialization part of a `For[ ]` loop. We can also have a `For[ ]` loop modify several variables in its increment part. In the following loop we use the `If[ ]` function to decide which controlling variables to change.

We want to print out the elements of a matrix, scanning the rows. That means we control the loop with the row index  $i$  and we either increment the column index  $j$  (when the scan of

the current row is not complete yet) or we increment the row index  $i$  and reset the column index  $j$  to 1. This control decision is made in the second line of the `For[ ]` statement.

In order to quickly generate nonsquare matrices, we use the definition of the function `uRandomMatrix[ ]` from Example 6 in the notebook `ListFcts.nb`, “List Processing Functions.”

```
Clear[uRandomMatrix]
uRandomMatrix[{rows_, columns_}, type_, {min_, max_}] :=
  Table[Random[type, {min, max}], {rows}, {columns}]

Clear[m, n, A, i, j]
{m, n} = {2, 3};
A = uRandomMatrix[{m, n}, Integer, {-10, 10}];
MatrixForm[A]
```

To demonstrate the control action of this loop we print the current element of the matrix while in a row and a blank line when the loop control resets to the next row in the matrix.

```
For[{i, j} = {1, 1}, i ≤ m, If[j < n, j = j + 1, {i, j} = {i + 1, 1}],
  If[j == 1, Print[" "]]; Print[A[[i, j]]]
```

We could have put the assignment of values to  $m$ ,  $n$ , and  $A$  inside the `For[ ]` loop, in its initialization part. However, we think of those three objects as global data on which the `For[ ]` loop acts and that must be known before the loop is entered. The row and column counters  $i$  and  $j$ , on the other hand, are integral, or local, to the iteration process of the `For[ ]` loop and we initialize them inside the loop.

## *Using List Processing Functions as Alternatives for Loops*

---

In this section, we will take several of the problems posed in previous sections of this notebook and provide alternative solutions. We have two goals here. One goal is to use built-in iterators such as `NestList[ ]`, `Map[ ]`, and `Table[ ]` instead of the loop functions `Do[ ]`, `For[ ]`, and `While[ ]` as much as possible. The second goal is to minimize the number of variables necessary for the control of the loops.

How do we decide which control function to use in the implementation of an iteration process? If the process manipulates entire lists that are guaranteed to be finite, then the list processing functions should be explored first. The computation of function tables are examples of such lists. If the process needs several different kinds of conditions in order to guarantee termination, then loop constructs may be more appropriate. The Bisection method is such a process.



**Example 18:** Print a list of values with `Map[ ]` (Example 1).

In the following implementation we do not mention any iteration variable explicitly. We use a `Range[ ]` function instead and define an anonymous function with anonymous arguments to replace the `Do[ ]` loop of Example 1.

```
Map[Print[#, " : ", #^2] &, Range[3, 21, 4]];
```

**Example 19:** Accumulate a sum with `Apply[ ]` and `Map[ ]` (Example 3).

The following expression implements the inner `Do[ ]` loop in Example 3 for the specific value  $i = -2$ ; that is, we add terms of the form  $-20 + j$ . The terms in the sum are not printed; only the final value of the sum is returned.

```
Apply[Plus, Map[(-20 + #) &, Range[2, 3]]]
```

If we want to implement the outer `Do[ ]` loop from Example 3, then we first turn this computation for the nested loop into a function, `uIntermediate[ ]`, of one variable. Then we map `uIntermediate[ ]` over the range of the control variable of the outer loop.

```
Clear[uIntermediate]  
uIntermediate[i_] := Apply[Plus, Map[(10i + #) &, Range[2, 3]]]  
  
Map[uIntermediate, Range[-2, 1]]
```

**Example 20:** Generate a nonarithmetic sequence of numbers with `NestList[ ]` (Example 9).

We construct lists of specified lengths of successive powers of two and three with `NestList[ ]`.

```
?NestList  
  
NestList[(3 #) &, 1, 10]  
  
NestList[(2 #) &, 7, 5]
```

We cannot specify a termination property in `NestList[ ]`, only the number of applications. Therefore, the last invocation does not implement the loop of Example 9. However, we can get around this difficulty by first guessing a sufficiently large nesting level and then selecting an appropriate initial segment of the generated list with the built-in function `Select[ ]`.

```
?Select  
  
NestList[(2 #) &, 7, 15]
```

The computation in the following input cell implements the `While[ ]` loop of Example 9.

```
Select[NestList[(2#) &, 7, 15], (# < 10^3) &]
```

**Example 21:** Computing a list of sine values with the `Array[ ]` function (Example 14).

```
?Array
```

```
Array[ (# + 2) &, 5]
```

```
Array[ (# + 2) &, 5, 0]
```

The third argument in `Array[ ]` starts the indexing at zero, not at the default value one.

In the next two computations we replace the `For[ ]` loop of Example 14 by the iterator in the `Array[ ]` function.

```
Array[Sin[# Pi/12] &, 7, 0]
```

```
Array[{# Pi/12, Sin[# Pi/12]} &, 7, 0]
```

Note that in the `Array[ ]` invocations the last value  $\frac{\pi}{2}$  for the computations is hidden in the second argument value 7 and the step size of  $\frac{\pi}{12}$ .

We conclude by printing out the list of values as approximate real numbers in tabular and in matrix forms.

```
Array[N[{# Pi/12, Sin[# Pi/12]}] &, 7, 0];
TableForm[%]
MatrixForm[%]
```

## *The Collatz Function*

---

We demonstrate the `For[ ]` loop with an iteration of a function; that is, the value computed by a function is used as an argument for which to evaluate the function again. When we apply a function  $f$  repeatedly to itself we get a sequence of values  $x, f(x), f(f(x)), f(f(f(x)))$ , and so on.

**Example 22:** Iterating the Collatz function.

The Collatz function is an enigmatic function in Number Theory and the Theory of Computational Complexity. It is defined for integers as  $f(x) = \frac{x}{2}$ , if  $x$  is even, and  $f(x) = \frac{3x+1}{2}$ , if  $x$  is odd. This function maps positive integers to positive integers, negative integers to negative integers and zero to itself. Note that  $-1$  and  $0$  are fixed points, that is,  $f(-1) = -1$  and  $f(0) = 0$ . When we apply the function repeatedly to itself with different starting arguments  $x$ , then we get various sequences of integers.

We begin with a few computations of the sequences of values of the Collatz function using `NestList[ ]`.

```
NestList[If[EvenQ[#], # / 2, (3 # + 1) / 2] &, 1, 10]
NestList[If[EvenQ[#], # / 2, (3 # + 1) / 2] &, -5, 10]
NestList[If[EvenQ[#], # / 2, (3 # + 1) / 2] &, -15, 10]
NestList[If[EvenQ[#], # / 2, (3 # + 1) / 2] &, 7, 20]
NestList[If[EvenQ[#], # / 2, (3 # + 1) / 2] &, 12, 20]
NestList[If[EvenQ[#], # / 2, (3 # + 1) / 2] &, 2^15, 20]
```

Here is an interesting question about the Collatz function:

If we start with some integer  $k > 0$  and apply the Collatz function repeatedly to its result, do we eventually arrive at 1; that is, is every run finite?

We will use the `For[ ]` loop to generate examples of such sequences.

And further, how long are runs? What is the largest number in a run?

Since the iterated Collatz function can cycle, we must guard against an infinite loop. We limit the loop to 100 evaluations and use a loop counter `n` for the length of the list produced during the evaluation of the loop.

Note that the `NestList[ ]` function is short and well suited for experimental computation, but it always computes the indicated number of iterations. In the following `For[ ]` loop, we can and do give a condition for early termination of the loop when the cycling part of a run has been reached. This kind of iteration control is useful when we need to minimize the amount of computation.

Here are the actions in the `For[ ]` loop, line by line:

1. Clear: The start value `s`, the list `L` of iterates of the Collatz function, and the length `n` of the list.
2. Initialization: `s`, `L`, and `n`.
3. Termination test: Number 1 has not been encountered and the loop limit has not been reached.
4. Continuation expression: Increment the loop counter.
5. Loop body: Iterate the Collatz function.
6. Loop body: Append the last iterate of the Collatz function to the list.
7. After the loop: Length of the list of iterates.
8. After the loop: The list of iterates.

Evaluate the following input cell several times with different values for the starting argument `s` such as `-5`, `31`, or a power of 2.

```
Clear[s,L,n]
For[s = 17;L = {17};n = 1,
  s ≠ 1 && n < 100,
  n++,
  s = If[EvenQ[s],s/2,(3s + 1)/2];
  L = Append[L,s]
];
n
L
```

## Exercises

---

In each of the following exercises some iteration process needs to be implemented. When not specified otherwise, you should select the type of iteration that you think is most appropriate for the given problem.

You may use any of the list processing functions with their implicit iterators or the three loop constructs that we discussed in this notebook or any combination of list processing functions and loop constructs.

**EXERCISE 1:** An animation of cycloids.

A cycloid can be described by the parametric equations  $x(t) = t - r \sin t$  and  $y(t) = 1 - r \cos t$  where  $r$  is a constant. When  $r$  equals one, the curve represents the trace of a point on the rim of a unit circle as the circle rolls along the horizontal axis.

- (a) Plot the function for several values of  $r$ . Experiment with the aspect ratio of the plots.
- (b) Produce a set of graphs for some range of values for  $r$  and animate the results. Be sure that all graphs are drawn in the same coordinate rectangle.

**EXERCISE 2:** Playing dice.

- (a) Modify the computations in examples 10 and 11 from betting on a roulette wheel to betting that you will roll a six with a single die (six-sided die with numbers one through six).
- (b) Compute how often you have to roll a die until you roll a six followed by a one. Note that this question is very different from part (a) since it asks for the previous roll in addition to the current roll of the die.
- (c) Execute the implementations of parts (a) and (b) many times. Automate the process.
- (d) Give theoretical justifications for the experimental results of part (c).

**EXERCISE 3:** A flower on the hula hoop.

Consider the simulation of the hula hoop in Example 8 of this notebook. Suppose we mark the point on the rotating hoop that initially touches the stationary disk with a flower. We can represent the flower graphically as a fat point or a small disk. When the circle rotates, this flower will move through the plane on some curve (a cycloid).

- (a) For each of the angles for which the rotating circle is drawn, compute the position of the flower in the plane.
- (b) Expand the computations for the rotating hula hoop to include a drawing of the flower. Animate the result.

**EXERCISE 4:** Testing the transitivity of binary relations.

Every binary relation  $R$  on an  $n$ -element set can be represented by an  $n \times n$  matrix  $A$  with entries of 0 and 1. This matrix  $A$  is called the adjacency matrix of the relation and it is defined by  $A_{ij} = 1$  precisely when  $iRj$  holds and  $A_{ij} = 0$  otherwise. A binary relation  $R$  is called transitive if  $xRy$  and  $yRz$  imply  $xRz$ , for all  $x, y$ , and  $z$ .

- (a) Explore the built-in logic operators `And[ ]`, `Or[ ]`, `Xor[ ]`, `Not[ ]`, and `Implies[ ]`.
- (b) Formulate the condition of transitivity of a relation in terms of the entries of its adjacency matrix.
- (c) Use the `Do[ ]` loop to compute whether a given adjacency matrix is transitive.
- (d) Use the `While[ ]` loop to compute whether a given adjacency matrix is transitive.

- (e) Use the `For[ ]` loop to compute whether a given adjacency matrix is transitive.
- (f) Use a combination of list processing functions such as `Array[ ]`, `Flatten[ ]`, `Map[ ]`, `Outer[ ]`, or `Table[ ]` to compute whether a given adjacency matrix is transitive. Your goal should be to minimize or to eliminate all explicit variables in your implementation.

### EXERCISE 5: Polynomial approximation of the Gaussian bell curve.

The Gaussian bell curve is the graph of the function  $e^{-x^2}$ . Its Taylor expansion at the origin is the series

$$1 - x^2 + \frac{x^4}{2} - \frac{x^6}{6} + \frac{x^8}{2} - \frac{x^{10}}{120} + \dots$$

- (a) Investigate the built-in functions `Series[ ]` and `Normal[ ]`.
- (b) When we take a specific value of  $x$ , say  $x = 0.5$ , we can compute the successive partial sums of the Taylor series for that value of  $x$ . These partial sums give increasingly better approximations to the true value of  $e^{-0.5^2}$ . Implement this approximation to within a given accuracy, say three or six digits, and print out the approximate value and the number of steps in the iteration.
- (c) Compute approximations for some values of  $x$  with  $|x| < 1$ .
- (d) Compute an approximation for  $x = 1$ , that is, approximate  $\frac{1}{e}$ .
- (e) Compute approximations for some values of  $x$  with  $|x| > 1$ .
- (f) Compare the efficiency of the computations in parts (c) and (e) by comparing the number of approximations necessary to achieve a given accuracy.

### EXERCISE 6: Computations with the entries of vectors and matrices.

Use the `uRandomMatrix[ ]` function from Example 17 to generate sample  $m \times n$  matrices.

```
Clear[uRandomMatrix]
uRandomMatrix[{rows_, columns_}, type_, {min_, max_}] :=
Table[Random[type, {min, max}], {rows}, {columns}]
```

- (a) Write a function `uRandomVector[ ]` that generates a random vector. Design an interface that is similar to that of the function `uRandomMatrix[ ]`.
- (b) Let  $v$  be a vector with entries of 0 and 1, also called a 0-1 vector. A run of zeros in  $v$  is a maximal subsequence of zeros in  $v$ ; that is, it cannot be extended with zeros. Analogously, we define a run of ones. Consider the vector  $\{1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0\}$ . It has one run of zeros of length four, one run of zeros of length two, and three runs of zeros of length one. Furthermore, it has two runs of ones of length two and three runs of ones of length one. Implement an algorithm that returns the length of the first (left-most) run of zeros in a 0-1 vector.
- (c) Implement an algorithm that returns the length of the first (left-most) run of ones in a 0-1 vector.

- (d) Implement a computation that returns the lengths of all runs in a 0-1 vector.
- (e) Use as much of the implementation in part (d) as possible to compute the length of the longest run in a 0-1 vector, regardless whether it is a run of zeros or a run of ones.
- (f) Expand your implementation in part (e) to compute the length of the longest run in any row of an  $m \times n$  0-1 matrix.

**EXERCISE 7:** Approximations of the golden ratio.

- (a) Change the termination test in the implementation of the Bisection method in Example 13 so that the approximation to the root  $r$  is as accurate as the one to  $f(r)$ .
- (b) The golden ratio is defined as  $\frac{1+\sqrt{5}}{2}$ , the positive root of the quadratic equation  $x^2 - x - 1$ . Its reciprocal is used by Mathematica as the default value for the aspect ratio in two-dimensional plots. Use your implementation of the Bisection method from part (a) to approximate the two roots of  $x^2 - x - 1$  to four, six, and ten digits.
- (c) Compare the results of part (b) with the numeric values of the system variable `GoldenRatio` and of the fraction  $\frac{55}{34}$ .

**EXERCISE 8:** Plotting a list of complex numbers.

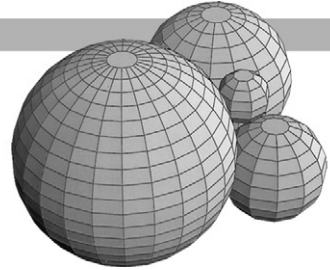
In Example 12 we calculated a list of complex numbers that represented a sequence of jumps starting from the origin with the quadratic rule  $f(z) = z^2 + c$  for each jump. In this exercise we will outline the steps to get from a list of complex numbers to their plot.

You may implement the computations in this exercise either with list functions or with loops.

- (a) Use the functions `Re[ ]` and `Im[ ]` to translate the complex number  $a + b i$  into the pair  $\{a, b\}$  of real numbers, the coordinates of the complex number.
- (b) Take your computation of part (a) and use it to translate an entire list of complex numbers into a list of coordinate pairs.
- (c) Take the last input cell from Example 12 to produce a list of complex numbers. Apply your function from part (b) to that list.
- (d) Take your resulting list from part (c) and wrap the graphics primitive `Point[ ]` around each pair. Hint: See Example 7 of the notebook `ListFcts.nb`, “List Processing Functions.”
- (e) Plot the results of part (d) with `Show [Graphics[. .]]`.
- (f) Put all computational pieces into a single input cell. Do not display the number of points computed or the actual list of points. Display only the final plot. Evaluate the input cell several times with different values for the constant  $c$ .
- (g) Embellish your plot by coloring the points, giving them different sizes, including coordinate axes or a coordinate frame.

This Page Intentionally Left Blank

# Computations with Modules and Local Variables



## Introduction

In the notebooks `ListFcts.nb`, “List Processing Functions,” and `Loops.nb`, “Iterations with Loops,” we discussed the interactive and immediate use of iteration functions. We now demonstrate how to write short programs with the `Module[ ]` function that perform iterative processes.

The main advantage of `Module[ ]` is that we can set up a computation whose details are hidden in the body of the `Module[ ]` function. For example, we can collect a sequence of computations in `Module[ ]`. We can also set up loops and their control variables locally with `Module[ ]`. In addition, `Module[ ]` can return values computed in its body, just like any other function we define in Mathematica.

The programming style embodied in this use of `Module[ ]` is called procedural programming. When only function evaluation is used, frequently with list processing functions, the programming style is called functional programming.

We use some of the examples from the two notebooks just mentioned. This time we use `Module[ ]` to define functions with appropriate parameters for the actions that were performed interactively in those notebooks.

## Using the `Module[ ]` Function: Basics

The general form of the `Module[ ]` function is `Module[{x, y, z, ...}, body]`. The symbols `x`, `y`, `z`, ... are called local variables of the `Module[ ]` function. They are objects hidden inside the `Module[ ]` function and not accessible from outside. The body represents the sequence of statements, separated by semicolons, that perform the computations in `Module[ ]`. We frequently use the form `Module[{x =  $x_0$ , y =  $y_0$ , z =  $z_0$ , ...}, body]` to assign initial values to the local variables. Typically, the expressions  $x_0$ ,  $y_0$ ,  $z_0$ , ... are constants. They may contain global variables, that is, variables already defined in the current Mathematica session, input parameters, and function evaluations.

### ?Module

The local variables do not conflict with global symbols having the same names. The extent of their meaning is the body of `Module[ ]`. Most importantly, the local variables are created when the `Module[ ]` function is entered and their values and names are destroyed when the computation in `Module[ ]` terminates.



**Example 1:** Playing Roulette, again.

We refer back to Examples 10 and 11 in the notebook `Loops.nb`, “Iterations with Loops.” Here we design a function `uRoulette[ ]`, which takes as its argument the number `k` on which we bet. It counts the number `n` of times the ball must be thrown into the roulette bowl until it comes to rest on the number `k`. The function returns that count `n`. Therefore, the function will have a single parameter and it will return a single number. The return value is computed with a counter in a `While[ ]` loop that is local to the computation.

Observe that the last expression in the body of the loop is the number `n` whose value will be returned by `Module[ ]` and therefore, is the value returned by the function `uRoulette[ ]`.

```
Clear[uRoulette]
uRoulette[k_Integer] :=
Module[{n = 1},
  While[Random[Integer, {0, 36}] ≠ k, n += 1];
  n] /; (0 ≤ k ≤ 36)
```

Evaluate the function several times on the same argument.

```
uRoulette[16]
```

Once we have this function, we can compute the results of a large number of rolls and compute their average.

```
Table[uRoulette[12], {100}];
N[Fold[Plus, 0, %]/100]
```

**Example 2:** Printing portions of a text string.

We want to take a text string, delete successive characters, and print the left-over portion of the text until no more text is left.

Mathematica has a large number of string processing functions:

```
? *String*
```

We use two built-in string processing functions.

```
? StringLength
```

```
? StringDrop
```

We design a function `uStringTriangleP[ ]` whose input parameter `s` is a string. Its action consists of printing successively smaller pieces of the input string `s`. Observe that we initialize the local counter `i` to zero since we want to print the entire original string at the beginning. We also print a string of six stars after the `For[ ]` loop terminated so that we can see the behavior of the loop.

```
Clear[uStringTriangleP]
uStringTriangleP[s_] :=
Module[{i},
  For[i = 0, i ≤ StringLength[s], i++, Print[StringDrop[s, i]]];
  Print["*****"]
]
```

```
uStringTriangleP["Rolling Pebbles"]
```

```
uStringTriangleP["0123456789"]
```

You can see that the last string printed by the `For[ ]` loop is the empty string, which shows up as an empty line in the Print cell. Observe also that there is no output cell since the `For[ ]` loop, as a control structure, returns the value `Null`.

We can implement the same action (except for printing the final string of stars) in the functional programming style as follows:

```
Clear[uStringTriangleF]  
uStringTriangleF[s_] := Map[Print[StringDrop[s, #]] &, Range[0, StringLength[s]]]
```

This function prints and returns a list.

```
uStringTriangleF["Mathematica"]
```

We can suppress printing of the list of Nulls with the semicolon.

```
uStringTriangleF["<<>>#<<>>"];
```

### Example 3: Printing a funnel of text from a text string

The printed output of Example 2 has a triangular form. In this example we want to print a symmetric funnel shape. We accomplish the visual form of a funnel by dropping one character from each end of the string, and by replacing all the characters that we dropped from the left end of the string by the same number of blanks. Since we drop characters from both ends of the string, the loop counter needs to advance only to the middle of the string.

DEFINITION :: `uStringFunnelP[ s ]`

INPUT :: A string `s` containing any number of characters.

LOCAL VARIABLES :: `i` is a loop counter, `b` is a string of blanks, and `t` is a substring of `s`.

1. Initialize `i` to 0.
2. Loop `i` until it reaches the middle position of the input string `s`.
  - (a) Drop `i` characters from both ends of `s` and store the rest in string `t`.
  - (b) Put `i` blanks in string `b`.
  - (c) Join `b` with `t` and print the resulting string.

OUTPUT :: No value is returned, but a funnel is printed from the characters in string `s`.

IMPLEMENTATION :: `uStringFunnelP[ ]:`

```

Clear[uStringFunnelP]
uStringFunnelP[s_String] := Module[{i, t, b},
  For[i = 0, i ≤ StringLength[s]/2, i++,
    t = StringDrop[StringDrop[s, i], -i];
    b = StringJoin[Table[" ", {i}]];
    Print[StringJoin[b, t]]
  ]
]

```

Here are several invocations of `uStringFunnelP[ ]`.

```

uStringFunnelP["-.-.-.-.-.-.-.-.-.-.-"]

uStringFunnelP["malayalam"]

```

We now show how the printing of the text funnel can be implemented without any local variables. We use a combination of list and string functions to achieve this goal. Note also that we invoke `TableForm[ ]` instead of `Print[ ]` in order to render the funnel.

```

Clear[uStringFunnelF]
uStringFunnelF[s_String] := TableForm[
  Map[StringJoin[StringJoin[Table[" ", {#}]],
    StringDrop[StringDrop[s, #], -#]] &,
    Range[0, StringLength[s]/2]]]

```

Here are several invocations of `uStringFunnelF[ ]`.

```

uStringFunnelF["^+^+^+^+^+^+^+^+^"]

uStringFunnelF["0123456789"]

```

## *Using the Module[ ] Function: Returning Numbers and Lists*

---

We can use `Module[ ]` to define functions that return a single numeric value or an entire list of values. We take three interactive computations, namely, the trace of a matrix, the Bisection method, and the Collatz function, from the notebook `Loops.nb`, “Iterations with Loops,” and define them as functions in this notebook.

For each example we present only the function definition and sample invocations. We refer you back to the notebook `Loops.nb`, “Iterations with Loops,” for questions you may have regarding the implementations.

**Example 4:** Computing the trace of a square matrix (Example 6 in the notebook `Loops.nb`).

**DEFINITION** :: `uMatrixTrace[ A ]`

**INPUT** :: `A` is a square matrix; the input is tested for the matrix and the squareness property.

LOCAL VARIABLES ::  $i$  is a loop counter,  $t$  accumulates the trace.

1. Initialize  $t$  to 0 and  $i$  to 1.
2. Vary  $i$  through the rows of the matrix  $A$  and add the current diagonal element to  $t$ .

OUTPUT :: Return the trace  $t$ .

IMPLEMENTATION :: `uMatrixTrace[ ]`:

```
Clear[uMatrixTrace]
uMatrixTrace[A_] :=
Module[{t,i},
  For[t = 0; i = 1, i ≤ Length[A], i++, t += A[[i, i]]];
  t
]; MatrixQ[A] && Length[A] == Length[Transpose[A]]
```

We use the function `uRandomMatrix[ ]` from Example 6 in notebook `Loops.nb` to create random square integer matrices.

```
Clear[uRandomMatrix]
uRandomMatrix[size_Integer, type_, {min_, max_}] :=
Table[Random[type, {min, max}], {size}, {size}]
```

Here are some computations of the trace for matrices with integer, real, and complex entries. Evaluate each example several times with different matrices.

```
Clear[m1]
m1 = uRandomMatrix[3, Real, {-5, 5}];
TableForm[m1]
uMatrixTrace[m1]

Clear[m2]
m2 = uRandomMatrix[8, Integer, {-10, 20}];
TableForm[m2]
uMatrixTrace[m2]

Clear[m3]
m3 = uRandomMatrix[2, Complex, {-1 - I, 1 + I}];
TableForm[m3]
uMatrixTrace[m3]
```

When we submit a nonsquare matrix, the computation of the function `uMatrixTrace[ ]` fails.

```
uMatrixTrace[{{3, 5, 7}, {1, -2, 4}}]
```

**Example 5:** Approximating a root with Bisection (Example 13 in the notebook Loops.nb).

We define the function `uBisect[ ]` with two arguments, the name `f` of a function and the interval `[a, b]` in which a root is located. It returns an approximate value `r` of the root. We also use a fixed accuracy of six digits to which the approximation is carried out. Since a root will be found only when  $f(a) * f(b) < 0$  holds, we will enforce this property on the input parameters.

DEFINITION :: `uBisect[f, {a, b}]`

INPUT :: `f` is a function of a single variable and `{a, b}` is an interval containing a root of `f`.

LOCAL VARIABLES ::

`u` is the left endpoint and `v` is the right endpoint of the current interval.  
`r` is the midpoint of the current interval and the current approximation of the root.

1. Initialize left endpoint, middle, and right endpoints of interval.
2. Loop as long as  $|f(r)|$  exceeds  $10^{-6}$ .
  - (a) Determine the half of the current interval that contains the root.
  - (b) Adjust the appropriate endpoint to the midpoint.
  - (c) Compute the midpoint of the new interval.

OUTPUT :: Return the approximation `r` to a root of `f` in the interval `{a, b}`.

IMPLEMENTATION :: `uBisect[ ]`:

```
Clear[uBisect]
uBisect[f_, {a_, b_}] := Module[{u = N[a], v = N[b], r = N[(a + b)/2]},
  While[N[Abs[f[r]]] ≥ 10^(-6), If[f[u] * f[r] > 0, u = r, v = r];
  r = N[(u + v)/2]
];
r
];/;N[f[a] * f[b]] < 0
```

Recall that a function cannot change its own input parameters. Therefore, we need local variables for the endpoints of the current interval and initialize them with the endpoints of the input interval.

We approximate all roots of a rational function and also compute the function value at each approximation.

```
Clear[f]
f[x_] := (0.8 ^3 - 4.6x + 3.2)/(0.5x^3 + 2.5x^2 - 1.0)
```

We plot the function for a narrow strip along the x-axis.

```
Plot[f[x], {x, -8, 8}, PlotRange → {-4, 4}];
```

```

uBisect[f, {-3, -2}]
f[%]

uBisect[f, {0.7, 1.0}]
f[%]

uBisect[f, {1.5, 2.5}]
f[%]

```

We now approximate two of the infinitely many roots of a transcendental function.

```

Clear[g]
g[x_] := Sin[Exp[x] / x^2]

Plot[g[x], {x, Pi/8, 2Pi}];

```

We compute approximations only for the roots close to 1 and 4.

```

uBisect[g, {0.7, 1.0}]
g[%]

uBisect[g, {3.5, 4.0}]
g[%]

```

**Example 6:** Computing runs of the Collatz function (Example 22 in the notebook `Loops.nb`).

We design the function `uCollatzRun[ ]` with a default bound for the length of all computed runs. Such a default value is necessary since every iteration sequence of the Collatz function is infinite and it is not known whether all sequences starting with a positive number eventually contain a 1. Indeed two adjacent starting numbers can have runs of very different lengths.

DEFINITION :: `uCollatzRun[ k, b ]`

INPUT ::

`k` is a positive integer, starting a run.  
`b` is a bound for the length of the run with a default value of fifty.

LOCAL VARIABLES ::

`r` is the current segment of the run and `n` is its length.  
`s` is the current iterated value of the Collatz function.

1. Initialize `r` to the list containing `k`, `s` to the start value `k`, and `n` to one.
2. Loop as long as `s` does not equal 1 and `n` is less than the bound `b`. Increment `n` by one.
  - (a) Compute the new value of `s` by applying the Collatz function to the current `s`.
  - (b) Append `s` to the list `r`.

OUTPUT :: Return the list `r`, which contains the run starting at `k` or the first `b` numbers in it.

IMPLEMENTATION :: `uCollatzRun[ ]`:

```
Clear[uCollatzRun]
uCollatzRun[k_Integer, b_Integer:50] := Module[{r, s, n},
  For[r = {k}; s = k; n = 1,
    s ≠ 1 && n < b,
    n++,
    s = If[EvenQ[s], s/2, (3s + 1)/2];
    r = Append[r, s]
  ];
  r
] /; (k > 0 && b > 0)
```

We compute a few values of `uCollatzRun[ ]` with the default bound of 50.

```
uCollatzRun[17]
uCollatzRun[27]
uCollatzRun[-10]
uCollatzRun[236]
```

We compute a few values of `uCollatzRun[ ]` where we specify the second argument for the bound.

```
uCollatzRun[27, 100]
Length[%]
Max[%]

uCollatzRun[97531, 200]
Length[%]
Max[%]
```

## *Using the Module[ ] Function: Rendering Graphics Objects*

---

In this section, we design two functions that draw two-dimensional graphics; one is an animation and the other draws many functions in a single plot. We need several computational steps to produce multiple graphics objects. Usually, we first create the graphics objects, for example with `Do[ ]` or with `Map[ ]`. Then we define graphics properties, such as color, point size, or thickness of lines for the objects.

We implement the plots with `Module[ ]` because it is convenient to store computed values in local variables for later use. The last action performed by `Module[ ]` is to render the graphics, for example with `Show[ ]` or `Plot[ ]`, using the quantities computed and stored earlier.

**Example 7:** Animation of a hula hoop.

We use the computations from Examples 7 and 8 in the notebook `Loops.nb`, “Iterations with Loops.”

**DEFINITION** :: `uHulaHoop[min, max, step]`

**INPUT** ::

min is the initial angular position for the hoop with default value 0.

max is the final angular position for the hoop with default value  $2\pi$ .

step is the increment step of the angular position for the hoop with default value  $\frac{\pi}{6}$ .

**LOCAL VARIABLES** ::

x is the x-coordinate and y is the y-coordinate of the center of the hoop.

t is the loop counter, hidden locally in the `Do[ ]` loop.

1. Vary t through the range from min to max, incrementing t by step amount.

(a) Compute the center {x, y} of the circle representing the hoop.

(b) Render the disk and the circle.

**OUTPUT** :: Render a sequence of  $\frac{\text{max} - \text{min}}{\text{step}} + 1$  drawings of hula hoops, each plot in a plotting frame of fixed extent.

**IMPLEMENTATION** :: `uHulaHoop[ ]`:

```
Clear[uHulaHoop]
uHulaHoop[min_:0, max_: (2Pi), step_: (Pi/6)] := Module[{x, y},
  Do[{x, y} = {Cos[t + Pi], Sin[t + Pi]};
    Show[Graphics[{RGBColor[1, 0, 0], Disk[{0, 0}, 1]}],
      Graphics[{RGBColor[0, 0, 1], Circle[{x, y}, 2]}],
      AspectRatio → Automatic, PlotRange → {{-3, 3}, {-3, 3}}],
    {t, min, max, step}
  ]
]
```

We have enclosed the default values in parentheses to make them easier to read.

Here are a few sample invocations of `uHulaHoop[ ]`. The first uses the default values, the second starts at an angular position different from zero, and the last gives a very smooth rolling effect for a small angle. If you can allocate enough memory to the front end of Mathematica, you might try the last invocation for a complete roll of  $2\pi$ .

```
uHulaHoop[ ]
```

To animate the hula hoop, select all the graphics output cells created by `uHulaHoop[ ]`, and double-click on the grouping bracket to close the group of plots. The first plot of the group will



be displayed on the screen. Now choose the animation tool in the menu bar **Cell ► Animate Selected Graphics**. Experiment with the three animation directions and with various animation speeds for the circle in the lower left-hand corner of the window. A click with the mouse anywhere will stop the animation.

```
uHulaHoop[Pi]
```

```
uHulaHoop[-Pi/4, Pi/4, Pi/32]
```

**Example 8:** Drawing a sequence of Taylor polynomials approximating a function.

Taylor polynomials can be used to approximate function values. Here we want to show graphically how well Taylor polynomials approximate a function. For that purpose we compute a sequence of Taylor polynomials of increasing degrees and plot them together with the function. We draw the polynomials in colors from red for the polynomial of smallest degree to blue for the polynomial of highest degree. The function being approximated is drawn last, in green and with a thicker pen.

In the following implementation we use two built-in functions that compute the Taylor series and an approximating polynomial.

```
?Series
```

```
?Normal
```

```
Clear[x]
```

```
Series[E^(-x^2), {x, 0, 10}]
```

```
Normal[Series[E^(-x^2), {x, 0, 10}]]
```

We define the function `uTaylor[ ]`, whose first two parameters are like those of `Series[ ]`. In addition, we allow all options of `Plot[ ]` as optional arguments so that we have full control over the graphics rendering.

**DEFINITION ::** `uTaylor[expr, {x, x0, n}, {a, b}]` or  
`uTaylor[expr, {x, x0, n}, {a, b}, opts]`

**INPUT ::**

`expr` is a function expression of one variable.  
`x` is the variable in `expr` and in the Taylor expansion.  
`x0` is the point of expansion.  
`n` is the order of the expansion.  
`a` is the left endpoint of the plotting interval.  
`b` is the right endpoint of the plotting interval where  $a < b$ .  
`opts` represents any number of optional arguments for `Plot[ ]`.

**LOCAL VARIABLES ::**

`appr` is the list of Taylor polynomials of orders 0 through `n`.  
`allf` is the list `appr` with `expr` appended at the end.

cols is the list of colors for the polynomials in appr.  
 allc is the list cols with the thickness and color green for expr appended at the end.

1. Compute in sequence appr, allf, cols, and allc.
2. Plot the functions in the list allf using the colors of allc.

OUTPUT :: Render the Taylor polynomials and the function in a single plot.

IMPLEMENTATION :: uTaylor[ ]:

```
Clear[uTaylor]
uTaylor[expr_, {x_, x0_, n_}, {a_, b_}, opts___] :=
  Module[{appr, allf, cols, allc},
    appr = Map[Normal[Series[expr, {x, x0, #}]] &, Range[0, n]];
    allf = Append[appr, expr];
    cols = Map[RGBColor[(n - #)/n, 0, #/n] &, Range[0, n]];
    allc = Append[cols, {Thickness[0.006], RGBColor[0, 1, 0]}];
    Plot[Evaluate[allf], {x, a, b}, opts, PlotRange -> All,
      PlotStyle -> allc];
  ] /; (n > 0 && a < b)
```

Recall from Example 16 in FunctDef.nb, “Functions,” that the pattern for an optional argument in a function definition consists of the name of the variable followed by three underscore symbols.

Here are a few sample plots. They visually demonstrate varying speeds of convergence for the approximating Taylor polynomials to the given function. For each plot we specify a plotting range in the invocation in order to get a clear separation of all approximating Taylor polynomials.

```
uTaylor[Sin[x], {x, 0, 10}, {-2Pi, 2Pi}, PlotRange -> {-2, 2}]

uTaylor[E^x, {x, 0, 3}, {-3, 2}, PlotRange -> {0, 4}]

uTaylor[E^x, {x, 0, 8}, {-3, 2}, PlotRange -> {0, 4}]

uTaylor[Log[x], {x, 1, 25}, {1, 3}, PlotRange -> {-0.5, 1.5}]

uTaylor[E^(-x^2), {x, 0, 40}, {-3, 3}, PlotRange -> {-1, 2}]
```

## Exercises

---

Some of the following exercises are extensions of examples in this notebook and of exercises and examples in the notebook Loops.nb, “Iterations with Loops.” The objective for those exercises is to take the existing interactive Mathematica computations and to build functions using Module[ ] from the computations.

For the other exercises you must determine the necessary computations first. It may be useful to initially do the computations interactively. Once you are satisfied with your algorithm and its implementation you can incorporate your computations in a function definition. Use `Module[ ]` whenever you need to compute with local variables.

### EXERCISE 1: A function for the hula hoop.

This is an extension of Example 7 in this notebook as well as of Examples 7 and 8 and Exercise 3 in the notebook `Loops.nb`, “Iterations with Loops.”

(a) Write a function `uHulaHoop[ ]` that permits you to choose a radius for the disk and for the hoop. An invocation of this function could have the form `uHulaHoop[{diskradius, hoopradius},{min, max, step}]`.

(b) Even though we would realistically assume that the disk radius is smaller than the hoop radius there is no mathematical reason for this restriction. If the disk radius is bigger than the hoop radius then the circle representing the hoop is rolling on the inside rim of the disk. Check whether your solution in part (a) allows all combinations of positive radii. If it does, demonstrate that fact in an animation, and if it does not, extend the function so that all combinations are possible.

### EXERCISE 2: Properties of binary relations.

This exercise is an extension of Exercise 4 in the notebook `Loops.nb`, “Iterations with Loops.” Use your solutions for parts (c), (d), or (e) in that exercise as a basis for answering parts (a) and (b), next.

(a) Recall that a binary relation  $R$  is called transitive if  $xRy$  and  $yRz$  imply  $xRz$ , for all  $x, y$ , and  $z$ . Write a Boolean-valued function `uTransitiveQ[ ]` whose single argument represents the adjacency matrix of a binary relation  $R$ , and which returns `True` when the relation  $R$  is transitive and `False` otherwise.

(b) Extend the function of part (a) to test that its argument is an adjacency matrix, that is, it is a square matrix with entries of zeros and ones only.

(c) A binary relation  $R$  on the set  $\{1, 2, \dots, n\}$  is called reflexive if  $kRk$  holds, for all  $k$ ,  $1 \leq k \leq n$ . Use the `uMatrixTrace[ ]` function from Example 4 in this notebook to write a Boolean-valued function `uReflexiveQ[ ]` that takes an adjacency matrix as its argument and returns `True` when the relation is reflexive.

### EXERCISE 3: Gerschgorin disks and eigenvalues of complex matrices.

Let  $A$  be an  $n \times n$  complex matrix. Let  $r_i$  be the sum of the absolute values of all entries in the  $i^{\text{th}}$  row of  $A$ , except for the diagonal entry. Let  $c_i$  be the diagonal element in the  $i^{\text{th}}$  row of  $A$ . Consider the disk  $D_i$  in the complex plane whose center is the point  $c_i$  and whose radius is  $r_i$ . Gerschgorin proved that every eigenvalue of  $A$  lies inside some disk  $D_i$ , for  $1 \leq i \leq n$ .

(a) Write a function `uRadius[ ]` of the two arguments  $A$  and  $i$  that computes the radius  $r_i$  for the disk  $D_i$ .

- (b) Write a function `uCenter[ ]` of the two arguments `A` and `i` that computes the coordinates of the center  $c_i$  for the disk  $D_i$ . You will have to use the functions `Re[ ]` and `Im[ ]` to convert a complex number to a pair of coordinates in the plane.
- (c) Write a function `uEigenCircle[ ]` of the two arguments `A` and `i` that returns the graphics object `Circle[ ]` for the disk  $D_i$ . Observe that one Gerschgorin disk may lie completely within another. In this situation the larger disk may entirely cover up the smaller disk. Therefore, drawing circles rather than disks keeps all  $n$  circles visible.
- (d) Write a function `uGerschgorin[ ]` whose single parameter is an  $n \times n$  complex matrix `A` and which draws, in a single plot, the  $n$  circles for the Gerschgorin disks of `A`.
- (e) Use the function `uRandomMatrix[ ]` from Example 4 in this notebook to produce complex matrices and plot their Gerschgorin disks. Furthermore, compute the eigenvalues for each of your matrices with the built-in function `Eigenvalues[ ]` and convince yourself of the correctness of Gerschgorin's theorem.
- (f) Expand your function from part (d) to draw the circles in color and to draw the centers of the circles. You can choose how to assign colors to the circles. One possibility is a random color assignment, another is to design some color progression for the circles from the first to the second, and so on, to the  $n^{\text{th}}$  row of the matrix.

#### EXERCISE 4: Trace out a vertical curve with a text string.

In this exercise we take a text string and print the entire string on successive lines. The first character of the text can trace out various curves going down vertically on the page. Therefore, the entire text string traces a wide curving band.

- (a) Write a function `uLineName[ ]` that takes a number `n` as its only argument and prints your name `n + 1` times on successive lines. The first character of your name must be flush with the left margin.
- (b) Write a function `uSineName[ ]` that takes a number `n` as its only argument and prints your name `n + 1` times on successive lines. The leading characters of the printed lines should be positioned along a sine curve; for example, if  $n = 4$ , the leading positions of the printed lines should be proportional to values of the sine at the angle  $0, \frac{\pi}{2}, \pi, \frac{3\pi}{2},$  and  $2\pi$ . For each line of print, compute an appropriate number of leading blanks, so that the printed image appears to trace a vertical sine curve for one full period.
- (c) Extend the function in part (b) by another parameter so that the input can be an arbitrary text string. A sample invocation could be `uSineText[ "Oscillation", 16 ]`.
- (d) Write a function `uCurveText[ ]` that prints a text string as in part (c). This function has two additional parameters specifying the function for the curve of the vertical trace and an interval for the extent of the curve to be traced out. A sample invocation could be `uCurveText[ "Damping Effect",  $e^{-x} \sin x$ ,  $\{x, 0, \pi, \frac{\pi}{4}\}$  ]`.
- (e) If you implemented `uCurveText[ ]` in part (d) with the `Module[ ]` function and some local variables, design an alternate implementation with list processing functions.

**EXERCISE 5:** Computing and plotting runs of the Collatz function.

This is an extension of Example 6 in this notebook. Use the function `uCollatzRun[ ]` from Example 6 for your solutions in this exercise.

- (a) Find several small numbers, in the teens and low hundreds, with long runs.
- (b) Find two adjacent numbers with runs of very different lengths. Compute also the largest value occurring in each run.
- (c) Write a function `uCollatzPlot[ ]` that first computes a run and then plots the numbers in it. For example, `ListPlot[ ]` can be used to plot the values against their positions in the run.
- (d) Write a function `uCollatzStats[ ]` that computes the length  $r$  of a run and the largest number  $m$  in it. The function should return the pair  $\{r, m\}$ . If the computed sequence is not a complete run ending in 1, the function should return the pair  $\{0, 0\}$ .
- (e) Write a function `uCollatzRange[ ]` that takes an interval of positive integers and returns the list of the lengths of the runs for the numbers in the input interval.

**EXERCISE 6:** Estimating intervals of convergence for Taylor series.

Use the function `uTaylor[ ]` from Example 8 in this notebook for your solutions in this exercise.

- (a) Plot Taylor polynomials for the functions  $e^{-x^2}$ ,  $\cos x$ ,  $\frac{x^3 - x + 3}{x^2 - 1}$ , and  $\log(1 + x^2)$ . Draw enough polynomials so that you can estimate the interval of convergence for the respective Taylor series expansion around  $x = 0$ .
- (b) Verify the correctness of your estimates in part (a).
- (c) Do part (a) with several functions of your own choice. If possible, verify your estimate of the radius of convergence.

**EXERCISE 7:** Plotting a list of complex numbers.

In Example 12 of the notebook `Loops.nb`, “Iterations with Loops,” we calculated a list of complex numbers that represents a sequence of jumps starting from the origin with the quadratic rule  $f(z) = z^2 + c$  for each jump. In this exercise, we will outline the steps to get from a list of complex numbers to their plot and use the `Module[ ]` function to hide all computations.

You should use list functions and loops in your computations. If you have completed Exercise 8 of the notebook `Loops.nb`, “Iterations with Loops,” you have completed most of the work for this exercise.

- (a) Use the functions `Re[ ]` and `Im[ ]` to translate the complex number  $a + b i$  into the pair  $\{a, b\}$  of real numbers, the coordinates of the complex number.
- (b) Take your computation of part (a) and use it to translate an entire list of complex numbers into a list of coordinate pairs.
- (c) Take the last input cell from Example 12 of the notebook `Loops.nb`, “Iterations with Loops,” to produce a list of complex numbers. Apply your function from part (b) to that list.

- (d) Take your resulting list from part (c) and wrap the graphics primitive `Point[ ]` around each pair. Hint: See Example 7 of the notebook `ListFcts.nb`, “List Processing Functions.”
- (e) Write a function `uEscapePoints[c]` that takes the complex constant  $c$  of the quadratic rule  $f(z) = z^2 + c$  as its only argument and returns the list of points in the iteration.
- (f) Write a function `uEscapePlot[c]` that takes the complex constant  $c$  of the quadratic rule  $f(z) = z^2 + c$  as its only argument and plots the points of the iteration. Use your function of part (e) together with `Show[Graphics[ . . . ]]`.
- (g) Write a function `uEscapePlot[c, r, n]` that takes the complex constant  $c$  of the quadratic rule  $f(z) = z^2 + c$  as its first argument, the radius  $r$  of the “escape circle” for  $f(z)$ , and the upper bound  $b$  for the number of iterations. The function plots the points of the iteration.

This Page Intentionally Left Blank

# IV

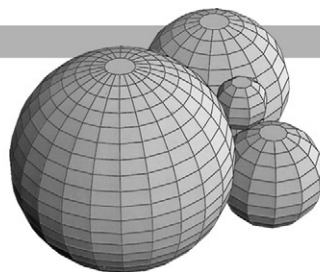
---

*Exploring Advanced Features*



This Page Intentionally Left Blank

# *Advanced Mathematica: Options*



## *Introduction*

In the notebooks of Parts I, II, and III we have made two kinds of assignments, immediate and delayed. We also applied substitution rules in two different forms, as results of solving systems of equations and as options to functions. The options are very useful because they permit a varying number of arguments for a function and they also permit an arbitrary ordering of the (optional) arguments for a function. We have used options extensively for the built-in graphing functions such as `Plot[ ]`, `Plot3D[ ]`, and `ListPlot[ ]`.

In Example 10 of the notebook `Rules.nb` in Part II, “Substitution Rules and Optional Arguments,” we defined a single option for the numeric function `uCubeRoot[ ]`. We will extend that example as a short review of defining options. The main topic of this notebook is to define functions that have a variety of options. The first step is to expand the options from a single option to several options. The next step is to include with the user-defined options for a function, options from a built-in function, such as the plotting function `Plot[ ]`. In this way we can use the power of Mathematica to control aspects of the user-defined function through user-defined options and, at the same time, have the full graphics control that Mathematica provides.

## *Defining a Single Option for a Function*

Recall from the notebook `Rules.nb` in Part II, that we must consider two aspects when defining an option for a function: An option has the syntactic form of a rule, as in “symbol  $\rightarrow$  value”, and an option always has a default value. When we invoke a function that has an option defined for it, then we can replace the default value of an option by naming the option with a new value.

We generalize the function `uCubeRoot[ ]` from Example 10 in the notebook `Rules.nb` of Part II.

**Example 1:** An option to enable real number computations for roots of a real number.

We want to compute  $x^y$  for a fractional exponent  $y$ , in reduced form, just as Mathematica computes it, but we also want to return a real number when it is mathematically possible to do so. For this purpose we define a function `uRoot[ ]` and the option `uCompute  $\rightarrow$  Real` to enable real number computation for it. The default value for the option is `uCompute  $\rightarrow$  Complex`.

```
Clear[uRoot, uCompute]
Options[uRoot] = {uCompute → Complex}
```

We split the design of the function `uRoot[ ]` into two parts. First we write a test function `uRealRootQ[ ]` that determines for the current arguments `x` and `y` of `uRoot[ ]` whether a computation over the real numbers is possible and whether the option was set for real computations. Note that the expression  $x^y$  admits a real value when neither `x` nor `y` are complex and when `y` represents a rational number with odd denominator, as for instance in  $x^y = (-2)^{\frac{4}{3}}$ . We need four distinct tests on the arguments `x` and `y` and one additional test to determine whether the real number computation was required.

```
Clear[uRealRootQ]
uRealRootQ[x_, y_, opts___] := Head[x] != Complex &&
Head[y] != Complex && Head[y] === Rational &&
OddQ[Denominator[y]] && (uCompute /. {opts} /. Options[uRoot]) === Real
```

We now explain how to perform the calculation of a real root. Mathematica carries out real number computations when the base `x` of the expression  $x^y$  is a non-negative real number. When the base is negative, an appropriate combination of the `Sign[ ]` and the `Abs[ ]` functions will ensure that the base for the computations of the power is non-negative.

We transform the root expression as follows:  $x^{\frac{p}{q}} = (\text{signum}(x) * |x|^{\frac{1}{q}})^p$ .

Here are two examples:  $(-2)^{\frac{5}{3}}$  is the negative real number  $((-1) * (2^{\frac{1}{3}}))^5$  and  $(-2)^{\frac{8}{5}}$  is the positive real number  $((-1) * 2^{\frac{1}{5}})^8$ .

When we want to perform computations over the complex numbers we invoke the built-in exponentiation function `Power[ ]` or `^`. This leads to the following definition of the function `uRoot[ ]`.

```
Clear[uRoot]
uRoot[x_, y_, opts___] :=
  If[uRealRootQ[x, y, opts], (Sign[x] Abs[x]^(1/Denominator[y]))
    ^Numerator[y], x^y] /; NumberQ[x] && NumberQ[y]
```

Note that we restrict `uRoot[ ]` to numbers only. We calculate complex and real roots for a variety of combinations of positive and negative values for the base `x` and the exponent `y`. All exponents will be rational numbers with odd denominators.

Positive base and positive rational exponent:

```
uRoot[2, 1/3]
N[%]

uRoot[2, 1/3, uCompute → Real]
N[%]
```

Negative base and positive rational exponent:

```
uRoot[-2, 5/3]
N[%]

uRoot[-2, 5/3, uCompute → Real]
N[%]
```

```
uRoot[-2, 8/5, uCompute → Real]  
N[%]
```

Positive base and negative rational exponent:

```
uRoot[2, -5/3]  
N[%]
```

```
uRoot[2, -5/3, uCompute → Real]  
N[%]
```

Negative base and negative rational exponent:

```
uRoot[-2, -5/3]  
N[%]
```

```
uRoot[-2, -5/3, uCompute → Real]  
N[%]
```

```
uRoot[-2, -8/5, uCompute → Real]  
N[%]
```

With symbolic numeric quantities the computation fails:

```
uRoot[-E, 1/3]
```

```
uRoot[-Pi, 2, uCompute → Real]
```

Independent of the number type of the argument—integer, real, or complex—the function `uRoot[ ]` always returns a number, and its default is complex.

```
uRoot[-3.0, 1+I]
```

```
uRoot[-3.0, 1+I, uCompute → Real]
```

```
uRoot[1+I, 1-I]  
N[%]
```

```
uRoot[4, 5]
```

Finally, we demonstrate the effect of `uRoot[ ]` with a graph of three root functions in a single plot.

Observe that we can use the variable (symbol) `x` as an argument for `uRoot[ ]` inside the `Plot[ ]` function. `Plot[ ]` keeps the function expression unevaluated and substitutes numbers from the plotting range into `uRoot[ ]` for each point on the graph of `uRoot[ ]`. Therefore, `uRoot[ ]` will be evaluated only on numbers.

```
Plot[{uRoot[x, 1/5, uCompute → Real],  
      uRoot[x, 3/7, uCompute → Real], uRoot[x, 5/9, uCompute → Real]},  
      {x, -27, 27},  
      PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 1, 0], RGBColor[0, 0, 1]}];
```

**Example 2:** An option for the Bisection method.

In this example we define a function `uBisection[ ]` that implements the Bisection method to approximate a root of a function. We use an option to control the type of information that `uBisection[ ]` returns. The function returns a list that contains either a substitution rule for the root approximation (Rule), or a sequence of approximations to the root (All or n), or the numeric value of the last approximation (Last), or the empty list when a meaningless value for the option was specified.

The function `uBisect[ ]` in Example 5 of the notebook `Modules.nb`, “Computations with Modules and Local Variables,” in Part III provides the basis for our discussions here. That function implements the Bisection method to approximate a root of a function in a given interval and it returns the final approximation to a root in that interval. We will modify it here.

Since we want to be able to return a rule we will replace all function evaluations of the form `f[z]` in the body of `uBisect[ ]` by substitution rules such as `v /. (x → z)`, where `v` is an expression. The function `uBisect[ ]` expects a function name as its first argument, whereas `uBisection[ ]` expects a function expression as its first argument. This makes the invocation of `uBisection[ ]` similar to `Plot[ ]` and `FindRoot[ ]`, whereas `uBisect[ ]` is similar to `Map[ ]` and `Fold[ ]`.

We first explain the different values expected for the option `uResult` of `uBisection[ ]`. The default will be to return a rule for the approximated root.

```
Clear[uBisection,uResult]
Options[uBisection]={uResult → Rule}
```

There will be three symbolic values and one numeric value for the option `uResult`:

- Rule** returns a rule for the approximation of the root.
- Last** returns the last value computed for the approximation of the root.
- All** returns the list of successive approximations to the root.
- n** returns the list of the last `n` approximations to the root or the entire list of approximations if there are fewer than `n` numbers in the list.

**DEFINITION ::** `uBisection[expr, {x, a, b}, opts]`

**INPUT ::**

- `expr` is a function expression of the variable `x`.
- `a` is the left endpoint and `b` is the right endpoint of an interval containing a root of `expr`.
- `opts` is the option `uResult`, which may or may not be present in an invocation.

**LOCAL VARIABLES ::**

- `u` is the left endpoint of the current interval.
- `v` is the right endpoint of the current interval.
- `r` is the midpoint of the current interval and the current approximation of the root.
- `z` is the list of current approximations to the root, starting with the midpoint of `[a, b]`.
- `p` is the actual value of the option `uResult`.

1. Initialize the left, middle, and right endpoints of the interval  $[a, b]$  and the list of approximations,  $z$ .
2. Loop as long as  $|\text{expr}/.(x \rightarrow r)|$  and  $(v - u)$  exceed  $10^{-6}$ .
  - (a) Determine the half of the current interval that contains the root.
  - (b) Adjust the appropriate endpoint.
  - (c) Compute the midpoint of the new interval.

OUTPUT :: Return an approximation to a root of  $\text{expr}$  in the interval  $(a, b)$  with six-digit accuracy and according to the options  $\text{opts}$ .

IMPLEMENTATION :: `uBisection[ ]`:

```
uBisection[expr_, {x_, a_, b_}, opts_] :=
Module[{u = N[a], v = N[b], r = N[(a + b)/2], z = {N[(a + b)/2]},
  p = uResult/.{opts}/.Options[uBisection]},
While[N[Abs[expr/.x→r]] ≥ 10^(-6) && N[v-u] ≥ 10^(-6),
  If[N[(expr/.x→u)*(expr/.x→r)]>0, u = r, v = r];
  r = N[(u+v)/2];
  AppendTo[z, r]];
Which[
  p === Rule, {x→r},
  p === All, z,
  p === Last, Last[z],
  Head[p] === Integer && 1 ≤ p && p ≤ Length[z], Take[z, -p],
  Head[p] === Integer && 1 ≤ p && p > Length[z], z,
  True, {}]
]; (N[(expr/.x→a)*(expr/.x→b)]<0)&&(N[a]<N[b])
```

As a first example we compute the roots of the equation  $e^{-\frac{x}{10}} = \sin x$  in the interval  $[0, 2\pi]$ . We graph the difference function  $e^{-\frac{x}{10}} - \sin x$  to find appropriate intervals for the start of the approximation.

```
Plot[E^(-x/10) - Sin[x], {x, 0, 2Pi}];
```

We return one root as a rule and also as a value.

```
uBisection[E^(-x/10) - Sin[x], {x, 1, 2}, uResult → Rule]
```

```
uBisection[E^(-x/10) - Sin[x], {x, 1, 2}, uResult → Last]
```

We return the list of all approximations, of the last five approximations, and of the last 30 approximations.

```
uBisection[E^(-x/10) - Sin[x], {x, 2, 3}, uResult → All]
```

```
uBisection[E^(-x/10) - Sin[x], {x, 2, 3}, uResult → 5]
```

```
uBisection[E^(-x/10) - Sin[x], {x, 2, 3}, uResult → 30]
```

Here are numeric values of the option that have no meaning for the computation. In those instances `uBisection[ ]` returns an empty list.

```
uBisection[E^(-x/10) - Sin[x], {x, 2, 3}, uResult → -4]
```

```
uBisection[E^(-x/10) - Sin[x], {x, 2, 3}, uResult → 2.65]
```

As a second example for the Bisection method we take the expression  $(\sin^2 2x + \cos 3x)$  and determine a root in the interval  $[0, \pi]$ .

```
Plot[Sin[2x]^2 + Cos[3x], {x, 0, Pi}];
```

```
uBisection[Sin[2x]^2 + Cos[3x], {x, 0, 1}, uResult → Rule]
```

```
uBisection[Sin[2x]^2 + Cos[3x], {x, 0, 1}, uResult → All]
```

The three symbolic values for the option `uResult`—`Rule`, `All`, and `Last`—have meaning of their own in Mathematica. You can find the meaning of the symbols in the Help Browser: [Rule](#), [All](#), and [Last](#). In `uBisection[ ]` we use the symbols for their names only, we never evaluate the symbols. Therefore, the role of the symbols in other parts of Mathematica does not influence the computations in `uBisection[ ]`.

## Defining Several Options for a Function

---

The next level of complexity in defining options is to define several options for one function. The built-in function `Plot[ ]`, for example, has 30 options. Mathematica lets us compute that number.

```
Options[Plot]  
Length[%]
```

In Example 5 we define a function `uAnalyze[ ]` to analyze a list of data statistically and graphically. We want to be able to compute various statistical measures for the data, for example, the mean, median, or variance. We use an option `uStats` for that purpose. We also want to display the data in various ways; for example, we plot the data points or we plot the best linear or quadratic fit for the data. This we do with the second option, `uGraph`. Throughout this section we assume that the data are numeric and given in the list  $L = \{(x_1, y_1), \dots, (x_n, y_n)\}$ . In order to keep the functions simple and free of testing code, we assume that the actual data always will be an  $n \times 2$  list.

**Example 3:** The statistical functions mean, median, and variance.

The mean of the list  $L$  is given by the formula  $(\sum_{i=1}^n x_i/n, \sum_{i=1}^n y_i/n)$ . Recall that the addition function `Plus[ ]` in Mathematica distributes over lists so that we can add vectors and, in particular, pairs of numbers. We can compute the mean with the `Apply[ ]` functions as follows:

```
Clear[uMeanFct]  
uMeanFct[data_] := N[Apply[Plus, data] / Length[data]]  
  
uMeanFct[{ {1, 5}, {-1, 7} }]
```

The median of a list requires a very different kind of computation. The median of a list of numbers is the middle entry in the list when it is sorted. When the list has an even number of entries we have two choices, and we choose the entry to the left of the middle. Given a list of data we first sort the data and then compute the middle position with the `Length[ ]` and the `Floor[ ]` functions.

```
Clear[uMedianFct]
uMedianFct[data_] := Sort[data][[Floor[Length[data]/2]]]
```

Evaluate the next input cell several times. Change the size of the list and the ranges in the random number generator.

```
Clear[ranges]
ranges = Map[{Random[Integer,{0,8}],Random[Integer,
  {-20,20}]}&, Range[10]]
Sort[ranges]
uMedianFct[ranges]
```

The variance of the x-coordinates of the list *L* of data is given by  $\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$ , where  $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$  is the mean. However, the actual expression commonly used for computations is  $\frac{1}{n-1} ((\sum_{i=1}^n x_i^2) - n\bar{x}^2)$ . Convince yourself that the two formulas are equivalent. The standard deviation of a list of data is the positive square root of the variance. As with the computations for the mean, we can use the Mathematica functions on the pairs of x-coordinates and y-coordinates simultaneously.

```
Clear[uVarianceFct]
uVarianceFct[data_] := Module[{n = Length[data],
  m = uMeanFct[data]},
  (Apply[Plus,Map[#^2&,data]] - n*m^2)/(n-1)]
```

We use the same random generation of lists as we did with `uMedianFct[ ]`. Evaluate the next input cell several times. Change the size of the list and the ranges in the random number generator.

```
Clear[ranges]
ranges = Map[{Random[Integer,{0,8}], Random[Integer,
  {-20,20}]}&, Range[10]]
uMeanFct[ranges]
uVarianceFct[ranges]
```

**Example 4:** Fitting a curve to data points and plotting the fit.

We generate a list of 10 numbers randomly as in Example 3.

```
Clear[ranges]
ranges = Sort[Map[{Random[Integer,{0,8}],
  Random[Integer,{ -20,20}]}&, Range[10]]]
```

The function `Fit[data, functions, variable]` will compute the best fit to the data given the list of functions in the variable. For example, the list of functions `{1, x}` will produce a linear fit.

```
Fit[ranges, {1, x}, x]
```



The list of functions  $\{1, x, x^2\}$  in the next input cell will produce a quadratic fit.

```
Fit[ranges, {1, x, x^2}, x]
```

We want to plot the curves for the best fits. Since Plot[ ] holds the evaluation of its arguments we must force evaluation with Evaluate[ ] inside the Plot[ ] command. We use the data from the first evaluation.

Here is the plot of the linear fit.

```
Clear[lin]  
lin = Plot[Evaluate[Fit[ranges, {1, x}, x]], {x, 0, 8},  
PlotRange → {{0, 8}, {-20, 20}}];
```

The next plot shows the quadratic fit on the same data.

```
Clear[quad]  
quad = Plot[Evaluate[Fit[ranges, {1, x, x^2}, x]], {x, 0, 8},  
PlotRange → {{0, 8}, {-20, 20}}];
```

Now, we plot the data points.

```
Clear[pts]  
pts = ListPlot[ranges, PlotStyle → PointSize[0.02],  
PlotRange → {{0, 8}, {-20, 20}}];
```

Finally, we plot the data points together with the linear fit and the quadratic fit. Since we selected random points, we cannot expect the fits to be good.

```
Show[lin, quad, pts];
```

Evaluate all input cells in this example several times in order to see a variety of data sets.

**Example 5:** Definition of the function uAnalyze[ ].

With Examples 3 and 4 we have assembled all the pieces necessary to write the function uAnalyze[ ] with its two options uStats and uGraph. We define several symbolic values for the two options. Here are the meanings of the option values:

uStats

Mean	computes the mean of the data.
Median	computes the median of the data.
Variance	computes the variance of the data.
None	returns Null (default value).

uGraph

ListPlot	plots the data points.
LinearPlot	plots the linear best fit.
QuadraticPlot	plots the quadratic best fit.
None	produces no plot (default value).

When any other value is entered for one of the two options then the default value None for that option takes effect.

```
Clear[uAnalyze,uStats,uGraph]
Options[uAnalyze] = {uStats → None,uGraph → None};
```

DEFINITION :: uAnalyze[data, opts]

INPUT ::

data is a numeric list of the form  $L = \{(x_1, y_1), \dots, (x_n, y_n)\}$ .  
 opts is the sequence of options that may or may not be present in an invocation.

LOCAL VARIABLES ::

s is the actual value of the option uStats.  
 p is the actual value of the option uGraph.  
 r is the two-dimensional range defined by the extremal x-values and y-values of the points in the data list.

1. Compute the actual values for the two options, s and p, and the numeric range of the data, r.
2. Compute the statistical quantity as indicated by the option uStats and print it.
3. Render the appropriate plot according to the value of the option uGraph.

OUTPUT :: A pair of numeric values and a plot as specified by the options.

IMPLEMENTATION :: uAnalyze[ ]:

```
Clear[uAnalyze]
uAnalyze[data_,opts___] := Module[{
  s = uStats/.{opts}/.Options[uAnalyze],
  p = uGraph/.{opts}/.Options[uAnalyze],
  r = Map[{Min[#],Max[#]}&,Transpose[data]]},

  Print["Statistics: ",s];
  Which[
    s === Mean,s = uMeanFct[data],
    s === Median,s = uMedianFct[data],
    s === Variance,s = uVarianceFct[data],
    s === None,s = Null
  ];
  Print["Value: ",s];

  Print["Type of plot: ",p];
  Which[p === ListPlot, ListPlot[data,PlotStyle → PointSize[0.02]],
    p === LinearPlot,
```

```

Plot[Evaluate[Fit[data,{1,x},x]],{x,r[[1,1]],r[[1,2]]},
PlotRange→r],
p===QuadraticPlot,Plot[Evaluate[Fit[data,{1,x,x^2},x]],
{x,r[[1,1]],r[[1,2]]},PlotRange→r]
]
]

```

**Example 6:** Invocations of the function `uAnalyze[ ]`.

We first create some test data as we did in the previous examples of this section.

```

Clear[testdata]
testdata=Sort[Map[{Random[Integer,{0,8}],
Random[Integer,{−20,20}]}&,Range[10]]]

```

Note that you must have evaluated the statistical function definitions in Exercise 3 as well as the definition of `uAnalyze[ ]` in Example 5 before you evaluate `uAnalyze[ ]` in the following input cells. We use the same data with different options. Evaluate the entire sequence of input cells with several different test data.

First, we test the statistics option `uStats`.

```

uAnalyze[testdata]

uAnalyze[testdata,uStats→Mean]

uAnalyze[testdata,uStats→Median]

uAnalyze[testdata,uStats→Variance]

```

Next, we test the plotting option `uGraph`.

```

uAnalyze[testdata,uGraph→ListPlot];

uAnalyze[testdata,uGraph→LinearPlot];

uAnalyze[testdata,uGraph→QuadraticPlot];

```

Here are invocations with both options in two different orderings.

```

uAnalyze[testdata,uStats→Median,uGraph→QuadraticPlot];
uAnalyze[testdata,uGraph→QuadraticPlot,uStats→Median];

```

## *Using Built-in Graphics Options in User-Defined Functions*

---

In this section we design a function `uVectorField2D[ ]` that plots a two-dimensional vector field. Mathematica implements many options that control the rendering of graphics objects with

Plot[ ] and Graphics[ ]. We exploit these capabilities by including the built-in graphics options in uVectorField2D[ ].

We first must decide how to represent the vectors at the grid points in a vector field. Our graphic image is the weather vane that is turning in the wind. The vane consists of two portions, the pivot on a pole around which it turns and the flat plate that catches the wind. We represent the pivot by the graphics primitive Point[ ] and draw it as a “fat” point with the PointSize[ ] graphics primitive. The plate of the vane we represent by the graphics primitive Line[ ]. One end of the line is at a grid point and its direction and length indicate the field vector at that grid point.

Mathematica provides in its graphics package a number of tools to draw vector fields that are different from the functions we are designing. You can find the two-dimensional vector field functions by selecting **Help** ► **Add-ons ...** in the menu bar and then choosing **Standard Packages** ► **Graphics** ► **PlotField** in the window from left to right.

**Example 7:** Drawing a line representation of a weather vane.

We suppose that the vector field  $v$  is given by a pair of function expressions  $(f(x, y), g(x, y))$  in the variables  $x$  and  $y$ . For a point  $p_0(x_0, y_0)$  the endpoints of the weather vane are  $p_0$  and  $p_0 + v_0$  where  $v_0 = (f(x_0, y_0), g(x_0, y_0))$  is the field  $v$  evaluated at  $p_0$ . We associate a specific size with the pivot and a specific thickness with the plate of the vane.

```
Clear[uVane]
uVane[p_, v_] := {PointSize[0.02], Point[p], Thickness[0.004],
  Line[{p, p + v}]}
```

As an example for a vector field, we use the standard inverse-square gravity field.

```
Clear[gravity]
gravity = {-x, -y} / (x^2 + y^2)
```

Here are drawings of a single vane and then of four vanes in the gravity field.

```
Show[Graphics[uVane[{1, 1}, gravity /. {x -> 1, y -> 1}],
  Frame -> True, PlotRange -> {{0, 2}, {0, 2}}]]];

Show[Graphics[{uVane[{1/2, 1}, gravity /. {x -> 1/2, y -> 1}],
  uVane[{1/2, 1/2}, gravity /. {x -> 1/2, y -> 1/2}], uVane[{1, -3/2},
  gravity /. {x -> 1, y -> -3/2}], uVane[{-1, 1},
  gravity /. {x -> -1, y -> 1}]}],
  Axes -> True, AspectRatio -> Automatic, PlotRange
  -> {{-1.5, 1.5}, {-1.5, 1.5}}]]];
```

The farther away the grid point is from the origin, the weaker the field and the shorter the vane. Observe that all vanes point to the center, the point of attraction in a gravity field.

The definition of the weather vane fixes its graphics properties of line thickness and point size to specific values. For each instance of a vane on the grid, the graphics primitives PointSize[ ] and Thickness[ ] are repeated over and over again with the same arguments for each grid point. The following evaluation lists the graphics primitives that were used in the previous plot.

```
InputForm[Show[%, DisplayFunction -> Identity]]
```

Now we draw eight vanes at the cardinal points of the compass of radius  $\frac{3}{2}$  in the gravity field.

```
Show[Graphics[Map[uVane[#, gravity/.{x→#[[1]],y→#[[2]]}]&,
  Map[{3/2 Cos [#], 3/2 Sin [#]}&, Range[0, 7/4 Pi, Pi/4]]],
Axes→False, Frame→True, AspectRatio→1,
PlotRange→{{-2, 2}, {-2, 2}}]]];
```

**Example 8:** Using built-in graphics options for a vector field function.

We will draw the two-dimensional vector field on a rectangular grid in the plane that we determine with range specifiers in the same manner as `Plot3D[]` does. However, we require that the extent and the increment step of the grid along the x-axis and along the y-axis must be specified. This simplifies the definition of the function interface.

We modify the function `uVane[]` to create only the two pieces of the vane, the point and the line segment, but no graphics primitives. This allows us to specify the vane and its graphics properties separately; for example, we can specify a set of graphics properties to apply to many vanes simultaneously.

```
Clear[uVaneParts]
uVaneParts[p_, v_] := {Point[p], Line[{p, p + v}]}
```

We generate the grid points and the vectors with the `Table[]` and the `uVane[]` functions. Furthermore, we specify values for the `AspectRatio` and the `Frame` options in the function `uVectorField2D[]`. However, any invocation of the function `uVectorField2D[]` can override these values by specifying different values for those options.

**DEFINITION ::** `uVectorField2D[{f,g}, horX, verY, opts]`

**INPUT ::**

<code>{f,g}</code>	is a two-dimensional vector field, where <code>f</code> and <code>g</code> are function expressions.
<code>horX</code>	specifies the range of x-coordinates of the grid in the form $\{x, x_0, x_1, dx\}$ .
<code>verY</code>	specifies the range of y-coordinates of the grid in the form $\{y, y_0, y_1, dy\}$ .
<code>opts</code>	is a sequence of zero or more options from the <code>Graphics[]</code> function.

1. Set the graphics primitives for the vanes.
2. Compute the table of vectors over the  $(horX, verY)$  grid with `uVaneParts[]`.
3. Include the option variable and some specific graphing options.

**OUTPUT ::** Render the two-dimensional vector field.

**IMPLEMENTATION ::** `uVectorField2D[]`:

```
Clear[uVectorField2D]
uVectorField2D[{f_, g_}, horX:{x_, x0_, x1_, dx_}, verY:{y_, y0_, y1_, dy_},
  opts___] := Show[Graphics[Join[{PointSize[0.02], Thickness[0.004]}],
```

```
Table[uVaneParts[{x,y},{f,g}],horX,verY]],opts,
      AspectRatio→Automatic,Frame→True]]
```

Note that the symbol `opts` must appear before any other options in the `Graphics[ ]` function so that the actual option values in an invocation will take effect. Note also that the `Table[ ]` function implicitly performs the substitutions into the vector field `{f,g}`, which we did explicitly in Example 7.

```
uVectorField2D[{-y,x},{x,-4,4,0.5},{y,-3,3,1}];
```

You should experiment with the visualization of the vector fields by drawing more grid points, larger regions of the field, and by including additional graphics options. Here are some examples of vector fields including the gravity field from Example 7. Remember that the gravity field is not defined at the origin, so `Plot[ ]` may give warnings that the value `Infinity` was encountered during computations.

```
Clear[gravity]
gravity = {-x,-y}/(x^2 + y^2)

gravity/.{x→0,y→0}

uVectorField2D[gravity,{x,-3.5,3.5,1},{y,-3.5,3.5,1}];

uVectorField2D[gravity,{x,-1,1,0.2},{y,-1,1,0.2},
  AspectRatio→Automatic,Background→RGBColor[0,0,1],
  PlotLabel→"The gravity vector field"];
```

The next vector field will be drawn on  $33 * 33 = 1089$  grid points with vanes.

```
uVectorField2D[{Cos[x-y],Sin[x+y]},{x,-2Pi,2Pi,Pi/8},
  {y,-2Pi,2Pi,Pi/8}, Background→RGBColor[1,0,0]];
```

## *Defining Options for One Function That Are Options in Several Other Functions*

---

We designed the function `uVectorField2D[ ]` in the previous section so that the length of the vanes, their thickness, and the size of the grid points are determined. When there are many points in the grid, the “fat” points of the vanes frequently wash out the field. We can get better control of the visualization of a vector field if we are able to specify a scaling factor and graphics primitives, such as thickness and color, for the vanes.

**Example 9:** Defining optional arguments with graphics primitives as their values.

Here is yet another definition of the `uVane[ ]` function. This time it includes options for specifying graphics primitives. We use the name `uFlexVane[ ]`, because we have more flexible graphics. Furthermore, we can easily distinguish the new function from the previous definition `uVane[ ]`.

First we explain the meaning of the options and describe their default values:

<code>uVaneScale</code>	accepts any numeric scaling factor for the length of the vane; its default value is 1.0.
<code>uLineStyle</code>	permits any list of graphics primitives that apply to lines; its default is 0.004 thickness (very thin) and black color.
<code>uPointStyle</code>	permits any list of graphics primitives that apply to points; its default value is 0.02 (fat point) and black color.

```
Clear[uFlexVane,uVaneScale,uLineStyle,uPointStyle]
```

```
Options[uFlexVane] = {uVaneScale → 1.0,  
  uLineStyle → {Thickness[0.004], RGBColor[0,0,0]},  
  uPointStyle → {PointSize[0.02], RGBColor[0,0,0]};
```

The definition of the function `uFlexVane[ ]` consists mostly of computations of option values.

DEFINITION :: `uFlexVane[ p, v, opts ]`

INPUT ::

<code>p</code>	is a pair of coordinates for the pivot of the vane.
<code>v</code>	is the value of the vector field at point <code>p</code> .
<code>opts</code>	is a sequence of zero or more options as defined previously for <code>uFlexVane[ ]</code> .

1. Compute the scaling options `sc`, the line options `ls`, and the point options `ps`.
2. Assemble the list of graphics primitives for the vane.

OUTPUT :: The list computed in step 2.

IMPLEMENTATION :: `uFlexVane[ ]`:

```
uFlexVane[p_,v_,opts___]:=  
  Module[{sc,ls,ps},sc = uVaneScale/.{opts}/.Options[uFlexVane];  
    ls = uLineStyle/.{opts}/.Options[uFlexVane];  
    ps = uPointStyle/.{opts}/.Options[uFlexVane];  
    Join[ps,{Point[p]},ls,{Line[{p,p + sc v}]}]]
```

We draw two vanes in one plot, once with the default option values and once with explicit values for all options. As you can see from the amount of code in the following input cells, the more control we want for the graphics the more we must specify.

```
Show[Graphics[{uFlexVane[{1,-1},{-y,x}/.{x→1,y→-1}],  
  uFlexVane[{-1/2,1/2},{-y,x}/.{x→-1/2,y→1/2}]}],  
  Frame→True,PlotRange→{{-2,3},{-2,1}}];
```

```
Show[Graphics[{uFlexVane[{1,-1},{-y,x}/.{x→1,y→-1},  
  uVaneScale→0.5,uLineStyle→{Thickness[0.01],RGBColor[1,1,0]},  
  uPointStyle→{PointSize[0.06],RGBColor[0,0,1]}},
```

```
uFlexVane[{-1/2,1/2},{-y,x}/.{x→-1/2,y→1/2},uVaneScale→2.5,
uLineStyle→{Thickness[0.025],RGBColor[1,0,1]},
uPointStyles→{PointSize[0.1],RGBColor[1,0,0]}},
Frame→True,PlotRange→{{-2,3},{-2,1}}];
```

Before we can continue with the development of the vector field function `uVectorField[ ]` we need two brief interludes. The first demonstrates how to manipulate sequences of options. The second explains the differences between clearing and removing functions with options.

**Example 10:** Separating options for different functions from a sequence of options.

We show how a sequence of options can be separated into several sequences according to the functions for which the options are defined. We achieve this separation with the function `FilterOptions[ ]` that is supplied in the standard Utilities package. Select **Help ▸ Add-ons ...** in the menu bar and then choose **Standard Packages ▸ Utilities ▸ FilterOptions** in the window from left to right. You can evaluate the “`≪ Utilities`FilterOptions``” command in the window of the Help Browser or the command in the following input cell.

```
Needs["Utilities`FilterOptions`"]
```

```
?FilterOptions
```

The function `FilterOptions[ ]` expects a sequence, not a list, of options as input and returns a sequence of options.

```
?Sequence
```

Now we apply `FilterOptions[ ]` to a function and to a sequence of options. Some of the options in the sequence apply to more than one function or to exactly one function.

```
FilterOptions[Plot,Frame→True,Modulus→2, Background
→ RGBColor[1,0,0]]
```

```
FilterOptions[Plot3D,Frame→True,Modulus→2, Background
→ RGBColor[1,0,0]]
```

```
FilterOptions[Factor,Frame→True,Modulus→2, Background
→ RGBColor[1,0,0]]
```

None of the options in the sequence apply to the following built-in functions, and the empty sequence is returned.

```
FilterOptions[TableForm,Frame→True,Modulus→2, Background
→ RGBColor[1,0,0]]
```

```
FilterOptions[NIntegrate,Frame→True,Modulus→2, Background
→ RGBColor[1,0,0]]
```



**Example 11:** A vector field function with options for the vanes and options for the field.

In order to avoid confusion with names we eliminate the names `uFlexVane` and `uVectorField2D` from the current session with the function `Remove[ ]`. Invoking the `Clear[ ]` function is not enough because it clears only the value of the symbol, that is in our case, the definition of the function `uFlexVane[ ]`. It does not clear the options defined for `uFlexVane[ ]`.

The next sequence of commands assumes that `uFlexVane[ ]` is defined. If it is not, evaluate the appropriate input cells in Example 9 first.

```
??uFlexVane

Clear[uFlexVane]

??uFlexVane

Remove[uFlexVane]

??uFlexVane

Remove[uVectorField2D]

??uVectorField2D
```

Now we are ready to define the new, more flexible version of the function `uVectorField2D[ ]`. We load the `FilterOptions` package from the `Utilities` folder. Then we repeat the definition of `uFlexVane[ ]` since we removed it in the previous example. The interface for `uVectorField2D[ ]` remains unchanged from the previous definition. However, now the options of `uFlexVane[ ]` are permitted in addition to all `Graphics[ ]` options.

```
Needs["Utilities`FilterOptions`"]

Options[uFlexVane] =
  {uVaneScale → 1.0, uLineStyle → {Thickness[0.004],
    RGBColor[0,0,0]},
    uPointStyles → {PointSize[0.02], RGBColor[0,0,0]}};

uFlexVane[p_, v_, opts___] :=
  Module[{sc, ls, ps}, sc = uVaneScale /. {opts} /. Options[uFlexVane];
    ls = uLineStyle /. {opts} /. Options[uFlexVane];
    ps = uPointStyles /. {opts} /. Options[uFlexVane];
    Join[ps, {Point[p]}, ls, {Line[{p, p + sc v}]}]]
```

DEFINITION :: `uVectorField2D[ { f, g }, horX, verY, opts ]`

INPUT ::

<code>{f,g}</code>	is a two-dimensional vector field, where <code>f</code> and <code>g</code> are function expressions.
<code>horX</code>	specifies the range of x-coordinates of the grid in the form $\{x, x_0, x_1, dx\}$ .
<code>verY</code>	specifies the range of y-coordinates of the grid in the form $\{y, y_0, y_1, dy\}$ .
<code>opts</code>	is a sequence of zero or more options from the <code>uFlexVane[ ]</code> and <code>Graphics[ ]</code> functions.

1. Select the options for `uFlexVane[ ]` with `FilterOptions[ ]`.
2. Select the options for `Graphics[ ]` with `FilterOptions[ ]`.
3. Compute the table of vanes over the  $(\text{horX}, \text{verY})$  grid with `uFlexVane[ ]` and its options.
4. Include the options for `Graphics[ ]` and some explicit rendering options.

OUTPUT :: Render the two-dimensional vector field.

IMPLEMENTATION :: `uVectorField2D[ ]`:

```
uVectorField2D[{f_,g_},horX:{x_,x0_,x1_,dx_},
  verY:{y_,y0_,y1_,dy_},opts___]:=
Module[{fvOpts,grOpts},fvOpts = FilterOptions[uFlexVane,opts];
grOpts = FilterOptions[Graphics,opts];
Show[Graphics[Table[uFlexVane[{x,y},{f,g},fvOpts],horX,verY],
  grOpts, Frame → True, AspectRatio → Automatic, PlotRange → All]]]
```

Observe that we apply `FilterOptions[ ]` twice to obtain sequences of options to control the graphics primitives for the vanes and to specify general graphics options.

Here is a vector field without any options.

```
uVectorField2D[{-y,x},{x,-2,2,0.25},{y,-2,2,0.5}];
```

Here is the same vector field with several options.

```
uVectorField2D[{-y,x},{x,-2,2,0.25},{y,-2,2,0.5},
  uVaneScale → 0.6, uLineStyle → {Thickness[0.008],
  RGBColor[1,0,0]},
  uPointStyles → {PointSize[0.04],RGBColor[0,0,1]},Frame → False,
  Axes → True,PlotRange → {-3,3},Background → RGBColor[0,1,0]];
```

Here is another vector field with different values for the options.

```
uVectorField2D[{Sin[x],Cos[y]},{x,-2Pi,2Pi,Pi/8},
  {y,-5/2Pi,3/2Pi,Pi/8},
  uVaneScale → 1/2,uLineStyle → {RGBColor[1,0,0],
  Thickness[0.004]},
  uPointStyles → {PointSize[0.01],RGBColor[0,0,1]},Frame → False,
  Axes → True,PlotRange → {{-2Pi,2Pi},{-5/2Pi,3/2Pi}},
  Background → RGBColor[1,1,0],PlotLabel → "A Periodic Field"];
```

The final example shows a vector field with some default values and some explicit values for the vane options as well as a general graphics option.

```
uVectorField2D[{Sin[x + y],Cos[x - y]},{x,-Pi,3/2Pi,0.1Pi},
  {y,-3/2Pi,Pi,0.1Pi},uLineStyle → {RGBColor[1,0,0]},
  uPointStyles → {PointSize[0.01]}, PlotRange → {{-4,6},{-6,4}}];
```

## Exercises

---

### EXERCISE 1: Analyzing a list of numbers.

For the questions in this exercise we assume that a list *L* of numbers (integer, rational, or real) is given. The objective of the exercise is to write functions `uArrange[ ]` and `uAnalyze[ ]` that have the list *L* as their first argument and an option as their second argument.

- (a) For the function `uArrange[ ]` design an option with the symbolic values `Automatic`, `Ascending`, `Descending`, and `NoDuplicates` that returns the list *L* as is, returns the list sorted in ascending or descending order, or returns the list (sorted or unsorted) with all duplicates eliminated. The default value of the option is `Automatic`.
- (b) For the function `uAnalyze[ ]` design an option with the symbolic values `None`, `Low`, `High`, `Median`, `Arithmetic Mean`, and `Geometric Mean` that returns the appropriate number as indicated by the option value. The default value for the option is `None`, and its effect is that the function `uAnalyze[ ]` returns the symbolic value `Indeterminate`.

### EXERCISE 2: Defining graphics options for a plot of random planar figures.

The functions `uRI[ ]`, `uRR[ ]`, and `uRandomCircles[ ]` are taken from Example 8 in the notebook `ListFct.nb`, “List Processing Functions,” in Part II; `uRandomCircles[ ]` draws a collection of random circles in the first quadrant of the plane.

```
Clear[uRI, uRR, uRandomCircles]

uRI[ ] := Random[Integer, {0,10}]
uRR[ ] := Random[Real, {0.0,1.0}]

uRandomCircles[n_Integer] :=
  Show[Graphics[Map[{RGBColor[uRR[ ],uRR[ ],uRR[ ]],
    Circle[{uRI[ ],uRI[ ]},uRR[ ]]}&,
    Range[n]], AspectRatio → Automatic, Frame → True]]/(n > 0)
```

- (a) Change the interface and the definition of `uRandomCircles[ ]` so that all options for `Graphics[ ]` can be used. Show several examples that use a variety of options.
- (b) Write a function `uRandomEllipses[ ]` that draws random ellipses rather than circles and that accepts all options of `Graphics[ ]`.
- (c) Write a function `uRandomFigures[ ]` that draws a random collection either of circles, of ellipses, of disks, of rectangles, or of polygons. The choice of planar figure should be made with an option for the function `uRandomFigures[ ]`.
- (d) Add all `Graphics[ ]` options to the function `uRandomFigures[ ]`. Show examples of plots with various types of figures and plotting options.

### EXERCISE 3: Approximating roots with Newton’s method.

In Example 2 we implemented the function `uBisection[ ]` that approximates roots using the Bisection method. Newton’s method provides another, more efficient method to

approximate roots. For example, it can find roots of even multiplicity. The following is an implementation of the function `uNewton[ ]` that returns a list of approximations.

`expr` is a function expression in a single variable `x`.  
`x` is the independent variable in `expr`.  
`x0` is the starting value for the iteration.  
`n` is the number of approximations to be computed.

```
Clear[uNewton]
uNewton[expr_, {x_, x0_}, n_] :=
Module[{g, xvals, yvals}, g = Simplify[x - expr / D[expr, x]];
  xvals = NestList[(g /. x -> #) &, N[x0], n];
  yvals = expr /. x -> xvals;
  Transpose[{xvals, yvals}]]
```

- (a) Look up Newton's method in a calculus text or in a book on numerical analysis. Write up an explanation of the method and then give the documentation for the function `uNewton[ ]` defined earlier. Follow the style that we have used in this text. See `uAnalyze[ ]` of Example 5 in this notebook with the five pieces: definition, input parameters, local variables, computations, and output.
- (b) Write a function that incorporates Newton's method and provide an option for this function. The option values select whether a table of the successive approximates is returned by the function or a plot of the list of approximations is drawn. The approximations should be drawn as points together with the graph of the function in a single plot.
- (c) Apply your function for Newton's method from part (a) to find the roots of functions such as  $x^2 - 4$  and  $\sin x$ . Explain the convergence behavior of the approximations for various starting values, for example for  $x_0 = 1, \frac{4}{3}$  and 0 with the parabola, and  $x_0 = 1, \frac{\pi}{2}$ , and 2 for the sine function.
- (d) Expand your definition of part (a) with appropriate options so that you can control the graphics properties of the approximating points as well as those of the graph of the function.

#### EXERCISE 4: A box-and-whisker plot.

This exercise extends Exercise 1 to visualize the characteristics of data. A list of numbers that is sorted in ascending order is referred to as a list of ranked data. Statisticians visualize the ranked data with a plot that consists of a box and two whiskers attached to the sides of the box. The whiskers end at the lowest and highest data values, respectively (the 0<sup>th</sup> and 100<sup>th</sup> percentile). The box is bounded on either side by the first and third quartile of the data. The median, which is the second quartile, is drawn as a vertical line inside the box.

Here is a box-and-whisker plot with whiskers at 1 and 19, and the three quartiles at 5, 12, and 15. The box is drawn as a green rectangle. All other markers are drawn in black. The ticks are drawn only at the five important locations.

```
Show[Graphics[{RGBColor[0, 1, 0], Rectangle[{5, 0.5}, {15, 1.5}],
  RGBColor[0, 0, 0],
  Line[{1, 0.9}, {1, 1.1}], Line[{5, 1}],
  Line[{1, 1}, {5, 1}],
  Line[{15, 1}, {19, 1}], Line[{19, 0.9}, {19, 1.1}]}],
```

```

Line[{{5,0.5},{5,1.5}}], Line[{{12,0.5},{12,1.5}}],
Line[{{15,0.5},{15,1.5}}], Line[{{5,1.5},{15,1.5}}],
Line[{{5,0.5},{15,0.5}}], Line[{{15,1},{19,1}}],
Line[{{19,0.9},{19,1.1}}]],
PlotRange → {{0,20},{0,2}}, Ticks → {{1,5,12,15,19},{0,1,2}},
Axes → True];

```

- (a) Use the sample plot as a guide to write a function `uBasicBoxPlot[ ]` whose only parameter is the list of five characteristic numbers for the box-and-whisker plot.
- (b) Expand the function `uBasicBoxPlot[ ]` with an option that controls the graphics of the box-and-whiskers. Examples for possible option values are the color of the rectangle or the size of the vertical endpiece of the whiskers.
- (c) Look up the precise definitions of percentiles and quartiles in a statistics textbook or reference book. Write a function `uBoxData[ ]` that takes a list `L` of numbers and returns the five characteristic numbers required for the function `uBasicBoxPlot[ ]`.
- (d) Based on your answers in parts (a), (b), and (c), write a function `uBoxPlot[ ]` that takes a list `L` of numbers as its parameter and draws the box-and-whisker plot for the data. Design graphics options for this function that you think are helpful in understanding the data better.
- (e) There are a number of variations of the box-and-whisker plot. Design an option for the different types of box-and-whisker plots; of particular interest to statisticians are plots that also indicate outliers in the data.

### EXERCISE 5: Giving names to rules.

We have used substitution rules and options in every notebook so far. However, we always just used them. Since rules are objects in Mathematica we can name them; for example, the assignment  $r = (x \rightarrow \frac{1}{1+x})$  names the rule `r`. We can use the name `r` in computations.

- (a) Apply the named rule  $r = x \rightarrow \frac{1}{1+x}$  repeatedly to symbol `x`. Evaluate the result when `x` is 1 (again using a rule substitution). The symbol `x` must stay a symbol without a value.
- (b) Use the list manipulation function `Nest[ ]`, so that you need to mention the rule `r` only once for the repeated application of the rule. Find the result of repeating the rule application 5, 10, 15, and 20 times.
- (c) Write a function `uMystery[ ]` that hides all the rule and nesting computations. `uMystery[n]` should return the result of nesting `n` times when `x` is 1.
- (d) Make a table of values for `uMystery[ ]` for  $n = 1, \dots, 12$ .
- (e) The results of part (d) should suggest to you that `uMystery[ ]` is related to a well-known function. What is that function?

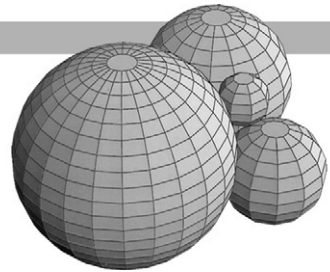
### EXERCISE 6: Chaotic behavior of function iteration.

Repeated application of the rule  $x \rightarrow \tan[x]$  will iterate the tangent function.

- (a) Use this rule together with `NestList[ ]` to define a function `uTanRule[x0,n]`. The first argument,  $x_0$ , is the starting value for the iteration and the second argument,  $n$ , is the number of iterations. `uTanRule[ ]` should return the list of the  $n$  iterates starting with  $x_0$ .
- (b) Compute lists of iterates for various starting values and graph them with `ListPlot[ ]`.

This Page Intentionally Left Blank

# *Advanced Mathematica: Hyperlinks and Buttons*



## *Introduction*

In the first part of this textbook, *Mathematica Basics*, we introduced the usage of hyperlinks and buttons that are part of Mathematica, version 4. This notebook is a short introduction to creating hyperlinks, buttons, and palettes. Hyperlinks are useful for lookups of additional information and for reminders of the usage of functions, for example. The button and palette objects are particularly helpful in computing environments that are confined to a narrow subject area and that use a small number of symbols and functions frequently.

Palettes are created from regular notebooks. Conversely, each palette can be turned back into a notebook and edited. In order to conveniently create hyperlinks and sample palettes we will use several auxiliary notebooks that are associated with this notebook.

## *Hyperlinks within a Notebook*

We begin the discussion of how to create a hyperlink with links inside this notebook. Since a hyperlink is a jump, it must have a named target to jump to. This named target is a “tag.” In order to get started we name the title cell of the section “Hyperlinks as Buttons” further on in this notebook with the tag name “Jump Buttons.” Tags may contain embedded blanks.

Recall that a cell tag is created by first selecting a cell, then choosing **Find ► Add/Remove Cell Tags ...** in the menu bar and finally typing the tag name into the text field of the tag window. Do that now with the title cell by adding another tag name for that cell.

Now we are ready to create a hyperlink to jump forward to the first cell in the section “Hyperlinks as Buttons.” We first decide on the word or phrase that appears as the hyperlink. We choose “Jump to Buttons Section”, type it, and select the phrase with the mouse. Then we choose **Input ► Create Hyperlink ...** in the menu bar and in the dialog window, first click the radio button labeled “Current notebook”, then the radio button labeled “Cells with the tag”, and then select from the list of tags the name Jump Buttons and click the OK button in the window.

Since we want to come back to this place in the notebook after the hyperlink jump we also create a tag for the text cell after the hyperlink as the target for the backwards hyperlink in the section “Hyperlinks as Buttons.”

Here is the hyperlink: [Jump to Buttons Section](#).



Here we are again! Note that we created the tag for this cell before we could define the hyperlink in the “Hyperlinks as Buttons” section. Obviously, we could have scrolled back to this cell after the previous hyperlink jump. However, hyperlinks are useful in long notebooks and get you where you want to go with one click of the mouse.

## *Jump within a Notebook without Hyperlinks*

---

There is an alternate method to jump within a notebook that does not require hyperlinks; instead tags are used directly. Once you have defined tags you can go to the menu bar and choose **Find ▸ Cell Tags**. Go to the submenu that pops up and select the tag that you want to jump to. The jump takes place when you release the mouse button.

One drawback to this method of jump is that you lose the textual context of the hyperlink since you must move the mouse cursor to the menu. Another drawback is that you must remember the location of the cell in the notebook associated with each tag.

## *Hyperlinks between Notebooks*

---

Jumping to some cell in another notebook is only slightly different from an intra-notebook hyperlink. In the dialog window, instead of clicking the radio button labeled “Current notebook” we click “Other notebook or URL.” If we know the filename of the notebook we can just type it into the text field; otherwise we click the Browse button, navigate to the folder of the target notebook, and select the notebook. As the example here, we select the notebook JumpTarget.nb associated with this notebook and click Open in the browse window. Then the list of tags defined for the JumpTarget.nb notebook (there is only one tag) is displayed in the combo field; we click the radio button “Cells with the tag” as the destination and select Target from BasButton.

Here is the example hyperlink: [Link to JumpTarget](#).

Here we are again!

There is one aspect of inter-notebook hyperlinks that can pose a serious problem. Before you define inter-notebook hyperlinks you need to carefully determine the locations of the notebooks that you are going to link. Once the hyperlink is created, the file path to the target notebook is determined and encoded in the link. If the file path changes—for example, if the notebook is moved to a different folder—then the hyperlink is broken. Mathematica only generates a beep when you click on the hyperlink and no jump is executed. You can temporarily create this situation by, for example, moving the notebook JumpTarget.nb into the Sample Palettes folder and then clicking on the hyperlink: [Link to JumpTarget](#). This link is identical to the previous one. Once you move the JumpTarget.nb notebook back to its original place, the hyperlink works again.

## *Hyperlinks to the Help Browser*

---

We repeat the method of creating hyperlinks into the Mathematica Help Browser that we discussed in the notebook BasNote.nb. Here is the hyperlink to the last section in that notebook

that discusses the Master Index in the Help Browser: [BasNote.nb](#). (Copy BasNote.nb into the folder containing BasButton.nb.)

For the sake of completeness we will also describe here how to create hyperlinks into the Help tools that Mathematica provides. The Help Browser contains six different categories: Master Index, Built-in Function, Add-ons, The Mathematica Book, GettingStarted/Demos, and Other Information. In the following text cells we create links into these tools. Each one has a similar mechanism. You select **Help ▸ Help Browser** in the menu bar, and then the appropriate category. In case the categories do not show, click the button Show Categories in the Help Browser.

Here is a link to the Master Index. We click on Master Index, then letter N, and then we scroll down to Natural logarithm. Finally, we copy the one link shown in the window and paste it into this notebook: [1.1.3](#). As you can see, the link puts us eventually into Section 1.1.3 of *The Mathematica Book*, as was advertised in the Help Browser window. The Master Index is the appropriate start when you do not know in which of the six help tools the particular term resides that you are looking for.

Suppose we want to know the options of NIntegrate[ ]. We do know that this is a built-in function so we select that button in the Help Browser. However, we also need to know or guess that NIntegrate[ ] as a numerical function is listed under Numerical Computation. Then we can select Integration and finally NIntegrate. This gives us the information in a manual search in the Help Browser. In order to define a hyperlink with NIntegrate as its target, we again go into the Master Index, select the letter N and then NIntegrate. This time we copy the hyperlink that is listed under the heading Built-in Function in the help window and paste it here into this notebook: [NIntegrate](#). If we want a hyperlink to some examples and the usage of NIntegrate[ ] we copy the link given under the heading *The Mathematica Book* in the help window: [1.6.2](#).

## *Hyperlinks to Internet Resources*

---

An even broader spectrum of help and support tools is available to a notebook through a hyperlink target that is an Internet address. All we need to know is the URL, the universal resource locator, of the notebook that we want to make available to users of this notebook at this point. We will choose as one example the obvious resource site of Mathematica: MathSource. A second obvious resource site is the folder of additional notebooks provided by an instructor for student use or demonstration purposes.

Here is a link to a notebook on the construction of sundials available on the Mathematica MathSource site. The URL to that notebook is <http://www.mathsource.com/MathSource/Applications/Other/0209-001/SundialConstruction.nb>. Hopefully, this URL is still active and correct when you read this. If not, choose another notebook on the MathSource site.

We name the link Sundial Construction. We typed that phrase below, selected it, and created the hyperlink as described before. This time we select the URL radio button and type in the URL given earlier: [Sundial Construction](#).

Clicking on the link will download the file to a folder that we specify in the pop-up folder/directory selection window. Assuming that the file was saved on the desktop with the same name SundialConstruction.nb as on the Mathematica site, we now open that file with another hyperlink: [Open Sundial Construction](#). (Or, double-click the SundialConstruction.nb icon.)

Note that you can select only Mathematica notebooks as target files of hyperlinks since a hyperlink opens its target file in Mathematica. As the default setting in Mathematica, the file navigation window will display only Mathematica notebook files when you browse for a filename.

## *Hyperlinks as Buttons*

---

Before we go into the topic of this section we give the hyperlink [Jump to Hyperlinks Section](#) to jump back to the section “Hyperlinks within a Notebook.”

Many of the functions that can be accomplished by hyperlinks are also implemented as buttons. There are several predefined types of buttons. You can inspect these types by choosing **Input ▸ Create Button** in the menu bar. The pop-up menu lists a hyperlink category at the top and then five of the categories that are available in the Help Browser.

As the first example we define a button that is a hyperlink to the previous subsection. We create an empty input cell or position the cursor below this text cell and choose **Input ▸ Create Button ▸ Hyperlink** in the menu bar. Then we type the words “Previous Section” in the placeholder on the button since that is the tag of the title cell of the previous subsection. When we click on the button nothing happens.

**Previous Section**

The reason is that we have not specified any script that determines the action of the button, namely what the target of the hyperlink action is. For that we need to select the button, choose **Input ▸ Edit Button...** in the menu bar, and enter in the text field for Button Data the string expression “Internet Hyperlinks”, which is the tag for the title cell of the previous section. Note that the selection box for Button Data gets checked when you type into the field. Then we click the Apply and OK buttons in the edit window. Be sure to include the double quotes since the function expects a string. Here is the redefined button:

**Previous Section**

Now the hyperlink button works. If we want to have this button always active, and thereby prevent any editing of the button, we need to choose **Cell ▸ Cell Properties ▸ Cell Active** in the menu bar. In order to show this we copy the input cell with the button from before and then make the cell active.

**Previous Section**

Observe that the shape of the cursor does not change anywhere in the last input cell; it always is the arrow. We can change the cell to inactive by clicking the checked item Cell Active in the menu bar **Cell ▸ Cell Properties ▸ Cell Active**.

If we want to jump into another notebook using a button, we follow the steps for creating and assigning an action to a button as explained earlier, except that we need to specify a list of two text strings, the name of the notebook and the name of the tag of the cell that we want to jump to. As an example we choose again the notebook `JumpTarget.nb` and jump to the same tagged text cell that we used in the hyperlinks earlier in this notebook.

**Button to JumpTarget**

When we specify the name of the notebook together with the empty string "" as the Button Data then the target of the hyperlink is the beginning of the notebook.

**Top of JumpTarget.nb**

To jump to a notebook in a different folder, we must provide the entire file path to the notebook. We use the sundial construction notebook, SundialConstruction.nb, as our file path example. We have downloaded this notebook previously from the Mathematica MathSource site on the Web and assume that we saved it on the desktop.

Obviously, the actual file path is likely to be different on any computer that you are going to use. Here is an effortless way to use Mathematica itself to compute the file path for us. We create an empty input cell below this text cell and leave the text insertion cursor in that cell. Then we choose **Input** ► **Get File Path...** in the menu bar and follow the standard file selection dialog down to the target file SundialConstruction.nb. The entire file path will be written into the input cell.

The file path for Macintosh OS 9.2 could be:

**"Hard Disk:Desktop Folder:SundialConstruction.nb"**

The file path for Macintosh OS X could be:

**"/Users/Guest:Desktop/SundialConstruction.nb"**

The file path for Windows 2000 could be:

**"C:\\Documents and Settings\\Guest\\Desktop\\SundialConstruction.nb"**

Instead of creating an empty input cell, we could have left the text insertion cursor in this text cell, for example after the colon in this cell. Then the file path will be inserted into this text cell here: "Hard Disk:Desktop Folder:Sundial-Construction.nb" for Macintosh OS 9.2.

Now we copy the entire text string for the file path, including the double quotes, select the button in the cell below, and choose **Input** ► **Edit Button ...** in the menu bar. Then we paste the file path into the Button Data text field and click the OK button. Since we did not specify a tag for a target cell, the sundial notebook will open at its beginning. The button contains the file path for Macintosh OS 9.2 created earlier.

**Sundial Resource**

## *Creating a Typesetting Palette*

---

In the notebook BasText.nb, "Mathematica Basics: Text and Typesetting," we explained and used the palettes that are provided with Mathematica. In this section we will demonstrate how we can construct our own individual typesetting buttons and arrays of such buttons that make up palettes.

The action that we associate with a typesetting button is to paste a character or symbol or sequence of objects into a notebook. The pasting occurs at the location of the insertion cursor.

We start with a button that contains the text “Pi”, and a button that contains the Greek letter  $\pi$ . We can enter the Greek letter  $\pi$  from the BasicTypesetting palette or we can enter the ASCII equivalent of the Greek letter `\ESC\p\ESC`. Observe that when we finished entering the three actual key strokes: ESCAPE-key, p-key, ESCAPE-key, the Greek letter  $\pi$  is rendered in the notebook.

Creating a button for typesetting requires the following steps:

1. Type the symbol or sequence of symbols in an input cell.
2. Select the content you just typed.
3. Choose **Input** ► **Create Button** ► **Paste** in the menu bar.
4. Select the newly created button for definition of action.
5. Choose **Input** ► **Edit Button...** in the menu bar.
6. Select Paste as the Type of the Button (top of the window).
7. Check Button always active (bottom of window).
8. [Optional] Click on Button Note and enter explanatory text in its field.
9. Check Apply and then check OK.
10. Choose **Cell** ► **Cell Active** in the menu bar.

**Pi**

Once the button has been created, we put the text insertion cursor into this cell and click the Pi button several times to see its action. For the button in the input cell above we skipped step 8.

$\pi$

For the second version of the button we included step 8 and typed Pi into the Button note text field. When we put the cursor over the button, the status field in the lower left-hand corner of the notebook window shows the text Pi that we entered during the creation of the button. Test out the use of the two buttons in this text cell.

Rather than specifying individual buttons, each in a separate cell, we can also create an array of buttons. We will create a button sequence of four buttons for the lower- and uppercase of  $\pi$  in Latin transcription and as Greek letters. For the purpose of defining the sequence of buttons we place the cursor into an empty input cell and choose **Input** ► **Create Table/Matrix/Palette** in the menu bar. In the pop-up window we click the radio button for Palette, specify the size of the palette as one row and four columns, and then click OK. Then we click on the placeholder in the first button and follow steps 4 through 7 for each button in the sequence. When we are done we make the entire input cell active so that we cannot accidentally change the buttons.

<b>pi</b>	$\pi$	<b>Pi</b>	$\Pi$
-----------	-------	-----------	-------

Note that at this point we have not created an actual palette, but the content in a notebook from which we can create a palette. In order to make a palette, all we need to do is to select the input cell containing the button sequence and choose **File** ► **Generate Palette from Selection** in the menu bar. A very small, separate, untitled window in the style of a palette window is created.

In order to save the palette, we close the palette window, click the Save button in the pop-up dialog window, and then save the palette in the appropriate folder.

We use the convention to start the name of any palette with the two characters p-, for palette, followed by a suggestive file name that describes the content of the palette. For this palette we choose the name p-Pi.nb and place it in the Sample Palettes folder.

Now we open the palette notebook again, position it on the desktop, and can use the palette freely. We position the insertion cursor into this text cell, click each button in the palette, and type a dash in between each click:  $\pi$ - $\pi$ -Pi- $\Pi$ .

We can also create a notebook from a palette, the reverse of the process of creating a palette from a notebook or selection in a notebook. For a sample conversion we select the palette p-Pi.nb, choose **File** ▸ **Generate Notebook from Palette** in the menu bar, and can then save the untitled notebook. For a notebook generated from a palette we use the convention to start its name with the two characters n-, for notebook. We save this notebook as n-Pi.nb. Note also that we can print a notebook generated from a palette, but not the associated palette.

When we enter each of the four symbols into an input cell as arguments of N[ ], then we see that the two middle symbols in the palette actually have numeric meaning in Mathematica, whereas the two outer symbols in the palette are just symbols for typesetting.

```
N[ $\pi$ ]
N[ $\pi$ ]
N[Pi]
N[ $\Pi$ ]
```

As you explore the Mathematica palettes further you will find several symbols with computational meaning. For example, the right arrow  $\rightarrow$  is determined by the ASCII sequence `\ESC>`. The rule arrow  $\rightarrow$  used in options looks the same, but is determined by the ASCII sequence `\ESC->\ESC`, missing the leading blank. Mathematica contains a list of 721 named characters in Appendix A.12 of the *Mathematica Book*.

## Creating a Palette of Expressions

---

In this section we create a palette with arithmetic operations and trigonometric functions. The actual Mathematica function to be evaluated is put as the data on the button and a selection placeholder object is used for each argument of the function. One common design and programming process is to take existing objects and rearrange or customize them for a new purpose in a new notebook. Often this is much easier and less time-consuming than creating a new object from scratch.

As an example, we will create a palette consisting of a  $1 \times 2$  button matrix containing the sine and the cosine functions, and a  $2 \times 1$  button matrix containing the addition and multiplication functions. We construct this from two arrays of buttons in the BasicCalculations palette. Choose **File** ▸ **Palettes** ▸ **Basic Calculations** in the menu bar and open Arithmetic and Numbers as well as Trigonometric and Exponential Functions and Trigonometric within it. Make the BasicCalculations palette active by clicking on its top bar and choose **File** ▸ **Generate Notebook from Palette** in the menu bar.

Exactly those cells that we opened in the palette are now open in the associated untitled notebook. We position the cursor at the end of the Basic Operations button array and delete the bottom row first. We can also select the entire bottom row and use **Edit ▸ Cut**. Then we delete two of the buttons in the remaining row so that the  $\square + \square$  and the  $\square \times \square$  operations are left. We perform a similar action in the Trigonometric section on the second column and the  $\text{Tan}[\square]$  button.

Now we cut all cells from the notebook until only the  $1 \times 2$  trigonometric row and the  $2 \times 1$  numeric column are left. The next step is to save the notebook in the Sample Palettes folder as n-Trig.nb and to generate the associated palette by selecting the two cells in the notebook and choosing **File ▸ Generate Palette from Selection** in the menu bar. Finally we close the new palette, name it p-Trig.nb, and put it in the Sample Palettes folder.

We open the palette p-Trig.nb, click on a button in it, and put a numeric or symbolic argument in the placeholder or click on another button in the palette. Two sample expressions follow. Pressing the TAB key will select the next placeholder and we can enter an argument into the function or expression. When we press the ENTER key (SHIFT-RETURN keys) the expression is evaluated.

```
Sin[Pi/2]
```

```
Sin[Cos[Pi/4]] + Cos[3 x Sin[Pi/8]]  
N[%]
```

## Creating a Palette of Characters and Expressions

---

We follow the basic sequence of steps described in the previous section to create a palette. We will add a few features to the palette and include buttons for typesetting characters in text or input cells as well as buttons for operations in expressions in input cells. As an example we will create a palette that provides tools for the lower- and uppercase letters in the Script alphabet, the standard logic operators, and some relational operations. All items are in the CompleteCharacters palette, which we open by choosing **File ▸ Palettes ▸ CompleteCharacters** in the menu bar. We generate the associated notebook from the menu and save it as n-CompleteCharacters.nb. Then we open a new untitled notebook in which we build the new palette and name it n-Logic.nb. The associated final palette we will name p-Logic.nb.

We start by copying the first cell group in the n-CompleteCharacters.nb notebook and change its title Letters to Logic Palette. Then we keep the *Script* cell group, including the title which we change to *Script Alphabet*. Next we copy the *General* cell group, change its title to *Logic Operations*, and select and delete the first, third, and fourth subarray of that button array. That leaves a  $3 \times 7$  button array in the cell. In the *Relational* cell group the button array consists of eight button arrays arranged as a  $4 \times 2$  matrix. We copy the entire cell group and delete all button subarrays except the one in the lower left, which contains 12 set-theoretic relational operators. We replace the *Relational* title by *Set-theoretic Relations*. We delete the *Arrows* cell group.

At this point we are done with the notebook n-CompleteCharacters.nb and close it. The final step is to create an actual palette from the notebook n-Logic.nb. We select all cells in the notebook n-LogicPalette.nb (that should be the single outermost cell level) and choose **File ▸ Generate Palette from Selection** in the menu bar. That action creates the new palette in an untitled window, which we close and save as p-Logic.nb through the dialog boxes.

We now close all notebooks and palettes, except this BasButton.nb notebook, and open the palette Logic.nb again. Unfortunately, the window of the palette is fixed in size and we must scroll to get to the different button arrays in the palette. However, we can use the Option Inspector to change the properties of the palette window to suit our needs here. Make the p-Logic.nb palette the active window by clicking in its top bar. Choose **Format ▸ Option Inspector ...** in the menu bar, set it to notebook, open the Notebook Options then the Notebook Properties, and set the Editable property to True by clicking in the box on the right. That action allows us to edit items in the Windows Properties. Add StatusArea to the WindowElements and add ResizeArea to the WindowFrameElements properties by clicking on those values in the respective pop-up windows on the right-hand side in the Option Inspector. When we are done we go back to the Notebook Properties and set the Editable property back to False by clicking in the box on the right.

The rest of this section was created with the help of the p-Logic.nb palette.

We formulate the Law of the Excluded Middle:  $x \vee \neg x = \text{True}$  and the Law of Contradiction:  $x \wedge \neg x = \text{False}$ .

The definition that set  $\mathcal{A}$  is a subset of set  $\mathcal{B}$ , denoted by  $\mathcal{A} \subset \mathcal{B}$  or  $\mathcal{B} \supset \mathcal{A}$  is  $\forall u(u \in \mathcal{A} \Rightarrow u \in \mathcal{B})$ .

It turns out that some symbols are not just alphabetic characters, but actually are the infix forms of the standard logic functions that are implemented in Mathematica. For example, the symbol  $\wedge$  is the function And[ ] as we can see in the next input cell. We associate the Script letters  $\mathcal{T}$  and  $\mathcal{F}$  with the values True and False, form a logic expression, and substitute the truth values  $\mathcal{T}$  and  $\mathcal{F}$  for the variables in the expression.

```
Clear [z, w]
```

```
 $\mathcal{T} = \text{True}; \mathcal{F} = \text{False};$ 
```

```
 $(z \wedge \neg w) /. \{z \rightarrow \mathcal{T}, w \rightarrow \mathcal{F}\}$ 
```

The next expression is a tautology since the conclusion is the constant True. However, the implication  $\Rightarrow$  does not simplify so we use the function LogicalExpand[ ] to force the evaluation of the expression.

```
 $(\neg z) \Rightarrow \mathcal{T}$ 
```

```
LogicalExpand[%]
```

We can see what the function LogicalExpand[ ] actually does by using an implication with symbolic arguments. Observe that we did not assign values to  $z$  and  $w$ , we just substituted truth values for them.

```
 $w \Rightarrow z$ 
```

```
LogicalExpand[%]
```

Mathematica applies the standard logic equivalence  $z \vee \neg w$  for the implication  $w \Rightarrow z$ .

## *Creating an Evaluation Palette*

---

All the buttons in palettes that we have created so far assist us in writing expressions in text cells as well as in input cells. No evaluation action is associated with the buttons. When we



created an expression in an input cell we still had to execute that cell by pressing the **[ENTER]**-key or the **[SHIFT+RET]**-keys. Buttons and palettes are also quite useful when they can be used to perform evaluations of expressions in a notebook. We demonstrate this feature with two simple, trigonometric, one-argument functions. Two types of evaluations are provided: in-place evaluation of a selected expression and an evaluation performed on a copy of a selected expression. The actual Mathematica function to be evaluated is put as the data on the button and a selection placeholder object is used for each argument of the function.

We create a small palette that has just two separate buttons, one for each type of evaluation, with a subsubsection cell before each button. We use the cosine function for in-place evaluation and the sine function for copy evaluation. We create each button in the notebook as a 1×1 palette. After we have entered the action function on the button, we select the entire button, choose **Input ▸ Edit Button ...** from the menu bar, and select the property Evaluate in the Button Style field. In addition, we added text into the Button Note field that will show in the Status area of the palette window when the cursor moves over the button in the palette.

Towards the end of the previous section we discussed how to modify the properties of a palette window. We added the Status Area and a scrollbar to the palette to improve the layout.

Here is an example of the action of an in-place evaluation button. We start with the value  $N[\text{Pi}/10]$  in the following two input cells, select the value in the second cell, and apply the cosine function six times from the palette p-Evaluate.nb. The in-place evaluation is useful for experimental iterative calculations where we do not need to keep the intermediate values.

**$N[\text{Pi}/10]$**

**0.7084082267527573`**

Note that the cell stays an input cell and that the expression in it remains selected to be acted upon again by some button or other manipulation such as a Copy action. The original value as well as the actual function applied to the last value are not displayed. There is no In[ ] and Out[ ] tag associated with the input cell. All that we are left with is the final result of the computation. We have to remember or document in writing which computations actually were performed to produce the answer displayed in the notebook.

The copy evaluation is similar, except that all computed values are stored in different input cells. We start with the value  $N[\text{Pi}/5]$  this time, select it, and apply the sine function six times. Again, no In[ ] and Out[ ] tags are associated with the computations. Note that as we invoke the sine function from the palette the result of the last button invocation is the new selected value.

**$N[\text{Pi}/5]$**

**0.5877852522924731`**

**0.5545193359484235`**

**0.5265347227580315`**

**0.5025404462268308`**

**0.4816534404406869`**

**0.4632451369283593`**

## *Printing a Palette*

---

As we have seen throughout this notebook we always dealt with two notebooks when we defined and manipulated a palette. There is the actual palette, say p-Logic.nb, and the notebook n-Logic.nb that we used to create the palette p-Logic.nb and that we can recreate from the palette with the menu action **File ▶ Generate Notebook from Palette**. You will find that the notebook generated from a palette has page breaks set before every button array and text cell. In order to avoid the many printed pages that this creates when you print n-Logic.nb we need to remove the page breaks and have Mathematica compute the location of the page breaks for us automatically.

We discussed page break preferences for entire notebooks in the notebook BasNote.nb, “Mathematica Basics: Notebooks.” The actions for palettes are slightly different since they affect individual cells, so we mention the details here also. Select a cell from which you want to remove the page break that was set. Open the Option Inspector **Format ▶ Option Inspector ...** from the menu bar and set the **Show options values for field to selection**. Now we select **Cell Options ▶ Page Breaking** and uncheck the property **PageBreakBefore**. That sets the property to **Automatic**. Alternatively you can set the value to **Automatic** by selecting that value directly from the pop-up menu of the button on the right. Now you can repeat the same process with any other cell for which you want to change the page break setting.

## *Exercises*

---

**EXERCISE 1:** Cell options in the Option Inspector.

The Option Inspector permits various settings for cells. Choose **Format ▶ Option Inspector ... ▶ Cell Options ▶ Display Options** from the menu bar. Make a copy of the notebook CellHierarchy.nb and use the copy for experiments in this exercise.

- (a) Choose different values for the CellDingbat property and observe the effect on the notebook.
- (b) Turn on the property ShowGroupOpenCloseIcon and observe the effect of the property on the notebook.

**EXERCISE 2:** A typesetting palette.

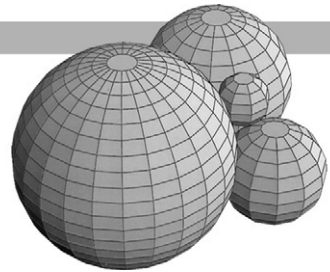
Create a palette for typesetting in an area of science in which you are interested—for example, a palette of constants in physics or engineering that you use frequently.

**EXERCISE 3:** Links to Mathematica documentation.

Take a notebook from this text and add links into the Help Browser for all Mathematica topics for which you need help and would want more detail than is provided in the notebook.

This Page Intentionally Left Blank

# *Advanced Mathematica: Packages*



## *Introduction*

A package is a mechanism to organize a collection of functions in a notebook. To get access to these functions, we can open the package, evaluate it, and then use the functions from the package in any other notebook, or we can load the package from within a notebook with the `Needs[ ]` function.

The package mechanism provides a single location for the functions and there is no need to copy functions into every notebook that uses them. If we want to change the implementation of a function then we need to make the change in only one place, the package in which the function is defined. When duplicate copies of a function exist in a variety of applications notebooks then, inevitably, inconsistent versions of the function will develop over time.

The functions that we put into packages are designed and written in the usual Mathematica style as we have done up to this point. In the previous notebooks we have followed the convention of using a lowercase `u` as the first letter of a user-defined function. There are good reasons for changing this convention when we design packages, and we will do so. We can think of the functions in a loaded package as an additional computing resource just like the built-in functions of Mathematica. Therefore, the same naming conventions as for the built-in functions ought to be used for the functions in packages.

We collect all packages from this notebook as individual notebooks in the folder (subdirectory) `PackageSupport`. In the section “Package Files” of this notebook we explain how user-defined packages can be created and loaded from within Mathematica. For that purpose we ask you to create a separate folder for the same packages as an experimentation folder. The process of loading packages is similar to loading standard packages that are shipped with Mathematica.

## *Contexts and Names*

The main purpose of a package is to set up a context for all the names in the package. We can use any name for a function or a data item without having to worry about whether this same name is used somewhere else, that is, in a different context within Mathematica. We have used something like this context mechanism when we used the function `Module[ ]`. The local variables that are defined inside `Module[ ]` are bound to the context of its body.

**Example 1:** Determining the context of a symbol.

```
Clear[x, Helix]
```

```
??Context
```

```
Context[x]
```

```
?x
```

The backquote symbol, ```, at the end of the context name is the context mark, which separates the name of a symbol defined in a particular context from the name of that context. The name of a symbol without its context name prepended to it is called the short symbol name. Normally, only the short names are used. All names that we define directly in Mathematica are in the `Global`` context, and all functions and symbols predefined in Mathematica have the context `System``.

```
Helix[a_:(2 Pi)] := ParametricPlot3D[{Cos[t], Sin[t], t/a}, {t, 0, a}]
```

```
Helix[3 Pi];
```

```
Context[Helix]
```

```
?Helix
```

```
Context[ParametricPlot3D]
```

We can inspect the contexts that are currently active with the system variable `$ContextPath`.

```
$ContextPath
```

Observe that the context `Global`` is “on top of” the context `System``. Therefore, before the context `System`` is searched, Mathematica searches the context `Global``.

**Example 2:** Adding the context of a package.

We load a package into the current Mathematica session.

```
Needs["Graphics`Shapes`"]
```

```
Names["Graphics`Shapes`*"]
```

Observe that the `Shapes` package contains the name `Helix`, which is listed with its full name. Once we evaluated the package we changed the list of contexts and we inspect this list.

```
$ContextPath
```

Actually, two new contexts were added because the package `Shapes.m` in the folder `Graphics` needs another package from the `Geometry` folder of packages. Observe the order of these contexts in the context path list.

As with the contexts established by packages we can also see all the names in the Global' context.

```
Names["Global`*"]
```

However, as soon as we have several contexts active we must be careful to remember the sequence of contexts in the context list. Mathematica will always search the current context first. Its value is stored in the system variable \$Context. If the symbol is not found in the current context then the context list is searched.

```
$Context
```

Indeed, when we loaded the package Shapes, Mathematica warned that the symbol Helix occurs in two different contexts. When we use the name Helix we get the meaning of the symbol Helix in the Global' context. Only when we give the full name Helix with the context Graphics'Shapes' do we get the Graphics3D object from the package Shapes.

```
?Helix
```

When we are at the interactive level of the notebook, as we have always been up to this point, the context is the Global' context. That explains the results of the previous evaluation.

```
?Graphics`Shapes`Helix
```

```
Show[Graphics3D[Graphics`Shapes`Helix[ ]]];
```

```
Helix[ ];
```

At this point we want to describe the action of the Clear[ ] function in more detail. Its effect is only to clear the definition, that is, the value of each of its arguments. Mathematica retains the knowledge that the symbol has been used and the context in which it has been used. It does not remove the symbol. We show this with the symbol Helix. If we want to remove the symbol Helix then we must use the function Remove[ ].

```
Clear[Helix]
```

```
Names["Global`*"]
```

```
?Helix
```

```
?Remove
```

```
Remove[Helix]
```

```
Names["Global`*"]
```

```
?Helix
```

Now we get the definition of the graphics object Helix[ ] from the Shapes package because it is the only context in which the name Helix is known.

## *Initialization Cells*

---

When we load a package into the current session, the definitions of all objects in the package must be evaluated. This evaluation process is automated in Mathematica with a special type of input cell, the Initialization cell. The Initialization cells can be set up in such a fashion that the evaluation takes place automatically or that it requires an affirmation before the evaluation is performed.

We show the use of Initialization cells in two notebooks associated with this one, `InitCellA.nb` and `InitCellM.nb`. The notebooks contain a single function definition for ease of demonstration, rather than a collection of definitions. Initialization cells are most useful when a large collection of functions is defined that set up a specific computing environment.

**Example 3:** Bisection defined in an Initialization cell with automatic evaluation.

Use this hyperlink to the notebook with automatic evaluation of Initialization cells: [InitCellA.nb](#). You have to click Yes in the pop-up window to evaluate all Initialization cells.

**Example 4:** Bisection defined in an Initialization cell with manual evaluation.

Use this hyperlink to the notebook with manual evaluation of Initialization cells: [InitCellM.nb](#). You will not see a pop-up window since the evaluation of Initialization cells is not tied to the opening of the notebook.

## *The Basic Scheme of a Package*

---

An introduction to the Standard Packages in the AddOns folder of Mathematica is available by selecting **Help** ► **Add-ons ...** in the menu bar and then selecting **Standard Packages** ► **Introduction** ► **The Standard Add-on Packages** in the scroll windows from left to right. For a particular group of packages such as the Calculus packages, click the **Add-ons** button and then select **Standard Packages** ► **Introduction** ► **Calculus Packages**.

We can get descriptions about the packages in this way and can load the functions defined in the packages into our current Mathematica session with the function `Needs[ ]`, or equivalently, the operator `<<`, as the functions are needed in computations.

```
Needs["Calculus`"]
```

```
<< Calculus`
```

A specific package is loaded immediately by its name.

```
<< Calculus `VectorAnalysis`
```

```
Names ["Calculus `VectorAnalysis`*"]
```

However, we do not see how the functions in the Calculus package are defined. In this notebook we describe a standard layout for a Mathematica package and explain how to define functions

in a package. We get some guidance when we open a package notebook from the Standard Packages.

In the next paragraphs we describe the path to the package `SurfaceOfRevolution.m` on a Macintosh and in Windows, respectively. In order to have uniformity in the file opening path sequence, we start at the desktop level. We locate the package through the menu **File** ▶ **Open ...** and then select in the file window the Desktop as the starting point.

In Macintosh OS 9

We assume that the disk is named Hard Disk and all application programs are in the folder Programming. Substitute the names that apply to your computer: **Desktop** ▶ **Hard Disk** ▶ **Programming** ▶ **Mathematica 4.1 Files** ▶ **AddOns** ▶ **StandardPackages** ▶ **Graphics** ▶ **SurfaceOfRevolution.m**. Click on the Open button. This is a notebook that was developed in 1990 for Mathematica version 2.0.

In Macintosh OS X

We assume that the disk is named Hard Disk and all application programs are at the top level of the disk. Substitute the names that apply to your computer: **Hard Disk** ▶ **Mathematica 4.1 Files** ▶ **AddOns** ▶ **StandardPackages** ▶ **Graphics** ▶ **SurfaceOfRevolution.m**. Click on the Open button or on the icon in the rightmost pane of the window. This notebook was developed in 1990 for Mathematica version 2.0.

In Windows 2000

The sequence of selections is **Desktop** ▶ **My Computer** ▶ **Local Disk (C:)** ▶ **Program Files** ▶ **Wolfram Research** ▶ **Mathematica** ▶ **4.1** ▶ **AddOns** ▶ **StandardPackages** ▶ **Graphics** ▶ **SurfaceOfRevolution.m**. Be sure that in the “files of type:” list box at the bottom of the window either “All files (\*.\*)” or “Packages (\*.m)” is selected. Click on the Open button. The package `SurfaceOfRevolution.m` was developed in 1990 for Mathematica version 2.0. This selection process for the package through the File menu occurs inside Mathematica, and the notebook is opened as a Mathematica notebook. If, however, the selection was done by double-clicking on the My Computer icon, the search for the package occurs in Windows through double-clicking on icons in Windows. The result of the search may be different. Since Mathematica saves packages as text files they appear with a Notepad icon, and double-clicking on that icon will open the package as a text file in Notepad, not as a package in Mathematica, unless files with .m extensions have been set on your computer to open with Mathematica through the Folder Options in the Control Panel.

Follow this process to another notebook of more recent design in the Graphics folder. Select **InequalityGraphics.m** and click on the Open button. It was developed in 1998 for Mathematica version 4.1.

Notice that each package opened in Mathematica consists of a single cell, an Initialization cell. That cell has to be evaluated to activate the functions in the package. Both packages start with introductory comments about title, author, history, and key words. After the comments, both have a line starting with `BeginPackage[ ]`, and both have the line `EndPackage[ ]` at the very end. Between these two lines we find the usage statements for the functions defined in the package. Between the `Begin[“Private”]` and the `End[ ]` statements is the Mathematica code for the definitions of the functions.

The layout provides a uniform structure for notebooks. It has three distinct parts and an optional fourth part.



### 1. Introductory Comments

This part contains the title, author, version, date of creation, key words, summary description, restrictions for applications, and so on, as detailed later. No particular order is required for these items.

### 2. Interface

This part contains all those symbols, called public symbols, that are defined in the package body and that are made available for use. This typically is done by defining the usage of the symbols. For example, the name of a function is revealed and the use of its arguments is described, but not its definition and the symbols that appear in it.

### 3. Implementation

This part contains the definition of every function mentioned in the Interface section. In addition, it contains the definition of, or reference to, every auxiliary function that is required in the definition of the public symbols. All such auxiliary functions are private to the context of the package and not intended for use outside of the package.

The entire package is implemented as a single input cell and the content is set up as an Initialization cell. This means that all comments and all documentation are the Initialization cell rather than as ordinary text in separate text cells. Explanatory text is enclosed in the formal comment brackets (\* and \*) of input cells. All packages that are included in Mathematica exhibit this organizational form.

### 4. Examples

Some packages include at the end explicit invocations of the public functions. These invocations are enclosed in comment brackets since they document sample applications that can be used for testing purposes of the functions defined in the package.

The first part and the optional fourth part of a package, as indeed all documentation, have no effect on the implementation of a package and therefore are not required. However, documentation is essential for understanding the content of a package and for remembering the design purposes of the functions in the package. For those reasons any package without this standard documentation is incomplete.

#### Example 5: A package template.

We design a template that contains the essential elements of the package framework. We include these as placeholders in every package that we write. In particular, the template contains a usage message for its one public function `Fpublic[ ]`, the definition of a private function `pFprivate[ ]`, and the definition of the public function `Fpublic[ ]`. The public function invokes the private function. For simplicity, the public function prints a simple message.

The following template is given in a single input cell; in this notebook the cell is not an Initialization cell. However, the template is reproduced as a package in the notebook `Template.m` in the `PackagesSupport` folder on the CD-ROM.

```
(* :Title: Template *)
(* :Author: Anonymous *)
(* :Date: Thanksgiving Day, 2001 *)
```

```

(* :Copyright: 2001, The Wizard *)
(* :Package Version: 0.5 *)
(* :Mathematica Version: 4.1 *)
(* :Keywords: String *)

(* :Summary: This package defines a private function that
    returns a message and a public function that prints the message
    of the private function. This package is intended as a template.
*)

BeginPackage["Template`"]

(* Usage message for the public function *)
Fpublic::usage = " Fpublic[ ] prints a message.";
Begin["`Private`"]

(* Definition of the private function *)
pFprivate[ ] := "the private function"

(* definition of the public function *)
Fpublic[ ] := Print["The public function prints ", pFprivate[ ]]
End[ ]

EndPackage[ ]

(* :Examples:
?Fpublic
Fpublic[ ]
*)

```

A package sets up a new context for all the symbols defined in the package. Its context name is stated as the argument of the `BeginPackage[ ]` function. In our example the name `Template`` serves that purpose. When the `EndPackage[ ]` function is evaluated the previous context is restored as the current context. If for any reason the function `BeginPackage[ ]` is evaluated while the function `EndPackage[ ]` is not evaluated then all subsequent computations occur in the context of that package because that is the current context after `End[ ]` has been evaluated. This situation can lead to confusion about active contexts and accessible names.

During the evaluation for the `Template` package the three bracketing functions `BeginPackage[ ]`, `Begin[ ]`, and `End[ ]` return the contexts that are opened or closed as the package is evaluated; `EndPackage[ ]` does not return a value, but it closes the context of the package and puts the context on the context search path.

### **?EndPackage**

Depending on the computer and the type of input cell, Initialization or not, Mathematica reports a different number of evaluations for the definition of a package context. However, the `Template`

package evaluation always produces three output cells, one for the context of the package, one for the opening of the private context, and one for its closing.

```
$ContextPath
```

```
$Context
```

```
Names["Template`*"]
```

We get the name of the single public function in the package. Here is an invocation of it.

```
Fpublic[ ]
```

When we use the `?` operator for the public function, its usage message is displayed.

```
?Fpublic
```

The private function in the package appears to be hidden.

```
?pFprivate
```

However, we can see the definition of the private function if we know its name and context path.

```
?Template`Private`pFprivate
```

It is common practice to include examples, inside a comment, at the end of a package. We will not do that in this notebook since we demonstrate several examples for each package explicitly. We do, however, include examples in the associated `.m` package files in the `PackagesSupport` folder on the CD-ROM.

**Example 6:** A conversion package for the Celsius and the Fahrenheit temperature scales.

Many thermometers display the Celsius scale as well as the Fahrenheit scale. In this example we write a package that converts between the two scales. We assume that the conversion is for casual use rather than scientific use. Therefore, the conversion functions will expect whole numbers as inputs and will return the temperature always to the nearest whole degree.

We write a package that conforms to the standards used in the template package and that implements a function `CtoF[ ]`, which converts a temperature given in degrees Celsius to degrees Fahrenheit, and a function `FtoC[ ]` that does the reverse conversion. As we mentioned earlier, we will use the same naming conventions for functions defined in packages as for the built-in functions.

We also want to exhibit the use of private functions. They are used as auxiliary functions inside the body of the package and are not made available for use by a usage message. As an example we define two private functions that test if the temperature entered is above absolute zero, that is, at least  $-273.15^{\circ}\text{C}$  (Celsius) or at least  $-459.67^{\circ}\text{F}$  (Fahrenheit), respectively.

The names of private functions will start with a lowercase `p`. This convention distinguishes them visually from the public functions defined in a package.

The conversion formula from  $^{\circ}\text{F}$  to  $^{\circ}\text{C}$  is  $\frac{5}{9}(t - 32)^{\circ}\text{C}$  for a temperature of  $t^{\circ}\text{F}$ . The reverse computation is  $\frac{9}{5}s + 32$  for a temperature of  $s^{\circ}\text{C}$ .

```

(* :Title: Temperature Conversion      *)
(* :Author: Gabriel Fahrenheit          *)
(* :Date: Summer of 1715                *)
(* :Copyright: 1730, Anders Celcius    *)
(* :Package Version: 1.1                *)
(* :Mathematica Version: 4.1           *)
(* :Keywords: Units, Conversion        *)

(* :Summary: This package implements the conversion
    between the Celsius and the Fahrenheit temperature scales.*/)

BeginPackage["Thermometer`"]

(* :Usage Messages: *)

CtoF::usage = " CtoF[s] converts s degrees
Celsius to degrees Fahrenheit. The argument s and
the result are integers.";

FtoC::usage = " FtoC[t] converts t degrees
Fahrenheit to degrees Celsius. The argument t and
the result are integers.";

Begin["`Private`"]

(* private functions for internal use *)

pCtoFTest[s_] := (s ≥ -273.15)

pFtoCTest[t_] := (t ≥ -459.67)

(* public functions to be exported *)

CtoF[s_?IntegerQ] := Round[N[9s/5 + 32]]/;pCtoFTest[s]

FtoC[t_?IntegerQ] := Round[N[(t - 32)5/9]]/;pFtoCTest[t]

End[ ]

EndPackage[ ]

(* :Examples:
    Names["Thermometer`*"]

```

```
?CtoF
CtoF[-15]
FtoC[67]
*)
```

Note that the previous cell must be evaluated before we can proceed further. When `Thermometer` is implemented as a package in a separate notebook, this cell will be made an initialization cell. We have done that in the package notebook `Temperature.m` in the `PackagesSupport` folder.

We first invoke the function `Names[ ]` to get a list of the public functions in the package.

```
Names["Thermometer`*"]
```

Then we use the `? help` facility to see the meaning and the interfaces of the public functions defined in the package.

```
?CtoF
```

```
?FtoC
```

With the `??` help facility we can even inspect the definition of a function inside a package, not just the usage message that we get with the `? facility`. Observe that all symbols are shown with their complete names describing the entire context of the symbols.

```
??CtoF
```

Here are several invocations of functions from the `Thermometer` package.

```
CtoF[0]
```

```
CtoF[100]
```

```
FtoC[77]
```

```
FtoC[-459]
```

Here are two temperature conversion tables.

```
TableForm[Table[{i,CtoF[i]},{i,-40,40,10}],
TableHeadings → {{},{ "C", "F"}}]
```

```
TableForm[Table[{i,FtoC[i]},{i,0,100,10}],
TableHeadings → {{},{ "F", "C\n"}}]
```

Next we give some unsuccessful invocations of the functions in the package since the arguments either have an incorrect type or are below the temperature of absolute zero.

```
CtoF[23.5]
```

```
CtoF[-300]
```

```
FtoC[-500]
```

**Example 7:** QuickThermometer, a preliminary version of the package Thermometer.

It is easy to make context mistakes when designing packages. The most frequent error is that a context, for example a `Private` context or the context of the package, is not closed. If that happens, symbols that you expect to be defined, for example in the `Global` context, may be shadowed. In such a case it is best to save all work from the current session, to quit Mathematica, and then to start a brand new session in which you can start with a context list that has not been corrupted by programming errors.

When we look at the listing of the package Thermometer in Example 6 we observe that more than half of the printed space is taken up by documentation for the design of the package and for the user of the package. Including documentation for function definitions and packages certainly is good methodology and we recommend that you write descriptive documentation for every package.

However, if we are in the initial stages of developing a function and are experimenting with the design of a prototype we may only want a quick implementation that we either discard or incorporate into some other, formally documented package later. In such a situation the formal layout for a package as described in Examples 5 and 6 is not really appropriate. In particular, usage messages are likely to be premature since we may not have settled on the final interface of a function.

In this example we show what might be a preliminary implementation of the package Thermometer. The package always contains the public symbols section as given by the pair of brackets `BeginPackage[ ]` and `EndPackage[ ]`. It also may or may not contain the nested private symbols section.

To avoid confusion with names of symbols in the package Thermometer we use the prefix Quick for the names in this package. The definitions of the two conversion functions in QuickThermometer are otherwise identical to those in the package Thermometer. All symbols are public in this example.

```
BeginPackage["QuickThermometer`"]

QuickCtoFTest[t_] := (t ≥ -273.15)
QuickFtoCTest[t_] := (t ≥ -459.67)

QuickCtoF[t_?IntegerQ] := Round[N[9t/5 + 32]]/;QuickCtoFTest[t]
QuickFtoC[t_?IntegerQ] := Round[N[(t - 32)5/9]]/;QuickFtoCTest[t]

EndPackage[ ]
```

Here are some invocations of the functions from the package QuickThermometer. We do not get any usage messages because we did not define any.

```
Names["QuickThermometer`*"]

?QuickCtoF

QuickCtoF[45]

QuickFtoC[32]

QuickCtoFTest[-165]
```

## Package Files

---

In the previous section we defined two package contexts, the `Template`` context and the `Thermometer`` context. We want to create actual packages for them such as the packages that are in the `AddOns` folder of Mathematica 4.1. Furthermore, we want to be able to use our packages in exactly the same way as we can use the `AddOns` packages. In this section we explain how to accomplish both objectives.

**Example 8:** Creating a package from an input cell.

To establish a common directory path for examples in this section, we assume that you created a folder called `MyPackages`. If you work in Windows, the following examples assume that the folder is in the `My Documents` folder of the user `Student` on C-drive. If you work on a Macintosh, we assume the folder is in the `Documents` folder of the hard disk in OS 9 and at the toplevel of the user space of user `Guest` in OS X. You may need to adjust file paths to suit the setup on your computer.

It may be convenient to close your current Mathematica session and start a new session so that only the `System`` and `Global`` contexts are defined. Select the input cell in Example 5 of the previous section that contains the `Template`` context. Copy the cell into a new untitled notebook, select the copied cell, and choose **Cell ▸ Cell Properties ▸ Initialization Cell**. Select **File ▸ Save As ...** in the menu bar and then save the notebook as `Template.nb` into the folder `MyPackages`. Then select in the menu bar **File ▸ Save As Special ... ▸ Package Format** and save the notebook as `Template.m` into the folder `MyPackages`. In the Macintosh environment you have to include the extension `.m`, but on the Windows platform this is not necessary. However, if your Windows computer is set up not to show extensions, then you still can distinguish the `Template` package from the `Template` notebook by the Notepad or Mathematica icon.

Open both notebooks and you see the following difference: the `.nb` notebook has the input cell as you copied it, but the `.m` notebook is an Initialization cell without the diagonal slash mark at the top of the cell bracket.

In the naming of these notebooks, we have followed the standard convention in Mathematica to name a package by its context name and give it the extension `.m`.

In order to access the package `Template.m` we need to learn how to navigate in the file directory system of the computer.

**Example 9:** File directories, current directories, and search paths.

Whenever you open a notebook, Mathematica establishes a home directory. This is the place where Mathematica looks (by default) for initialization and system files as well as for packages that you want to load with `Needs[ ]`.

```
?$HomeDirectory
```

```
$HomeDirectory
```

The last evaluation shows the entire directory path to the Mathematica application on your computer or the remote server from which you run Mathematica. In addition, the current directory

for your session is stored in Mathematica. This usually will be the home directory and contains all Mathematica files. We can check its setting with the following command.

```
?Directory
```

```
Directory[ ]
```

We want to change this directory to the one where we keep the folder MyPackages of packages. That is achieved by the following built-in function.

```
?SetDirectory
```

If we know the sequence of nested folders (subdirectories) to access MyPackages then we enter this sequence as a string. Here are two specific paths, one for a Macintosh computer and the other for a Windows computer.

We assume that the internal disk on the Macintosh computer is called Hard Disk and that the C drive is used on the Windows computer. If necessary, change the content of the appropriate string and evaluate the following cell. Observe that two back-slashes are required for the Windows path because the single back-slash is used as a control character inside a quoted string.

```
Clear[uMac9Path,uMacXPath,uWinPath]
```

```
uMac9Path = "Hard Disk:Documents:MyPackages:"
```

```
uMacXPath = "Desktop/MyPackages:"
```

```
uWinPath = "C:\\Documents and Settings\\Student\\My Documents\\  
MyPackages\\"
```

Now evaluate the appropriate cell.

```
SetDirectory[uMac9Path] (*Macintosh OS 9*)
```

```
SetDirectory[uMacXPath] (*Macintosh OS X*)
```

```
SetDirectory[uWinPath] (*Windows 2000*)
```

We changed the current directory, but Mathematica keeps a record of all directories used earlier as current directories in a stack. The home directory and the search path outside the current directory have not been changed by the switch to a new current directory.

```
Directory[ ]
```

```
DirectoryStack[ ]
```

```
$HomeDirectory
```

Now we are ready to load the Template package in MyPackages.

```
Needs["Template`"]
```

We can get the names of the public functions defined in the package with the function Names[ ]. We specify the context and then use the wildcard symbol \* to obtain all function names.

```
Names["Template`*"]
```



Here is an invocation of the one public function defined in the package `Template.m`.

```
Fpublic[ ]
```

We can also inspect the packages that have been loaded so far during this session. The list of the contexts corresponding to the names of the loaded packages is stored in the system variable `$Packages`.

```
$Packages
```

Even though we have changed the current directory, we still can load any of the standard Mathematica packages. The search path to those packages was never changed. We load the `Colors` package from the `Graphics` folder.

```
Needs["Graphics`Colors`"]
```

```
$Packages
```

```
LampBlack
```

```
MistyRose
```

The current directory has not changed.

```
Directory[ ]
```

However, we can always change back to the previous directory with the `ResetDirectory[ ]` function.

```
ResetDirectory[ ]
```

```
Directory[ ]
```

The home directory has not changed during these manipulations with the working directory.

```
$HomeDirectory
```

**Example 10:** The `Thermometer` package.

Take the input cell from Example 6 that contains the `Thermometer`` context and create the files `Thermometer.nb` and `Thermometer.m` in the folder `MyPackages`. We now follow the same steps as in Example 9. First we establish the correct current directory and then load the functions from the `Thermometer` package.

```
Directory[ ]
```

```
Clear[uMac9Path, uMacXPath, uWinPath]
```

```
uMac9Path = "Hard Disk:Documents:My Packages:"
```

```
uMacXPath = "Desktop/MyPackages:"
```

```
uWinPath = "C:\\Documents and Settings\\Student\\My Documents\\  
MyPackages\\"
```

Now evaluate the appropriate cell.

```
SetDirectory[uMac9Path] (*Macintosh OS 9*)

SetDirectory[uMacXPath] (*Macintosh OS X*)

SetDirectory[uWinPath] (*Windows 2000*)
```

Now we are ready to load the Thermometer package in MyPackages using the `<<` operator rather than the `Needs[ ]` function.

```
<< Thermometer.m
```

We can get the names of the public functions defined in the package with the function `Names[ ]`. We specify the context and then use the wildcard symbol `*` to obtain all function names.

```
Names["Thermometer`*"]
```

Here is an invocation of each of the two functions defined in the package `Thermometer.m`.

```
CtoF[100]

FtoC[32]
```

## *A Package for an Iteration Function*

---

We have used the built-in functions `Nest[ ]` and `NestList[ ]` that iterate a function a specified number of times. There are other built-in functions that iterate over a given list, such as `FoldList[ ]`, `Fold[ ]`, and `Map[ ]`. It would be convenient also to have a function where the iteration is controlled by a property on the objects that are computed during the iteration. We will define a pair of such functions, `IterateList[ ]` and `Iterate[ ]`, where the first function returns the list of all values computed and the second function returns the last value that was computed. These two functions will be the public functions in the package `Iteration`.

**Example 11:** A package for a function iteration controlled by a property.

In an iteration that is controlled by a property we must compute until the first value is encountered that violates the given property. In the following implementation we include this value in the list for `IterateList[ ]` and we return that value as the value for `Iterate[ ]`. We suggest alternate implementations in the exercises.

We design an interface for `IterateList[ ]` that is similar to that of `NestList[ ]`. The starting value for the iteration is some expression and the return value is a list. In our sample evaluations the expression will be numeric. However, no such restriction is imposed in the definition of `IterateList[ ]`.

We will implement the action of `IterateList[ ]` recursively. For this purpose we need an auxiliary function that takes a list, applies the function to the last element of the list, and appends that value to the list. This function we call `pNext[ ]`, and we hide it as a private function in the package `Iteration`.

Since there is no usage for the private function `pNext[ ]` we give the documentation of its implementation in the form that we have used throughout the textbook. An iteration can be done with loops or through recursion. We elected to use recursion since it allows a short function definition in a single `If[ ]` statement.

The third parameter `y` in the interface of the function `pNext[ ]` is the list of values computed so far. In order to perform the recursive invocation we need to refer only to the last value in that list. Therefore, we name that value `z`. Since we do not care what or how many elements come before `z` we use the anonymous pattern of three underscore symbols for zero or more objects to denote everything else in the list `y`.

DEFINITION :: `pNext[ expr_, x_, y: {___, z_}, p_]`

INPUT ::

`expr`    is a function expression in the variable `x`.  
`x`        is the variable.  
`y`        is the nonempty list of iterates computed so far.  
`z`        is the last element in the list `y`.  
`p`        is a property in the variable `x` that controls the recursion.

1. Test whether the property `p` holds for the last element `z` in list `y`.
  - (a) If so, compute `expr /. (x → z)`, append it to `y`, and recurse with the appended list.
  - (b) If not, terminate the recursion and return the list of iterates.

OUTPUT :: The list of iterates where all elements in the list, except the last, satisfy property `p`.

IMPLEMENTATION `pNext[ ]`:

```
(* :Title: Iteration of Functions *)
(* :Author: George Boole *)
(* :Date: April 1,1847 *)
(* :Copyright: 1847, George Boole *)
(* :Package Version: 1.2 *)
(* :Mathematica Version: 4.1 *)
(* :Keywords: Iteration, Iterated Functions *)

(* :Summary: Compute the list of iterates x, f(x), f(f(x)),
    f(f(f(x))),... of a function of one argument while a Boolean
    expression holds for the iterates.
*)
```

```

BeginPackage["Iteration`"]

(* :Usage Messages: *)

IterateList::usage = " IterateList[expr,{x,x0},p] iterates the
expression expr of the variable x with a starting value x0 as
long as the property p holds for the current iterate
of expr. The list of all iterates is returned.";

Iterate::usage = " Iterate[expr,{x,x0},p] returns the last value
computed by the function IterateList[expr,{x,x0},p].";

Begin["`Private`"]

(* Here is the auxiliary function *)

pNext[expr_,x_,y:{___,z_},p_] :=
  If[p/.x → z,pNext[expr,x,Append[y,expr/.x → z],p],y]

(* Here are the two public functions *)

IterateList[expr_,{x_,x0_},p_] := pNext[expr,x,{N[x0]},p]

Iterate[expr_,{x_,x0_},p_] := Last[IterateList[expr,{x,x0},p]]

End[]

EndPackage[]

(*
IterateList[Sqrt[x],{x,100},x > 1.1]
*)

```

We list the names of the public functions in the Iteration package.

```
Names["Iteration`*"]
```

**Example 12:** Computing with the Iteration package.

Remember that the input cell defining the Iteration package in Example 11 must have been evaluated before we can proceed with this example.

We calculate successive square roots and logarithms.

```

IterateList[Sqrt[x], {x,100}, x > 1.1]

IterateList[Log[2.,x], {x,1000}, x > 10]

IterateList[Log[2.,x], {x,1000000}, x > 2]

```

We show how to implement Newton's method with the `Iterate[ ]` function. Recall that given an approximation  $x$ , the next value in Newton's method is  $x - \frac{f(x)}{f'(x)}$ . That expression is the first argument for the iteration function.

First we approximate the root of a polynomial.

```
Clear[f]
f[x_] := 3x^5 - 17x^3 + 2x - 11

Plot[f[x], {x, -3, 3}, PlotRange → {-50, 50}];

Iterate[x - f[x]/f'[x], {x, 5}, Abs[f[x]] > 0.001]
f[%]
```

Note which root of  $f$  we approximate in the next computation.

```
IterateList[x - f[x]/f'[x], {x, 1.0}, Abs[f[x]] > 0.001]
f[Last[%]]
```

Here is an approximation of the third root of  $f$ .

```
IterateList[x - f[x]/f'[x], {x, 0.5}, Abs[f[x]] > 0.001]
f[Last[%]]
```

As a final iteration example we approximate a root of  $\sin x$ .

```
IterateList[x - Sin[x]/Sin'[x], {x, 2}, Abs[Sin[x]] > 0.001]
Sin[Last[%]]
```

Evaluate the previous iteration several times with different termination conditions, for example,  $|\sin x| > 0.01$  or  $0.0001$ .

## *A Package for Gram-Schmidt Orthogonalization*

---

In this section we present a package containing the function `GramSchmidt[ ]` that computes a set of orthogonal or orthonormal vectors  $\{v_1, \dots, v_n\}$  from a given set  $\{u_1, \dots, u_n\}$  of linearly independent vectors. The computations for the vectors  $v_k$ ,  $1 \leq k \leq n$ , are done iteratively as follows:

$$v_1 = u_1$$

$$v_k = u_k - \frac{[u_k | v_1]}{[v_1 | v_1]} v_1 - \dots - \frac{[u_k | v_{k-1}]}{[v_{k-1} | v_{k-1}]} v_{k-1}$$

for all  $1 < k \leq n$ , where  $[x | y]$  stands for the inner product of the two vectors  $x$  and  $y$ .

We divide the computation of the  $k^{\text{th}}$  orthogonal vector into several levels:

1. The first level is to compute the quotient of the inner products  $\frac{[u_i | v_i]}{[v_i | v_i]}$ , for all  $i$  such that  $1 \leq i < k$ . These determine the lengths of the orthogonal projections. The computation is done in the private function `pProject[ ]`.

2. At the next level we need to compute this projection for all vectors  $v_1, \dots, v_{k-1}$ . We map `pProject[ ]` across the list of vectors  $\{v_1, \dots, v_{k-1}\}$ . Then we add all vectors in the resulting list and subtract the sum from  $u_k$ .
3. Once the vector  $v_k$  has been computed we append it to the list of the  $k - 1$  orthogonal vectors that have already been computed. Steps (2) and (3) are combined in the private function `pNextVector[ ]`.
4. Finally, we return the entire list of vectors  $\{v_1, \dots, v_n\}$ , either normalized or not, depending on the option chosen. We do this by folding the `pNextVector[ ]` function across the given list of linearly independent vectors.

**Example 13:** Implementation of the `GramSchmidt` function and a 3D-plotting function.

The definition of the `GramSchmidt[ ]` function follows the computations described in the introduction. Its option `Normalized` has the default value `False`.

We include in the package the public function `EuclideanNorm[ ]` that computes the (Euclidean) length of a single vector or the list of lengths of all vectors in a list. It is a useful tool when we want to check the lengths of the vectors returned by the function `GramSchmidt[ ]`.

For the special case of three dimensions, the function `GramSchmidtPlot3D[ ]` draws the three vectors  $\{u_1, u_2, u_3\}$  and their orthogonalization  $\{v_1, v_2, v_3\}$ , where  $u_1 = v_1$ . We believe that it is more effective to draw the non-normalized orthogonal projections computed in the Gram-Schmidt method than to draw the normalized projections. Therefore, we do not make the option `Normalized` available in `GramSchmidtPlot3D[ ]`.

However, we want to guard against the case where the `Normalized` option is entered by mistake in `GramSchmidtPlot3D[ ]`. For that reason we will use the `FilterOptions` package. We load that package automatically when the `Normalization` package is evaluated. This is achieved by mentioning the context `Utilities`FilterOptions`` in the `BeginPackage[ ]` function of the `Normalization` package.

```
(* :Title: Gram-Schmidt Orthonormalization *)
(* :Author: Jorgen P.Gram & Erhardt Schmidt *)
(* :Date: March 15, 1907 *)
(* :Copyright: 1907, J.P.Gram & E.Schmidt *)
(* :Package Version: 1.3 *)
(* :Mathematica Version: 4.1 *)

(* :Keywords: Orthogonalization, Orthonormalization, Gram-Schmidt
*)

(* :Summary: Given a set of linearly independent vectors the Gram-
Schmidt method is used to compute an orthogonal or an
orthonormal set of vectors that have the same span as the given
set of vectors.*)
```

```

BeginPackage["GramSchmidt`",{ "Utilities`FilterOptions`"}]

(* :Usage Messages: *)

GramSchmidt::usage = " GramSchmidt[{u1,...,un},opt] calculates
an orthogonal basis {v1,...,vn} for the span of {u1,...,un}. The
vectors {v1,...,vn} are normalized when the option Normalized is
set to True.";

Normalized::usage = " Normalized is an option for GramSchmidt[ ]
which determines whether the orthogonal vectors are normalized.
The default value for Normalized is False.";

EuclideanNorm::usage = "EuclideanNorm[v] computes the standard
Euclidean norm for a vector or a list of vectors v.";

GramSchmidtPlot3D::usage = " GramSchmidtPlot3D[{u1,u2,u3},opts]
plots the three 3-dimensional vectors and their orthogonalization
computed by the Gram-Schmidt method. Any option of Graphics can be
used to control the plot.";

Begin["`Private`"]

(* Here are the private auxiliary functions *)

pProject[u_,v_] := (u.v)/(v.v) v

pNextVector[vlist_,u_] := Append[vlist,u-Apply[Plus,
  Map[pProject[u,#]&,vlist]]]

(* Here are the public functions *)

Options[GramSchmidt] = {Normalized→False};

GramSchmidt[ulist_,opts___]:=
  If[Normalized/.{opts}/.Options[GramSchmidt],Map[#/Sqrt[#. #]&,
    Fold[pNextVector,{},ulist]], Fold[pNextVector,{},ulist]]/;
  MatrixQ[ulist]

EuclideanNorm[v_] := Sqrt[Apply[Plus,Map[(# #)&,v]]]/;VectorQ[v]
EuclideanNorm[m_] := Map[EuclideanNorm,m]/;MatrixQ[m]

GramSchmidtPlot3D[ulist_,opts___]:=
  Show[Graphics3D[Join[{RGBColor[0,0,1],Thickness[0.008]},
    Map[Line[{{0,0,0},#}]&,GramSchmidt[ulist]],
    {RGBColor[0,1,0],Thickness[0.003]},
    Map[Line[#]&,Transpose[{ulist,GramSchmidt[ulist]}]],
    {Thickness[0.003]},
    Transpose[{{RGBColor[0,0,1],RGBColor[1,0,0],RGBColor[1,1,0]}},
    Map[Line[{{0,0,0},#}]&,ulist}]]],
    FilterOptions[Graphics3D,opts],
    Axes→True, PlotLabel→"GramSchmidt: Orthogonal Basis",

```

```

AxesLabel → {"x-axis", "y-axis", "z-axis"}]; Dimensions[ulist]
== {3,3}

End[ ]

EndPackage[ ]

```

**Example 14:** Usage of the GramSchmidt package in two and three dimensions.

Remember that the input cell defining the GramSchmidt package in Example 13 must have been evaluated before we can proceed with this example.

```
Names["GramSchmidt`*"]
```

We orthogonalize two vectors in the plane.

```

GramSchmidt[{{1,7},{-3,3}}]
N[%]
EuclideanNorm[%]

GramSchmidt[{{1.,7.},{-3.,3.}},Normalized→True]
EuclideanNorm[%]

```

We orthogonalize three vectors in space.

```

GramSchmidt[{{1,1,1},{-3,2,1},{0,9,-4}}]
N[%]
EuclideanNorm[%]

GramSchmidt[{{1,0,0},{2,2,0},{1,1/2,1}}]
EuclideanNorm[%]

GramSchmidt[{{1,0,0},{2,2,0},{1,1/2,1}},Normalized→True]
EuclideanNorm[%]

```

We graph the orthogonalization process in three dimensions.

```

GramSchmidtPlot3D[{{1,0,0},{1,1,0},{1,0.5,1}}];

GramSchmidtPlot3D[{{1,0,0},{1,1,0.5},{1,0.25,1}},
ViewPoint→{1.300,-2.400,2.000}];

```

Here is an animation of the orthogonalization process in three dimensions. We vary the z-coordinate of one vector between  $-1$  and  $+1$ . This animation draws 21 pictures in 3D so that the front end of Mathematica needs several MB of RAM for rendering the pictures. If you find it difficult to follow the various angles in the animation, you might want to choose a different viewpoint.

```

Table[GramSchmidtPlot3D[
  {{1,0,0},{1,1,z},{1,0.25,0.5}},
  ViewPoint→{3.510,1.530,-0.130},
  PlotRange→{{-0.5,1},{-0.5,1},{-1,1}},{z,-1,1,0.1}];

```



## Loading Packages

---

In this section we use some of the packages that we designed earlier in this notebook. All packages have been collected as separate notebooks in the folder (subdirectory) `PackagesSupport`. The standard convention in Mathematica is to name a package by its context name and give it the extension `.m`.

To access packages we need to learn how to navigate in the file directory system of the computer. We suggest that you quit your current Mathematica session if you have evaluated any of the packages in this notebook and restart Mathematica. This ensures that only the two contexts `Global`` and `System`` are defined.

Sometimes we know the name of a package that we want to load, but we may have forgotten the precise access path. We will write a function `uFindPackage[ ]`, which computes the search path for a package when we know the name of the package and the name of the disk on which the package is stored. The built-in function `FileNames[ ]` will perform the disk search for us.

**Example 15:** A function to compute the directory and file path for a package.

The function `FileNames[ ]` has several distinct uses.

### ?FileNames

We search for the package `Thermometer.m` on the search disk `Hard Disk` that we used in Example 6. This computation may take some time if the search disk is large. You should not search for the file if it resides on a server disk. The value `Infinity` for the third parameter indicates that the search will be carried to arbitrary nesting levels of the folders. If you know how deeply the `Thermometer.m` package is nested then you can enter that number as the third argument and speed up the execution of the search.

```
FileNames["Thermometer.m", "Hard Disk:", Infinity]
(*Macintosh OS 9*)
FileNames["Thermometer.m", "/users/Guest/Desktop/", Infinity]
(*Macintosh OS X*)
FileNames["Thermometer.m", "C:\\", Infinity] (*Windows*)
```

We demonstrate how error messages can be created for a function. The message text is defined like a usage message in a package interface. It is printed with the `Message[ ]` function.

### ?Message

Instead of the predefined usage tag name for public symbols in packages we specify the tag name for the message. Here we choose the tag `BadExtension` when the extension of a filename is not the expected `.m` for packages.

```
Clear[uFindPackage, BadExtension]

uFindPackage::BadExtension = " Expected file extension .m";

Message[uFindPackage::BadExtension]
```

We designed the interface of `uFindPackage[ ]` similar to that of the built-in function `FileNames[ ]`. The output of `uFindPackage[ ]` will be a list of file path names.

DEFINITION :: `uFindPackage[package, disk, depth]`

INPUT ::

`package` is the quoted string of the name of the file containing the package; the filename is expected to have the extension `.m`.  
`disk` is the quoted string of the name of the disk to be searched.  
`depth` is a default parameter for the search depth of nested folders; the default value is `Infinity`.

1. If the extension `.m` is not present,
  - (a) Issue an error message.
  - (b) Return the empty list.
2. Otherwise, search for all paths to a file with the name specified in the parameter `package`, and
  - (a) Strip the name of the package from each file path.
  - (b) Return the resulting list of directories.

OUTPUT :: The list of directories containing the package.

IMPLEMENTATION :: `uFindPackage[ ]`:

```
uFindPackage[package_String,disk_String,deep_:Infinity]:=
If[StringTake[package,-2] ≠ ".m", Message[uFindPackage::
BadExtension]; {}, Map[StringReplace[#,package → ""]&,
FileNames[package,disk,deep]]]
```

Here are some invocations of this function. The first evaluation in each of the following groups should succeed. The second will fail because we provide an incorrect extension for a package. Evaluate the group appropriate for your computer.

For the Macintosh OS 9 platform: one successful search and one search that triggers the error message.

```
uFindPackage["GramSchmidt.m","Hard Disk:",4]
```

```
uFindPackage["Normalization.nb","Hard Disk:",1]
```

For the Macintosh OS X platform: one successful search and one search that triggers the error message.

```
uFindPackage["GramSchmidt.m","/Users/Guest/",6]
```

```
uFindPackage["Normalization.nb","/Users/Guest/",1]
```

For the Windows 2000 platform: one successful search and one search that triggers the error message.

```
uFindPackage["GramSchmidt.m", "C:\\", 6]
```

```
uFindPackage["Normalization.nb", "C:\\"]
```

Finally, we extract the appropriate directory from the output cell and assign it to the variable `curdir`. We change the current directory to the one just computed and load the package context `GramSchmidt`` into the current session of Mathematica.

```
Clear[curdir]
curdir = (* paste the directory here *)

SetDirectory[curdir]

Needs["GramSchmidt`"]

GramSchmidt[{{2, 1}, {1, 2}}]
```

## Exercises

---

**EXERCISE 1:** Expansion of the Thermometer package.

Add functions to the Thermometer package that allow conversion to and from the Kelvin temperature scale.

**EXERCISE 2:** Expansion of the Iteration package.

(a) Add an option for the function `Iterate[ ]` that specifies whether the first value that does not satisfy the property is returned or whether the last value that does satisfy the property is returned. The package in the notebook always includes the first value that does not satisfy the property.

(b) Add an option for the function `IterateList[ ]` that specifies whether the first value that does not satisfy the property is included in the list of all computed values or not. The package in the notebook always includes that value.

**EXERCISE 3:** A package to test properties of binary relations.

Binary relations occur in many different applications of mathematics and of computer science. They are represented by adjacency matrices, that is, square matrices with entries of zeros and ones. You can perform Boolean arithmetic with the numbers 0 and 1 as follows. The Boolean sum (logical Or) of  $a$  and  $b$  is  $a \oplus b = a + b - a \times b$ , and the Boolean product (logical And) of  $a$  and  $b$  is  $a \otimes b = a \times b$ .

Since functions that test a property return `True` or `False`, all test functions that you implement in this package must return a Boolean value.

(a) Implement the functions `BooleanPlus[ ]` and `BooleanTimes[ ]` for two 0-1 valued arguments. Use the `Fold[ ]` function and the 3-underscore pattern `x___` to allow an arbitrary number of arguments in these two functions.

- (b) Use the two functions from part (a) to write a function `BooleanDot[ ]` that computes the matrix product of two adjacency matrices.
- (c) Write functions that test a binary relation that is represented as an adjacency matrix for reflexivity, symmetry, anti-symmetry, and transitivity.
- (d) Use the functions from part (c) to write a function `EquivalenceQ[ ]` that tests whether a binary relation that is represented as an adjacency matrix is an equivalence relation, that is, the relation is reflexive, symmetric, and transitive.
- (e) Use the functions from part (c) to write a function `PartialOrderQ[ ]` that tests whether a binary relation that is represented as an adjacency matrix is a partial order, that is, the relation is reflexive, antisymmetric, and transitive.
- (f) Collect all functions that you defined in parts (a) through (e) in a package named `Relations`. Make the functions from parts (a) and (b) private functions.
- (g) Include a function `ToAdjacency[ ]` in the package that constructs an adjacency matrix from the list of pairs of elements of a relation. Use the list of pairs as the single parameter of the function.
- (h) Include a function `ToRelation[ ]` in the package that constructs a relation as a list of pairs of numbers 1 through  $n$  from an  $n \times n$  adjacency matrix.
- (i) Include a function `AllProperties[ ]` in the package that will test an adjacency matrix for all properties listed in parts (c), (d), and (e). The output of this function should be in tabular form, give the name of each property, and state whether the property holds for the adjacency matrix.

#### EXERCISE 4: A package for tridiagonal matrices.

The Mathematica packages folder `LinearAlgebra` contains the package `Tridiagonal.m`. You can access it by selecting **Help** ► **Add-ons ...** in the menu bar and then selecting **Standard-Packages** ► **Linear Algebra** ► **Tridiagonal** in the dialog window. This package implements one function `TridiagonalSolve[ ]`, which solves a linear system  $Ax = r$  where  $A$  is an  $n \times n$  tridiagonal matrix and  $x$  and  $r$  are  $n$ -dimensional vectors.

- (a) Take a textbook on linear algebra or on numerical analysis and learn about the special algorithm to solve a linear tridiagonal system.
- (b) Study the implementation of the solution algorithm implemented in the package `Tridiagonal.m`. Follow the steps at the beginning of the section “The Basic Scheme of a Package” in this notebook to copy and open the package.
- (c) The parameters for `TridiagonalSolve[ ]` are the subdiagonal, the diagonal, and the superdiagonal vectors of the matrix  $A$  and the right-handside vector  $r$ . Write a function that converts a tridiagonal matrix to the three vectors that represent the subdiagonal, the diagonal, and the superdiagonal. It may be useful to return the three vectors as a `Sequence[ ]` object rather than as a `List[ ]` object.
- (d) Write a function that converts three vectors that represent a subdiagonal, a diagonal, and a superdiagonal into a tridiagonal matrix.

- (e) Make a copy of the Mathematica package `Tridiagonal.m` and give the package a new name. Add the two functions from parts (c) and (d) to it. Modify the appropriate documentation and usage messages in the package.
- (f) Load the package you implemented in part (e) and demonstrate the use of its functions with several tridiagonal linear systems.

**EXERCISE 5:** A package “Riemann” of approximations for the Riemann integral.

In this exercise we will use the function `uGetCoords[ ]`, which gives us points on a graph that we then use to approximate the definite integral over the specified range. It was introduced in Example 18 of the notebook `ListFcts.nb`, “List Processing Functions,” with a slightly different implementation. Here we will use it as a private function in the package `Riemann`.

```
pGetCoords[expr_, r:{_,_,_}]:=
  Nest[Part[#,1]&,InputForm[Plot[expr,r,DisplayFunction
    → Identity]]],5]
```

This exercise is also Exercise 6 in the notebook `ListFcts.nb`, “List Processing Functions.” If you completed that exercise then you can reuse the functions you developed in that exercise and you need to complete only parts (i), (j), and (k).

(a) Write a function `pMapList[ ]` that takes two lists of equal length, a list  $\{f_1, f_2, \dots, f_n\}$  of functions and a list  $\{v_1, v_2, \dots, v_n\}$  of values, and that returns the list  $\{f_1(v_1), f_2(v_2), \dots, f_n(v_n)\}$ . For example, the invocation `pMapList[{Sqrt, Log[10, #]&, Point}, {49, 1000, {1, 2}}]` should produce the list  $\{7, 3, \text{Point}[\{1, 2\}]\}$ .

(b) Given is a list of numbers  $\{x_1, x_2, \dots, x_n\}$  that you may think of as the points of a subdivision of the interval  $x_1 = a \leq b = x_n$ . Write a function `pWidth[ ]` that returns the list of the lengths of the subdivision, that is, the  $(n - 1)$ -element list  $\{x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}\}$ .

(c) Given is a list of pairs of numbers such as the list of coordinates that the function `pGetCoords[ ]` produces:  $\{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\}\}$ . Write a function `pLeftHeight[ ]` that takes such a list as its input and returns the list  $\{\{x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}\}, \{y_1, y_2, \dots, y_{n-1}\}\}$  of widths of the subintervals defined by the x-coordinates of the points and the heights at the left endpoints of the subintervals. The two functions `pMapList[ ]` and `pWidth[ ]` may be useful in the implementation of `pLeftHeight[ ]`.

(d) If you take the pair of lists that you computed in part (c) and apply the dot product to the two lists you will produce an approximation of the definite integral with the left Riemann sums. Write a function `LeftRiemann[ ]` that implements these computations and that has two parameters; the first is some real-valued function expression and the second specifies an interval. Therefore, its interface is as in `Plot[ ]`. Choose several functions and intervals for them and approximate the definite integrals with `LeftRiemann[ ]` as well as with `NIntegrate[ ]`. Compare and explain your results.

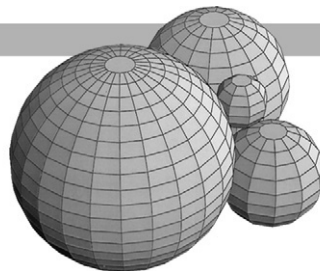
(e) Write a function `pRightHeight[ ]` that takes the list  $\{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\}\}$  as its input and returns the list  $\{\{x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}\}, \{y_2, y_3, \dots, y_n\}\}$ .

(f) Write the function `RightRiemann[ ]` in the manner of part (d). Use your function of part (e).

- (g) Write a function `pTrapezoidHeight[ ]` that takes the list  $\{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\}\}$  as input and that has the list  $\{x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}\}, \{\frac{y_1+y_2}{2}, \frac{y_2+y_3}{2}, \dots, \frac{y_{n-1}+y_n}{2}\}$  as its return value.
- (h) Write the function `Trapezoid[ ]` in the manner of part (d). Use your function of part (g).
- (i) Write a single function `ApproxTable[ ]` that returns the approximations of the four functions `LeftRiemann[ ]`, `RightRiemann[ ]`, `Trapezoid[ ]`, and `NIntegrate[ ]`. Print the name of the method together with the numeric value computed.
- (j) Incorporate the functions of parts (a) through (i) in a package. Design appropriate usage messages for the four public functions in the package.
- (k) Demonstrate the package of part (j) with several functions and intervals of integration.

This Page Intentionally Left Blank

# *Advanced Mathematica: Files, Data Exchange, and Conversions*



## *Introduction*

So far, we have used the computing environment as presented by Mathematica when we open one of the notebooks or the application itself. Whenever we define a function in a notebook we extend the computing power of Mathematica and change the environment by introducing new names. On occasion we added functions from packages with the `Needs[ ]` function, and we also demonstrated how to write new packages and thereby expand the computing environment.

Any data that we needed for computations in earlier notebooks we either entered interactively from the keyboard or generated internally in Mathematica with functions such as `Table[ ]` and `Random[ ]`. In this notebook we present methods for importing data from files and exporting data to files.

To make this notebook independent of other notebooks in Part IV we will repeat some topics that have also been covered elsewhere. One such topic is the various directories that Mathematica uses for its operations.

To access data files from inside a Mathematica notebook we need to know some details about folders on a Macintosh and directories in Windows on a PC. When necessary, the examples in this notebook include separate input cells for the Macintosh OS 9 and OS X platforms and for the Windows 2000 platform. In some cases we must make an assignment to a variable where the value depends on the computing platform. We leave the assignment open and indicate that fact with a comment in the input cell. Only the file paths are platform dependent.

Associated with this notebook is the folder `DataTests`. It contains all the files needed for input and also the files generated as output by the examples in this notebook. We suggest that you create a folder `DataTests` at the top level of the internal disk as indicated and copy the Excel spreadsheet file `Survey.xls` into it. Then there is a short filepath for the examples in this notebook.

In order to establish a common directory path for examples in this section, we assume that you created a folder called `MyPackages`. If you work in Windows, the following examples assume that the folder is in the `My Documents` folder of the user `Student` on C-drive. If you work on a Macintosh, we assume the folder is at the toplevel of the hard disk in OS 9 and at the toplevel of the user space of user `Guest` in OS X. You may need to adjust file paths to suit the setup on your computer.



Evaluate the following Input cell before you work on any of the examples in this notebook.

```
Clear[uMac9Path, uMacXPath, uWinPath]

uMac9Path = "Macintosh HD:Documents:DataTests:"

uMacXPath = "Desktop/DataTests:"

uWinPath = "C:\\Documents and settings\\Student\\My Documents\\
DataTests\\"
```

## *Directories and File Paths*

---

Data required by functions normally are entered from the keyboard into the current notebook. When there is a large amount of data from an experiment or a survey that is stored, for example, in a spreadsheet, then we need an automated mechanism to read the data.

To access data files from inside a Mathematica notebook we need to know some details about folders on a Macintosh and directories in Windows on a PC. Example 1 will deal with questions concerning directories; see also Example 9 in PackDef.nb, “Advanced Mathematica: Packages.”

**Example 1:** File directories and the current directory.

When we initially open Mathematica, it establishes all file paths to directories that it searches for external files. This list of directories required by Mathematica is stored in the system variable `$Path`.

```
?$Path
```

```
$Path
```

Two additional directories are established at the start of a Mathematica session: the home directory and the current working directory. The home directory is the place where Mathematica looks (by default) for the executable and initialization files.

```
?$HomeDirectory
```

```
$HomeDirectory
```

The last evaluation shows the entire directory path to the Mathematica application on your computer or the remote server from which you run Mathematica. In addition, the current directory for your session is stored in Mathematica. This usually will be the first part of the file path to the home directory that contains all Mathematica files. We can check its setting with the following command.

```
?Directory
```

```
Directory[ ]
```

**Example 2:** Changing the current directory.

We want to change the current directory to the folder DataTests, which contains a set of data files. That is achieved by the built-in function `SetDirectory[ ]`.

**?SetDirectory**

In the Introduction we assigned the platform-dependent file path for the folder DataTests to the variables `uMac9Path`, `uMacXPath`, and `uWinPath`, respectively. We assume that these variables have been computed. You may need to adjust file paths to suit the setup on your computer.

Now change the current directory to the DataTests folder by evaluating the input cell appropriate for your computer.

```
SetDirectory[uMac9Path] (*Macintosh OS 9*)
```

```
SetDirectory[uMacXPath] (*Macintosh OS X*)
```

```
SetDirectory[uWinPath] (*Windows 2000*)
```

Even though we changed the current directory, Mathematica keeps a record of all directories used earlier as current directories in a stack. The home directory and the search path outside the current directory have not been changed by the switch to a new current directory.

```
$HomeDirectory
```

```
DirectoryStack[ ]
```

```
Directory[ ]
```

The built-in function `FileNames[ ]` lists all files that are contained in the current directory.

```
?FileNames
```

```
FileNames[ ]
```

At this point we assume that the Excel spreadsheet `Survey.xls` has been copied into the DataTests folder. If you have not done this, do it now. When you evaluate `FileNames[ ]` again, it shows your changes to the current directory.

```
FileNames[ ]
```

## ***Import and Export of Data***

---

`Import[ ]` and `Export[ ]` are two versatile functions that can be used with any type of object, with numbers, as well as with Mathematica expressions and images. We will demonstrate their usage here with numeric data.

```
?Import
```

```
?Export
```

If you have not established the current directory needed to access the DataTests folder, evaluate the next input cell, otherwise skip to the next example.

```
Clear[uMac9Path,uMacXPath,uWinPath]

uMac9Path = "Macintosh HD:Documents:DataTests:"

uMacXPath = "Desktop/DataTests:"

uWinPath = "C:\\Documents and Settings\\Student\\My Documents\\
  DataTests\\"
```

Now evaluate the input cell that applies to your computing platform.

```
SetDirectory[uMac9Path] (*Macintosh OS 9*)

SetDirectory[uMacXPath] (*Macintosh OS X*)

SetDirectory[uWinPath] (*Windows 2000*)
```

**Example 3:** Computations with data from an input file.

We import the survey data from the file Survey.xls into the DataTests folder in Table format. Then the data is copied as a two-dimensional array into the current Mathematica session.

```
Clear[indata]

indata = Import["Survey.xls","Table"];
```

We display the imported data in table format. The data is from a survey of 149 students, where the third of seven questions (columns) represents the age responses of the students surveyed. Therefore, the table is quite lengthy.

```
Length[indata]

TableForm[indata]
```

We extract the third column, the age responses of the students surveyed, from the table and compute some statistical characteristics for it. We name the variable into which we extract the data rawAges since the file Survey.xls contains the actual student responses on the survey. A value of zero denotes that the student did not provide an answer for the age item on the survey.

```
Clear[rawAges]
rawAges = Transpose[indata][[3]]
Length[rawAges]
```

We use the function Select[ ] to purge the bad data from the list by removing all zeros. The result of the selection is stored in the variable goodAges.

```
?Select
```

```

Clear[goodAges]
goodAges = Select[rawAges, (# ≠ 0)&];
Length[goodAges]

```

Now we compute the minimum, maximum, mean, and median ages for the set of students participating in the survey. In order to compute the median of the data, we need to sort the age data. We do this first and save the sorted data in the variable sortedAges.

```

Clear[sortedAges]
sortedAges = Sort[goodAges];

```

We performed computations in Examples 11 and 15 of the notebook ListFcts.nb, “List Processing Functions,” that are similar to the ones in the following input cell. Alternatively, we could load functions from the package Statistics in the AddOns folder for more sophisticated statistical evaluations.

```

Clear[stats]
stats =
  {Min[goodAges], Max[goodAges],
   N[Apply[Plus, goodAges] / Length[goodAges]],
   sortedAges[[Length[sortedAges] / 2]]}

```

Here is a labeled table for the four values computed previously.

```

TableForm[stats, TableHeadings
  → {{ "Minimum", "Maximum", "Average", "Median" }}]

```

Finally, we visualize the goodAges data in two different ways. First we plot it in the order in which the ages occur in the data file Survey.xls. Then we plot the data in ascending order of age to get a picture of the age distribution of the students. Since we know the range of ages, we specify an appropriate plotting range for the data.

```

ListPlot[goodAges, PlotRange → {{0, 150}, {15, 55}},
  AxesLabel → {"Students", "Age"}];

ListPlot[sortedAges, PlotRange → {{0, 150}, {15, 55}},
  AxesLabel → {"Students", "Age"}];

```

**Example 4:** Writing computed data to an output file.

This example is a continuation of Example 3 and we assume that the variables goodAges and sortedAges have been evaluated. We take the data in the variable goodAges and write it to the file GoodAges.dat. We also take the data in the variable sortedAges and write it to the file SortedAges.dat.

We will save both files in the same directory DataTests that contains the input data file Survey.xls. Therefore, the file path from the previous example still applies.

```

Export["GoodAges.dat", goodAges]

Export["SortedAges.dat", sortedAges]

```

At this point you should look into the DataTests folder and convince yourself that the two files have been created. Depending on the characteristics of your computing platform they may appear with the Mathematica icon. When you open the files from Mathematica you see that each value in the list is on a separate line in the file.

We can save data in spreadsheet, tabular, or list form. The next four commands save the first 100 student responses stored in the variable `indata` using four different formats. The designator CSV produces a comma-separated data format, and the Table designator produces a tabular format. Both can be used for importing data into spreadsheet and statistics applications programs. The designators Lines and List produce one list of seven entries per line in the output file.

```
Export["Std100CSV", Take[indata, 100], "CSV"]
```

```
Export["Std100Lines", Take[indata, 100], "Lines"]
```

```
Export["Std100List", Take[indata, 100], "List"]
```

```
Export["Std100Table", Take[indata, 100], "Table"]
```

Look for the four files in the DataTests folder.

**Example 5:** Using imported data in a spreadsheet program.

This example is a continuation of Examples 3 and 4, and we assume that the variables `indata` and `sortedAges` have been evaluated and that the file `Std100CSV` has been created.

Open the Microsoft Excel application and open the file `Std100CSV` by selecting **File ► Open ...** in the menu. Then the text Import Wizard in Excel will open a window. We follow the steps according to the format in which the data were saved. In the window a preview of the data in the file `Std100CSV` is presented. Click on the Next button for the next step in the data conversion. Click on Comma as the delimiter in the window and unclick the Tab box, since the data in the import file is comma delimited. The Preview window will show the spreadsheet representation of the data being imported into Excel. Click on the Next button for the next step in the data conversion. Leave the data format as General and click the Finish button. Finally save the spreadsheet in the Excel Worksheet format, for example, as the file `Std100.xls`. Inspect the DataTests folder to check that this file was saved with the Microsoft Excel icon.

Now we can double-click on the file `Std100.xls` and Excel will open it, since Excel created the file `Std100.xls`, not Mathematica. Further manipulations can now be conducted in Excel on the data in the file `Std100.xls`.

We will also save the sorted age data in an Excel spreadsheet. For that purpose we open the file `SortedAges.dat` from inside Excel and proceed as before, except that this time we check as the delimiter the TAB box, not the COMMA box. We save the converted data as a spreadsheet in the Excel Worksheet format in the directory DataTests using the filename `SortedAges.xls`. Inspect the DataTests folder to check that this file was saved with the Microsoft Excel icon.

## *Conversions to Other File Formats*

---

Mathematica notebooks can be saved in several other file formats. These converted files are created by choosing **File ► Save as Special ...** in the menu bar and then selecting one of the file

types in the pop-up menu, for example, Package Format,  $\text{T}_{\text{E}}\text{X}$ , HTML, or HTML+MathML. In the section “Package Files” of the notebook PackDef.nb, “Advanced Mathematica: Packages,” we saved notebooks in the Package Format. In this section we will create HTML files from Mathematica notebooks. We will demonstrate the conversion with the three sample notebooks in the folder HTMLDemos.

**Example 6:** Creating an HTML document containing text and formulas only.

We use the notebook SphereCenters.nb in the folder HTMLDemos. It contains three text cells and one input cell. The input cell contains mathematical formulas in standard mathematical notation.

Open the notebook SphereCenters.nb and select **File ▶ Save as Special ...** in the menu bar and then select HTML. Choose a location for the HTML files in the file directory window that pops up. We chose the folder HTMLDemos, and chose the name SphereCenters for the folder that will contain the conversion files. The folder SphereCenters contains an index.html document and the two folders Images and Links.

Now open your preferred browser and load the index.html file into it. The contents of the SphereCenters.nb notebook are now rendered in the browser. You can also create a link from another Web page to this index.html file.

**Example 7:** Creating an HTML document containing a 3D graphics cell.

We use the notebook SphereDrawings.nb in the folder HTMLDemos. It contains several text cells, input cells, output cells, and one graphics output cell.

Follow the same steps as in Example 6. This time we name the folder SphereDrawings.

Now open your preferred browser and load the index.html file into it. The contents of the SphereDrawings.nb notebook are now rendered in the browser.

**Example 8:** Creating an HTML document containing typeset text and 2D graphics.

We use the notebook Views2D-Drawings.nb in the folder HTMLDemos. It contains several text cells, input cells, and 2D graphics output cells.

Follow the same steps as in Example 6. This time we name the folder Views2D-Drawings.

Now open your preferred browser and load the index.html file into it. The contents of the Views2D-Drawings.nb notebook are now rendered in the browser.

## *Protected Functions*

---

The built-in functions in Mathematica are protected to guard against accidental changes. For example, we know that Times[ ] is the name for multiplication. So we might decide to use the name Product in some computation and make the following initial assignment for it.

```
Product = 100
```

Mathematica tells us that the symbol Product is protected, that is, it is used for another purpose.

```
?Product
```

We first must unprotect a built-in symbol before we can change its value or add to its definition. After the definition has been made we protect the symbol again.

```
?Unprotect
```

```
?Protect
```

**Example 9:** Extending the definition of the built-in `Power[ ]` function.

We want to compute  $x^y$  for a fractional exponent  $y$ , in reduced form, just as Mathematica computes it, but we also want to return a real number when it is mathematically possible to do so. In Example 1 of the notebook `Options.nb`, “Advanced Mathematica: Options,” we designed a function `uRoot[ ]` that computes a root in this way. In this example we show how we can change the definition of the built-in function `Power[ ]` to suit our purpose rather than defining a function with a different name such as `uRoot[ ]`. Note that the short name of the `Power[ ]` function is “`^`” and the value of  $x^y$  is the principal complex root of the expression `Exp[ y Log[x] ]`.

```
??Power
```

```
Power[-1,1/3]  
(-1)^(1/3)
```

We will extend the definition of the `Power[ ]` function by a third, optional argument. The value of the third argument is either the symbol `Real`, which indicates that the computation should return a real number whenever that is possible, or the symbol `Complex`, which indicates that the built-in function `Power[ ]` is to be used in the computations.

We copy the test function `uRealRootQ[ ]` from the Example 1 in the notebook `Options.nb` and modify it slightly. It determines for the current arguments  $x$  and  $y$  of `Power[ ]` whether a computation over the real numbers is possible and whether the option was set for real computations. Note that the expression  $x^y$  admits a real value when neither  $x$  nor  $y$  are complex and when  $y$  represents a rational number with odd denominator, as for example, in  $x^y = (-2)^{\frac{4}{3}}$ . We need four distinct tests on the arguments  $x$  and  $y$  and one additional test to determine whether the real number computation was required.

```
Clear[uRealRootQ]  
uRealRootQ[x_,y_,type_] := Head[x] != Complex &&  
Head[y] != Complex && Head[y] === Rational &&  
OddQ[Denominator[y]] && type === Real
```

We now explain how to perform the calculation of a real root. Mathematica carries out real number computations when the base  $x$  of the expression  $x^y$  is a non-negative real number. When the base is negative, an appropriate combination of the `Sign[ ]` and the `Abs[ ]` functions will ensure that the base for the computations of the power is non-negative.

We transform the root expression as follows:  $x^{\frac{p}{q}} = (\text{signum}(x) * |x|^{\frac{1}{q}})^p$ .

Here are two examples:  $(-2)^{\frac{5}{3}}$  is the negative real number  $((-1) * (2^{\frac{1}{3}}))^5$  and  $(-2)^{\frac{8}{5}}$  is the positive real number  $((-1) * 2^{\frac{1}{5}})^8$ .

When we want to perform computations over the complex numbers we invoke the built-in exponentiation function `Power[ ]` or `^`. This leads to the following additional definition for

the function `Power[ ]`. We wrap the `Unprotect[ ]` and `Protect[ ]` functions around the expanded definition.

```
Unprotect [Power]

Power[x_,y_,type_] :=
  If[uRealRootQ[x,y,type],
    (Sign[x] Abs[x]^(1/Denominator[y]))^Numerator[y], x^y];
    NumberQ[x]&& NumberQ[y]
Protect [Power]
```

Once we have evaluated the input cell, this extension to the `Power[ ]` function has been added to Mathematica's internal definition in the current session.

```
??Power
```

Note that we restrict the extended definition of the `Power[ ]` function to numbers only. We calculate complex and real roots for a variety of combinations of positive and negative values for the base  $x$  and the exponent  $y$ . All exponents will be rational numbers with odd denominators. The numeric examples are taken from Example 1 of the notebook `Options.nb`, "Advanced Mathematica: Options."

Positive base and positive rational exponent:

```
Power[2,1/3]
N[%]
```

Negative base and positive rational exponent:

```
Power[-2,3/5]
N[%]

Power[-2,3/5,Real]
N[%]
```

Positive base and negative rational exponent:

```
Power[2,-2/3]
N[%]
```

Negative base and negative rational exponent:

```
Power[-2,-2/3]
N[%]

Power[-2,-2/3,Real]
N[%]
```

Finally, we plot the `Power[ ]` function over a range that is symmetric with respect to the origin. The graphs of the functions using the built-in `Power[ ]` function are colored red, and the graphs of the functions using the extended definition of `Power[ ]` are colored blue.

```
Plot[{2 Power[x,1/3], Power[x,1/3,Real]},
{x,-27,27}, PlotStyle→{RGBColor[1,0,0], RGBColor[0,0,1]}];
```



Note the error messages that are created by the built-in `Power[]` function when Mathematica computes in complex number mode.

```
Plot[{2 Power[x, 2/3], Power[x, 2/3, Real]},  
{x, -27, 27}, PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 0, 1]}];
```

## Exercises

---

**EXERCISE 1:** Statistical Analysis and Visualization of data from an input file.

Use the data file `Survey.xls` in the folder `DataTests` as the input file for this exercise. This file contains 149 lines with 7 numbers in each line, which represent responses of students to a class questionnaire.

(a) Follow the steps of Example 3, importing the data, selecting a subset of the data, performing computations with the selected data, graphing the selected data, and exporting the selected data. Use the fourth column of the data. It represents codes for concentrations of study. A value of zero means that the student has not declared any concentration. Find out if there are any concentrations that have been chosen by at least 5% of the students.

(b) Follow the steps of Example 3, importing the data, selecting a subset of the data, performing computations with the selected data, graphing the selected data, and exporting the selected data. Use the fifth and sixth columns of the data. Each column denotes a test, and the values in each column represent the scores that the student received on that test. First compute the maximum score on each of the two tests. Then compute common statistical characteristics such as the lowest score, the highest score, the average score, the median score, and the variance. Also plot the data, for example in ascending order of test scores.

**EXERCISE 2:** Extensions to the definitions of some built-in, protected functions.

(a) Write a function `Log[b, x, n]` that extends the standard built-in `Log[]` function. The argument `b` is the base of the logarithm, `x` is the argument of which to take the logarithm, and `n` is the number of times that the logarithm is applied. For example:

```
Log[2, 65, 3] means Log[2, Log[2, Log[2, 65]]]  
Log[E, 10^6, 4] means Log[Log[Log[Log[10^6]]]]
```

(b) Extend the `Map[]` function to a function `Map[fList, aList]` whose first parameter `fList` is a list of functions of one argument and whose second parameter `aList` is a list of arguments. The extended `Map[]` function should apply the functions in `fList` to each member of `aList` in turn and return the resulting matrix. For example, `Map[{f1, f2}, {a, b, c}]` returns the 3-by-2 matrix `{{f1[a], f2[a]}, {f1[b], f2[b]}, {f1[c], f2[c]}}`.

**EXERCISE 3:** Locating a file and reading data from it.

Mathematica provides an interactive tool to find the entire file path for any file and to insert it where we need it for further manipulation. We do not have to type in the lengthy string. Select **Input ▸ Get File Path ...** in the menu bar and then navigate through the directories as you

would in the search for a file in any application on your computer. Insert the cursor after the = sign in the next input cell and find the file Survey.xls.

```
Clear[infilepath]  
infilepath = (* <- put file path here *)
```

The entire file path will be inserted as a quoted string.

- (a) Mathematica provides another data import and export mechanism. Use the Help Browser to learn about the `ReadList[ ]` function and its option `RecordLists`.
- (b) Use the function `ReadList[ ]` to import the data from the file Survey.xls.
- (c) Select the seventh column from the data. It represents another test score. Compute a frequency count for the scores and write the scores and their frequencies to a file called Frequency7.

#### EXERCISE 4: Transferring graphics from Mathematica to Microsoft Word.

Mathematica provides tools for copying a graphics output cell into Microsoft Word as well as other software. The sequence of steps for copying graphics into Microsoft Word is as follows: Select the graphics cell to be copied, then select **Edit ▸ Copy As** from the menu bar, and finally choose either PICT or Bitmap PICT or PICT with Embedded Postscript. Now switch to Microsoft Word. When you select **Edit ▸ Paste Special As ...**, and then select Picture in the pop-up window, the graphics will be inserted at the current insertion point in your Microsoft Word document. Text from text cells and input cells in Mathematica can be imported to Word through the regular copy/paste mechanism.

- (a) Create a notebook with an input cell containing a `Plot3D[ ]` command and its graphics output cell as well as a text cell explaining a significant feature of your picture. Export all three cells into a Microsoft Word Document.
- (b) Repeat part (a) with a two-dimensional graph.

#### EXERCISE 5: Incorporating Mathematica graphics into a Web page.

Create a notebook which contains a single graphics output cell. Follow the process of Examples 7 and 8 to create the folder with a .gif and an index.html file. Copy the folder to the folder for your personal Web page and create a link to it from one of your pages.

This Page Intentionally Left Blank

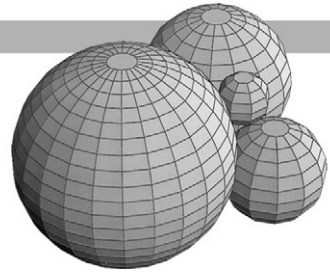
V

---

## *Student Projects*

This Page Intentionally Left Blank

# *Student Projects*



**Introduction**

**Arnold's Cat Map and Chaotic Mapping**

**Bouncing Balls**

**Collatz's Function**

**Conway's Challenge Sequence**

**Exponentially Damped Surfaces**

**Finite Automata**

**Fractals and Chaotic Boundary Sets**

**Fractals and Iterated Function Systems**

**Geometric Optics and Lens Systems**

**Groups of Rigid Motions**

**Growth Rates of Functions**

**Harmonic Coupled Oscillations**

**Hidden Patterns**

**Implementation of a Package**

**Interpolation of Curves with Cubic Splines**

**Juggling Balls**

**Leasing a Car**

**Markov Chains and Dynamic Models**

**Moiré Fringes**

**Oscillating Mass System**

**Pell's Equation**

**Public Key Cryptography**

**Rainbows**

**Recurrence Relations**

**Spanning Trees of a Graph**

## ***Introduction***

---

On the following pages we have assembled a number of suggestions for student projects. This is a work in progress with new titles being added to the list each time the course is taught. These projects reflect the interests of the instructors and of the students who enrolled in the course. Some of the projects included here were developed by students. Needless to say, we have also tried out a number of projects that did not quite work out as expected, and those have not been included here. We have had good luck with the projects on this list.

Most of the projects require material that was not presented in the notebooks of the text. In particular, for some of the projects it will be necessary to know about Mathematica's mechanisms for solving differential equations. We have found that at this point in the course, students have enough competence with Mathematica to use the Help Browser and the examples in the Mathematica book to extract the necessary material.

Different instructors and different groups of students would probably have designed a very different set of projects. For new ideas and alternatives to these projects, try some of the recent Mathematica-based books on modeling and simulation and the large number of Web sites that feature Mathematica-based projects. A wealth of information is available from the extensive Mathematica Web site at <http://library.wolfram.com>. You definitely want to look at the examples that can be found at <http://library.wolfram.com/explorations>, and you may want to check out <http://library.wolfram.com/forums/student-support/list>.

We have found that most of our students were willing to work very hard on their projects but had little experience with independent work. They needed help and guidance to get started and to keep on a schedule. Following is an outline of what we do to keep the students on track and to establish the standards we expect for the final project. Students get this information on the first day of classes. The sample is from the Fall term 2000 at the University of Michigan-Dearborn.

## ***Final Project Time Table***

Week of October 30: Choose a topic for the final project.

Weeks of October 30 and November 6: Collect materials and read about the topic; study the necessary mathematics.

Friday, November 10: Submit a two-page proposal for your final project and an outline of what you plan to do. Be very specific. Also attach a detailed time table that lists the dates when you want to have parts of the project completed. Think about this carefully. Be realistic.

Wednesday, November 29: Start writing the final version of the final report. Start preparing for the oral presentation.

The final project will be concluded by submitting a printed project report of approximately 20 pages and an oral presentation of approximately 15 to 20 minutes.

Printed reports are due on Friday, December 15.



Oral presentations of final reports will be scheduled in Room 106 SSC during class time on Wednesday, December 6, Friday, December 8, and the final exam time slot Friday, December 15, 9:00 A.M. to 12:00 P.M.

## ***Final Project Printed Report***

The printed project report will contain the following items:

1. **Title page**
2. **Table of contents** List sections of the project with page numbers.
3. **Introduction** A narrative description of the main features of the project. What is the topic? Why was it chosen? What are the problems to be solved? What are the mechanisms developed to solve the problems? (Approximately two pages in the notebook.)
4. **Project** Presentation of the algorithms, programs, mathematical ideas, conjectures, and examples that lead to the solutions of the problems. This is the main part of the project. It should contain plenty of explanatory text to make the programs and the mathematics understandable to the reader. Submit this as a printout of a Mathematica notebook in evaluated form. Include only those evaluations that are essential for understanding the solutions.
5. **Conclusion** A summary of the results achieved in step 4. (No more than two pages, but at least one full page.)
6. **References** List all sources of materials including articles, books, software, manuals, and URLs with their publication dates.
7. **Evaluation** A description of the problems you encountered with the mathematics, the programs, the hardware, or the software. A description of what went particularly well. How much time did you spend on the project? Where was the work done, at home or in a campus computer lab? How much help did you need? Did you get enough help if you needed it? Include whatever else you believe is significant.

The oral presentation should be timed to last approximately 15 minutes. In your presentation concentrate on the highlights and main aspects of your project. Explain the tricky parts and show off how much you learned. Be flashy and entertaining. This is your opportunity to shine. Come prepared to show your notebook using the projection equipment in Room 106 SSC. Either store your file on the public part of the file server and remember how to find it quickly, or bring a diskette.

Class attendance is mandatory on days when projects are presented.

## ***Final Project Evaluation***

The final project will be evaluated according to the following criteria:

- Class attendance on three presentation days: 10%
- Timely completion of the project: 5%
- Quality of the oral presentation (well prepared, well organized, explains what the project is about, how it was solved, highlights the main points of the project, knowledge of subject matter, can answer a question or two): 20%
- Quality of the printed report (followed the expected outline, no parts missing, report has page numbers, use of palettes for mathematical symbols where appropriate, etc.): 10%
- Sufficient depth in part 7 (project evaluation): 5%
- Quality of the project (User-defined functions conform to their specifications; in other words, they do what you claim they will do. Adequate examples are expected. Mathematical and programming sophistication will be judged. Graduating seniors with significant mathematics background will be expected to do more sophisticated work than others. Adequate explanatory text and pictures are expected.): 50%

## *Arnold's Cat Map and Chaotic Mapping*

---

### *Goal*

Implement chaotic mappings such as Arnold's cat map. Examine the effect of repeated applications (iterations) of Arnold's cat map on some images other than the cat. Investigate the periodic behavior of the map and the streaking that appears in some of the iterations.

### *Resources*

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Howard Anton & Chris Rorres. *Elementary Linear Algebra: Applications Version*, 7th edition. John Wiley & Sons Inc., 1994, Section 11.5 "Chaos," pages 717–731.
- (2) Robert Devaney. *An Introduction to Chaotic Dynamical Systems*. Addison-Wesley, 1989.
- (3) F. J. Dyson & H. Falk. "Period of a Discrete Cat Mapping," *The American Mathematical Monthly*, vol. 99, August-September 1992, pages 603–614.

### *Prerequisites*

This project requires familiarity with the following:

- (1) Linear algebra: Eigenvectors, eigenvalues, and linear transformations of the plane

## ***Bouncing Balls***

---

### ***Goal***

Consider a small ball that is dropped onto a table and bounces off elastically. When the table is a cone or a paraboloid then the ball may bounce back and forth on the table. Further physical constraints may be added to the ideal model by adding air resistance for the ball or modeling inelastic collisions between the ball and the table or letting the table vibrate. However, the physical assumptions should guarantee that the motion of the ball is confined to a plane perpendicular to the table surface.

The goal of this project is to solve (numerically) for the motion of the bouncing ball and to discuss the behavior of the trajectory of the ball under various initial conditions. The solutions should be rendered graphically and the trajectories of the bouncing ball should be animated.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Nicholas B. Tufillaro, Tyler Abbott & Jeremiah Reilly.  
*An Experimental Approach to Nonlinear Dynamics and Chaos*.  
Addison-Wesley, 1992.
- (2) Gareth Williams. *Linear Algebra with Applications*, 4th edition.  
Wm. C. Brown Publishers, 2000.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Physics: Mechanics
- (2) Differential equations: Systems of equations
- (3) Dynamic systems: Chaotic orbits and attractors

## Collatz's Function

---

### Goal

The Collatz function is defined as  $f(x) = \frac{x}{2}$  when  $x$  is even, and  $f(x) = \frac{3x+1}{2}$  when  $x$  is odd. We introduced this function in Example 22 of the notebook `Loops.nb`, “Iterations with Loops,” in Part III. We computed the runs created by the successive applications of the Collatz function to a starting value.

The objective of this project is to study the computational complexity of the Collatz function. To this end you should implement a number of functions in Mathematica that provide experimental data for the complexity studies. For example, these functions will compute the runs, their length, tables of these objects, the frequency distribution of the lengths, and so on.

### Resources

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Roman E. Maeder. *Programming in Mathematica*, 3rd edition. Addison-Wesley Longman, 1997.
- (2) Jeffrey C. Lagarias. “The  $3x + 1$  Problem and Its Generalizations,” *American Mathematical Monthly*, vol. 92, 1985, pages 3–32.
- (3) Theodore W. Gray & Jerry Glynn. *Exploring Mathematics with Mathematica*. Addison-Wesley, 1991, pages 284–288 and 295–297.
- (4) Ilan Vardin. *Computational Recreations in Mathematica*. Addison-Wesley, 1991, Chapter 7.

### Prerequisites

This project requires familiarity with the following:

- (1) Elementary number theory

## ***Conway's Challenge Sequence***

---

### ***Goal***

In a lecture at AT&T Labs in 1988 Conway introduced the sequence 1, 1, 2, 2, 3, 4, 4, 4, 5, 6, 7, 7, 8, 8, 8, 8, 9, . . . which is defined by the recursion  $a(n) = a(a(n-1)) + a(n - a(n-1))$ , for  $n \geq 3$ . Study the structure of this sequence. Make small changes to the recursive formula or to the initial values and study the structure of the resulting sequences. Explain some of the properties of these sequences such as monotonicity, rate of growth, and asymptotic behavior.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc. New York, 1979, pages 137–138.
- (2) Colin L. Mallows. "Conway's Challenge Sequence," *The American Mathematical Monthly*, vol. 98, no. 1, January 1991, pages 5–20.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Combinatorics
- (2) Elementary number theory

## ***Exponentially Damped Surfaces***

---

### ***Goal***

This project is a variation of Example 8 in the notebook Assign.nb, “Values, Variables, and Assignments,” in Part II. Generalize the visualization of the characteristics of curves in two dimensions to surfaces in three dimensions. Use the Plot3D[ ], ContourPlot[ ], and DensityPlot[ ] functions to experimentally locate critical points, saddle points, and regions of uniform concavity. Then apply the theory of multivariable calculus to support your experimental findings.

Start with a personal number such as your Student ID, your Social Security number, or your birthdate. Construct a function in two variables from this number that includes the exponential damping factor  $e^{-(x^2+y^2)}$ . The damping ensures that the critical points of the function occur in a circular area close to the origin. You may want to choose a trigonometric function such as  $f(x, y) = e^{-(x^2+y^2)} \sin(ax + \cos(by))$  or a damped polynomial function in two variables such as  $p(x, y) = e^{-(x^2+y^2)}(3x^5y^2 - 2x^3y^3 + 7xy - 1)$ .

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Larry Riddle. “Individualized Computer Investigations for Multivariate Calculus,” *The College Mathematics Journal*, vol. 26, no. 3. 1995, pages 235–237.
- (2) James Stewart. *Calculus*. Brooks/Cole, 1999.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Multivariable calculus

## *Finite Automata*

---

### *Goal*

Deterministic finite automata (DFA) are used in computer science to implement language recognizers and in mathematics to model finite state systems. Study another, more general, class of finite automata such as nondeterministic automata or pushdown automata.

Understand the techniques used in the accompanying notebook `DFAIntro.nb` and package `DFA.m` in the folder `ProjectsSupport`, which implement deterministic finite automata and trace computations in them. Generalize these techniques to the class of automata that you choose to implement.

### *Resources*

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) John B. Fraleigh. *A First Course in Abstract Algebra*, 6th edition. Addison-Wesley, 1999.
- (2) Peter Linz. *An Introduction to Formal Languages and Automata*, 3rd edition. Jones and Bartlett Publishing Inc., 2001.
- (3) John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.
- (4) Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Co., 1997.
- (5) Thomas A. Sudkamp. *Languages and Machines*, 2nd edition. Addison-Wesley, 1997.
- (6) Hartmut F. W. Höft & Margret H. Höft. *Computing with Mathematica*, `ProjectsSupport` folder in the `Projects` folder on the CD.  
Mathematica notebook: `DFAIntro.nb`, “Deterministic Finite Automata,” 1998.  
Mathematica package: `DFA.m`, 1998.

### *Prerequisites*

This project requires familiarity with the following:

- (1) Formal languages and finite automata



## *Fractals and Chaotic Boundary Sets*

---

### *Goal*

This project is an extension of Examples 12 and 15 as well as of Exercise 8 in the notebook *Loops.nb*, “Iteration with Loops,” in Part III. Many objects in nature have a self-similar structure, that is, their global form appears to be similar to a small piece. Examples are river systems, coast lines, trees, and clouds. Many of these structures have complicated boundaries that can be realized as the limit of an iteration process. Frequently, the dimension of the resulting object is not a whole number, like two or three, but rather a fraction, like  $\frac{3}{2}$  or  $\log_2 3$ , giving rise to the name fractal.

One process for generating fractals starts with a complex quadratic polynomial  $p(z) = z^2 + c$ , where  $z$  is a complex variable and  $c$  is a constant complex number. This polynomial is iterated with a starting value of  $z_0 = 0$ . The sequence of iterates of the polynomial  $p$  may converge to a point, diverge to infinity, or do neither of these, but define some boundary set. These boundaries are called Julia sets, and the collection of values for the constant  $c$  for which the iterated polynomial sequence does not diverge is called the Mandelbrot set.

The goal of this project is to study some Julia sets and the Mandelbrot set in detail and to produce graphic images of these sets.

### *Resources*

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Michael Barnsley. *Fractals Everywhere*. Academic Press, 1988.
- (2) Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. Freeman, 1983.
- (3) H.-O. Peitgen & P. H. Richter. *The Beauty of Fractals*. Springer Verlag, 1986.  
“Frontiers of Chaos,” pages 1–22.  
Special Section 2: “Julia Sets and Their Computergraphical Generation,” pages 27–52.
- (4) A. David Wunsch. *Complex Variables and Applications*, 2nd edition. Addison-Wesley, 1994.  
Appendix A: “Sequences, Fractals and the Mandelbrot Set,” pages 286–295.
- (5) Hartmut F. W. Höft & Margret H. Höft. *Computing with Mathematica*, ProjectsSupport folder in the Projects folder on the CD.  
Mathematica notebook: IFSIntro.nb, “Iterated Function Systems,” 2002.  
Mathematica package: IFS.m, 2002.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Complex variables

## ***Fractals and Iterated Function Systems***

---

### ***Goal***

This project is an extension of Examples 12 and 15 as well as of Exercise 8 in the notebook *Loops.nb*, “Iteration with Loops,” in Part III. Many objects in nature have a self-similar structure, that is, their global form appears to be similar to a small piece. Examples are river systems, coast lines, trees, and clouds. Many of these structures have complicated boundaries that can be realized as the limit of an iteration process. Frequently, the dimension of the resulting object is not a whole number, like two or three, but rather a fraction, like  $\frac{3}{2}$  or  $\log_2 3$ , giving rise to the name fractal.

In one such limit process a set of linear transformations is applied iteratively, starting from a single point. The process creates a trajectory of points. The limiting trajectory then defines the fractal. This project researches the generating process in iterated function systems. The goal is to implement the iteration process and graphically render fractals in the Euclidean plane.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Howard Anton & Chris Rorres. *Elementary Linear Algebra: Applications Version*, 7th edition. John Wiley & Sons Inc., 1994.  
Section 11.14, “Fractals,” pages 699–717.
- (2) Gareth Williams. *Linear Algebra with Applications*, 3rd edition. William C. Brown Publ., 1994.  
From Section 7.2, “Computer Graphics and Fractals,” pages 342–347 and 350.
- (3) Michael Barnsley. *Fractals Everywhere*. Academic Press, 1988.
- (4) Hartmut F. W. Höft & Margret H. Höft. *Computing with Mathematica*, ProjectsSupport folder in the Projects folder on the CD.  
Mathematica notebook: IFSIntro.nb, “Iterated Function Systems,” 2002.  
Mathematica package: IFS.m, 2002.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Linear algebra: Linear and affine transformations

## ***Geometric Optics and Lens Systems***

---

### ***Goal***

One useful experimental tool to design optical systems (that is, assemblies of lenses) is the tracing of rays of light. This method of approximating the behavior of optical systems is called geometric optics. Take different lens forms, convex, concave, spherical, and so on, and trace rays of one color or of several colors in single and multiple lens systems. Study and demonstrate the effects of spherical and of chromatic aberration.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) David Halliday, Robert Resnick & Jearl Walker. *Fundamentals of Physics*, 6th edition. John Wiley & Sons, Inc., 2000.  
Sections on Optics, Interference, and Diffraction.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Physics: Optics
- (2) Calculus I

## ***Groups of Rigid Motions***

---

### ***Goal***

Generate the multiplication (addition) tables for some finite groups of small cardinalities. Possible choices include dihedral groups, groups of rigid motions, and matrix groups. Provide functions that create the group tables and functions to compute with these tables.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) John B. Fraleigh. *A First Course in Abstract Algebra*, 6th edition. Addison-Wesley Publ. Co., 1999.  
Chapters 1–5, “Groups and Fields.”
- (2) Joseph A. Gallian. *Contemporary Abstract Algebra*, 4th edition. Houghton Mifflin Co., 1998.  
Chapters on Groups  
Sections 27 & 28 : Symmetry Groups, Frieze Groups, and Crystallographic Groups.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Modern algebra or Group theory
- (2) Linear algebra: Linear transformations
- (3) Geometry: Rigid motions and symmetries of planar figures

## ***Growth Rates of Functions***

---

### ***Goal***

One important area in computer science and in computational mathematics is the study of algorithms and their complexity. This includes the time and the space required for the successful execution of an algorithm. Select a number of algorithms and compare their computational complexities. Examples are searches in files of data, sorting a given set of data, and constructing objects that satisfy a given property (such as spanning trees or search trees). Discuss different growth rates and classify your examples accordingly.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Richard Johnsonbaugh. *Discrete Mathematics*, 5th edition. Prentice Hall, Upper Saddle River, NJ, 2001.
- (2) Gregory J. E. Rawlins. *Compared to What, an Introduction to the Analysis of Algorithms*. Computer Science Press, 1992.
- (3) James Stewart. *Calculus*. Brooks/Cole 1999.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Discrete mathematics
- (2) Data structures
- (3) Calculus

## ***Harmonic Coupled Oscillations***

---

### ***Goal***

Consider two masses suspended in series by two springs. For example, the first mass is suspended with a spring from a ceiling and the second mass is suspended from the first mass with another spring. The system is set into motion by positioning one or both masses at points away from the equilibrium and then let go of the masses.

Set up differential equations or systems of differential equations for such configurations and derive their solutions to describe the motion of the system. Render this motion graphically and animate the graphics.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) An introductory textbook in physics.
- (2) An introductory textbook in differential equations.
- (3) Robert L. Zimmerman & Fredrick I. Olness. *Mathematica for Physics*. Addison-Wesley, 1995, Chapter 3.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Physics: Mechanics
- (2) Differential equations

## ***Hidden Patterns***

---

### ***Goal***

Consider a periodic function such as  $y = \sin x$ . When it is computed for integer arguments different patterns emerge in the rendered graph depending on the range of numbers chosen. Study and explain these patterns. Also experiment with other functions.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Doug Hardin & Gilbert Strang, "A Thousand Points of Light," *The College Mathematics Journal*, vol. 21, no. 5, 1990, pages 406–409.
- (2) Norman Richert, "Strang's Strange Figures," *American Mathematical Monthly*, vol. 99, no. 2, 1992, pages 101–107.
- (3) Gilbert Strang. *Calculus*. Wellesley: Cambridge Press, 1991.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Calculus



## ***Implementation of a Package***

---

### ***Goal***

With the package tool Mathematica provides a mechanism to set up a context for a collection of functions. Study the notebook PackDef.nb, “Advanced Mathematica: Packages,” in Part IV. Use the example packages in that notebook as a guide, particularly the Gram-Schmidt Orthogonalization since that includes algebraic functions as well as a plotting function. Select an area from computer science, mathematics, or the physical sciences that is of particular interest to you. Design and implement a package for a collection of functions from that area that you think are a useful. Exercise 3 on binary relations and Exercise 5 on the Riemann integral are examples of specific collections of functions that might be included in a package.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Roman Maeder. *Programming in Mathematica*, 3rd edition. Addison-Wesley, 1997.  
Chapter 2: Packages.
- (2) Stephen Wolfram. *The Mathematica Book*, 4th edition, Mathematica Version 4. Wolfram Media & Cambridge University Press, 1999.  
Sections 2.6.8–2.6.11.
- (3) Wolfram Research. *Mathematica 4 Standard Add-on Packages*. Wolfram Media & Cambridge University Press, 1999.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Linear algebra
- (2) Calculus
- (3) An area of your choice for the package implementation

## *Interpolation of Curves with Cubic Splines*

---

### *Goal*

It is a common situation in science and engineering that a finite sequence of points is given to which a smooth curve needs to be fitted. Frequently the approximating curve must pass through all the points. The curve is constructed in pieces from subsequences of the given points, and each piece is represented by a polynomial of low degree. Such curves are called splines.

Design and implement several methods for interpolating a given set of points in the plane. Include cubic splines, Hermetian splines, and also parametric splines.

### *Resources*

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Howard Anton & Chris Rorres. *Elementary Linear Algebra: Applications Version*, 7th edition. John Wiley & Sons, 1994.  
Section 11.5: "Cubic Spline Interpolation," pages 603–614.
- (2) Wilbur J. Hildebrand.  
"Connecting the Dots Parametrically: An Alternative to Cubic Splines,"  
*The College Mathematics Journal*, vol. 21, no. 3, May 1990, pages 208–215.
- (3) John H. Mathews. *Numerical Methods for Computer Science, Engineering & Mathematics*.  
Prentice Hall, 1987.  
Chapter 5: "Curve Fitting."
- (4) The package Spline.m in the Graphics folder of the Mathematica packages folder.
- (5) The package SplineFit.m in the NumericalMath folder in the Mathematica packages folder.
- (6) Hartmut F. W. Höft & Margret H. Höft. *Computing with Mathematica*,  
ProjectsSupport folder in the Projects folder on the CD.  
Mathematica notebook: LPSIntro.nb, "Local Parametric Splines," 2002.  
Mathematica package: LPS.m, 2002.

### *Prerequisites*

This project requires familiarity with the following:

- (1) Linear algebra
- (2) Calculus I

## ***Juggling Balls***

---

### ***Goal***

Juggling objects, say balls, with two hands is a game requiring considerable skills. There are two basic questions about juggling: How many balls can one juggle? Given a number of balls, what are the different patterns in which to juggle the balls?

Investigate the notations that jugglers use to describe patterns, create patterns, and represent them graphically.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Peter J. Beek & Arthur Lewbel. “The Science of Juggling,”  
*Scientific American*, November 1995, pages 92–97.
- (2) The bibliography cited in that article.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Calculus
- (2) Combinatorics

## ***Leasing a Car***

---

### ***Goal***

This project is an extension of Example 14 and Exercise 8 in the notebook FunctDef.nb, “Functions,” in Part II. Cars are pervasive in American society and a necessary commodity for most people. Therefore the purchase of a car, whether new or old, is an activity that cannot be avoided. However, in today’s market a large portion of all new cars are leased rather than sold. The goal of this project is to make a complete analysis of the costs associated with the lease of a car.

Search for advertisements of car dealers in local newspapers. Investigate the terms of leases offered for various types of cars and by different dealers.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) JoAnn Muller. “When a Driver and a Lease Must Part Company,” *The New York Times*, October 8, 1995, Business section.
- (2) “New Light on Leases,” *US News & World Report*, July 3, 1995, page 62.
- (3) “Lease Instead of Buy?” *Consumer Reports*, April, 1995, pages 268–269.
- (4) Tracey Longo. “Is That New Car Lease a Fair Deal?” *Kiplinger’s Personal Finance Magazine*, October, 1995, pages 113–114.
- (5) James S. Schallheim. *Lease or Buy?: Principles for Sound Decision Making*. Harvard Business School Press, Boston, 1994.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Algebra: Solving linear and nonlinear equations
- (2) Calculus: Finding roots

## ***Markov Chains and Dynamic Models***

---

### ***Goal***

A mathematical model that tries to make predictions for a system over a period of time is called a dynamic model. A Markov chain is a probability model simulating the behavior of a system that moves among different states over a period of time according to fixed transition probabilities.

Investigate and implement Markov chain models that occur in a variety of mathematical models for population movement, genetics, gambling, and economics.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Howard Anton & Chris Rorres. *Elementary Linear Algebra: Applications Version*, 7th edition. John Wiley & Sons Inc., 1994.  
Section 11.6, “Markov Chains,” pages 614–625.
- (2) Alan Tucker. *A Unified Introduction to Linear Algebra: Models, Methods, and Theory*. Macmillan Publishing, 1988.  
Section 1.3, “Markov Chains and Dynamic Models,” pages 21–37.  
Section 4.4, “Markov Chains,” pages 303–321.
- (3) Gareth Williams. *Linear Algebra with Applications*, 3rd edition. William C. Brown Publ., 1994.  
Section 2.6, “Stochastic Matrices: A Population Movement Model,” pages 102–109.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Linear algebra

## *Moiré Fringes*

---

### *Goal*

Moiré patterns appear when geometric patterns overlap, for example, when two wavefronts interfere or two line gratings on pieces of glass are overlaid. Even with lines and circles many interesting patterns emerge. The goal of this project is to implement a variety of Moiré patterns, animate some patterns and understand the theory for the patterns created by lines and circles.

### *Resources*

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Yasuhiko Arai, Tomoharu Yamada & Shunsuke Yokozeki.  
“Fringe-Scanning Method Using a General Function for Shadow Moiré,”  
*Applied Optics*, vol. 34, no. 22, 1995, pages 4877–4881.
- (2) Mike Cullen, “Moiré Fringes and the Conic Sections,”  
*The College Mathematics Journal*, vol. 21, no. 5, 1990, pages 370–378.
- (3) Hongkai Lai, Shou Liu & Xiangsu Zhang.  
“Artistic Effect and Application of Moiré Patterns in Security Holograms,”  
*Applied Optics*, vol. 34, no. 22, 1995, pages 4700–4702.
- (4) Theodore W. Gray & Jerry Glynn. *Exploring Mathematics with Mathematica*.  
Addison-Wesley, 1991.  
Chapter 12, “Density Plots,” pages 217–236.
- (5) Gerald Oster, “Moiré Patterns,”  
*Encyclopedia of Science and Technology*, 1985 edition.
- (6) Gerald Oster & Yasunori Nishijima. “Moiré Patterns,”  
*Scientific American*, vol. 208, no. 5, 1963, pages 54–63.
- (7) P. S. Theocaris. *Moiré Fringes in Strain Analysis*.  
Pergamon Press, Elmsford N.Y., 1969.

***Prerequisites***

This project requires familiarity with the following:

- (1) Geometry
- (2) Calculus
- (3) Optics

## ***Oscillating Mass System***

---

### ***Goal***

Model a spring and mass system and animate the result. Start with a single mass that has four springs attached to it at right angles to each other. Each of the springs is attached to a fixed wall with the four walls forming a square. Incorporate different combinations of spring constants as well as different combinations of initial velocities and initial positions of the mass.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) William E. Boyce & Richard C. DiPrima. *Elementary Differential Equations*, 7th edition. John Wiley & Sons, Inc., 2000.
- (2) David Halliday, Robert Resnick & Jearl Walker. *Fundamentals of Physics*, 6th edition. John Wiley & Sons, Inc., 2000.
- (3) Marvin deJong. *Mathematica for Calculus-based Physics*. Addison-Wesley Publishing Co., 1999.
- (4) Robert L. Zimmerman & Fredrick I. Olness. *Mathematica for Physics*. Addison-Wesley Publishing Co., 1995.  
Chapter 3, "Oscillating Systems."

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Calculus
- (2) Differential equations: Numerical methods for solving ordinary differential equations
- (3) Physics: Hooke's law and Newton's second law



## *Pell's Equation*

---

### *Goal*

This project relates to Example 5 and Exercise 4 in the notebook *Recurse.nb*, “Recursive Definitions,” in Part II. Study the solutions of Pell’s equation  $x^2 - ky^2 = a$ , where  $k$  and  $a$  are integers. That is, find all integer values for  $x$  and  $y$  that solve the equation. For what values of  $k$  does one get finitely many solutions? When does one get infinitely many solutions? Investigate the relations between the solutions and Fibonacci sequences. Investigate the relations between the solutions and continuous fractions.

### *Resources*

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Laurent Béeckmans. “Squares Expressible as Sums of Consecutive Squares,” *American Mathematical Monthly*, May 1994, pages 437–442.
- (2) C. F. Gauss. *Disquisitiones Arithmeticae*. Yale University Press, 1966 (republished).
- (3) Niven & Zuckerman & Montgomery. *An Introduction to the Theory of Numbers*, 5th edition. John Wiley & Sons, New York, 1991.
- (4) David A. Cox. *Primes of the form  $x^2 + dy^2$ : Fermat, Class Field Theory, and Complex Multiplication*. John Wiley & Sons, New York, 1997.

### *Prerequisites*

This project requires familiarity with the following:

- (1) Elementary number theory
- (2) Recursion

## Public Key Cryptography

---

### Goal

During the 1980s, a new method to encode and decode messages was developed. This method, called public key cryptography, does not require the two parties who want to communicate in secret ever to meet. The name “public key” derives from the fact that a portion of the key, two large numbers, can be made public without revealing the secret part of the key. By today’s computing standards the numbers in the key must have more than 200 digits in order for the key to be unbreakable.

Implement a public key encryption system that is based on the RSA algorithm. The algorithm is named after Rivest, Shamir, and Adleman who developed the algorithm and then patented an implementation in a hardware device. Discuss and also implement a mechanism to authenticate encrypted messages.

### Resources

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Wayne Patterson. *Mathematical Cryptology for Computer Scientists and Mathematicians*. Totowa, New Jersey, 1987.
- (2) R. L. Rivest & A. Shamir & L. M. Adleman. “A Method for Obtaining Digital Signatures and Public Key Cryptosystems,” *Communications ACM*, vol. 21, no. 2, 1978, pages 120–126.
- (3) Dominic Welsh. *Codes and Cryptography*. Clarendon Press, Oxford, 1989.
- (4) Lance J. Hoffman. “Clipping Clipper,” *Communications ACM*, vol. 36, no. 9, 1993, pages 15–17.
- (5) Steven Levy. “The Cyberpunks vs Uncle Sam,” *The New York Times Magazine*, June 12, 1994.
- (6) Hartmut F. W. Höft & Margret H. Höft. *Computing with Mathematica*, ProjectsSupport folder in the Projects folder on the CD.  
Mathematica notebook: RSAIntro.nb, “Hiding Information Publicly,” 1998.  
Mathematica package: RSA.m, 1998.

***Prerequisites***

This project requires familiarity with the following:

- (1) Elementary number theory: Modular arithmetic
- (2) Elementary group theory: Euler's and Fermat's theorems

## ***Rainbows***

---

### ***Goal***

Trace the development of some of the mathematical and physical theories of rainbows through history. Produce color simulations of the refractive and reflective decomposition of sunlight and produce color simulations of the primary and secondary bows.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) René Descartes. "Optics," *The Philosophical Writings of Descartes*, vol. 1, translated by John Cottingham, Robert Stoothoff & Dugald Murdoch. Cambridge University Press, New York, 1985.
- (2) Robert Greenleer. *Rainbows, Halos and Glories*. Cambridge University Press, New York, 1980.
- (3) David K. Lynch & William Livingston. *Color and Light in Nature*. Cambridge University Press, New York, 1995.
- (4) Isaac Newton. *Opticks*. Dover, New York, 1952.
- (5) James Stewart. *Single Variable Calculus*, 4th edition. Brooks/Cole, 1999. Applied Project: The Calculus of Rainbows, page 232.
- (6) The Rainbow Lab developed at the Geometry Center at the University of Minnesota.  
<http://www.geom.umn.edu/locate/lab>
- (7) Arthur Zajonc. *Catching the Light*. Bantam Books, New York, 1993.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Physics: Optics
- (2) Calculus

## ***Recurrence Relations***

---

### ***Goal***

Many biological, mathematical, and physical processes can be expressed in terms of a recursion or a recurrence relation when they are modeled in finitely many steps over a finite domain. In this project you select a particular recurrence relation. Perform computational experiments with the recurrence and formulate hypotheses from the results of the experiments. This leads to a closed formula, patterns in the sequence of numbers generated, and formal proofs for the properties discovered experimentally.

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) Dmitri Thoro & Linda Valdez. "Investigation of a Recurrence Relation," *The College Mathematics Journal*, vol. 25, no. 4, 1994, pages 322–324.
- (2) K. H. Rosen. *Discrete Mathematics and Its Applications*, 4th edition. McGraw-Hill, 1999. Chapter V: Advanced Counting Techniques.

### ***Prerequisites***

This project requires familiarity with the following:

- (1) Combinatorics: Solutions to recurrence relations
- (2) Elementary number theory

## ***Spanning Trees of a Graph***

---

### ***Goal***

Study algorithms that construct a spanning tree for a graph and implement at least one algorithm. Visualize the actions of the algorithm with appropriate graphics. A viewer should understand the underlying concepts of the algorithm from the graphics. Animate the construction of the spanning tree.

Still pictures and animated graphics can demonstrate different aspects of an algorithm. The implementation should be guided by the proverb: “A picture is worth a thousand words.”

### ***Resources***

Here are some references to textbooks, journal articles, news and magazine articles, and Mathematica packages and notebooks that can be used for the project.

- (1) R. Johnsonbaugh. *Discrete Mathematics*, 5th edition. Prentice Hall, 2001.
- (2) K. H. Rosen. *Discrete Mathematics and Its Applications*, 4th edition. McGraw-Hill, 1999.
- (3) M. Weiss. *Data Structures and Algorithm Analysis in C++*, 2nd edition. Addison-Wesley, 1999.
- (4) The package Combinatorica.m in the DiscreteMath folder of the Mathematica Standard Packages folder.

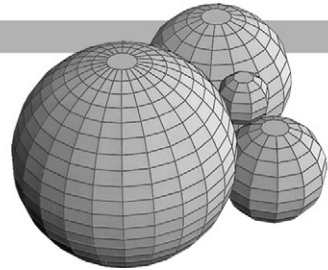
### ***Prerequisites***

This project requires familiarity with the following:

- (1) Graph theory: Adjacency matrices
- (2) Combinatorics

This Page Intentionally Left Blank

# *Index*



- abort an evaluation, 21
- Ackermann function, 97
- Add-ons, 37
- animation of graphics, 164
- approximation
  - bisection, 186, 202
  - Bisection method, 168
  - Newton's method, 216
  - visualization, 190
- arithmetic, 3
- assignment
  - delayed, 49
  - graphical value, 48
  - immediate, 46
  - to nested levels, 49
  - numeric value, 46
  - simultaneous, 49
  - structured value, 47
  - symbolic value, 48
- brace, 12
- bracket
  - double square, 12
  - single square, 5
- button, 28
- cell
  - grouping, 26
  - initialization, 236
  - tag, 27
- clear a value, 6
- Collatz function, 175
- context
  - global, 235
  - package, 240
  - system, 235
- context mark, 234
- cube root
  - complex computation of, 116
  - real computation of, 116
- derivative, 7
- directory
  - current, 244
  - home, 244
  - path, 244
  - reset, 246
  - set, 245
- Do loop
  - animation with, 163
  - structure, 160
- equation
  - solve approximate, 7
  - solve exact, 7
  - system of, 8
- evaluation
  - conditional, 65
  - conditional evaluation, 65



evaluation (*Continued*)

- efficiency of, 56
- forcing of, 56
- limit of recursive, 95
- order of, 54
- recursive, 88
- timing of, 57
- tracing of, 58

## export

- data, 263
- data formats, 266
- graphics to HTML, 267
- text to HTML, 267

## Fibonacci numbers, 90

## file name, 246

## For loop

- iterator increment, 172
- multiple iterators, 173
- structure, 170

## Four Spheres Problem, 119

## function

- anonymous, 129
- conditional evaluation, 65
- delayed assignment, 54
- evaluation of, 54
- immediate assignment, 54
- multiple case definition, 68
- naming of user-defined, 46
- numeric value, 64
- one definition with cases, 70
- protect, 268
- recursive definition, 88
- several numeric arguments, 58
- single numeric argument, 64
- structured argument, 71
- structured value, 70
- unprotect, 268
- varying number of arguments, 81

## functions

- composition of, 75

## Gram Schmidt algorithm, 250

## graphics animation, 188

## Graphics Shapes, 119

## help

- ?, 19
  - ??, 19
  - browser, 20
- hyperlink, 28
- as a button, 224
  - to a cell, 222
  - to the Help Browser, 222
  - to an Internet resource, 223
  - to a notebook, 223

## import

- data, 263
- from a spreadsheet, 264

## Infinity, 52

## initialization cell, 236

## integral, 7

## iteration with

- Array, 174
- Do loop, 163
- Fold, 141
- For loop, 172
- Map, 135, 163, 173
- NestList, 175
- Table, 130
- While loop, 167

## list

- counting elements, 144
- element extraction, 140
- element positions, 147
- heterogeneous, 128
- nested, 149
- nesting depth, 149
- sorting, 143
- sublist extraction, 143

## lists

- combining, 144

## load a package, 37

## loop

- animation, 164
- Boolean controlled, 163
- continuation expression, 170
- Do, 159
- enumerated, 159

- For, 170
  - multiple increments, 174
  - nested, 161
  - termination, 167
  - While, 165
- matrix, 12
  - element, 13
  - operations, 13
- memory management, 21
- message
  - usage, 241
  - warning, 255
- notebook
  - options, 25
  - toolbars, 25
- option
  - default value, 114
  - definition of, 114
  - setting a value of, 200
- options,
  - filtering of, 213
- package
  - context, 240
  - definition, 242
  - function names in, 234
  - GramSchmidt', 251
  - Iteration', 249
  - layout, 239
  - load a, 39
  - private function in, 239
  - public function in, 242
  - QuickThermometer', 243
  - standard packages, 39
  - Template', 238
  - Thermometer', 240
  - usage, 242
- packages
  - standard add-ons, 236
- palette
  - AlgebraicManipulation, 33
  - BasicCalculations, 34
  - BasicInput, 33
  - BasicTypesetting, 31
  - of characters, 228
  - CompleteCharacters, 31
  - creating a, 225
  - of expressions, 228
  - printing a, 231
  - use of a, 229
- parameter
  - anonymous, 129
  - condition on, 66
  - default value, 76
  - definition of default, 76
  - definition of optional, 80, 114
  - non-standard evaluation of, 79
  - single recursive, 88
  - structured, 76
  - usage of default, 76
  - usage of optional, 80, 114
- parameters
  - multiple recursive, 92
- parenthesis
  - curly, 12
  - double square, 12
  - round, 5
  - single square, 5
- Pascal's triangle, 93
- pattern variable, 50
- plot
  - 2D contour, 11
  - 2D examples, 9
  - 3D examples, 11
  - 3D view point, 11
  - evaluation of arguments, 56
  - options, 9, 52
  - parametric, 10
- power function
  - complex, 269
  - real, 269
- preferences
  - global, 26
  - notebook, 26

- programming
  - functional, 151
- quit the kernel, 22
- random number generator, 13
- recursion
  - depth of, 95
  - efficient evaluation, 88
  - multiple parameters, 92
  - schema, 87
  - single parameter, 88
- remove a name, 39
- replacement
  - nested, 108
  - numeric, 108
  - simultaneous, 108
  - successive, 109
  - symbolic, 109
- series expansion, 190
- simulation
  - roulette, 182
- Stirling numbers, 94
- substitution
  - see replacement, 107
- suppress display of value, 46
- symbol
  - clear, 235
  - context of, 234
  - remove, 235
- tag
  - name, 27
- Taylor polynomials
  - visualization, 190
- uAnalyze, 207
  - option uGraph, 207
  - option uStats, 207
- uAngle, 114
- uArea, 108
- uArithmeticMean, 141
- uBalance, 88
- uBisect, 186
- uBisection
  - option uResult, 203
- uBlueYellow, 139
- uButterfly, 75
- uButterflyPlot, 80
- uCarLoan, 78
- uCollatzRun, 187
- uColMaxs, 148
- uCommutator, 74
- uCoordinatePairs, 134
- uCountOfPi
  - version 1, 144
  - version 2, 144
- uCubeRoot, 117
- uDamping, 51
- uDelayed, 57
- uDigitsOfPi, 114
- uDouble, 96
- uEuclidean, 145
- uFastBalance, 88
- uFib
  - version 1, 90
  - version 2, 90
- uFindPackage, 255
- uFlexBalance, 89
- uFlexVane, 211, 212
  - option uLineStyle, 212
  - option uPointStyle, 212
  - option uVaneScale, 212
- uFlyPlot, 76
- uGetCoords, 151
- uHeight, 108
- uHulaHoop, 189
- uImmediate, 57
- uIntermediate, 174
- uLinearScale, 138
- uLogCos1, 69
- uLogCos2, 69
- uMatrixTrace, 185
- uMeanFct, 204
- uMedianFct, 205
- uNewton, 217
- uPascal, 93

- uPattern, 52
- uPhase, 77
- uPolar, 70
- uPoly, 50
- uPoly1, 58
- uPoly2, 58
- uRandomCircles, 137
- uRandomMatrix
  - version 1, 72
  - version 2, 73
  - version 3, 133
  - version 4, 163
- uRandomScores, 142
- uRealRootQ, 200, 269
- uRI, 135
- uRoot, 200
  - option uCompute, 200
- uRoulette, 182
- uRowMins, 148
- uRR, 136
- uSCurve, 68
- uShowSegments, 139
- uSlide, 68
- uSquare, 129
- uSquareRec, 91
- uStirling, 94
- uStraight, 64
- uStringFunnelF, 184
- uStringFunnelP, 184
- uStringTriangleF, 183
- uStringTriangleP, 182
- uTax, 67
- uTaylor, 190
- uVane, 209
- uVaneParts, 210
- uVarianceFct, 205
- uVectorField2D
  - version 1, 214
  - version 2, 215
- uVerticalLines, 138
- uVerticals, 132
- value
  - suppress display of, 46
- variable
  - global, 181
  - local, 181
- vector
  - Euclidean norm, 133, 145
  - maximum norm, 133
  - orthogonalization, 251
- warning message
  - turning off, 71
  - turning on, 71
- While loop
  - structure, 165
  - termination, 167