

# MATLAB

## Guide

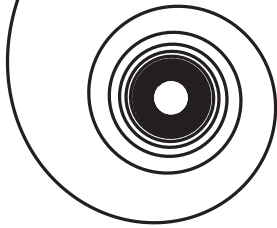
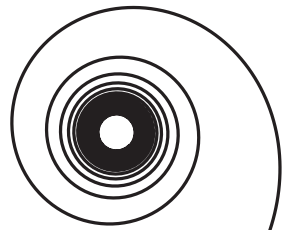
THIRD EDITION

DESMOND J. HIGHAM  
NICHOLAS J. HIGHAM

siam

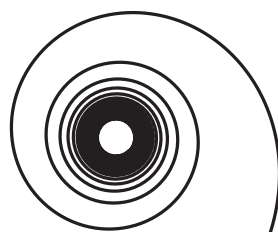
# **MATLAB**

*Guide*



# MATLAB

## Guide



THIRD EDITION

**DESMOND J. HIGHAM**

University of Strathclyde  
Glasgow, Scotland

**NICHOLAS J. HIGHAM**

University of Manchester  
Manchester, England

**siam**<sup>®</sup>

Society for Industrial and Applied Mathematics  
Philadelphia

Copyright © 2017 by the Society for Industrial and Applied Mathematics

10 9 8 7 6 5 4 3 2 1

All rights reserved. Printed in the United States of America. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, write to the Society for Industrial and Applied Mathematics, 3600 Market Street, 6th Floor, Philadelphia, PA 19104-2688 USA.

MATLAB is a registered trademark of The MathWorks, Inc. For MATLAB product information, please contact The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098 USA, 508-647-7000, Fax: 508-647-7001, [info@mathworks.com](mailto:info@mathworks.com), [www.mathworks.com](http://www.mathworks.com).

No warranties, express or implied, are made by the publisher, authors, and their employers that the programs contained in this volume are free of error. They should not be relied on as the sole basis to solve a problem whose incorrect solution could result in injury to person or property. If the programs are employed in such a manner, it is at the user's own risk and the publisher, authors, and their employers disclaim all liability for such misuse.

<i>Publisher</i>	David Marshall
<i>Acquisitions Editor</i>	Elizabeth Greenspan
<i>Developmental Editor</i>	Gina Rinelli Harris
<i>Managing Editor</i>	Kelly Thomas
<i>Production Editor</i>	David Riegelhaupt
<i>Copy Editor</i>	Sam Clark, T&T Productions Ltd, London
<i>Production Manager</i>	Donna Witzleben
<i>Production Coordinator</i>	Cally Shrader
<i>Graphic Designer</i>	Lois Sellers

### Library of Congress Cataloging-in-Publication Data

Names: Higham, Desmond J., 1964- | Higham, Nicholas J., 1961-  
Title: MATLAB guide / Desmond J. Higham, University of Strathclyde, Glasgow, Scotland, United Kingdom, Nicholas J. Higham, University of Manchester, Manchester, United Kingdom.


Description: Third edition. | Philadelphia : Society for Industrial and Applied Mathematics, [2016] | Series: Other titles in applied mathematics ; 150 | Includes bibliographical references and index.

Identifiers: LCCN 2016039121 (print) | LCCN 2016039323 (ebook) | ISBN 9781611974652 (print) | ISBN 9781611974669 (e-book)

Subjects: LCSH: MATLAB. | Numerical analysis--Data processing.

Classification: LCC QA297 .H5217 2016 (print) | LCC QA297 (ebook) | DDC 518.0285/53--dc23

LC record available at <https://lccn.loc.gov/2016039121>

 **siam** is a registered trademark.

*In memory of our mother and father, Doris and Ken.*

# Contents

<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xvii</b>
<b>List of Program Files</b>	<b>xix</b>
<b>Preface</b>	<b>xxi</b>
<b>1 A Brief Tutorial</b>	<b>1</b>
<b>2 Basics</b>	<b>23</b>
2.1 MATLAB Desktop . . . . .	23
2.2 Interaction and Script Files . . . . .	23
2.3 More Fundamentals . . . . .	25
2.4 Help . . . . .	28
2.5 Variables and the Workspace . . . . .	30
<b>3 Distinctive Features of MATLAB</b>	<b>35</b>
3.1 Automatic Storage Allocation . . . . .	35
3.2 Variable Arguments Lists . . . . .	35
3.3 Complex Arrays and Arithmetic . . . . .	37
<b>4 Arithmetic</b>	<b>39</b>
4.1 IEEE Arithmetic . . . . .	39
4.2 Precedence . . . . .	41
4.3 Mathematical Functions . . . . .	42
4.4 Other Data Types . . . . .	42
<b>5 Matrices</b>	<b>47</b>
5.1 Matrix Generation . . . . .	47
5.2 Subscripting and the Colon Notation . . . . .	54
5.3 Matrix and Array Operations . . . . .	57
5.3.1 Implicit Expansion . . . . .	61
5.4 Empty Matrices . . . . .	63
5.5 Matrix Manipulation . . . . .	64
5.6 Data Analysis . . . . .	66
<b>6 Operators and Flow Control</b>	<b>71</b>
6.1 Relational and Logical Operators . . . . .	71
6.2 Flow Control . . . . .	78

<b>7</b>	<b>Program Files</b>	<b>83</b>
7.1	Scripts and Functions . . . . .	83
7.2	Naming and Editing Program Files . . . . .	90
7.3	Working with Program Files and the MATLAB Path . . . . .	91
7.4	Startup . . . . .	92
7.5	Command/Function Duality . . . . .	93
<b>8</b>	<b>Graphics</b>	<b>97</b>
8.1	Two-Dimensional Graphics . . . . .	97
8.1.1	Basic Plots . . . . .	97
8.1.2	Axes and Annotation . . . . .	102
8.1.3	Multiple Plots in a Figure . . . . .	109
8.2	Three-Dimensional Graphics . . . . .	113
8.3	Specialized Graphs for Displaying Data . . . . .	125
8.4	Saving and Printing Figures . . . . .	129
8.5	On Things Not Treated . . . . .	131
<b>9</b>	<b>Linear Algebra</b>	<b>135</b>
9.1	Matrix Properties . . . . .	135
9.2	Norms and Condition Numbers . . . . .	136
9.3	Linear Equations . . . . .	138
9.3.1	Square System . . . . .	138
9.3.2	Overdetermined System . . . . .	140
9.3.3	Underdetermined System . . . . .	141
9.4	Inverse, Pseudoinverse, and Determinant . . . . .	142
9.5	LU, LDL*, and Cholesky Factorizations . . . . .	143
9.6	QR Factorization . . . . .	145
9.7	Singular Value Decomposition . . . . .	146
9.8	Eigenvalue Problems . . . . .	148
9.8.1	Eigenvalues . . . . .	148
9.8.2	More about Eigenvalue Computations . . . . .	150
9.8.3	Generalized Eigenvalues . . . . .	151
9.9	Iterative Linear Equation and Eigenproblem Solvers . . . . .	153
9.10	Functions of a Matrix . . . . .	156
<b>10</b>	<b>More on Functions</b>	<b>159</b>
10.1	Function Handles . . . . .	159
10.2	Anonymous Functions . . . . .	160
10.3	Local Functions . . . . .	161
10.4	Default Input Arguments . . . . .	163
10.5	Variable Numbers of Arguments . . . . .	165
10.6	Argument Checking and Parsing . . . . .	167
10.7	Nested Functions . . . . .	168
10.8	Private Functions . . . . .	169
10.9	Recursive Functions . . . . .	170
10.10	Global and Persistent Variables . . . . .	173
10.11	Exemplary Functions in MATLAB . . . . .	174

<b>11 Numerical Methods: Part I</b>	<b>175</b>
11.1 Polynomials and Data Fitting . . . . .	175
11.2 Nonlinear Equations . . . . .	180
11.3 Optimization . . . . .	184
11.4 The Fast Fourier Transform . . . . .	185
<b>12 Numerical Methods: Part II</b>	<b>189</b>
12.1 Numerical Integration . . . . .	189
12.2 Ordinary Differential Equations . . . . .	193
12.2.1 Examples with Ode45 . . . . .	193
12.2.2 Case Study: Pursuit Problem with Event Location . . . . .	201
12.2.3 Stiff Problems, Differential-Algebraic Equations, and the Choice of Solver . . . . .	205
12.3 Boundary-Value Problems . . . . .	213
12.4 Delay-Differential Equations . . . . .	221
12.5 Partial Differential Equations . . . . .	225
<b>13 Input and Output</b>	<b>233</b>
13.1 User Input . . . . .	233
13.2 Output to the Screen . . . . .	234
13.3 File Input and Output . . . . .	236
13.4 Fine Tuning the Display of Arrays . . . . .	238
<b>14 Troubleshooting</b>	<b>241</b>
14.1 Errors and Assertions . . . . .	241
14.2 Warnings . . . . .	243
14.3 Debugging . . . . .	245
14.4 Pitfalls . . . . .	246
<b>15 Sparse Matrices</b>	<b>249</b>
15.1 Sparse Matrix Generation . . . . .	249
15.2 Linear Algebra . . . . .	252
<b>16 More on Coding</b>	<b>257</b>
16.1 Elements of Coding Style . . . . .	257
16.2 Cleaning Up . . . . .	258
16.3 Checking and Comparing Code Files . . . . .	259
16.4 Profiling . . . . .	260
16.5 P-Code . . . . .	261
16.6 Source Control . . . . .	264
16.7 Live Editor . . . . .	264
16.8 Creating a Toolbox . . . . .	265
16.9 Distributing Code Files . . . . .	268
16.10 Unit Tests . . . . .	269
<b>17 Advanced Graphics</b>	<b>273</b>
17.1 Objects, Handles, and Properties . . . . .	273
17.2 Root and Default Properties . . . . .	278
17.3 Animation . . . . .	279
17.4 Examples . . . . .	281



<b>18 Other Data Types and Multidimensional Arrays</b>	<b>291</b>
18.1 Character Vectors and Arrays	292
18.2 String Arrays	295
18.3 Multidimensional Arrays	297
18.4 Categorical Arrays	299
18.5 Datetime and Duration Arrays	300
18.6 Tables and Timetables	304
18.7 Structures and Cell Arrays	308
<b>19 Object-Oriented Programming</b>	<b>315</b>
19.1 Max-Plus Algebra Class	315
19.2 Circulant Matrix Class	321
19.3 On Things Not Treated	324
<b>20 The Symbolic Math Toolbox</b>	<b>325</b>
20.1 Creating Symbolic Variables and Expressions	325
20.2 Equation Solving	327
20.3 Calculus	330
20.3.1 Integration	330
20.3.2 Differentiation	332
20.3.3 Solving Differentiation Equations	335
20.3.4 Taylor Expansions	336
20.4 Linear Algebra	337
20.5 Polynomials and Rationals	339
20.6 Variable-Precision Arithmetic	343
20.7 Other Features	347
<b>21 Graphs</b>	<b>349</b>
21.1 Undirected Graphs	349
21.2 Directed Graphs	351
<b>22 Large Data Sets</b>	<b>361</b>
22.1 Datastores	361
22.2 MapReduce	364
22.3 Tall Arrays	364
<b>23 Optimizing Codes</b>	<b>369</b>
23.1 Timing Code	369
23.2 Vectorization	370
23.3 Accessing Matrices by Column	372
23.4 Preallocating Arrays	374
23.5 Miscellaneous Optimizations	374
23.6 Illustration: Bifurcation Diagram	375
23.7 External Codes	375
<b>24 Tricks and Tips</b>	<b>379</b>
24.1 Empty Arrays	379
24.2 Exploiting Infinities	380
24.3 Permutations	380
24.4 Rank-1 Matrices	382

24.5	Set Operations . . . . .	383
24.6	Subscripting Matrices as Vectors . . . . .	384
24.7	Avoiding If Statements . . . . .	385
<b>25</b>	<b>The Parallel Computing Toolbox</b>	<b>387</b>
25.1	The Parfor Loop . . . . .	388
25.2	Asynchronous Computing with Parfeval . . . . .	392
25.3	Batch Computations . . . . .	393
25.4	Single Program, Multiple Data . . . . .	395
25.5	Distributed and Codistributed Arrays . . . . .	397
25.6	GPU Computing . . . . .	398
25.7	On Things Not Treated . . . . .	401
<b>26</b>	<b>Case Studies</b>	<b>403</b>
26.1	Introduction . . . . .	403
26.2	Brachistochrone . . . . .	403
26.3	Small-World Networks . . . . .	404
26.4	Performance Profiles . . . . .	409
26.5	Multidimensional Calculus . . . . .	416
26.6	L-Systems and Turtle Graphics . . . . .	420
26.7	Black–Scholes Delta Surface . . . . .	422
26.8	Chutes and Ladders . . . . .	425
26.9	Pythagorean Sum . . . . .	430
26.10	Fisher’s Equation . . . . .	432
<b>A</b>	<b>The Top 111 MATLAB Functions</b>	<b>439</b>
	<b>Glossary</b>	<b>445</b>
	<b>Bibliography</b>	<b>447</b>
	<b>Index</b>	<b>459</b>

# List of Figures

1.1	MATLAB desktop at start of tutorial. . . . .	2
1.2	Basic 2D picture produced by <code>plot</code> . . . . .	8
1.3	Histogram produced by <code>histogram</code> . . . . .	9
1.4	Growth of a random Fibonacci sequence. . . . .	10
1.5	Plot produced by <code>collatz.m</code> . . . . .	13
1.6	Plot produced by <code>collbar.m</code> . . . . .	14
1.7	Mandelbrot set approximation produced by <code>mandel.m</code> . . . . .	15
1.8	Phase plane plot from <code>ode45</code> . . . . .	17
1.9	Removal process for the Sierpinski gasket and Sierpinski gasket approximation from <code>gasket.m</code> . . . . .	17
1.10	Sierpinski gasket approximation from <code>barnsley.m</code> . . . . .	19
1.11	3D picture produced by <code>sweep.m</code> . . . . .	20
2.1	Help browser. . . . .	30
2.2	Workspace browser. . . . .	32
2.3	Array Editor. . . . .	32
7.1	Histogram produced by <code>rouldist</code> . . . . .	85
7.2	MATLAB Editor/Debugger. . . . .	91
8.1	Simple $x$ - $y$ plots: default and nondefault. . . . .	98
8.2	Default color order for lines and markers. . . . .	98
8.3	Color wheel showing how cyan, magenta, and yellow are obtained by combining red, green, and blue. . . . .	100
8.4	Two nondefault $x$ - $y$ plots. . . . .	101
8.5	<code>loglog</code> example. . . . .	102
8.6	<code>plot(fft(eye(17)))</code> with four variations of <code>axis</code> . . . . .	103
8.7	Use of <code>ylim</code> to change automatic $y$ -axis limits. . . . .	104
8.8	Epicycloid example. . . . .	105
8.9	Legendre polynomial example, using <code>legend</code> . . . . .	107
8.10	Legendre polynomial example (revised), using <code>legend</code> . . . . .	108
8.11	Plot with text produced using the MATLAB L <sup>A</sup> T <sub>E</sub> X interpreter. . . . .	110
8.12	Bezier curve and control polygon. . . . .	111
8.13	Example with <code>subplot</code> and <code>fplot</code> . . . . .	112
8.14	Irregular grid of plots produced with <code>subplot</code> . . . . .	113
8.15	3D plot created with <code>plot3</code> . . . . .	115
8.16	Contour plots with <code>fcontour</code> and <code>contour</code> . . . . .	116
8.17	Contour plot labeled using <code>clabel</code> . . . . .	117
8.18	Surface plots with <code>mesh</code> and <code>meshc</code> . . . . .	117
8.19	Surface plots with <code>surf</code> , <code>surfz</code> , and <code>waterfall</code> . . . . .	118

8.20	Surface plot with <code>fsurf</code> . . . . .	119
8.21	3D view of a 2D plot. . . . .	120
8.22	Color maps <code>jet</code> and <code>parula</code> . . . . .	120
8.23	Fractal landscape views. . . . .	123
8.24	<code>surf</code> plot of matrix containing NaNs. . . . .	124
8.25	Riemann surface for $z^{1/3}$ . . . . .	124
8.26	2D bar plots. . . . .	126
8.27	3D bar plots. . . . .	127
8.28	Histograms produced with <code>histogram</code> , for a 1000-by-1 data vector. . . . .	128
8.29	Pie charts. . . . .	129
8.30	Area graphs. . . . .	130
8.31	Three versions of the same plot: original, tuned, and converted to $\LaTeX$ . . . . .	132
8.32	From the 1964 Gatlinburg Conference on Numerical Algebra. . . . .	133
10.1	Sample output from <code>rosy</code> . . . . .	162
10.2	Koch curves created with function <code>koch</code> . . . . .	171
10.3	Koch snowflake created with function <code>koch</code> . . . . .	172
11.1	Least-squares polynomial fit of degree 3 and cubic spline for data from $1/(x + (1 - x)^2)$ . . . . .	177
11.2	Interpolation with <code>pchip</code> and <code>spline</code> . . . . .	179
11.3	Interpolating a sine curve at five points using <code>interp1</code> . . . . .	180
11.4	Interpolation with <code>griddata</code> . . . . .	181
11.5	Plot produced by <code>fplot(@(x)x-tan(x),[-pi,pi]), grid</code> . . . . .	182
12.1	Fresnel spiral. . . . .	192
12.2	Scalar ODE example. . . . .	194
12.3	Vector field for scalar ODE example. . . . .	196
12.4	Pendulum phase plane solutions. . . . .	197
12.5	Rössler system phase space solutions. . . . .	199
12.6	Attractor reconstruction using <code>deval</code> . . . . .	200
12.7	Pursuit example. . . . .	202
12.8	Pursuit example, with capture. . . . .	203
12.9	Chemical reaction solutions with <code>ode45</code> and <code>ode15s</code> . . . . .	206
12.10	Zoom of chemical reaction solution from <code>ode45</code> . . . . .	206
12.11	Stiff ODE example, with Jacobian information supplied. . . . .	210
12.12	DAE solution components from <code>chemakzo</code> in Listing 12.6. . . . .	211
12.13	Water droplet BVP solved by <code>bvp4c</code> . . . . .	215
12.14	Liquid crystal BVP solved by <code>bvp4c</code> . . . . .	217
12.15	Skipping rope eigenvalue BVP solved by <code>bvp4c</code> . . . . .	220
12.16	Predator–prey model with delay and harvesting. . . . .	223
12.17	Neural network DDE. . . . .	225
12.18	Black–Scholes solution with <code>pdepe</code> . . . . .	227
12.19	Reaction–diffusion system solution with <code>pdepe</code> . . . . .	229
15.1	Wathen matrix and its Cholesky factor. . . . .	254
15.2	Wathen matrix and its Cholesky factor with symmetric reverse Cuthill–McKee ordering ( <code>symrcm</code> ). . . . .	254

15.3	Wathen matrix and its Cholesky factor with symmetric minimum-degree ordering ( <code>symamd</code> ). . . . .	254
16.1	<code>profile viewer</code> report for <code>membrane</code> example. . . . .	261
16.2	More from <code>profile viewer</code> report for <code>membrane</code> example. . . . .	262
16.3	<code>profile viewer</code> report for <code>ops</code> example. . . . .	263
16.4	Calculus example in the Live Editor. . . . .	266
16.5	First page of PDF file exported from the live script in Figure 16.4. . . . .	267
17.1	Hierarchical structure of graphics objects (simplified). . . . .	274
17.2	Original plot and plot modified by <code>set</code> commands. . . . .	275
17.3	Straightforward use of <code>subplot</code> . . . . .	277
17.4	Modified version of Figure 17.3 postprocessed by modifying object properties. . . . .	277
17.5	One frame from a movie. . . . .	280
17.6	Animated figure upon completion. . . . .	281
17.7	Plot with default and modified settings. . . . .	283
17.8	Word frequency bar chart created by script <code>wfreq</code> . . . . .	284
17.9	Selected Chebyshev polynomials $T_k(x)$ on $[-1, 1]$ , created by script <code>cheb3plot</code> . . . . .	285
17.10	Example with superimposed axes created by script <code>garden</code> . . . . .	287
17.11	Diagram created by <code>sqrt_ex</code> . . . . .	287
18.1	Hierarchy of fundamental MATLAB data types. . . . .	292
18.2	Histogram of categorical array. . . . .	301
18.3	Plot of interpolated data with <code>datetime</code> vector on the $x$ -axis. . . . .	304
18.4	<code>cellplot(testmat)</code> . . . . .	313
20.1	The integrand in (20.1). . . . .	334
20.2	<code>taylor</code> tool window. . . . .	337
20.3	$\sin x + \arcsin x$ . . . . .	338
20.4	Jacobi polynomials. . . . .	343
21.1	Undirected graph. . . . .	350
21.2	Weighted undirected graph and a minimum spanning tree. . . . .	351
21.3	Random graph from preferential attachment model. . . . .	352
21.4	Shortest path tree for graph in Figure 21.3. . . . .	352
21.5	Directed graph. . . . .	355
21.6	Neuronal network of <i>C. elegans</i> . . . . .	356
21.7	Subnetwork from <i>C. elegans</i> data. . . . .	356
21.8	Visualization of PageRank centrality in <i>C. elegans</i> subnetwork. . . . .	358
22.1	Histograms of days and times of tweets. . . . .	364
22.2	Histogram of pages of index commands. . . . .	367
23.1	Approximate Brownian path. . . . .	373
23.2	Numerical bifurcation diagram. . . . .	376
25.1	Error for <code>integral</code> function, for integral $\int_1^2 0.1/((x - \lambda)^2 + 0.01) dx$ depending on $\lambda$ . . . . .	390

26.1	Output from <code>brach</code> . . . . .	406
26.2	Output from the small-world simulations of <code>small_world</code> . . . . .	407
26.3	Performance profile produced by <code>ode_pp</code> . . . . .	413
26.4	Performance profile for fictitious data in 12-by-4 array <code>A</code> . . . . .	417
26.5	Contours and stationary points of camel function (26.4). . . . .	420
26.6	Members of the genus <i>Matlabius Floribundum</i> produced by <code>lsys</code> . . . . .	421
26.7	Black–Scholes delta picture from <code>bsdelta</code> . . . . .	425
26.8	<code>spy</code> plot of transition matrix from <code>chute</code> . . . . .	428
26.9	Probability of finishing chutes and ladders game in exactly $n$ rolls and at most $n$ rolls. . . . .	428
26.10	Execution time of <code>pythag</code> versus requested accuracy. . . . .	433
26.11	Traveling-wave solutions for Fisher’s equation, from <code>fisher</code> . . . . .	434
26.12	Solution of Fisher’s equation for initial conditions (26.10) in moving coordinate system, from <code>fisher</code> . . . . .	436

# List of Tables

0.1	Selected highlights of MATLAB releases. . . . .	xxiii
2.1	<code>10*exp(1)</code> displayed in several output formats. . . . .	26
2.2	Command line editing keypresses. . . . .	27
2.3	Information and demonstrations. . . . .	27
2.4	MATLAB directory structure (under Windows). . . . .	29
4.1	Arithmetic operator precedence. . . . .	41
4.2	Elementary and special mathematical functions. . . . .	42
4.3	Parameters for single- and double-precision data types. . . . .	43
5.1	Elementary matrices. . . . .	48
5.2	Special matrices. . . . .	52
5.3	Matrices available through <code>gallery</code> . . . . .	53
5.4	Matrices classified by property. . . . .	55
5.5	Elementary matrix and array operations. . . . .	59
5.6	Matrix manipulation functions. . . . .	64
5.7	Basic data analysis functions. . . . .	67
6.1	Selected logical <code>is*</code> functions. . . . .	73
6.2	Logical operators. . . . .	74
6.3	Operator precedence. . . . .	76
8.1	Options for the <code>plot</code> command. . . . .	99
8.2	RGB coordinates for the colors in Table 8.1, as used for setting the <code>color</code> property. . . . .	99
8.3	Default values for some properties. . . . .	101
8.4	Some commands for controlling the axes. . . . .	103
8.5	Some of the <code>T<sub>E</sub>X</code> commands supported in text strings. . . . .	107
8.6	2D plotting functions. . . . .	114
8.7	3D plotting functions. . . . .	123
9.1	Logical <code>is*</code> functions for matrices. . . . .	136
9.2	Some examples of how to set the <code>opts</code> structure in <code>linsolve</code> . . . . .	140
9.3	Iterative linear equation solvers. . . . .	155
11.1	Top ten algorithms based on <i>The Princeton Companion to Applied Mathematics</i> and Dongarra and Sullivan’s list. . . . .	187
12.1	Default values for absolute and relative error tolerances. . . . .	189
12.2	The MATLAB ODE solvers. . . . .	208

18.1	Multidimensional array functions. . . . .	298
18.2	Subset of identifiers supported in the <code>datetime</code> 'Format' and 'InputFormat' specifiers. . . . .	302
20.1	Linear algebra functions in the Symbolic Math Toolbox. . . . .	340
20.2	Special polynomials. . . . .	342
21.1	Selected graph functions. . . . .	353
26.1	Data in transpose of array T from <code>ode_pp</code> . . . . .	413
A.1	Elementary and specialized vectors and matrices. . . . .	439
A.2	Special variables and functions. . . . .	439
A.3	Array information and manipulation. . . . .	440
A.4	Logical operators. . . . .	440
A.5	Flow control. . . . .	440
A.6	Basic data analysis. . . . .	440
A.7	Graphics. . . . .	441
A.8	Linear algebra. . . . .	441
A.9	Functions connected with program files. . . . .	441
A.10	Miscellaneous. . . . .	442
A.11	Data types and conversions. . . . .	442
A.12	Managing the workspace. . . . .	442
A.13	Input and output. . . . .	442
A.14	Numerical methods. . . . .	443



# List of Program Files

1.1	Script <code>rfib.m</code> . . . . .	11
1.2	Script <code>collatz.m</code> . . . . .	11
1.3	Script <code>collbar.m</code> . . . . .	13
1.4	Script <code>mandel.m</code> . . . . .	14
1.5	Function <code>lorenz_de.m</code> . . . . .	15
1.6	Script <code>lorenz_run.m</code> . . . . .	15
1.7	Function <code>gasket.m</code> . . . . .	16
1.8	Script <code>barnsley.m</code> . . . . .	19
1.9	Script <code>sweep.m</code> . . . . .	20
7.1	Script <code>rouldist</code> . . . . .	84
7.2	Function <code>maxentry</code> . . . . .	86
7.3	Function <code>flogist</code> . . . . .	87
7.4	Function <code>cheby</code> . . . . .	87
7.5	Function <code>sqrtn</code> . . . . .	89
7.6	Function <code>marks2</code> . . . . .	90
8.1	Script <code>legendre_plot</code> . . . . .	108
8.2	Function <code>pnorm_plot</code> . . . . .	110
8.3	Function <code>bezier_plot</code> . . . . .	111
8.4	Function <code>land</code> . . . . .	121
10.1	Function <code>fd_deriv</code> . . . . .	160
10.2	Function <code>poly1err</code> containing local function <code>f</code> . . . . .	163
10.3	Function <code>rosy</code> containing local function <code>spiro</code> . . . . .	164
10.4	Script <code>test_solver</code> containing local function <code>test</code> . . . . .	164
10.5	Function <code>companb</code> . . . . .	166
10.6	Function <code>moments</code> . . . . .	167
10.7	Function <code>arg_checks</code> . . . . .	169
10.8	Function <code>rational_ex</code> containing a nested function <code>r</code> . . . . .	170
10.9	Function <code>koch</code> . . . . .	172
12.1	Function <code>rossler_ex</code> . . . . .	198
12.2	Function <code>rossler_attract2</code> . . . . .	200
12.3	Function <code>fox1</code> . . . . .	202
12.4	Function <code>fox_rabbit</code> . . . . .	204
12.5	Function <code>rcd</code> . . . . .	209
12.6	Function <code>chemakzo</code> . . . . .	212
12.7	Function <code>lcrun</code> . . . . .	218
12.8	Function <code>skiprun</code> . . . . .	219
12.9	Function <code>harvest</code> . . . . .	222

12.10	Script <code>neural</code> .	224
12.11	Function <code>bs</code> .	228
12.12	Function <code>mbiol</code> .	230
13.1	Script <code>print_matrix</code> .	239
14.1	Script <code>fib</code> that generates a runtime error.	242
16.1	Script <code>badfun</code> .	260
16.2	Script <code>ops</code> .	263
16.3	Script <code>calculus.m</code> .	265
16.4	Script <code>test_acos</code> .	271
17.1	Script <code>wfreq</code> .	284
17.2	Script <code>cheb3plot</code> .	285
17.3	Script <code>garden</code> .	286
17.4	Script <code>sqrt_ex</code> .	288
19.1	Code file <code>maxplus</code> ; version for scalars.	317
19.2	Code file <code>maxplus</code> ; version for matrices.	319
19.3	Code file <code>circulant</code> .	323
23.1	Script <code>bif1</code> .	376
23.2	Script <code>bif2</code> .	377
25.1	Function <code>specrad_randn</code> .	392
25.2	Function <code>parfeval_specrad</code> .	394
26.1	Function <code>brach</code> .	405
26.2	Script <code>small_world</code> .	408
26.3	Function <code>perfprof</code> .	411
26.4	Function <code>ode_pp</code> .	414
26.5	Script <code>camel_solve</code> .	418
26.6	Function <code>lsys</code> .	423
26.7	Script <code>lsys_run</code> .	424
26.8	Script <code>bsdelta</code> .	426
26.9	Script <code>chute</code> .	429
26.10	Function <code>pythag</code> .	431
26.11	Script <code>fisher</code> .	435

# Preface

MATLAB<sup>®</sup><sup>1</sup> is an interactive system for numerical computation. Numerical analyst Cleve Moler wrote the initial Fortran version of MATLAB in the late 1970s as a teaching aid. It became popular for both teaching and research and evolved into a commercial software package written in C. For many years now, MATLAB has been widely used in universities and industry.

MATLAB has several advantages over more traditional means of numerical computing (e.g., writing Fortran or C programs and calling numerical libraries).

- It allows quick and easy coding in a very high-level language.
- Data structures require minimal attention; in particular, arrays need not be declared before first use.
- An interactive interface allows rapid experimentation and easy debugging.
- High-quality graphics and visualization facilities are available.
- MATLAB programs are completely portable across a wide range of platforms.
- Toolboxes can be added to extend the system, giving, for example, specialized signal processing facilities and a symbolic manipulation capability.
- A wide range of user-contributed MATLAB programs is freely available on the Internet.

Furthermore, MATLAB is a modern programming language and problem-solving environment: it has sophisticated data structures, contains built-in editing and debugging tools, and supports object-oriented programming. These factors make MATLAB an excellent language for teaching and a powerful tool for research and practical problem-solving. Being interpreted, MATLAB inevitably suffers some loss of efficiency compared with compiled languages, but built-in performance acceleration techniques (including some runtime compilation) reduce the inefficiencies and users have the possibility of calling code and libraries written in other languages.

This book has two purposes. First, it aims to give a lively introduction to the most popular features of MATLAB, covering all that most users will ever need to know. We assume no prior knowledge of MATLAB, but the reader is expected to be familiar with the basics of programming and with the use of the operating system under which MATLAB is being run. We describe how and why to use MATLAB functions but do not explain the mathematical theory and algorithms underlying them; instead, references are given to the appropriate literature.

The second purpose of the book is to provide a compact reference to MATLAB. The scope of MATLAB has grown dramatically as the package has been developed

---

<sup>1</sup>MATLAB is a registered trademark of The MathWorks, Inc.

(see Table 0.1), and even experienced MATLAB users may be unaware of some of the functionality of the latest versions. Indeed, the PDF documentation for MATLAB runs to well over ten thousand pages. Hence we believe that there is a need for a manual that is wide-ranging yet concise. We hope that our approach of focusing on the most important features of MATLAB, combined with the book’s logical organization and detailed index, will make *MATLAB Guide* a useful reference.

The book is intended to be used by students, researchers, and practitioners alike. Our philosophy is to teach by giving informative examples rather than to treat every function comprehensively. Full documentation is available in the MATLAB help system, which can be accessed from the Home tab of the MATLAB Toolstrip, by typing `doc` in the Command Window, or on the website of The Mathworks. The contents of the help system are also available as PDF files, accessible via “PDF Documentation” links in the help system. When we refer to “the help system” we mean any one of these sources.

Our treatment includes many “hidden” or easily overlooked features of MATLAB and we provide a wealth of useful tips, covering such topics as customizing graphics, coding style, code optimization, and debugging. However, we discuss only officially documented MATLAB features (undocumented features can change without warning and cannot be relied on).

The main subjects omitted are Graphical User Interface (GUI) tools, which can be useful as front-ends to MATLAB computations, and MEX files.

We have not included exercises; MATLAB is often taught in conjunction with particular subjects, and exercises are best tailored to the context.

We have been careful to show complete, undoctored MATLAB output and to test every piece of MATLAB code listed. The only editing we have done has been to omit some lines of output (to save space) and replace them by a line consisting of “...”.

MATLAB runs on several operating systems and we concentrate on features common to all. We do not describe how to install or run MATLAB, or how to customize it—the documentation should be consulted for this system-specific information.

A web page for the book can be found at <http://www.siam.org/books/ot150>. It includes links to all the codes used as examples in the book, errata, and links to various MATLAB-related web resources. It also includes the bibliography of the book as a BibTeX bib file and in PDF form with embedded links.

## What This Book Describes

This book describes MATLAB 9.1 (Release 2016b). If you are not sure which version of MATLAB you are using, type `ver` or `version` at the MATLAB prompt.

## How This Book Is Organized

The book begins with a tutorial that provides a quick tour of MATLAB. The rest of the book is independent of the tutorial, so the tutorial can be skipped—for example, by readers already familiar with MATLAB.

The chapters are ordered so as to introduce topics in a logical fashion, with the minimum of forward references. A principal aim was to cover MATLAB programs and graphics as early as possible, subject to being able to provide meaningful examples. Later chapters contain material that is more advanced or less likely to be needed by the beginner.

Table 0.1. *Selected highlights of MATLAB releases.*

Year	Version	Notable features
1978	Classic	Original Fortran version.
1984	1	Rewritten in C.
1985	2	30% more commands and functions, typeset documentation.
1987	3	Faster interpreter, color graphics, high-resolution graphics printing.
1992	4	Sparse matrices, animation, visualization, user interface controls, debugger, Handle Graphics <sup>®</sup> ,* Microsoft Windows support.
1997	5	Profiler, object-oriented programming, multidimensional arrays, cell arrays, structures, more sparse linear algebra, new ordinary differential equation solvers, browser-based help.
2000	6, R12	MATLAB desktop including Help browser, matrix computations based on LAPACK with optimized BLAS, function handles, <code>eigs</code> interface to ARPACK, boundary-value problem and partial differential equation solvers, graphics object transparency, Java support.
2002	6.5, R13	Performance acceleration, more control in warning and error handling.
2004	7.0, R14	Mathematics on nondouble operands (single precision, integer), anonymous functions, nested functions, publishing an M-file to HTML, L <sup>A</sup> T <sub>E</sub> X, etc., enhanced plot annotation.
2008	7.6, R2008a	Enhanced object-oriented programming capabilities.
2008	7.7, R2008b	Upgraded random number generators.
2012	8.0, R2012b	Redesigned desktop with Toolstrip, new help system.
2013	8.2, R2013b	<code>table</code> data type and <code>categorical</code> arrays.
2014	8.4, R2014b	Source control, updated graphics system, <code>datetime</code> arrays, <code>datastore</code> .
2015	8.6, R2015b	New execution engine, <code>graph</code> and <code>digraph</code> classes
2016	9.0, R2016a	Live Editor, performance testing framework
2016	9.1, R2016b	Local functions in scripts, string arrays, tall arrays, implicit expansion of arrays with dimensions of length 1.

\* Handle Graphics is a registered trademark of The MathWorks, Inc.

## Using the Book

Readers new to MATLAB should begin by working through the tutorial in Chapter 1. The tutorial gives a fast-paced overview of the capabilities of MATLAB, with all its topics being covered in greater detail in subsequent chapters. Although it is designed to be read sequentially, with most chapters building on material from earlier ones, the book can be read in a nonsequential fashion by following cross-references and making use of the index. It is difficult to do serious MATLAB computation without a knowledge of arithmetic, matrices, colon notation, operators, flow control, and program files, so Chapters 4–7 contain information essential for all users.

Appendix A lists our choice of the top 111 MATLAB functions—those that we think every MATLAB user should know about. The beginner may like to tick off these functions as they are learned, while intermediate users can pick out for study those functions with which they are not already familiar.

From time to time we make reference to the extensive MATLAB documentation. Reference information for a particular function, `fun`, can be obtained by typing `help fun` or `doc fun`, but sometimes we need to refer to a page in HTML documentation that is not directly accessible with a `doc` command. In this case we point to the precise page in question by specifying a command such as

```
web([docroot '/matlab/numeric-types.html'])
```

Here, `docroot` refers to the location of the documentation on the system in question, so this command should work on any MATLAB installation. Note that the quote symbol, `'`, which displays in this way in MATLAB and is typeset this way in all the MATLAB code in this book, is typed as the right or closing quote, `'`, on the keyboard.

## What’s New in the Third Edition

This third edition of the book is 25 percent longer than the second edition (2005) and differs from it in several respects.

1. Many changes and new features introduced in MATLAB are incorporated.
2. All figures are now in color (they were monochrome in the second edition).
3. New “Asides”, highlighted in gray boxes, contain discussions on MATLAB-related topics, such as anonymous functions, reproducibility, and color maps.
4. Our continuing experience in using MATLAB for teaching and research has led to numerous improvements and additions—in particular, more examples.
5. The “Advanced Graphics” chapter (Chapter 17) (previously title “Handle Graphics”) has been rewritten to reflect the major update to the graphics system introduced in MATLAB 2014b.
6. A new chapter “Object-Oriented Programming” (Chapter 19) presents an introduction to object-oriented programming in MATLAB through two examples of classes.
7. The chapter “The Symbolic Math Toolbox” (Chapter 20) has been revised to reflect the use of MuPAD as the underlying symbolic engine (previously the engine was Maple), and the chapter has been extended.

8. A new chapter “Graphs” (Chapter 21) describes the new MATLAB classes `graph` and `digraph`, for representing and manipulating undirected graphs and directed graphs.
9. A new chapter “Large Data Sets” (Chapter 22) describes MATLAB features for handling data sets so large that they do not fit into the memory of the computer.
10. A new chapter “The Parallel Computing Toolbox” (Chapter 25) describes this toolbox, which exploits multicore processors, clusters, and graphics processing units (GPUs).
11. New sections have been added, including “Empty Matrices” (Section 5.4), “Matrix Properties” (Section 9.1), “Argument Checking and Parsing” (Section 10.6), “Fine Tuning the Display of Arrays” (Section 13.4), “Live Editor” (Section 16.7), “Unit Tests” (Section 16.10), “String Arrays” (Section 18.2), “Categorical Arrays” (Section 18.4), “Tables and Timetables” (Section 18.6), and “Timing Code” (Section 23.1), and many existing sections contain new or reorganized material.
12. Changes in MATLAB terminology have been incorporated. For example, the terms “program files” and “local functions” replace what were previously called “M-files” and “subfunctions”.

## Future Versions of MATLAB

MATLAB will continue to evolve. New versions are released twice a year, denoted “R20xya” and “R20xyb”. It is a good habit to inspect the release notes of each new version in order to see what has changed. They can be found by typing `doc` then selecting MATLAB or one of its installed toolboxes and following the link. The release notes also give advanced notice of changes planned for the future, listed under “Functionality being removed or changed”, enabling you to avoid using functions or syntax that will become obsolete.

## Acknowledgments

We are grateful to a number of people who offered helpful advice and comments during the preparation of the book.

For the third edition: Penny Anderson, Bobby Cheng, Mike Croucher, Massimiliano Fasi, Heather Gorr, Stefan Güttel, Nick Hale, Richard Lang, Jasmina Lazic, Steve Lord, Jos Martin, Umberto Noe, Sarah Palfreyman, Sam Relton, Ben Tordoff.

For the second edition: Penny Anderson, Paolo Bientinesi, David Carlisle, Jacek Kierzenka, Cleve Moler, Jorge Moré, Jim Nagy, Larry Shampine.

For the first edition: Penny Anderson, Christian Beardah, Tom Bryan, Brian Duffy, Cleve Moler, Damian Packer, Harikrishna Patel, Larry Shampine, Françoise Tisseur, Nick Trefethen, Jack Williams.

It has been a pleasure working with Elizabeth Greenspan, Gina Rinelli Harris, David Riegelhaupt, and Lois Sellers (SIAM), and Sam Clark (T&T Productions Ltd, London), for the third edition; Beth Gallagher, Sara Murphy, Linda Thiel, and Kelly Thomas for the second edition; and Beth Gallagher, Vickie Kearn, Michelle Montgomery, Deborah Poulson, Lois Sellers, Kelly Thomas, and Marianne Will for the first edition.

*For those of you that have not experienced MATLAB,  
we would like to try to show you what everybody is excited about....  
The best way to appreciate PC-MATLAB is, of course, to try it yourself.*

— JOHN LITTLE and CLEVE B. MOLER, *A Preview of PC-MATLAB* (1985)

*In teaching, writing and research,  
there is no greater clarifier than  
a well-chosen example.*

— CHARLES F. VAN LOAN, *Using Examples to Build Computational Intuition* (1995)

*A new era in scientific computing  
has been ushered in by the development of MATLAB.*

— LLOYD N. TREFETHEN, *Spectral Methods in MATLAB* (2000)



# Chapter 1

## A Brief Tutorial

The best way to learn MATLAB is by trying it yourself, and hence we begin with a whirlwind tour. Working through the examples below will give you a feel for the way that MATLAB operates and an appreciation of its power and flexibility.

The tutorial is entirely independent of the rest of the book—all the MATLAB features introduced are discussed in greater detail in the subsequent chapters. Indeed, in order to keep this chapter brief, we have not explained all the functions used here. You can use the index to find out more about particular topics that interest you.

The tutorial contains commands for you to type at the command line. In the last part of the tutorial we give examples of script and function files—the MATLAB versions of programs and functions, subroutines, or procedures in other languages. These files are short, so you can type them in quickly. Alternatively, you can download them from the website mentioned in the preface on p. xxii. You should experiment as you proceed, keeping the following points in mind.

- Uppercase and lowercase characters are not equivalent (MATLAB is case sensitive).
- Typing the name of a variable will cause MATLAB to display its current value.
- A semicolon at the end of a command suppresses the display of any output that would otherwise be produced in the Command Window.
- MATLAB uses parentheses, `()`, square brackets, `[]`, and curly braces, `{}`, and these are not interchangeable.
- The up arrow and down arrow keys can be used to scroll through your previous commands. Also, an old command can be recalled by typing the first few characters followed by up arrow.
- You can type `help topic` to access online help on the command, function, or symbol `topic`. Note that hyperlinks, indicated by underlines, are provided that will take you to related help items and the more complete documentation in the Help browser. Type `doc topic` to go directly to the Help browser.
- If you press the `tab` key after partially typing a function or variable name, MATLAB will attempt to complete it, offering you a selection of choices if there is more than one possible completion.
- You can quit MATLAB by typing `exit` or `quit`.

Having entered MATLAB, you should work through this tutorial by typing in the text that appears after the MATLAB prompt, `>>`, in the Command Window. After showing you what to type, we display the output that is produced. We begin with

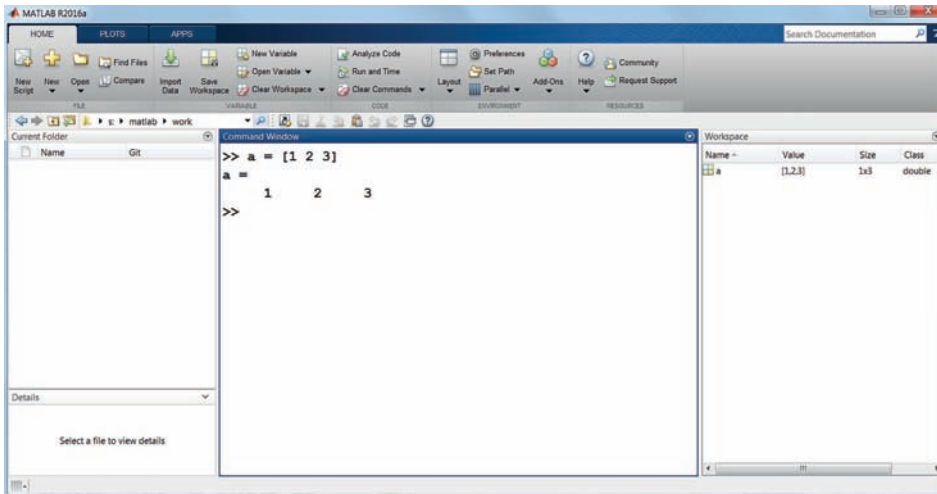


Figure 1.1. *MATLAB desktop at start of tutorial.*

```
>> a = [1 2 3]

a =

     1     2     3
```

This means that you are to type “`a = [1 2 3]`”, after which you will see the output “`a =`” and “`1 2 3`” on separate lines separated by a blank line, as shown in Figure 1.1. (To save space we will subsequently omit blank lines in MATLAB output. You can tell MATLAB to suppress blank lines by typing `format compact`.) This example sets up a 1-by-3 array `a` (a row vector). In the next example, semicolons separate the entries:

```
>> c = [4; 5; 6]
c =

     4
     5
     6
```

A semicolon tells MATLAB to start a new row, so `c` is 3-by-1 (a column vector). Now you can multiply the arrays `a` and `c`:

```
>> a*c
ans =

    32
```

Here, you performed an inner product: a 1-by-3 array multiplied into a 3-by-1 array. MATLAB automatically assigned the result to the variable `ans`, which is short for answer. An alternative way to compute an inner product is with the `dot` function:

```
>> dot(a,c)
```

```
ans =
    32
```

Inputs to MATLAB functions are specified after the function name and within parentheses. You may also form the outer product:

```
>> A = c*a
A =
     4     8    12
     5    10    15
     6    12    18
```

Here, the answer is a 3-by-3 matrix that has been assigned to `A`.

The product `a*a` is not defined, since the dimensions are incompatible for matrix multiplication:

```
>> a*a
Error using *
Inner matrix dimensions must agree.
```

Arithmetic operations on matrices and vectors come in two distinct forms. Matrix-sense operations are based on the normal rules of linear algebra and are obtained with the usual symbols `+`, `-`, `*`, `/`, and `^`. Array-sense operations are defined to act elementwise and are generally obtained by preceding the symbol with a dot. Thus if you want to square each element of `a` you can write

```
>> b = a.^2
b =
     1     4     9
```

Since the new vector `b` is 1-by-3, like `a`, you can form the array product of it with `a`:

```
>> a.*b
ans =
     1     8    27
```

MATLAB has many mathematical functions that operate in the array sense when given a vector or matrix argument. For example,

```
>> exp(a)
ans =
    2.7183    7.3891   20.0855

>> log(ans)
ans =
     1     2     3

>> sqrt(a)
ans =
    1.0000    1.4142    1.7321
```

MATLAB displays floating-point numbers to 5 decimal digits, by default, but always stores numbers and computes with them to the equivalent of 16 decimal digits. The output format can be changed using the `format` command:

```
>> format long

>> sqrt(a)
ans =
    1.0000000000000000    1.414213562373095    1.732050807568877

>> format
```

The last command reinstates the default output format of 5 digits (and loose line spacing, which we always suppress in this book, as noted above). Large or small numbers are displayed in exponential notation, with a power of 10 scale factor preceded by **e**:

```
>> 2^(-24)
ans =
    5.9605e-08
```

Various data analysis functions are also available:

```
>> sum(b), mean(c)
ans =
    14
ans =
     5
```

As this example shows, you may include more than one command on the same line by separating them with commas. If a command is followed by a semicolon then MATLAB suppresses the output:

```
>> pi
ans =
    3.1416

>> y = tan(pi/6);
```

The function `pi`, which is built into MATLAB, returns the value  $\pi$ . The variable `ans` always contains the most recent unassigned expression, so after the assignment to `y`, `ans` still holds the value  $\pi$ .

You may set up a two-dimensional array by using spaces to separate entries within a row and semicolons to separate rows:

```
>> B = [-3 0 -1; 2 5 -7; -1 4 8]
B =
    -3     0    -1
     2     5    -7
    -1     4     8
```

At the heart of MATLAB is a powerful range of linear algebra functions. For example, recalling that `c` is a 3-by-1 vector, you may wish to solve the linear system  $\mathbf{B}\mathbf{x} = \mathbf{c}$ . This can be done with the backslash operator:

```
>> x = B\c
x =
```

```

-1.2995
 1.3779
-0.1014

```

You can check the result by computing the relative residual in the Euclidean norm:

```

>> norm(B*x-c)/(norm(B)*norm(x))
ans =
 9.6513e-17

```

While nonzero because of rounding errors in the computations, this residual is about as small as we can expect, given that MATLAB computes to the equivalent of about 16 decimal digits.

The eigenvalues of  $B$  can be found using `eig`:

```

>> e = eig(B)
e =
-3.1361
 6.5680 + 5.1045i
 6.5680 - 5.1045i

```

Here,  $i$  is the imaginary unit,  $\sqrt{-1}$ . You may also specify two output arguments for the function `eig`:

```

>> [V,D] = eig(B,'nobalance')
V =
 0.9829 + 0.0000i  -0.0385 - 0.0393i  -0.0385 + 0.0393i
-0.1266 + 0.0000i  -0.8005 + 0.0000i  -0.8005 + 0.0000i
 0.1337 + 0.0000i   0.1683 + 0.5725i   0.1683 - 0.5725i
D =
-3.1361 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
 0.0000 + 0.0000i   6.5680 + 5.1045i   0.0000 + 0.0000i
 0.0000 + 0.0000i   0.0000 + 0.0000i   6.5680 - 5.1045i

```

In this case the columns of  $V$  are eigenvectors of  $B$  and the diagonal elements of  $D$  are the corresponding eigenvalues. Here, we gave `eig` the optional input argument `'nobalance'`, which disables the default balancing that attempts to improve the scaling of the matrix. Single (closing) quotes act as string delimiters, so `'nobalance'` is a string. Many MATLAB functions take string arguments.

The colon notation is useful for constructing vectors of equally spaced values. For example,

```

>> v = 1:6
v =
 1     2     3     4     5     6

```

Generally, `m:n` generates the vector with entries  $m, m+1, \dots, n$ . Nonunit increments can be specified with `m:s:n`, which generates entries that start at  $m$  and increase (or decrease) in steps of  $s$  as far as  $n$ :

```

>> w = 2:3:10, y = 1:-0.25:0
w =
 2     5     8
y =
 1.0000    0.7500    0.5000    0.2500    0

```

You may construct big matrices out of smaller ones by following the conventions that (a) square brackets enclose an array, (b) spaces or commas separate entries in a row, and (c) semicolons separate rows:

```
>> C = [A, [8;9;10]], D = [B;a]
C =
     4     8    12     8
     5    10    15     9
     6    12    18    10

D =
    -3     0    -1
     2     5    -7
    -1     4     8
     1     2     3
```

The element in row  $i$  and column  $j$  of the matrix  $C$  (where  $i$  and  $j$  always start at 1) can be accessed as  $C(i,j)$ :

```
>> C(2,3)
ans =
    15
```

More generally,  $C(i1:i2, j1:j2)$  picks out the submatrix formed by the intersection of rows  $i1$  to  $i2$  and columns  $j1$  to  $j2$ :

```
>> C(2:3,1:2)
ans =
     5    10
     6    12
```

You can build certain types of matrix automatically. For example, identities and matrices of zeros and ones can be constructed with `eye`, `zeros`, and `ones`:

```
>> I3 = eye(3,3), Y = zeros(3,5), Z = ones(2)
I3 =
     1     0     0
     0     1     0
     0     0     1

Y =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0

Z =
     1     1
     1     1
```

Note that for these functions the first argument specifies the number of rows and the second the number of columns; if both numbers are the same then only one need be given. The functions `rand` and `randn` work in a similar way, generating random entries from the uniform distribution over  $[0, 1]$  and the normal  $(0, 1)$  distribution, respectively. The numbers generated depend on the state of the random number generator. By seeding the generator you can make your experiments repeatable. Here, the seed is set to 20:

```
>> rng(20)
>> F = rand(3), g = randn(1,5)
F =
    0.5881    0.8158    0.3787
    0.8977    0.0359    0.5185
    0.8915    0.6918    0.6580
g =
   -1.3543   -0.9625    0.8736    0.8499    1.6579
```

#### BEFORE THE DAYS OF `rand`

In 1955, the appropriately named RAND Corporation published the book *A Million Random Digits with 100,000 Normal Deviates* [144] as a resource for researchers conducting large-scale randomized experiments. The book, also available in punched card format, listed “random” numbers generated via an electric roulette wheel. The book was reissued in 2001, attracting a range of amusing online reviews.

By this point several variables have been created in the workspace. You can obtain a list with the `who` command:

```
>> who
```

Your variables are:

```
A    C    F    V    Z    ans  c    g    w    y
B    D    I3  Y    a    b    e    v    x
```

You can obtain a more detailed list showing the size and class of each variable by typing `whos`. Alternatively, look at the Workspace browser, which is displayed by default in the MATLAB desktop (see Figure 1.1).

Like most programming languages, MATLAB has loop constructs. The following example uses a `for` loop to evaluate the continued fraction

$$1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + 1}}}}}}}}}}}}}}}}},$$

which approximates the golden ratio,  $(1 + \sqrt{5})/2$ . The evaluation is done from the bottom up:

```
>> r = 2;
```

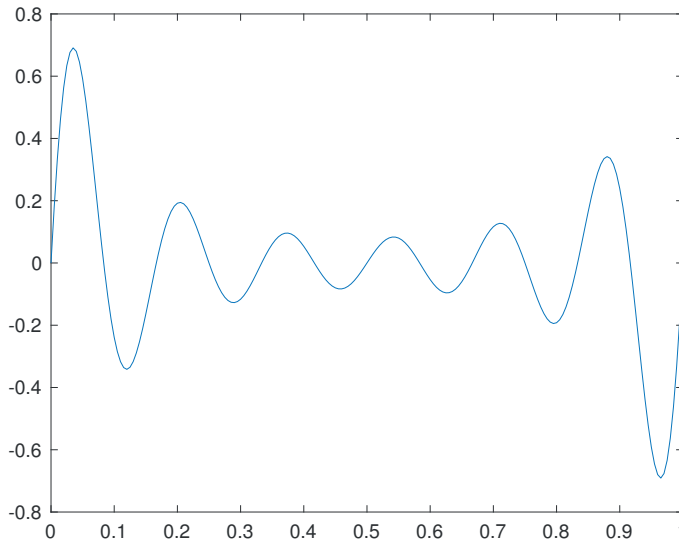


Figure 1.2. *Basic 2D picture produced by plot.*

```
>> for k = 1:10, r = 1 + 1/r; end
>> r
r =
    1.6181
```

Loops involving `while` can be found later in this tutorial.

The `plot` function produces two-dimensional (2D) pictures:

```
>> t = 0:0.005:1; z = exp(10*t.*(t-1)).*sin(12*pi*t);
>> plot(t,z)
```

Here, `plot(t,z)` joins the points  $t(i), z(i)$  using the default solid linetype. MATLAB opens a figure window in which the picture is displayed. Figure 1.2 shows the result. You can close a figure window by typing `close` at the command line.

You can produce a histogram with the function `histogram`:

```
>> histogram(randn(1000,1))
```

Here, `histogram` is given 1000 points from the normal (0,1) random number generator. The result is shown in Figure 1.3.

You are now ready for more challenging computations. A random Fibonacci sequence  $\{x_n\}$  is generated by choosing  $x_1$  and  $x_2$  and setting

$$x_{n+1} = x_n \pm x_{n-1}, \quad n \geq 2.$$

Here, the  $\pm$  indicates that  $+$  and  $-$  must have equal probability of being chosen. Viswanath [175] analyzed this recurrence and showed that, with probability 1, for large  $n$  the quantity  $|x_n|$  increases like a multiple of  $c^n$ , where  $c = 1.13198824\dots$  (see also [45]). You can test Viswanath's result as follows:

```
>> clear
```



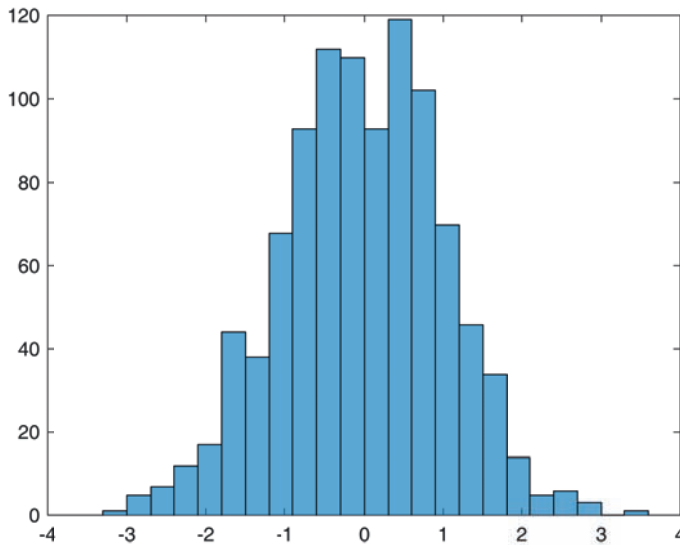


Figure 1.3. Histogram produced by `histogram`.

```
>> rng(100)
>> x = [1 2];
>> for n = 2:999, x(n+1) = x(n) + sign(rand-0.5)*x(n-1); end
>> semilogy(1:1000,abs(x))
>> c = 1.13198824;
>> hold on
>> semilogy(1:1000,c.^[1:1000])
>> hold off
```

Here, `clear` removes all variables from the workspace. The `for` loop stores a random Fibonacci sequence in the array `x`; MATLAB automatically extends `x` each time a new element `x(n+1)` is assigned. The `semilogy` function then plots `n` on the  $x$ -axis against `abs(x)` on the  $y$ -axis, with logarithmic scaling for the  $y$ -axis. Typing `hold on` tells MATLAB to superimpose the next picture on top of the current one. The second `semilogy` plot produces a line of slope `c`. The overall picture, shown in Figure 1.4, is consistent with Viswanath’s theory.

The MATLAB commands used to generate Figure 1.4 stretched over several lines. This is inconvenient for a number of reasons, not least because if a change is made to the experiment then it is necessary to reenter all the commands. To avoid this difficulty you can employ a script. Create a file named `rfib.m` identical to Listing 1.1 in your current directory.<sup>2</sup> You can call up the MATLAB Editor/Debugger from the home tab of the Toolstrip or by typing `edit` in the Command Window; `pwd` displays the current directory and `ls` or `dir` lists its contents. Now type

```
>> rfib
```

at the command line. This will reproduce the picture in Figure 1.4. Running `rfib` in this way is essentially the same as typing the commands in the file at the command

<sup>2</sup>“Directory” is a synonym for “folder”. We use the former term throughout this book.

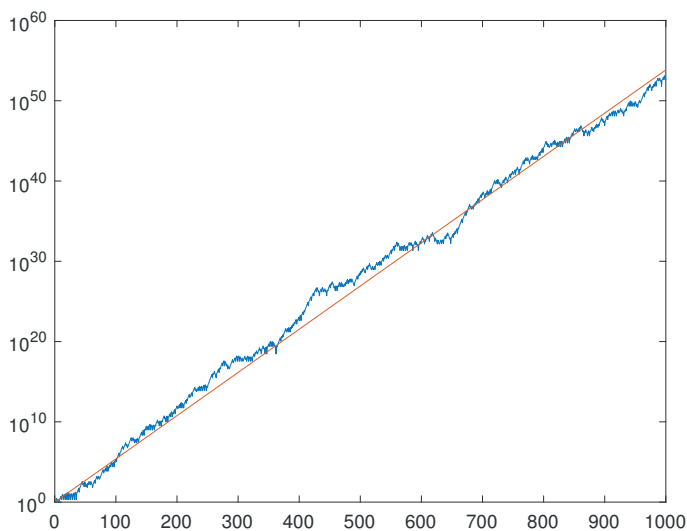


Figure 1.4. *Growth of a random Fibonacci sequence.*

line, in sequence. Note that in Listing 1.1 blank lines and indentation are used to improve readability, and we have made the number of iterations a variable, `m`, so that it can be more easily changed. The script also contains helpful comments—all text on a line after the `%` character is ignored by MATLAB. Having set up these commands in a script you are now free to experiment further. For example, changing `rng(100)` to `rng(101)` generates a different random Fibonacci sequence, and adding the line `title('Random Fibonacci Sequence')` at the end of the file will put a title on the graph.

Our next example involves the Collatz iteration, which, given a positive integer  $x_1$ , has the form  $x_{k+1} = f(x_k)$ , where

$$f(x) = \begin{cases} 3x + 1, & \text{if } x \text{ is odd,} \\ x/2, & \text{if } x \text{ is even.} \end{cases}$$

In words: if  $x$  is odd, replace it by  $3x + 1$ , and if  $x$  is even, halve it. It has been conjectured that this iteration will always lead to a value of 1 (and hence thereafter cycle between 4, 2, and 1) whatever starting value  $x_1$  is chosen. There is ample computational evidence to support this conjecture, which is variously known as the Collatz problem, the  $3x + 1$  problem, the Syracuse problem, Kakutani’s problem, Hasse’s algorithm, and Ulam’s problem. However, a rigorous proof has so far eluded mathematicians. For further details, see [111] or type “Collatz problem” into your favorite search engine. You can investigate the conjecture by creating the script `collatz.m` shown in Listing 1.2. In this file a `while` loop and an `if` statement are used to implement the iteration. The `input` command prompts you for a starting value. The appropriate response is to type an integer and then hit return or enter:

```
>> collatz
Enter an integer bigger than 2: 27
```

Here, the starting value 27 has been entered. The iteration terminates and the resulting picture is shown in Figure 1.5.

Listing 1.1. *Script rfib.m.*

```

%RFIB   Random Fibonacci sequence.

rng(100)           % Set random number state.
m = 1000;          % Number of iterations.

x = [1 2];         % Initial conditions.
for n = 2:m-1      % Main loop.
    x(n+1) = x(n) + sign(rand-0.5)*x(n-1);
end

semilogy(1:m,abs(x))
c = 1.13198824;    % Viswanath's constant.
hold on
semilogy(1:m,c.^(1:m))
hold off

```

Listing 1.2. *Script collatz.m.*

```

%COLLATZ   Collatz iteration.

n = input('Enter an integer bigger than 2: ');
narray = n;

count = 1;
while n ~= 1
    if rem(n,2) == 1 % Remainder modulo 2.
        n = 3*n+1;
    else
        n = n/2;
    end
    count = count + 1;
    narray(count) = n; % Store the current iterate.
end

plot(narray,'*-') % Plot with * marker and solid line style.
title(['Collatz iteration starting at ' int2str(narray(1))])

```

To investigate the Collatz problem further, the script `collbar` in Listing 1.3 plots a bar graph of the number of iterations required to reach the value 1, for starting values  $1, 2, \dots, 29$ . The result is shown in Figure 1.6. For this picture, the function `grid` adds grid lines that extend from the axis tick marks, and `xlabel` and `ylabel` add labels to the  $x$ - and  $y$ -axes.

The well-known and much-studied Mandelbrot set can be approximated graphically in just a few lines of MATLAB. It is defined as the set of points  $c$  in the complex plane for which the sequence generated by the map  $z \mapsto z^2 + c$ , starting with  $z = c$ , remains bounded [140, Chap. 14]. The script `mandel` in Listing 1.4 produces the plot of the Mandelbrot set shown in Figure 1.7. The script contains calls to `linspace` of the form `linspace(a,b,n)`, which generate an equally spaced vector of  $n$  values between  $a$  and  $b$ . The `meshgrid` and `complex` functions are used to construct a matrix  $C$  that represents the rectangular region of interest in the complex plane. The `waitbar` function plots a bar showing the progress of the computation (the variable  $h$  is a “handle” to the wait bar). The plot itself is produced by `contourf`, which plots a filled contour. The expression `abs(Z)<Z_max` in the call to `contourf` detects points that have not exceeded the threshold  $Z_{\max}$  and that are therefore assumed to lie in the Mandelbrot set; the `double` function is applied in order to convert the resulting logical array to numeric form. You can experiment with `mandel` by changing the region that is plotted, via the `linspace` calls, the number of iterations `it_max`, and the threshold  $Z_{\max}$ . Note that `mandel` changes the color map so that the image is displayed in gray and white instead of the default blue and yellow; this makes it much easier to see the boundary of the set. The default color map can be restored with `colormap('default')`.

Next we solve the ordinary differential equation (ODE) system

$$\begin{aligned}\frac{d}{dt}y_1(t) &= 10(y_2(t) - y_1(t)), \\ \frac{d}{dt}y_2(t) &= 28y_1(t) - y_2(t) - y_1(t)y_3(t), \\ \frac{d}{dt}y_3(t) &= y_1(t)y_2(t) - 8y_3(t)/3.\end{aligned}$$

This is an example from the Lorenz equations family (see [161]). We take initial conditions  $y(0) = [0, 1, 0]^T$  and solve over  $0 \leq t \leq 50$ . The program file `lorenz_de` in Listing 1.5 is an example of a MATLAB function. Given  $t$  and  $y$ , this function returns the right-hand side of the ODE as the vector `yprime`. This is the form required by the MATLAB ODE solving functions. The script `lorenz_run` in Listing 1.6 uses the MATLAB function `ode45` to solve the ODE numerically and then produces the  $(y_1, y_3)$  phase plane plot shown in Figure 1.8. You can see an animated plot of the solution by typing `lorenz`, which calls one of the MATLAB demonstrations (type `demos` for the complete list).

Now we give an example of a recursive function, that is, a function that calls itself. The Sierpinski gasket [139, Sec. 2.2] is based on the following process. Given a triangle with vertices  $P_a$ ,  $P_b$ , and  $P_c$ , we remove the triangle with vertices at the midpoints of the edges,  $(P_a + P_b)/2$ ,  $(P_b + P_c)/2$ , and  $(P_c + P_a)/2$ . This removes the “middle quarter” of the triangle, as illustrated in Figure 1.9(a). Effectively, we have replaced the original triangle with three “subtriangles”. We can now apply the middle quarter removal process to each of these subtriangles to generate nine subsubtriangles, and so on. The Sierpinski gasket is the set of all points that are never removed by repeated application of this process. The function `gasket` in Listing 1.7 implements

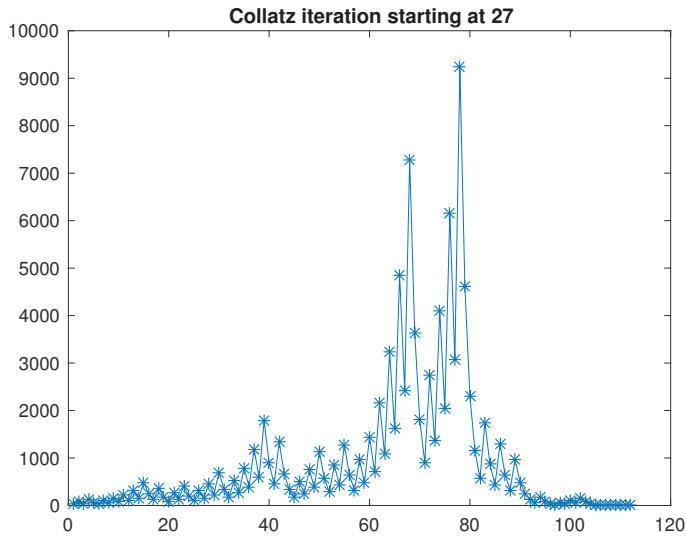


Figure 1.5. Plot produced by collatz.m.

Listing 1.3. Script collbar.m.

```

%COLLBAR    Collatz iteration bar graph.

N = 29;          % Use starting values 1,2,...,N.
niter = zeros(N,1); % Preallocate array.
for i = 1:N
    count = 0;
    n = i;
    while n ~= 1
        if rem(n,2) == 1
            n = 3*n+1;
        else
            n = n/2;
        end
        count = count + 1;
    end
    niter(i) = count;
end
bar(niter) % Bar graph.
grid % Add horizontal and vertical grid lines.
title('Collatz iteration counts')
xlabel('Starting value','FontSize',12) % Label x-axis.
ylabel('Number of iterations','FontSize',12) % Label y-axis.

```

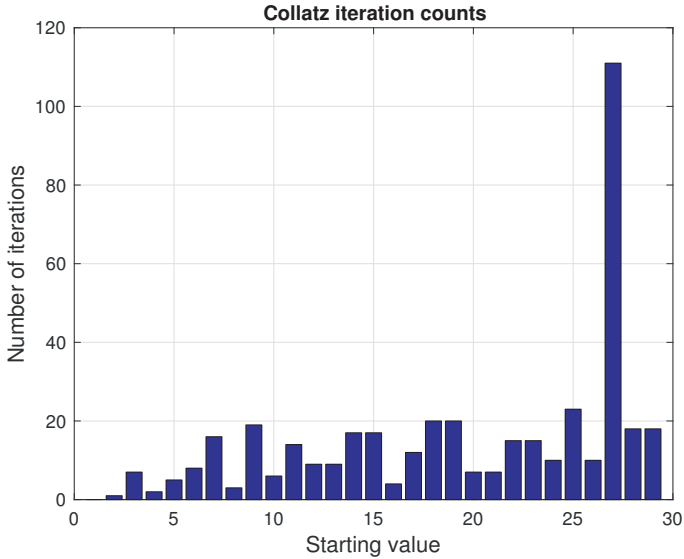


Figure 1.6. Plot produced by `collbar.m`.

Listing 1.4. Script `mandel.m`.

```

%MANDEL    Mandelbrot set.

h = waitbar(0,'Computing...');
x = linspace(-2.1,0.6,2001);
y = linspace(-1.1,1.1,2001);
[X,Y] = meshgrid(x,y);
C = complex(X,Y);

Z_max = 1e6; it_max = 50;
Z = C;
for k = 1:it_max
    Z = Z.^2 + C;
    waitbar(k/it_max)
end
close(h)

contourf(x,y,double(abs(Z)<Z_max))
colormap([1 1 1; 1/2 1/2 1/2]) % Gray inside, white outside.
title('Mandelbrot Set','FontSize',16,'FontWeight','normal')

```

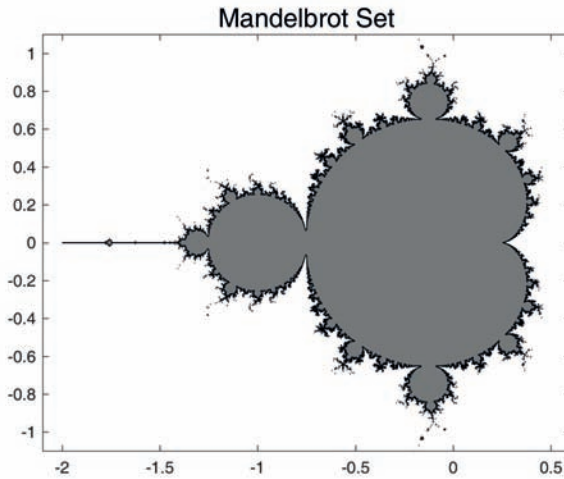


Figure 1.7. *Mandelbrot set approximation produced by mandel.m.*

Listing 1.5. *Function lorenz\_de.m.*

```
function yprime = lorenz_de(t,y)
%LORENZ_DE   Lorenz equations.
%   yprime  = lorenz_de(t,y).

yprime = [10*(y(2)-y(1))
          28*y(1)-y(2)-y(1)*y(3)
          y(1)*y(2)-8*y(3)/3];
```

Listing 1.6. *Script lorenz\_run.m.*

```
%LORENZ_RUN   ODE solving example: Lorenz.

tspan = [0 50];           % Solve for 0 <= t <= 50.
yzero = [0;1;0];         % Initial conditions.
[t,y] = ode45(@lorenz_de,tspan,yzero);
plot(y(:,1),y(:,3))      % (y_1,y_3) phase plane.
xlabel('y_1','FontSize',14)
ylabel('y_3 ','FontSize',14,'Rotation',0,'HorizontalAlignment','right')
title('Lorenz equations','FontSize',16,'FontWeight','normal')
```

Listing 1.7. *Function gasket.m.*

```

function gasket(Pa,Pb,Pc,level)
%GASKET    Recursively generated Sierpinski gasket.
%  GASKET(Pa, Pb, Pc, level) generates an approximation to
%  the Sierpinski gasket, where the 2-vectors Pa, Pb, and Pc
%  define the triangle vertices.
%  level is the level of recursion.

if level == 0
    % Fill the triangle with vertices Pa, Pb, Pc.
    fill([Pa(1),Pb(1),Pc(1)], [Pa(2),Pb(2),Pc(2)], [0.5 0.5 0.5]);
    hold on
else
    % Recursive calls for the three subtriangles.
    gasket(Pa, (Pa+Pb)/2, (Pa+Pc)/2, level-1)
    gasket(Pb, (Pb+Pa)/2, (Pb+Pc)/2, level-1)
    gasket(Pc, (Pc+Pa)/2, (Pc+Pb)/2, level-1)
end

```

the removal process. The input arguments `Pa`, `Pb`, and `Pc` define the vertices of the triangle, and `level` specifies how many times the process is to be applied. If `level` is nonzero then `gasket` calls itself three times with `level` reduced by 1, once for each of the three subtriangles. When `level` finally reaches zero, the appropriate triangle is drawn and filled with gray. The following code generates Figure 1.9(b).

```

level = 5;
Pa = [0;0];
Pb = [1;0];
Pc = [0.5;sqrt(3)/2];
gasket(Pa,Pb,Pc,level)
hold off
title_string = ['Gasket level = ' num2str(level)];
title(title_string,'FontSize',16,'FontWeight','normal')
axis('equal','off')

```

(Figure 1.9(a) was generated in the same way with `level = 1`.) In the last line, the call to `axis` makes the units of the  $x$ - and  $y$ -axes equal and turns off the axes and their labels. You should experiment with different initial vertices `Pa`, `Pb`, and `Pc`, and different levels of recursion, but keep in mind that setting `level` bigger than 8 may overstretch either your patience or your computer's resources.



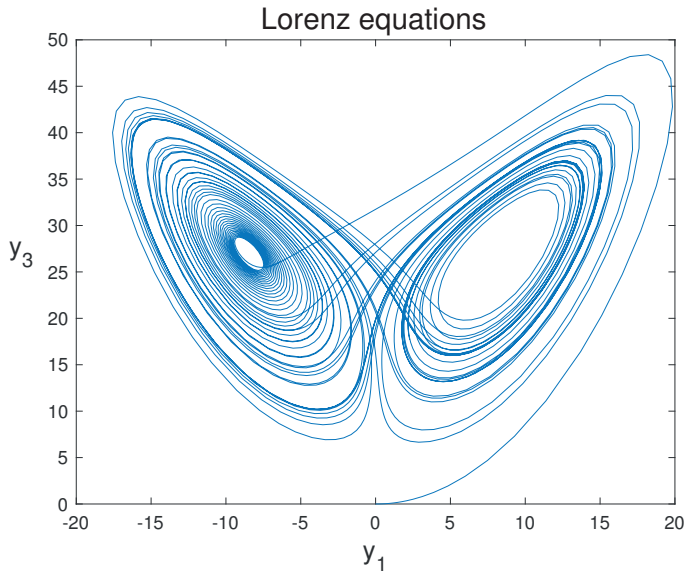


Figure 1.8. Phase plane plot from ode45.

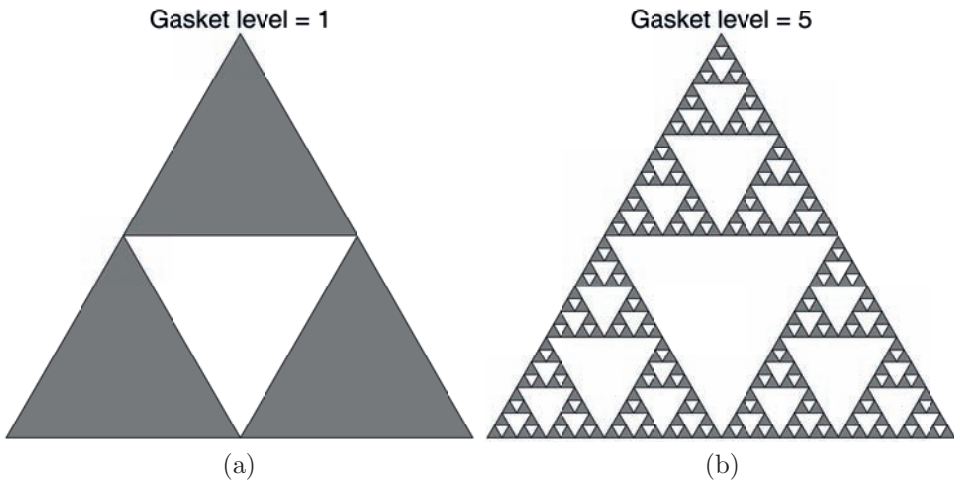


Figure 1.9. Removal process for the Sierpinski gasket (a) and Sierpinski gasket approximation (b), from gasket.m.

The Sierpinski gasket can also be generated by playing Barnsley's "chaos game" [139, Sec. 1.3]. We choose one of the vertices of a triangle as a starting point. Then we pick one of the three vertices at random, take the midpoint of the line joining this vertex with the starting point, and plot this new point. Then we take the midpoint of the line joining this point and a randomly chosen vertex as the next point, which is plotted, and the process continues. The script `barnsley` in Listing 1.8 implements the game. Figure 1.10 shows the result of choosing 1000 iterations:

```
>> barnsley
Enter number of points (try 1000) 1000
```

The reason for the colorful markers is that MATLAB provides several colors for lines and markers and cycles among them if the color is not explicitly specified. Try experimenting with the number of points, `n`, the type and size of marker in the `plot` command, and the location of the starting point.

We finish with the script `sweep` in Listing 1.9, which generates the volume-swept three-dimensional (3D) object shown in Figure 1.11. Here, the command `surf(X,Y,Z)` creates a 3D surface where the height  $Z(i,j)$  is specified at the point  $(X(i,j), Y(i,j))$  in the  $(x,y)$ -plane. The script is not written in the most obvious fashion, which would use two nested `for` loops. Instead it is vectorized. To understand how it works you will need to be familiar with Chapter 5 and Section 24.4. You can experiment with the script by changing the parameter `N` and the function that determines the variable `radius`: try replacing `sqrt` by other functions, such as `log`, `sin`, or `abs`.

Listing 1.8. *Script barnsley.m.*

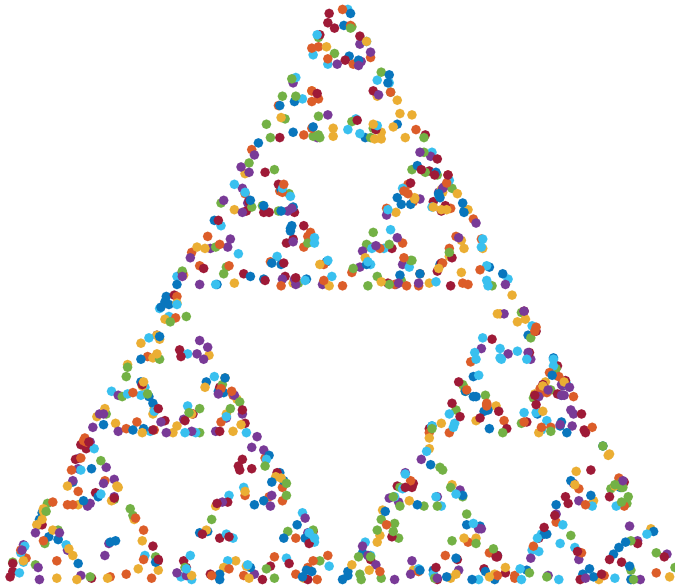
```
%BARNSELEY   Barnsley's game to compute Sierpinski gasket.

rng(1)                               % Set random number state.
V = [0, 1, 0.5; 0, 0, sqrt(3)/2]; % Columns give triangle vertices.

point = V(:,1);                        % Start at a vertex.
n = input('Enter number of points (try 1000) ');

for k = 1:n
    node = ceil(3*rand);                % node is 1, 2, or 3 with equal prob.
    point = (V(:,node) + point)/2;
    plot(point(1),point(2),'.','MarkerSize',15)
    hold on
end

axis('equal','off')
hold off
```

Figure 1.10. *Sierpinski gasket approximation from barnsley.m.*

Listing 1.9. *Script sweep.m.*

```

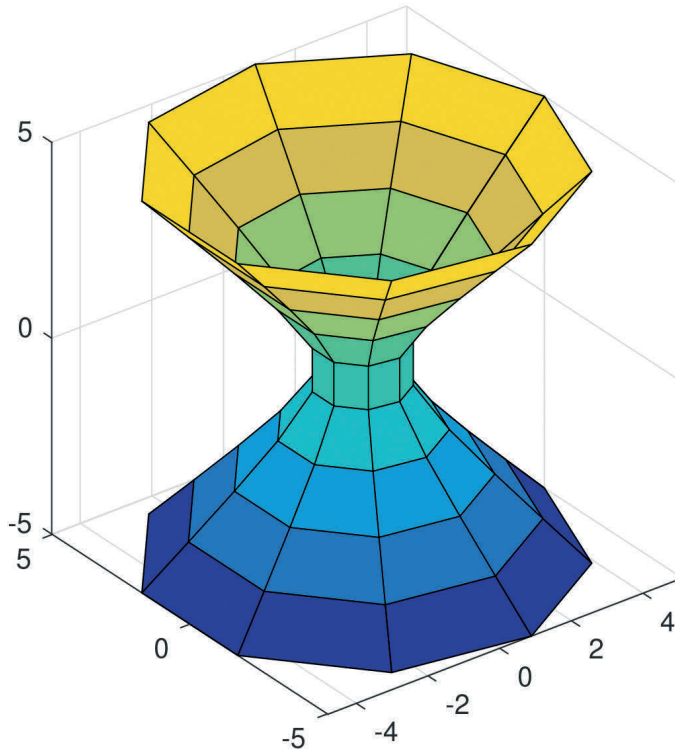
%SWEEP    Generates a volume-swept 3D object.

N = 10;           % Number of increments - try increasing.

z = linspace(-5,5,N)';
radius = sqrt(1+z.^2); % Try changing sqrt to some other function.
theta = 2*pi*linspace(0,1,N);
X = radius*cos(theta);
Y = radius*sin(theta);
Z = z(:,ones(1,N));

surf(X,Y,Z)
axis equal

```

Figure 1.11. *3D picture produced by sweep.m.*

*If you are one of those experts who  
wants to see something from MATLAB right now  
and would rather read the instructions later,  
this page is for you.*

— 386-MATLAB User's Guide (1989)

*Do not be too timid and squeamish about your actions.  
All life is an experiment.  
The more experiments you make the better.*

— RALPH WALDO EMERSON

# Chapter 2

## Basics

### 2.1. MATLAB Desktop

The MATLAB desktop, shown in its default layout in Figure 1.1, contains a Toolstrip at the top with tabs labeled “Home”, “Plots”, and “Apps”. Clicking on a tab opens up a strip or ribbon of tools, and clicking on the tab again closes it. Alternatively, the icon containing a triangle located at the top right-hand corner of the desktop can be used to open or close the active tab.

The window arrangement can be customized using the Layout item on the Home tab. The Preferences item allows many aspects of MATLAB to be customized, including font type, font size, colors, and Command History behavior, amongst many other things.

### 2.2. Interaction and Script Files

MATLAB is an interactive system. You type commands at the prompt (`>>`) in the Command Window and computations are performed when you press the enter or return key. At its simplest level, MATLAB can be used like a pocket calculator:

```
>> (1+sqrt(5))/2
ans =
    1.6180
```

```
>> 2^(-53)
ans =
    1.1102e-016
```

The first example computes  $(1 + \sqrt{5})/2$  and the second  $2^{-53}$ . Note that the second result is displayed in exponential notation: it represents  $1.1102 \times 10^{-16}$ . The variable `ans` is created (or overwritten, if it already exists) when an expression is not assigned to a variable. It can be referenced later, just like any other variable. Unlike in most programming languages, variables are not declared prior to use but are created by MATLAB when they are assigned:

```
>> x = sin(22)
x =
   -0.0089
```

Here, we have assigned to `x` the sine of 22 radians. The printing of output can be suppressed by appending a semicolon. The next example assigns a value to `y` without displaying the result:

```
>> y = 2*x + exp(-3)/(1+cos(0.1));
```

Commas or semicolons are used to separate statements that appear on the same line:

```
>> x = 2, y = cos(1/4), z = 3*x*y
```

```
x =
    2
y =
    0.9689
z =
    5.7320
```

```
>> x = 5; y = cos(0.5); z = x*y^2
```

```
z =
    3.8508
```

Note again that the semicolon causes output from the preceding command to be suppressed.

MATLAB is case sensitive. This means, for example, that `x` and `X` are distinct variables and that the exponential function must be typed as `exp`, not `Exp` or `EXP`.

To perform a sequence of related commands, you can write them into a script, which is a text file with a `.m` filename extension. For example, suppose you wish to process a set of exam marks using the MATLAB functions `sort`, `mean`, `median`, and `std`, which, respectively, sort into increasing order and compute the arithmetic mean, the median, and the standard deviation. You can create a file in the current directory, say `marks.m`, of the form

```
%MARKS
exmark = [12 0 5 28 87 3 56];
exsort = sort(exmark)
exmean = mean(exmark)
exmed = median(exmark)
exstd = std(exmark)
```

The `%` denotes a comment line. Typing

```
>> marks
```

at the command line then produces the output

```
exsort =
    0    3    5   12   28   56   87
exmean =
    27.2857
exmed =
    12
exstd =
    32.8010
```

Note that calling `marks` is entirely equivalent to typing each of the individual commands in sequence at the command line. More details on creating and using script files can be found in Chapter 7.

Throughout this book, unless otherwise indicated, the prompt `>>` signals an example that has been typed at the command line and it is immediately followed by

the corresponding MATLAB output (if any). A sequence of MATLAB commands without the prompt should be interpreted as forming a script file (or part of one).

To quit MATLAB type `exit` or `quit`.

## 2.3. More Fundamentals

MATLAB has many useful functions in addition to the usual ones found on a pocket calculator. For example, you can set up a random matrix of order 3 by typing

```
>> A = rand(3)
A =
    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

Here, each entry of `A` is chosen independently from the uniform distribution on the interval  $[0, 1]$ . The `inv` command inverts `A`:

```
>> inv(A)
ans =
   -1.9958    3.0630   -1.1690
    2.8839   -2.6919    0.6987
   -0.0291   -0.1320    1.1282
```

The inverse has the property that its product with the matrix is the identity matrix. We can check this property for our example by typing

```
>> ans*A
ans =
    1.0000    0.0000    0.0000
   -0.0000    1.0000   -0.0000
         0    0.0000    1.0000
```

The product has ones on the diagonal, as expected. The off-diagonal elements displayed as plus or minus 0.0000 are, in fact, not exactly zero. MATLAB stores numbers and computes to a relative precision of about 16 decimal digits. By default it displays numbers in a 5-digit fixed-point format. While concise, this is not always the most useful format. The `format` command can be used to set a 5-digit floating-point format (also known as scientific or exponential notation):

```
>> format short e
>> ans
ans =
    1.0000e+00    1.1102e-16    2.2204e-16
   -4.9960e-16    1.0000e+00   -1.1102e-16
         0    1.3878e-17    1.0000e+00
```

Now we see that some of the off-diagonal elements of the product are nonzero but tiny—the result of rounding errors. The default format can be reinstated by typing `format short`, or simply `format`. The `format` command has many options, which can be seen by typing `help format`: see Table 2.1 for some examples. All the MATLAB output shown in this book was generated with `format compact` in effect, which suppresses blank lines.



Table 2.1.  $10*\exp(1)$  displayed in several output formats. The space between short or long and e or g can be omitted.

format short	27.1828
format long	27.18281828459045
format short e	2.7183e+001
format long e	2.718281828459045e+001
format short g	27.183
format long g	27.1828182845905
format hex	403b2ecd2dd96d44
format bank	27.18
format rat	2528/93

If you want to change the format and later restore the original format (which may have been changed from the MATLAB default) you can do so using `get` and `set`, as follows:

```
>> format_saved = get(0,'Format'); % Save current format.
>> format longe           % Or any other format option.
% Computations ...
>> set(0,'Format',format_saved) % Restore previous format.
```

If you make an error when typing at the prompt you can correct it using the arrow keys and the backspace or delete keys. Previous command lines can be recalled using the up arrow key, and the down arrow key takes you forward through the command list. If you type a few characters before hitting up arrow then the most recent command line beginning with those characters is recalled. You can scroll through commands previously typed in the current and past sessions in the Command History window, which pops up by default when up arrow is pressed and can be docked or closed. Double-clicking on a command in this window executes it. Table 2.2 summarizes the command line editing keypresses. Many of these keypresses have alternatives that can be seen, and if desired changed, from Preferences-Keyboard-Shortcuts.

Tab completion is a valuable command line time-saver and is also useful when you don't remember the precise name of a function or variable. If you press the tab key after typing a few characters of a function or variable name then MATLAB will attempt to complete the name, and it will offer you a menu of choices if there is more than one possible completion.

It is possible to enter multiple lines at the command line and run them all at once: press Shift-Enter at the end of each line and then press Enter at the end of the last line to run all of the lines.

Type `clc` to clear the Command Window.

A MATLAB computation can be aborted by pressing Ctrl-c. If MATLAB is executing a built-in function it may take some time to respond to this keypress.

A line can be terminated with three periods (`...`), which causes the next line to be a continuation line:

```
>> x = 1 + 1/2 + 1/3 + 1/4 + 1/5 + ...
      1/6 + 1/7 + 1/8 + 1/9 + 1/10
```

Table 2.2. *Command line editing keypresses.*

Key	Operation
Up arrow	Recall previous line
Down arrow	Recall next line
Left arrow	Back one character
Right arrow	Forward one character
Ctrl left arrow	Left one word
Ctrl right arrow	Forward one word
Home	Beginning of line
End	End of line
Esc	Clear line
Del	Delete character under cursor
Backspace	Delete previous character
Ctrl-k	Delete (kill) to end of line
Insert	Toggle insert mode
Shift-home	Select to beginning of line
Shift-end	Select to end of line

Table 2.3. *Information and demonstrations.*

<b>bench</b>	Benchmarks to test the speed of your computer
<b>computer</b>	Computer on which MATLAB is running
<b>demo</b>	A collection of demonstrations
<b>license</b>	License number
<b>ver</b>	Version number and release dates of MATLAB and toolboxes
<b>version</b>	Version number and release dates of MATLAB

```
x =
    2.9290
```

The value of `x` illustrates the fact that, unlike in some other programming languages, arithmetic on integers is done in floating-point arithmetic and so can be written in the natural way.

Commands are available for interacting with the operating system: `cd` (change directory), including `cd ..` to change to the parent directory; `copyfile` (copy file); `mkdir` (make directory); `pwd` (print working directory); `dir` or `ls` (list directory); and `delete` (delete file). A command can be issued to the operating system by preceding it with an exclamation mark, `!`.

Some MATLAB commands giving access to information and demonstrations are listed in Table 2.3.

## 2.4. Help

MATLAB is renowned for the richness of its documentation and the ease of access to it. Help is provided within both the Command Window and the Help browser.

The Help browser (see Figure 2.1) provides complete documentation, which includes help for all MATLAB functions, including numerous examples; release and upgrade notes; and links to the documentation in PDF form on the MathWorks website. The Help browser is accessed by clicking the “?” icon on the toolbar of the MATLAB desktop or by selecting Help from the Home tab. From the Command Window you can type `doc foo` to call up help on the command or function `foo` in the Help browser. Type

```
web([docroot '/matlab/functionlist.html'])
```

for a complete list of MATLAB functions, which can be arranged alphabetically or by category.

Typing `help foo` displays information about `foo` in the Command Window. For example:

```
>> help sqrt
```

```
sqrt    Square root.
        sqrt(X) is the square root of the elements of X. Complex
        results are produced if X is not positive.
```

```
See also sqrtm, realsqrt, hypot.
```

```
Reference page for sqrt
Other functions named sqrt
```

The terms `sqrtm`, `realsqrt`, and `hypot`, and all of the last two lines, are underlined in the Command Window to indicate that they are hyperlinks: clicking on them takes you to the relevant entry.

Note that it is a convention that function names are capitalized within the help lines in a program file. However, when you type `help fun`, any occurrences of uppercase `FUN` in the leading comment lines are converted into lowercase.

Typing `help` by itself produces a list of directories, of which a subset is shown in Table 2.4 (extra directories will be shown for any toolboxes that are available, and if you have added your own directories to the path they will be shown as well). This list provides an overview of how MATLAB functions are organized. Typing `help` followed by a directory name (e.g., `help general`) gives a list of functions in that directory. Type `help help` for further details on the `help` command.

Type `phrase` in the search box of the Help browser, or type `docsearch phrase` in the Command Window, to execute a full-text search of the Help browser documentation for the indicated phrase. You can also type `lookfor keyword` to search for functions relating to the keyword. Example:

```
>> lookfor elliptic
pdepe    - Solve initial--boundary-value problems for
          parabolic-elliptic PDEs in 1-D.
ellipj   - Jacobi elliptic functions.
ellipke  - Complete elliptic integral.
```

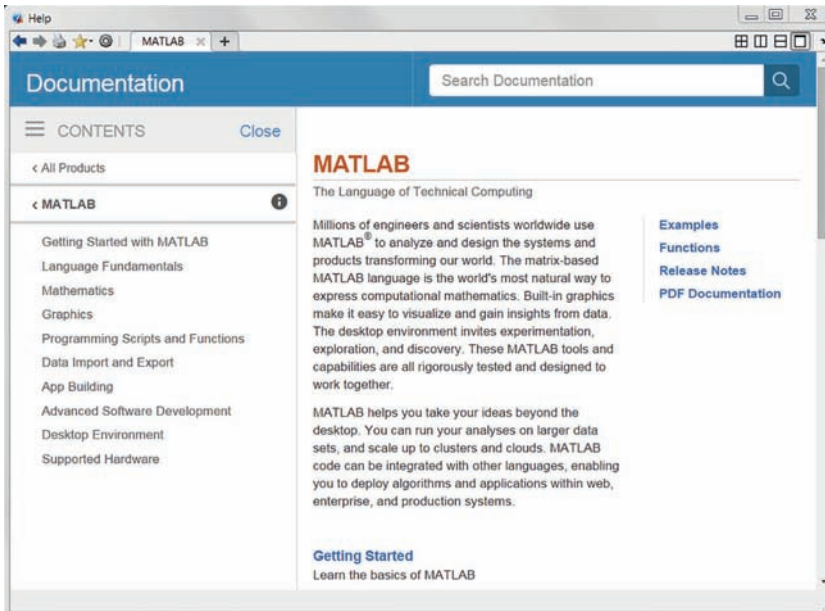
Table 2.4. *MATLAB directory structure (under Windows).*

```

>> help
HELP topics:

matlab\datafun           - Data analysis and Fourier transforms.
matlab\datatypes        - Data types and structures.
matlab\elfun            - Elementary math functions.
matlab\elmat            - Elementary matrices and matrix manipulation.
matlab\funfun           - Function functions and ODE solvers.
matlab\general          - General purpose commands.
matlab\iofun           - File input and output.
matlab\lang             - Programming language constructs.
matlab\matfun           - Matrix functions - numerical linear algebra.
matlab\ops              - Operators and special characters.
matlab\polyfun          - Interpolation and polynomials.
matlab\randfun          - Random matrices and random streams.
matlab\sparfun          - Sparse matrices.
matlab\specfun          - Specialized math functions.
matlab\strfun           - Character strings.
matlab\timefun          - Time and dates.
matlab\demos            - Examples.
matlab\graph2d          - Two dimensional graphs.
matlab\graph3d          - Three dimensional graphs.
matlab\graphics         - Handle Graphics.
matlab\plottools        - Graphical plot editing tools
matlab\scribe           - Annotation and Plot Editing.
matlab\specgraph        - Specialized graphs.
matlab\uitools          - Graphical user interface components and tools
matlab\images           - (No table of contents file)
matlab\optimfun         - Optimization and root finding.
matlab\codetools        - Commands for creating and debugging code
matlab\datamanager      - (No table of contents file)
matlab\datastoreio      - (No table of contents file)
matlab\graphfun         - (No table of contents file)
matlab\guide            - Graphical user interface design environment
matlab\helptools        - Help commands.
matlab\mapreduceio      - (No table of contents file)
testframework\core      - (No table of contents file)
testframework\performance - (No table of contents file)
matlab\verctrl          - Version control.
matlab\winfun           - Windows Operating System Interface Files (COM/DDE)
matlab\apps             - (No table of contents file)
matlab\audiovideo       - Audio and Video support.

```

Figure 2.1. *Help browser.*

## 2.5. Variables and the Workspace

Several functions provide special values:

- `pi` is  $\pi = 3.14159\dots$ ;
- `i` is the imaginary unit,  $\sqrt{-1}$ , as is `j`. Complex numbers are entered as, for example, `2-3i`, `2-3*i`, `2-3*sqrt(-1)`, or `complex(2,-3)`. Note that the form `2-3*i` may not produce the intended results if `i` is being used as a variable. Ambiguity can be avoided by writing `1i` instead of `i`, since the former is always interpreted as the imaginary unit. Therefore `2-3*1i`, or simply `2-3i`, always produces the expected complex number.

Functions generating constants related to floating-point arithmetic are described in Chapter 4. It is possible to override existing variables and functions by creating new ones with the same names. This practice should be avoided as it can lead to confusion. However, the use of `i` and `j` as counting variables is widespread.

MATLAB fully supports complex arithmetic, with `conj`, `real`, and `imag` taking the conjugate and the real and imaginary parts, respectively. For example,

```
>> w = (-1)^0.25
w =
    0.7071 + 0.7071i

>> z = conj(w)
z =
    0.7071 -0.7071i
```

```
>> [real(z) imag(z)]
ans =
    0.7071   -0.7071

>> exp(i*pi)
ans =
-1.0000 + 0.0000i
```

A list of variables in the workspace can be obtained by typing `who`, while `whos` shows the size and class of each variable as well. For example, after executing the commands so far in this chapter, `whos` produces

Name	Size	Bytes	Class	Attributes
A	3×3	72	double	
ans	1×1	16	double	complex
exmark	1×7	56	double	
exmean	1×1	8	double	
exmed	1×1	8	double	
exsort	1×7	56	double	
exstd	1×1	8	double	
w	1×1	16	double	complex
x	1×1	8	double	
y	1×1	8	double	
z	1×1	16	double	complex

To display the total number of bytes occupied type `w = whos; sum([w.bytes])`. (In earlier versions of MATLAB this total was shown automatically at the end of the list.) An existing variable `var` can be removed from the workspace by typing `clear var`, while `clear` or `clearvars` clears all existing variables.

The workspace can also be examined via the Workspace browser (see Figure 2.2), which is invoked from the Layout menu on the Home tab or by typing `workspace`. A variable, `A`, say, can be edited interactively in spreadsheet format in the Array Editor by double-clicking on the variable name (see Figure 2.3); alternatively, typing `openvar('A')` calls up the Array Editor on `A`.

Variable names are case sensitive (as mentioned above) and consist of a letter followed by any combination of letters, digits, and underscores, up to a maximum number of characters given by the value returned by the `namelengthmax` function:

```
>> namelengthmax
ans =
    63
```

To save variables for recall in a future MATLAB session type `save filename`; all variables in the workspace are then saved to `filename.mat`. Alternatively, select the Save Workspace option on the Home tab. The command

```
save filename A x
```

saves just the variables `A` and `x`. The command `load filename` loads in the variables from `filename.mat`, and individual variables can be loaded using the same syntax as for `save`. The default is to save and load variables in binary form, but options allow

Name	Value	Size	Bytes	Class
A	[0.8147,0.9134,0...	3x3		double
ans	-1.0000 + 0.0000i	1x1		double (complex)
exmark	[12,0,5,28,87,3,56]	1x7		double
exmean	27.2857	1x1		double
exmed	12	1x1		double
exsort	[0,3,5,12,28,56,87]	1x7		double
exstd	32.8010	1x1		double
w	0.7071 + 0.7071i	1x1		double (complex)
x	2.9290	1x1		double
y	0.8776	1x1		double
z	0.7071 - 0.7071i	1x1		double (complex)

Figure 2.2. *Workspace browser.* Here, we have added extra columns “Size”, “Bytes”, and “Class” by right-clicking on the bar containing the column headings.

	1	2	3	4	5	6	7	8
1	0.8147	0.9134	0.2785					
2	0.9058	0.6324	0.5469					
3	0.1270	0.0975	0.9575					
4								

Figure 2.3. *Array Editor.*

ASCII form to be specified. MAT-files can be ported between MATLAB implementations running on different computer systems. An Import Wizard, accessible from the Import Data option on the Home tab, or by typing `uiimport`, provides a graphical interface to the MATLAB import functions.

Often you need to capture MATLAB Command Window output for incorporation into a report. This is most conveniently done with the `diary` command. If you type `diary filename` then all subsequent input and (most) text output is copied to the specified file; `diary off` turns off the diary facility. After typing `diary off` you can later type `diary on` to cause subsequent output to be appended to the same diary file.

To print the value of a variable or expression without the name of the variable or `ans` being displayed, you can use `disp`:

```
>> A = eye(2); disp(A)
     1     0
     0     1
```

```
>> disp('Result:'), disp(1/7)
Result:
    0.1429
```

CLASSIC MATLAB

While MATLAB today has thousands of functions, the original Fortran version [122] had only 80, which are listed here.

< M A T L A B >  
Version of 01/10/84

HELP is available

<>

help

Type HELP followed by

INTRO (To get started)

NEWS (recent revisions)

ABS	ANS	ATAN	BASE	CHAR	CHOL	CHOP	CLEA	COND	CONJ	COS
DET	DIAG	DIAR	DISP	EDIT	EIG	ELSE	END	EPS	EXEC	EXIT
EXP	EYE	FILE	FLOP	FLPS	FOR	FUN	HESS	HILB	IF	IMAG
INV	KRON	LINE	LOAD	LOG	LONG	LU	MACR	MAGI	NORM	ONES
ORTH	PINV	PLOT	POLY	PRIN	PROD	QR	RAND	RANK	RCON	RAT
REAL	RETU	RREF	ROOT	ROUN	SAVE	SCHU	SHOR	SEMI	SIN	SIZE
SQRT	STOP	SUM	SVD	TRIL	TRIU	USER	WHAT	WHIL	WHO	WHY

< > ( ) = . , ; \ / ' + - \* :

Most of those functions have survived into today's MATLAB. Two that didn't are CHOP and FLOP. CHOP(*p*) caused the least significant *p* digits to be chopped off after each floating-point operation, and was useful for simulating machines with shorter word lengths. FLOP provided a count of the floating-point operations done and was a convenient tool in algorithm development. With today's processors and software neither function is practical to implement.

*Help!*

— Title of a song by LENNON and MCCARTNEY (1965)

*If ifs and ans were pots and pans,  
there'd be no trade for tinkers.*

— Proverb



# Chapter 3

## Distinctive Features of MATLAB

MATLAB has three features that distinguish it from most other modern programming languages and problem-solving environments. We introduce them in this chapter and elaborate on them later in the book.

### 3.1. Automatic Storage Allocation

As we saw in Chapter 2, variables are not declared prior to being assigned. This applies to arrays as well as scalars. Moreover, MATLAB automatically expands the dimensions of arrays in order for assignments to make sense. Thus, starting with an empty workspace, we can set up a 1-by-3 vector  $x$  of zeros with

```
>> x(3) = 0
x =
     0     0     0
```

and then expand it to length 6 with

```
>> x(6) = 0
x =
     0     0     0     0     0     0
```

Automatic allocation of storage is one of the most convenient and distinctive features of MATLAB. For efficiency, however, it can be desirable to preallocate arrays; see Section 23.4.

### 3.2. Variable Arguments Lists

MATLAB contains a large (and user-extendible) collection of functions. They take zero or more input arguments and return zero or more output arguments. MATLAB enforces a clear distinction between input and output: input arguments appear on the right of the function name, within parentheses, and output arguments appear on the left, within square brackets. Functions can support a variable number of input and output arguments, so that on a given call not all arguments need be supplied. Functions can even vary their behavior depending on the precise number and type of arguments supplied. We illustrate with some examples.

The `norm` function computes the Euclidean norm, or 2-norm, of a vector (the square root of the sum of squares of the absolute values of the elements):

```
>> x = [3 4];
```

```
>> norm(x)
ans =
     5
```

A different norm can be obtained by supplying `norm` with a second input argument. For example, the 1-norm (the sum of the absolute values of the elements) is obtained with

```
>> norm(x,1)
ans =
     7
```

If the second argument is not specified then it defaults to 2, giving the 2-norm. The `max` function has a variable number of output arguments. With an input vector and one output argument it returns the largest element of the vector:

```
>> m = max(x)
m =
     4
```

If a second output argument is supplied then the index of the largest element is assigned to it:

```
>> [m,k] = max(x)
m =
     4
k =
     2
```

For another example of the versatility of MATLAB functions consider `size`, which returns the dimensions of an array. In the following example we set up a 5-by-3 random matrix and then request its dimensions:

```
>> A = rand(5,3);

>> s = size(A)
s =
     5     3
```

With one output argument, `size` returns a 1-by-2 vector with first element the number of rows of the input argument and second element the number of columns. However, `size` can also be given two output arguments, in which case it sets them to the number of rows and columns individually:

```
>> [m,n] = size(A)
m =
     5
n =
     3
```

Some functions vary their computations significantly depending on the number of output arguments requested. For example, the code

```
A = rand(100);
[V,D] = eig(A);
```

computes the eigenvalues of  $\mathbf{A}$  (placed in the diagonal elements of  $\mathbf{D}$ ) and the corresponding eigenvectors (the columns of  $\mathbf{V}$ ), but if we replace the function call by  $\mathbf{e} = \mathbf{eig}(\mathbf{A})$  then just a vector of eigenvalues is computed, which is a much less expensive computation. (For more on `eig`, see Section 9.8.)

### 3.3. Complex Arrays and Arithmetic

The fundamental data type in MATLAB is a multidimensional array of complex numbers, with real and imaginary parts stored in double-precision floating-point format. Important special cases are matrices (two-dimensional arrays), vectors, and scalars. Most computation in MATLAB is performed in floating-point arithmetic, and complex arithmetic is used automatically when the data is complex. There is no separate real data type (though for reals the imaginary part is not stored). This can be contrasted with Fortran, in which different data types are used for real and complex numbers, and with C, C++, and Java, which support only real numbers and real arithmetic.

MATLAB also has integer data types, which are intended mainly for memory-efficient storage, rather than computation; see Section 4.4.

*The guts of MATLAB are written in C.  
Much of MATLAB is also written in MATLAB,  
because it's a programming language.*

— CLEVE B. MOLER (in [118]) (1999)

*In some ways, MATLAB resembles SPEAKEASY and, to a lesser extent, APL.  
All are interactive terminal languages that ordinarily  
accept single-line commands or statements,  
process them immediately,  
and print the results.  
All have arrays as the principal data type.*

— CLEVE B. MOLER, *Demonstration of a Matrix Laboratory* (1982)

# Chapter 4

## Arithmetic

### 4.1. IEEE Arithmetic

Floating-point arithmetic in MATLAB conforms to the IEEE standard [87], [88]. Numeric variables are, by default, of the data type `double` (double precision) and occupy a 64-bit word.

Nonzero double-precision numbers range in magnitude between approximately  $10^{-308}$  and  $10^{+308}$ , and the unit roundoff is  $2^{-53} \approx 1.11 \times 10^{-16}$ . (See [70, Chap. 2] or [137] for a detailed explanation of floating-point arithmetic.) The significance of the unit roundoff is that it is a bound for the relative error in converting a real number to floating-point form and also a bound for the relative error in adding, subtracting, multiplying, or dividing two floating-point numbers or taking the square root of a floating-point number. In simple terms, MATLAB stores floating-point numbers and carries out elementary operations to an accuracy of about 16 significant decimal digits.

The function `eps` returns the distance from 1.0 to the next larger floating-point number:

```
>> eps
ans =
    2.2204e-16
```

This distance,  $2^{-52}$ , is *twice* the unit roundoff. More generally, `eps(x)` returns the (positive) distance from `x` to the next larger (in magnitude) floating-point number:

```
>> eps(1/2)
ans =
    1.1102e-16
```

```
>> eps(2)
ans =
    4.4409e-16
```

Because MATLAB implements the IEEE standard, every computation produces a floating-point number, albeit possibly one of a special type. If the result of a computation is larger than the value returned by the function `realmax` then overflow occurs and the result is `Inf` (also written `inf`), representing infinity. Similarly, a result more negative than `-realmax` produces `-Inf`. Example:

```
>> realmax
ans =
    1.7977e+308
```

```
>> -2*realmax
ans =
    -Inf

>> 1.1*realmax
ans =
     Inf
```

A computation whose result is not mathematically defined produces a NaN, standing for Not a Number. The result NaN (also written nan) is generated by expressions such as `0/0`, `inf/inf`, and `0 * inf`:

```
>> 0/0
ans =
    NaN

>> inf/inf
ans =
    NaN

>> inf-inf
ans =
    NaN
```

Once generated, a NaN propagates through all subsequent computations:

```
>> NaN-NaN
ans =
    NaN

>> 0*NaN
ans =
    NaN
```

The function `realmin` returns the smallest positive normalized floating-point number. Any computation whose result is smaller than `realmin` either underflows to zero if it is smaller than `eps*realmin` or produces a subnormal number—one with leading zero bits in its mantissa. To illustrate:

```
>> realmin
ans =
    2.2251e-308

>> realmin*eps
ans =
    4.9407e-324

>> realmin*eps/2
ans =
     0
```

Table 4.1. *Arithmetic operator precedence.*

Precedence level	Operator
1 (highest)	Exponentiation ( $\wedge$ )
2	Unary plus (+), unary minus (-)
3	Multiplication (*), division (/)
4 (lowest)	Addition (+), subtraction (-)

To obtain further insight, repeat all the above computations after typing `format hex`, which displays the binary floating-point representation of the numbers in hexadecimal format.

## 4.2. Precedence

Arithmetic operators in MATLAB obey the same precedence rules as those in most calculators and computer languages. The rules are shown in Table 4.1. (For a more complete table, showing the precedence of all MATLAB operators, see Table 6.3.) For operators of equal precedence, evaluation is from left to right. Parentheses can always be used to overrule priority, and their use is recommended to avoid ambiguity. Examples:

```
>> 2^10/10
ans =
    102.4000

>> 2 + 3*4
ans =
     14

>> -2 - 3*4
ans =
    -14

>> 1 + 2/3*4
ans =
     3.6667

>> 1 + 2/(3*4)
ans =
     1.1667

>> [2^2^3 2^(2^3)]
ans =
     64    256
```

Table 4.2. *Elementary and special mathematical functions* (“**fun\***” indicates that more than one function name begins “**fun**”).

cos, sin, tan, csc, sec, cot	Trigonometric
cosd, sind, tand, cscd, secd, cotd	
acos, asin, atan, atan2, asec, acsc, acot	Inverse trigonometric
acosd, asind, atand, asecd, acscd, acotd	
cosh, sinh, tanh, sech, csch, coth	Hyperbolic
acosh, asinh, atanh, asech, acsch, acoth	Inverse hyperbolic
log, log2, log10, log1p, exp, expm1	Exponential
sqrt, hypot, nthroot, nextpow2, pow2	Roots, powers
ceil, fix, floor, round	Rounding
abs, angle, conj, imag, real, unwrap	Complex
mod, rem, sign	Remainder, sign
airy, <b>bessel*</b> , <b>beta*</b> , <b>ellipj</b> , <b>ellipke</b> , <b>erf*</b> , <b>expint</b> , <b>gamma*</b> , <b>legendre</b> , <b>psi</b>	Mathematical
<b>factor</b> , <b>factorial</b> , <b>gcd</b> , <b>isprime</b> , <b>lcm</b> , <b>primes</b> , <b>nchoosek</b> , <b>perms</b> , <b>rat</b> , <b>rats</b>	Number theoretic
<b>cart2sph</b> , <b>cart2pol</b> , <b>pol2cart</b> , <b>sph2cart</b>	Coordinate transforms

### 4.3. Mathematical Functions

MATLAB contains a large set of mathematical functions. Typing `help elfun` and `help specfun` calls up full lists of elementary and special functions. A selection is listed in Table 4.2. The trigonometric functions take arguments in radians, with “**\*d**” versions taking arguments in degrees. Of particular note are `expm1` and `log1p`, which accurately compute  $e^x - 1$  and  $\log(1 + x)$ , respectively, for  $|x| \ll 1$ , avoiding the cancellation that would affect the direct calculation. (Explanations of the formulas underlying these two functions can be found in [70, Sec. 1.14.1, Problem 1.5].)

### 4.4. Other Data Types

The MATLAB `double` is not the only data type on which arithmetic can be performed. A `single` data type, corresponding to IEEE single-precision arithmetic, also exists. There are two main reasons for working with single-precision numbers.

- To save storage. A scalar `single` occupies a 32-bit word rather than the 64-bit word occupied by a `double`. Hence, for example, a `single` vector of length 2000 can be stored in the same space as a `double` vector of length 1000.
- To obtain faster execution, the lower storage requirement of single precision over double-precision results in less data movement and therefore reduces data transfer costs. Moreover, in principle, elementary operations in single-precision arithmetic require only about half the time of the corresponding double-precision operations. Whether the latter benefit is seen depends on how the arithmetic is implemented in hardware.

Another use of single-precision arithmetic is to explore the accuracy of a numerical algorithm. It is a standard technique to apply an algorithm in both single- and double-

Table 4.3. *Parameters for single- and double-precision data types.*

Data type	Size	eps	Range
<code>single</code>	32 bits	$2^{-23} \approx 1.19 \times 10^{-7}$	$10^{\pm 38}$
<code>double</code>	64 bits	$2^{-52} \approx 2.22 \times 10^{-16}$	$10^{\pm 308}$

precision arithmetic and use the difference of the results as an estimate of the error in the single-precision solution.

The drawbacks of single precision are that it has half the precision of double precision (about 8 significant decimal digits versus 16) and it has a much smaller range, so overflow and underflow are much more likely to occur. The parameters of single-precision arithmetic can be obtained as follows:

```
>> eps('single')
ans =
    single
    1.1921e-07

>> realmax('single')
ans =
    single
    3.4028e+38

>> realmin('single')
ans =
    single
    1.1755e-38
```

Note that for most data types other than `double`, when MATLAB displays the value of a variable (here `ans`) it includes a header containing the data type, as well as the dimensions if the result is an array. Table 4.3 compares the key parameters for the single and double precisions.

Single-precision numbers can be created in MATLAB in two ways: by conversion from another arithmetic data type using the `single` function, or by using the functions `eye`, `ones`, or `zeros` with an extra `'single'` argument (these functions are described in Section 5.1). For example,

```
>> format long, pi_s = single(pi), pi_d = pi
pi_s =
    single
    3.1415927
pi_d =
    3.141592653589793

>> delta = pi_s - pi_d
delta =
    single
    8.7422777e-08
```



```

>> A = ones(2,'single')
A =
    2×2 single matrix
     1     1
     1     1

>> whos
  Name      Size      Bytes  Class  Attributes

  A         2×2         16   single
  delta     1×1           4   single
  pi_d      1×1           8   double
  pi_s      1×1           4   single

```

This example shows several things. First, the single-precision version of `pi` differs from the double-precision version by about  $10^{-7}$ , which is consistent with the respective precisions. Second, the `whos` output confirms that a `single` occupies half the storage of a `double` of the same dimension. Third, `delta`, which is the difference of a `single` and `double`, has the type `single`. This illustrates the rule that when `single` and `double` arrays interact in arithmetic, the type of the result is `single`.

Single and double-precision numbers can be distinguished with the `class` function:

```

>> class(pi)
ans =
    1×6 char array
double

>> class(single(pi))
ans =
    1×6 char array
single

```

MATLAB is consistent in its use of these two precisions. For example, if a single-precision computation overflows, then the resulting `inf` or `-inf` has the class `single`.

MATLAB also has eight integer data types: `int8`, `int16`, `int32`, and `int64` store signed integers of 8, 16, 32, and 64 bits, respectively, and their analogs `uint8`, `uint16`, `uint32`, and `uint64` store unsigned integers. One of the main uses of these data types is for efficient storage of image data. Limited arithmetic operations are supported, including a number of bitwise operations.

Variables can be created using `eye`, `ones`, and `zeros` with an extra argument:

```

>> E =
    1×3 int8 row vector
     0     0     0

>> A = 5*ones(1,3,'uint16')
A =
    1×3 uint16 row vector
     5     5     5

```

```
>> whos
  Name      Size      Bytes  Class      Attributes

  A         1×3         6  uint16
  E         1×3         3  int8
```

Functions of the same name as the data type convert into these storage formats, and the range of numbers supported can be obtained with `intmin` and `intmax`:

```
>> int8(30.8)
ans =
  int8
   31

>> [intmin('int8') intmax('int8')]
ans =
  1×2 int8 row vector
 -128   127

>> int8(128)
ans =
  int8
  127

>> [intmin('uint8') intmax('uint8')]
ans =
  1×2 uint8 row vector
    0  255

>> [uint8(-1) uint8(256)]
ans =
  1×2 uint8 row vector
    0  255
```

As these examples show, numbers outside the range are mapped to one of the ends of the range.

To obtain a full list of MATLAB data types issue the command `doc datatypes`, and see Figure 18.1 for the interrelations between the data types.

*Round numbers are always false.*

— SAMUEL JOHNSON, *Boswell's Life of Johnson* (1791)

*Minus times minus is plus.*

*The reason for this we need not discuss.*

— W. H. AUDEN, *A Certain World* (1971)

# Chapter 5

## Matrices

An  $m$ -by- $n$  matrix is a two-dimensional array of numbers consisting of  $m$  rows and  $n$  columns. Special cases are a column vector ( $n = 1$ ), a row vector ( $m = 1$ ), and a scalar ( $m = n = 1$ ).

Matrices are fundamental to MATLAB, and even if you are not intending to use MATLAB for linear algebra computations you need to become familiar with matrix generation and manipulation. In versions 3 and earlier of MATLAB there was only one data type: the complex matrix.<sup>3</sup> Nowadays, MATLAB has several data types (see Chapter 18) and matrices are special cases of a multidimensional numeric array.

### 5.1. Matrix Generation

Matrices can be generated in several ways. Many elementary matrices can be constructed directly with a MATLAB function (see Table 5.1). The matrix of zeros, the matrix of ones, and the identity matrix (which has ones on the diagonal and zeros elsewhere) are returned by the functions `zeros`, `ones`, and `eye`, respectively. All have the same syntax. For example, `zeros(m,n)` or `zeros([m,n])` produces an  $m$ -by- $n$  matrix of zeros, while `zeros(n)` produces an  $n$ -by- $n$  matrix. Examples:

```
>> zeros(2)
ans =
     0     0
     0     0

>> ones(2,3)
ans =
     1     1     1
     1     1     1

>> eye(3,2)
ans =
     1     0
     0     1
     0     0
```

A common requirement is to set up an identity matrix whose dimensions match those of a given matrix `A`. This can be done with `eye(size(A))`, where `size` is the function introduced in Section 3.2. Related to `size` is the `length` function:

---

<sup>3</sup>Cleve Moler used to joke that “MATLAB is a strongly typed language: it only has one data type!”

Table 5.1. *Elementary matrices.*

<code>zeros</code>	Zeros array
<code>ones</code>	Ones array
<code>eye</code>	Identity matrix
<code>repmat</code>	Replicate and tile array
<code>rand</code>	Uniformly distributed random numbers
<code>randn</code>	Normally distributed random numbers
<code>randi</code>	Uniformly distributed random integers
<code>linspace</code>	Linearly spaced vector
<code>logspace</code>	Logarithmically spaced vector
<code>meshgrid</code>	X and Y arrays for 3D plots
<code>:</code>	Regularly spaced vector and index into matrix

`length(A)` is the larger of the two dimensions of `A`. Thus for an `n`-by-1 or 1-by-`n` vector `x`, `length(x)` returns `n`.

The functions `nan` and `inf` can also generate matrices in the same way:

```
>> nan(2,3)
ans =
    NaN    NaN    NaN
    NaN    NaN    NaN

>> inf(2)
ans =
    Inf    Inf
    Inf    Inf
```

Depending on the computation, initializing matrix elements to `NaN` or `Inf` may be more likely to reveal bugs in a code than initializing to zeros.

Two other very important matrix generation functions are `rand` and `randn`, which generate matrices of independent (pseudo-)random numbers using the same syntax as `eye`. The function `rand` produces a matrix of numbers from the uniform distribution over the interval  $[0, 1]$ . For this distribution on an interval  $[a, b]$  the proportion of numbers with  $0 < a < b < 1$  is  $b - a$ . To produce uniformly distributed numbers on  $[a, b]$  use `a + rand*(b-1)`.

The function `randn` produces a matrix of numbers from the standard normal (0,1) distribution (mean 0, variance 1). Called without any arguments, both functions produce a single random number:

```
>> rand
ans =
    0.8147

>> rand(3)
ans =
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
    0.9134    0.2785    0.9649
```

The periods of `rand` and `randn`, that is, the number of terms generated before the sequences start to repeat, are  $2^{19937} - 1 \approx 10^{6002}$ .

In carrying out experiments with random numbers it is often important to be able to regenerate the same numbers on a subsequent occasion. The numbers produced by a call to `rand` and `randn` depend on the state of the generator. The generator can be seeded with the command `rng(j)` for a nonnegative integer `j`, after which a predictable sequence of random numbers is generated. For `j = 0` the generator is set to the state it has when MATLAB starts. Typing `rng` on its own produces

```
>> rng
ans =
    Type: 'twister'
    Seed: 0
    State: [625×1 uint32]
```

MATLAB contains seven different random number generator algorithms, and the default is the Mersenne twister. Three of those supported are legacy generators provided for backward compatibility (over the years the generator has been changed several times). See `doc rng` for details of the generators. For most purposes, the default generator is entirely adequate.

#### REPRODUCIBLE RESEARCH

It is good practice to make your computational experiments reproducible, in the sense that you, or someone else working independently, can repeat the experiment and reproduce the results. One aspect of this is recording which codes you used and how they were run. If your computations involve random numbers then it is essential to set the state of the generator using `rng` at the start of the experiment, so that the same random numbers are generated on each run (we have done so throughout this book).

In reporting results you need to state the version of MATLAB used (you can find it out by typing `ver`) and list any toolboxes that were used. Computational results may depend to a greater or lesser extent on the release, since as time passes MATLAB functions are improved, the underlying libraries that MATLAB uses are updated, and the compiler used to generate MATLAB is itself updated. It may not be obvious whether you have used any of the toolboxes available on your system; see Section 16.9 for some ways to determine what functions are used in a particular computation.

For an overview of the many issues in reproducible research see [37].

The function `randi` generates matrices with random integer entries: `randi(imax,n)` generates an `n`-by-`n` matrix with entries drawn from the uniform distribution on `1:imax`.

```
>> randi(25,4)
ans =
     4     21     20     22
    25     4     24     24
    24    11     17     17
    13    23     1     19
```

Prior to Release 2011a of MATLAB, a different syntax was used for setting the state of the random number generator, namely `rand('state',s)`, or `randn('seed',s)`

in much earlier versions. If these commands are used now they cause legacy generators to be activated and a warning message is generated if `rng` is subsequently called. Type `rng default` to escape legacy mode.

Matrices can be built explicitly using the square bracket notation. For example, a 3-by-3 matrix comprising the first nine primes can be set up with the command

```
>> A = [2 3 5
        7 11 13
        17 19 23]
A =
     2     3     5
     7    11    13
    17    19    23
```

The end of a row can be specified by a semicolon instead of a carriage return, so a more compact command with the same effect is

```
>> A = [2 3 5; 7 11 13; 17 19 23]
```

Within a row, elements can be separated by spaces or by commas. In the former case, if numbers are specified with a plus or minus sign take care not to leave a space after the sign, else MATLAB will interpret the sign as an addition or subtraction operator. To illustrate with vectors:

```
>> v = [-1 2 -3 4]
v =
    -1     2    -3     4

>> w = [-1, 2, -3, 4]
w =
    -1     2    -3     4

>> x = [-1 2 - 3 4]
x =
    -1    -1     4
```

Matrices can be constructed in block form. With `B` defined by `B = [1 2; 3 4]`, we may create

```
>> C = [B      zeros(2)
        ones(2) eye(2)]
C =
     1     2     0     0
     3     4     0     0
     1     1     1     0
     1     1     0     1
```

Block diagonal matrices can be defined using the function `blkdiag`, which is easier than using the square bracket notation. Example:

```
>> A = blkdiag(2*eye(2),ones(2))
A =
     2     0     0     0
```

```

0     2     0     0
0     0     1     1
0     0     1     1

```

Useful for constructing “tiled” block matrices is `repmat`: `repmat(A,m,n)` creates a block  $m$ -by- $n$  matrix in which each block is a copy of  $A$ . If  $m$  is omitted, it defaults to  $n$ . Example:

```

>> A = repmat(eye(2),2)
A =
     1     0     1     0
     0     1     0     1
     1     0     1     0
     0     1     0     1

```

MATLAB provides a number of special matrices, which are listed in Table 5.2. These matrices have interesting properties that make them useful for constructing examples and for testing algorithms. One of the most famous is the Hilbert matrix, whose  $(i, j)$  element is  $1/(i + j - 1)$ . The matrix is generated by `hilb`, and its inverse (which has integer entries) by `invhilb`. The function `magic` generates magic squares, which are fun to investigate using MATLAB [124]. The `toeplitz` function constructs a Toeplitz matrix: one for which the elements down each diagonal are constant. Similarly, `hankel` constructs a Hankel matrix: one for which the elements down each *antidiagonal* are constant. To construct a Toeplitz matrix, specify the first column and the first row:

```

>> toeplitz([1 0 -1 -2],[1 2 4 8])
ans =
     1     2     4     8
     0     1     2     4
    -1     0     1     2
    -2    -1     0     1

```

For a Hankel matrix it is the first column and the last row that are specified:

```

>> hankel([3 1 2 0],[0 -1 -2 -3])
ans =
     3     1     2     0
     1     2     0    -1
     2     0    -1    -2
     0    -1    -2    -3

```

The function `gallery` provides access to a large collection of test matrices created by N. J. Higham [69] (an earlier version of the collection was published in [68]). Table 5.3 lists the matrices; more information is obtained by typing `help private\matrix_name`. As indicated in the table, some of the matrices in `gallery` are returned in the `sparse` data type (see Chapter 15). Example:

```

>> help private/moler
moler Moler matrix (symmetric positive definite).
     A = GALLERY('moler',N,ALPHA) is the symmetric positive definite
     N-by-N matrix U'*U, where U = GALLERY('TRIW',N,ALPHA).

```

Table 5.2. *Special matrices.*

<code>compan</code>	Companion matrix
<code>gallery</code>	Large collection of test matrices
<code>hadamard</code>	Hadamard matrix
<code>hankel</code>	Hankel matrix
<code>hilb</code>	Hilbert matrix
<code>invhilb</code>	Inverse Hilbert matrix
<code>magic</code>	Magic square
<code>pascal</code>	Pascal matrix
<code>rosser</code>	Classic symmetric eigenvalue test problem
<code>spiral</code>	Matrix of integers arranged in spiral pattern
<code>toeplitz</code>	Toeplitz matrix
<code>vander</code>	Vandermonde matrix
<code>wilkinson</code>	Wilkinson's eigenvalue test matrix

For the default `ALPHA = -1`,  $A(i,j) = \text{MIN}(i,j)-2$ , and  $A(i,i) = i$ . One of the eigenvalues of  $A$  is small.

```
>> A = gallery('moler',5)
A =
     1     -1     -1     -1     -1
    -1      2      0      0      0
    -1      0      3      1      1
    -1      0      1      4      2
    -1      0      1      2      5
```

By default, the matrices produce by `gallery` are of type `double`. If you append an input argument `'single'`, or provide a numeric parameter (other than the dimension) of type `single`, then a matrix of type `single` is produced. Example:

```
>> A = gallery('moler',4,single(1)), class(A)
A =
     1      1      1      1
     1      2      2      2
     1      2      3      3
     1      2      3      4

ans =
single
```

Table 5.4 lists matrices from Tables 5.2 and 5.3 having certain properties; in most cases the matrix has the property for the default arguments, but in some cases, such as for `gallery`'s `randsvd`, the arguments must be suitably chosen. For definitions of these properties see Chapter 9 and the textbooks listed at the start of that chapter.

Another way to generate a matrix is to load it from a file using the `load` command (see p. 31).



Table 5.3. *Matrices available through gallery.*

binomial	Binomial matrix—multiple of involutory matrix
cauchy	Cauchy matrix
chebspec	Chebyshev spectral differentiation matrix
chebvand	Vandermonde-like matrix for the Chebyshev polynomials
chow	Chow matrix—a singular Toeplitz lower Hessenberg matrix
circul	Circulant matrix
clement	Clement matrix—tridiagonal with zero diagonal entries
compar	Comparison matrices
condex	Counterexamples to matrix condition number estimators
cycol	Matrix whose columns repeat cyclically
dorr	Dorr matrix—diagonally dominant, ill-conditioned, tridiagonal (one or three output arguments, <b>sparse</b> )
dramadah	Matrix of 1s and 0s whose inverse has large integer entries
fiedler	Fiedler matrix—symmetric
forsythe	Forsythe matrix—a perturbed Jordan block
frank	Frank matrix—ill-conditioned eigenvalues
gcdmat	GCD matrix
gearmat	Gear matrix
grcar	Grcar matrix—a Toeplitz matrix with sensitive eigenvalues
hanowa	Matrix whose eigenvalues lie on a vertical line in the complex plane
house	Householder matrix (three output arguments)
integerdata	Array of arbitrary data from uniform distribution on specified range of integers
invhess	Inverse of an upper Hessenberg matrix
invol	Involutory matrix
ipjfact	Hankel matrix with factorial elements (two output arguments)
jordbloc	Jordan block matrix
kahan	Kahan matrix—upper trapezoidal
kms	Kac–Murdock–Szego Toeplitz matrix
krylov	Krylov matrix
lauchli	Läuchli matrix—rectangular
lehmer	Lehmer matrix—symmetric positive definite
leslie	Leslie matrix
lesp	Tridiagonal matrix with real, sensitive eigenvalues
lotkin	Lotkin matrix
minij	Symmetric positive definite matrix $\min(i, j)$
moler	Moler matrix—symmetric positive definite
neumann	Singular matrix from the discrete Neumann problem ( <b>sparse</b> )
normaldata	Array of arbitrary data from standard normal distribution
orthog	Orthogonal and nearly orthogonal matrices
parter	Parter matrix—a Toeplitz matrix with singular values near $\pi$
pei	Pei matrix
poisson	Block tridiagonal matrix from Poisson’s equation ( <b>sparse</b> )
prolate	Prolate matrix—symmetric, ill-conditioned Toeplitz matrix
qmult	Premultiply matrix by random orthogonal matrix

Table 5.3. (*continued*)

<code>randcolu</code>	Random matrix with normalized columns and specified singular values
<code>randcorr</code>	Random correlation matrix with specified eigenvalues
<code>randhess</code>	Random, orthogonal upper Hessenberg matrix
<code>randjorth</code>	Random $J$ -orthogonal matrix
<code>rando</code>	Random matrix with elements $-1$ , $0$ , or $1$
<code>randsvd</code>	Random matrix with preassigned singular values and specified bandwidth
<code>redheff</code>	Matrix of 0s and 1s of Redheffer
<code>riemann</code>	Matrix associated with the Riemann hypothesis
<code>ris</code>	Ris matrix—a symmetric Hankel matrix
<code>sampling</code>	Nonsymmetric matrix with integer, ill-conditioned eigenvalues
<code>smoke</code>	Smoke matrix—complex, with a “smoke ring” pseudospectrum
<code>toeppd</code>	Symmetric positive definite Toeplitz matrix
<code>toeppen</code>	Pentadiagonal Toeplitz matrix ( <b>sparse</b> )
<code>tridiag</code>	Tridiagonal matrix ( <b>sparse</b> )
<code>triw</code>	Upper triangular matrix discussed by Wilkinson and others
<code>uniformdata</code>	Array of arbitrary data from standard uniform distribution
<code>wathen</code>	Wathen matrix—a finite-element matrix ( <b>sparse</b> , random entries)
<code>wilk</code>	Various specific matrices devised/discussed by Wilkinson (two output arguments)
<code>gallery(3)</code>	Badly conditioned 3-by-3 matrix
<code>gallery(5)</code>	Interesting eigenvalue problem

## 5.2. Subscripting and the Colon Notation

To enable access and assignment to submatrices MATLAB has a powerful notation based on the colon character. The colon is used to define vectors that can act as subscripts. For integers  $i$  and  $j$ ,  $i:j$  denotes the row vector of integers from  $i$  to  $j$  (in steps of 1). A nonunit step (or stride)  $s$  is specified as  $i:s:j$ . This notation is valid even for noninteger  $i$ ,  $j$ , and  $s$ . Examples:

```
>> 1:5
ans =
     1     2     3     4     5

>> 4:-1:-2
ans =
     4     3     2     1     0    -1    -2

>> 0:.75:3
ans =
     0    0.7500    1.5000    2.2500    3.0000
```

Single elements of a matrix are accessed as  $A(i,j)$ , where  $i \geq 1$  and  $j \geq 1$  (zero or negative subscripts are not supported in MATLAB). The submatrix comprising the intersection of rows  $p$  to  $q$  and columns  $r$  to  $s$  is denoted by  $A(p:q,r:s)$ . As

Table 5.4. *Matrices classified by property. Most of the matrices listed here are accessed through gallery.*

Defective	chebspec, gallery(5), gearmat, jordbloc, triw
Hankel	hilb, ipjfact, ris
Hessenberg	chow, frank, grcar, randhess, randsvd
Idempotent	invol
Inverse of tridiagonal matrix	kms, lehmer, minij
Involutory	binomial, invol, orthog, pascal
Nilpotent	chebspec, gallery(5)
Normal*	circul
Orthogonal	hadamard, orthog, randhess, randsvd
Rectangular	chebvand, cycol, kahan, krylov, lauchli, rando, randsvd, triw
Symmetric indefinite	clement, fiedler
Symmetric positive definite	gcdmat, hilb, invhilb, ipjfact, kms, lehmer, minij, moler, pascal, pei, poisson, prolate, randcorr, randsvd, toeppd, tridiag, wathen
Toeplitz	chow, dramadah, grcar, kms, parter, prolate, toeppd, toeppen
Totally positive/nonnegative	cauchy, <sup>†</sup> hilb, lehmer, pascal
Tridiagonal	clement, dorr, lesp, randsvd, tridiag, wilk, wilkinson
Triangular	dramadah, jordbloc, kahan, pascal, triw

\* But not symmetric or orthogonal.

<sup>†</sup> `cauchy(x,y)` is totally positive if  $0 < x_1 < \dots < x_n$  and  $0 < y_1 < \dots < y_n$  [70].

a special case, a lone colon as the row or column specifier covers all entries in that row or column; thus `A(:,j)` is the *j*th column of *A* and `A(i,:)` is the *i*th row. The keyword `end` used in this context denotes the last index in the specified dimension; thus `A(end,:)` picks out the last row of *A*. In effect, a lone colon is short for `1:end`. Finally, an arbitrary submatrix can be selected by specifying the individual row and column indices. For example, `A([i j k],[p q])` produces the submatrix given by the intersection of rows *i*, *j*, and *k* and columns *p* and *q*. Here are some examples, using the matrix of primes set up above:

```
>> A
A =
     2     3     5
     7    11    13
    17    19    23

>> A(2,1)
ans =
     7

>> A(2:3,2:3)
```

```

ans =
    11    13
    19    23

>> A(:,1)
ans =
     2
     7
    17

>> A(2,:)
ans =
     7    11    13

>> A(end:-1:1,end)
ans =
    23
    13
     5

>> A([1 3],[2 3])
ans =
     3     5
    19    23

```

#### ZERO-BASED VERSUS ONE-BASED INDEXING

Arrays start at index 1 in MATLAB, so  $x(1)$  and  $A(1,1)$  are the first elements of a vector  $x$  and matrix  $A$ . The same is true of Fortran, Julia, and Pascal. Many other languages, including C, Python, and Scala, use zero-based indexing, with indices starting at 0. If you translate from, or compare with, code in other languages, or with pseudocode, make sure to check which form of indexing is in effect.

A further special case is  $A(:)$ , which denotes a vector comprising all the elements of  $A$  taken down the columns from first to last:

```

>> B = A(:)
B =
     2
     7
    17
     3
    11
    19
     5
    13
    23

```

When placed on the left side of an assignment statement,  $A(:)$  fills  $A$ , preserving its shape. Using this notation, another way to define our 3-by-3 matrix of primes is

```
>> A = zeros(3); A(:) = primes(23); A = A'
A =
     2     3     5
     7    11    13
    17    19    23
```

The function `primes` returns a vector of the prime numbers less than or equal to its argument. The transposition `A = A'` (see the next section) is necessary to reorder the primes across the rows rather than down the columns.

#### THE COLON NOTATION

The colon notation `A(i:j,p:q)` for array subscripting (or slicing) was used in specific forms in early languages such as Algol 68. Following its introduction in its most general form in MATLAB many other programming languages have adopted it. The colon notation is also widely used as a mathematical notation in textbooks and research papers. An early and influential example was the first edition (1983) of Golub and Van Loan's book *Matrix Computations* [53].

In one circumstance—when the right-hand side is a single element—the number of elements in a subscripted assignment can be different on the two sides of the assignment. In this case the scalar is “expanded” to match the number of elements on the left:

```
>> A = ones(3);
>> A(2:3,2:3) = 0 % Scalar expansion.
A =
     1     1     1
     1     0     0
     1     0     0
```

Related to the colon notation for generating vectors of equally spaced numbers is the function `linspace`, which accepts the number of points rather than the increment: `linspace(a,b,n)` generates `n` equally spaced points between `a` and `b`. If `n` is omitted it defaults to 100. Example:

```
>> linspace(-1,1,9)
ans =
Columns 1 through 7
-1.0000   -0.7500   -0.5000   -0.2500         0    0.2500    0.5000
Columns 8 through 9
 0.7500    1.0000
```

### 5.3. Matrix and Array Operations

For scalars `a` and `b`, the operators `+`, `-`, `*`, `/`, and `^` produce the obvious results. As well as the usual right division operator, `/`, MATLAB has a left division operator, `\`:

MATLAB notation	Mathematical equivalent
Right division: $a/b$	$\frac{a}{b}$
Left division: $a \backslash b$	$\frac{b}{a}$

For matrices, all these operations can be carried out in a matrix sense (according to the rules of matrix algebra) or an array sense (elementwise). Table 5.5 summarizes the syntax.

Addition and subtraction, which are identical operations in the matrix and array senses, are defined for matrices of the same dimension. The product  $A*B$  is the result of matrix multiplication, defined only when the number of columns of  $A$  and the number of rows of  $B$  are the same. The backslash and the forward slash define solutions of linear systems:  $A \backslash B$  is a solution  $X$  of  $A*X = B$ , while  $A/B$  is a solution  $X$  of  $X*B = A$  (see Section 9.3 for more details). Examples:

```
>> A = [1 2; 3 4], B = ones(2)
A =
     1     2
     3     4
B =
     1     1
     1     1

>> A+B
ans =
     2     3
     4     5

>> A*B
ans =
     3     3
     7     7

>> A \ B
ans =
    -1    -1
     1     1
```

Multiplication and division in the array, or elementwise, sense are specified by preceding the operator with a period. If  $A$  and  $B$  are matrices of the same dimensions then  $C = A.*B$  sets  $C(i,j) = A(i,j)*B(i,j)$  and  $C = A./B$  sets  $C(i,j) = A(i,j)/B(i,j)$ . The assignment  $C = A.\backslash B$  is equivalent to  $C = B./A$ . With the same  $A$  and  $B$  as in the previous example:

```
>> A.*B
ans =
     1     2
     3     4

>> B./A
```

Table 5.5. *Elementary matrix and array operations.*

Operation	Matrix sense	Array sense
Addition	+	+
Subtraction	-	-
Multiplication	*	.*
Left division	\	.\
Right division	/	./
Exponentiation	^	.^

```
ans =
    1.0000    0.5000
    0.3333    0.2500
```

Exponentiation with  $\wedge$  is defined as matrix powering, but the dot form exponentiates elementwise. Thus if  $A$  is a square matrix then  $A^2$  is the matrix product  $A*A$ , but  $A.^2$  is  $A$  with each element squared:

```
>> A^2, A.^2
ans =
     7     10
    15     22
ans =
     1     4
     9    16
```

The dot form of exponentiation allows the power to be an array when the dimensions of the base and the power agree, or when the base is a scalar:

```
>> x = [1 2 3]; y = [2 3 4]; Z = [1 2; 3 4];

>> x.^y
ans =
     1     8    81

>> 2.^x
ans =
     2     4     8

>> 2.^Z
ans =
     2     4
     8    16
```

Matrix exponentiation is defined for all powers, not just for positive integers. If  $n < 0$  is an integer then  $A^n$  is defined as  $\text{inv}(A)^{-n}$ . For noninteger  $p$ ,  $A^p$  is evaluated using the eigensystem of  $A$ ; results can be incorrect or inaccurate when  $A$  is not diagonalizable or when  $A$  has an ill-conditioned eigensystem.

The conjugate transpose of the matrix  $A$  is obtained with  $A'$ . If  $A$  is real this is simply the transpose. The transpose without conjugation is obtained with  $A.'$ .

The functional alternatives `ctranspose(A)` and `transpose(A)` are sometimes more convenient.

For the special case of column vectors  $x$  and  $y$ ,  $x'*y$  is the inner, scalar, or dot product, which can also be obtained using the `dot` function as `dot(x,y)`. The vector or cross product of two 3-by-1 or 1-by-3 vectors (as used in mechanics) is produced by `cross`. Example:

```
>> x = [-1 0 1]'; y = [3 4 5]';

>> x'*y
ans =
     2

>> dot(x,y)
ans =
     2

>> cross(x,y)
ans =
    -4
     8
    -4
```

The `kron` function evaluates the Kronecker product of two matrices. The Kronecker product of an  $m$ -by- $n$   $A$  and  $p$ -by- $q$   $B$  has dimensions  $mp$ -by- $nq$  and can be expressed as a block  $m$ -by- $n$  matrix with  $(i,j)$  block  $a_{ij}B$ . Example:

```
>> A = [1 10; -10 100]; B = [1 2 3; 4 5 6; 7 8 9];

>> kron(A,B)
ans =
     1     2     3    10    20    30
     4     5     6    40    50    60
     7     8     9    70    80    90
   -10   -20   -30   100   200   300
   -40   -50   -60   400   500   600
   -70   -80   -90   700   800   900
```

The `repelem` function enables a larger matrix to be formed from a given matrix by repeating its elements:

```
>> repelem(A,2,3)
ans =
     1     1     1    10    10    10
     1     1     1    10    10    10
   -10   -10   -10   100   100   100
   -10   -10   -10   100   100   100
```

If a scalar is added to a matrix MATLAB will expand the scalar into a matrix with all elements equal to that scalar. For example:

```
>> [4 3; 2 1] + 4
```



```

ans =
     8     7
     6     5

>> A = [1 -1] - 6
A =
    -5    -7

```

However, if an assignment makes sense without expansion then it will be interpreted in that way. Thus if the previous command is followed by `A = 1` then `A` becomes the scalar 1, not `ones(1,2)`.

If a matrix is multiplied or divided by a scalar, the operation is performed elementwise. For example:

```

>> [3 4 5; 4 5 6]/12
ans =
    0.2500    0.3333    0.4167
    0.3333    0.4167    0.5000

```

Most of the functions described in Section 4.3 can be given a matrix argument, in which case the functions are computed elementwise. For example, here we verify that  $\cos^2 x + \sin^2 x = 1$  for six random  $x$ :

```

>> A = randn(2,3);
>> hypot(cos(A),sin(A))
ans =
    1.0000    1.0000    1.0000
    1.0000    1.0000    1.0000

```

Functions of a matrix in the linear algebra sense are signified by names ending in `m` (see Section 9.10): `expm`, `funm`, `logm`, `sqrtnm`. For example, for `A = [2 2; 0 2]`,

```

>> sqrt(A)
ans =
    1.4142    1.4142
         0    1.4142

>> sqrtnm(A)
ans =
    1.4142    0.7071
         0    1.4142

>> ans*ans
ans =
    2.0000    2.0000
         0    2.0000

```

### 5.3.1. Implicit Expansion

MATLAB has a feature called implicit expansion that applies to addition, subtraction, certain other arithmetic, relational, and logical operators, and certain mathematical

functions with two input arguments. If one array argument,  $x$  say, has a unit dimension where the other argument has a non-unit dimension then  $x$  is (effectively) replicated along that dimension in order that the two arrays have the same dimension; this can even result in both arguments being replicated. A special case is scalar expansion, which we have seen already:

```
>> zeros(2) + 1
ans =
     1     1
     1     1
```

Here are some examples where vectors are implicitly expanded:

```
>> A = ones(3), b = [1 2 3]
A =
     1     1     1
     1     1     1
     1     1     1
b =
     1     2     3
```

```
>> A - b
ans =
     0    -1    -2
     0    -1    -2
     0    -1    -2
```

```
>> A - b'
ans =
     0     0     0
    -1    -1    -1
    -2    -2    -2
```

```
>> b - b'
ans =
     0     1     2
    -1     0     1
    -2    -1     0
```

The `min` and `max` functions support implicit expansion, as exploited by the function `gallery('minij',...)`, which forms the matrix with  $(i,j)$  element  $\min(i,j)$  as

```
a = 1:n; A = min(a,a');
```

Implicit expansion also works with elementwise arithmetic operators such as `.*` and `.\`. Here, we use it to carry out multiplication by a diagonal matrix on the left or the right, and we check the answer:

```
>> A = magic(4); d = [-10 -1 2 20];
>> BD = A.*d; % Or d.*A
>> DB = A.*d'; % Or d'.*A
>> [isequal(BD, A*diag(d)) isequal(DB, diag(d)*A)]
```

```
ans =
    1×2 logical array
     1     1
```

The `.*` forms execute more quickly than the expressions that require formation of the diagonal matrix `diag(d)`. Multiplication by the inverse of a diagonal matrix can be performed in an analogous way, and here we can use either the left or the right division operator:

```
>> B1 = d.\A; B2 = A./d; B3 = A/diag(d); norm([B1-B3, B2-B3])
ans =
     0
```

```
>> B1 = d'.\A; B2 = A./d'; B3 = diag(d)\A; norm([B1-B3, B2-B3])
ans =
     0
```

Before implicit expansion was fully implemented in MATLAB it was recommended to use the function `bsxfun` (binary singleton expansion function), which provides a fast way to carry out binary operations on two arrays with implicit expansion. For example, the previous subtraction and the formation of `BD` can also be expressed as

```
bsxfun(@minus,b,b')
BD = bsxfun(@times,A,d);
```

## 5.4. Empty Matrices

An  $m$ -by- $n$  matrix with one or both of  $m$  and  $n$  equal to zero is called an empty matrix:

```
>> ones(0,0), zeros(3,0), rand(0,5)
ans =
    []
ans =
    3×0 empty double matrix
ans =
    0×5 empty double matrix
```

The notation `[]` stands for the 0-by-0 matrix. If the use of `ones`, `zero`, or `rand` to construct an empty matrix seems arbitrary and unsatisfactory, you can use `double.empty` instead:

```
>> double.empty(0,2)
ans =
    0×2 empty double matrix
```

MATLAB defines operations on empty matrices by extrapolating the rules for normal matrices to the case of a zero dimension. In particular, multiplication of an  $m$ -by- $n$  matrix by an  $n$ -by- $p$  matrix to produce an  $m$ -by- $p$  matrix remains valid when one or more of the dimensions is zero: an empty matrix is produced, except in the special case when only  $n$  is zero, in which case the product is an  $m$ -by- $p$  matrix of zeros:

Table 5.6. *Matrix manipulation functions.*

<code>reshape</code>	Change size
<code>diag</code>	Diagonal matrices and diagonals of matrix
<code>tril</code>	Extract lower triangular part
<code>triu</code>	Extract upper triangular part
<code>flip</code>	Flip (reverse) order of elements
<code>fliplr</code>	Flip matrix in left/right direction
<code>flipud</code>	Flip matrix in up/down direction
<code>rot90</code>	Rotate matrix 90 degrees

```
>> m = 0; n = 2; p = 4; A = rand(m,n); B = rand(n,p); C = A*B
C =
Empty matrix: 0-by-4
```

```
>> m = 3; n = 0; p = 4; A = rand(m,n); B = rand(n,p); C = A*B
C =
    0    0    0    0
    0    0    0    0
```

Assigning `[]` to a row or column is one way to delete that row or column from a matrix:

```
>> A = spiral(3)
A =
    7    8    9
    6    1    2
    5    4    3

>> A(2,:) = []
A =
    7    8    9
    5    4    3
```

In this example the same effect is achieved with `A = A([1 3],:)`. The empty matrix is also useful as a placeholder in argument lists, as we will see in Section 5.6.

The empty matrix is an elegant concept that can remove the need to treat edge cases specially. See Section 24.1 for some examples.

## 5.5. Matrix Manipulation

Several commands are available for manipulating matrices (commands more specifically associated with linear algebra are discussed in Chapter 9); see Table 5.6.

The `reshape` function changes the dimensions of a matrix: `reshape(A,m,n)` produces an  $m$ -by- $n$  matrix whose elements are taken columnwise from `A`. For example:

```
>> A = [1 4 9; 16 25 36], B = reshape(A,3,2)
A =
    1    4    9
```

```

      16    25    36
B =
      1    25
     16     9
      4    36

```

The function `diag` deals with the diagonals of a matrix and can take a vector or a matrix as argument. For a vector  $x$ , `diag(x)` is the diagonal matrix with main diagonal  $x$ :

```

>> diag([1 2 3])
ans =
      1     0     0
      0     2     0
      0     0     3

```

More generally, `diag(x,k)` puts  $x$  on the  $k$ th diagonal, where  $k > 0$  specifies diagonals above the main diagonal and  $k < 0$  diagonals below the main diagonal ( $k = 0$  gives the main diagonal):

```

>> diag([1 2], 1)
ans =
      0     1     0
      0     0     2
      0     0     0

```

```

>> diag([3 4], -2)
ans =
      0     0     0     0
      0     0     0     0
      3     0     0     0
      0     4     0     0

```

For a matrix  $A$ , `diag(A)` is the column vector comprising the main diagonal of  $A$ . To produce a diagonal matrix with diagonal the same as that of  $A$  you must therefore write `diag(diag(A))`. Analogously to the vector case, `diag(A,k)` produces a column vector made up from the  $k$ th diagonal of  $A$ . Thus if

```

A =
      2     3     5
      7    11    13
     17    19    23

```

then

```

>> diag(A)
ans =
      2
     11
     23

>> diag(A,-1)
ans =

```

```

7
19

```

Triangular parts of a matrix can be extracted using `tril` and `triu`. The lower triangular part of `A` (the elements on and below the main diagonal) is specified by `tril(A)` and the upper triangular part of `A` (the elements on and above the main diagonal) is specified by `triu(A)`. More generally, `tril(A,k)` gives the elements on and below the `k`th diagonal of `A`, while `triu(A,k)` gives the elements on and above the `k`th diagonal of `A`. With `A` as above:

```

>> tril(A)
ans =
    2     0     0
    7    11     0
   17    19    23

>> triu(A,1)
ans =
    0     3     5
    0     0    13
    0     0     0

>> triu(A,-1)
ans =
    2     3     5
    7    11    13
    0    19    23

```

## 5.6. Data Analysis

Table 5.7 lists functions for basic data analysis computations. The simplest usage is to apply these functions to vectors. For example:

```

>> x = [4 -8 -2 1 0]
x =
    4    -8    -2     1     0

>> [min(x) max(x)]
ans =
   -8     4

>> sort(x)
ans =
   -8    -2     0     1     4

>> sum(x)
ans =
   -5

```

The `sort` function sorts into ascending order by default. Descending order is obtained by appending an extra argument `'descend'`. For complex vectors, `sort` sorts by absolute value:

Table 5.7. *Basic data analysis functions.*

<code>max</code>	Largest component
<code>min</code>	Smallest component
<code>mean</code>	Average or mean value
<code>median</code>	Median value
<code>mode</code>	Mode (most frequent value)
<code>std</code>	Standard deviation
<code>var</code>	Variance
<code>sum</code>	Sum of elements
<code>prod</code>	Product of elements
<code>movmax, movmin</code>	Moving maximum and minimum of elements
<code>movmean, movmedian</code>	Moving mean and median of elements
<code>movstd, movsum</code>	Moving standard deviation and sum of elements
<code>cummax, cummin</code>	Cumulative maximum and minimum of elements
<code>cumsum, cumprod</code>	Cumulative sum and product of elements
<code>diff</code>	Difference of elements
<code>sort</code>	Sort in ascending order

```
>> x = [1+i -3-4i 2i 1];
>> sort(x,'descend')
ans =
-3.0000 - 4.0000i    0.0000 + 2.0000i    1.0000 + 1.0000i
 1.0000 + 0.0000i
```

Any NaN elements are placed by `sort` at the high end, while `max` and `min` ignore NaNs.

For matrices the functions are defined columnwise. Thus `max` and `min` return a vector containing the maximum and minimum element, respectively, in each column, `sum` returns a vector containing the column sums, and `sort` sorts the elements in each column of the matrix into ascending order. The functions `min` and `max` can return a second argument that specifies in which components the minimum and maximum elements are located. For example, if

```
A =
    0    -1     2
    1     2    -4
    5    -3    -4
```

then

```
>> max(A)
ans =
    5     2     2

>> [m,i] = min(A)
m =
    0    -3    -4
i =
    1     3     2
```

As this example shows, if there are two or more minimal elements in a column then the index of the first is returned. The smallest element in the matrix can be found by applying `min` twice in succession:

```
>> min(min(A))
ans =
    -4
```

An alternative, which has the advantage that it also works for arrays of dimension greater than 2, is

```
>> min(A(:))
ans =
    -4
```

Functions `max` and `min` can be made to act row-wise via a third argument:

```
>> max(A, [], 2)
ans =
     2
     2
     5
```

The 2 in `max(A, [], 2)` specifies the maximum over the second dimension, that is, over the column index. The empty second argument, `[]`, is needed because with just two arguments `max` and `min` return the elementwise maxima and minima of the two arguments:

```
>> max(A, 0)
ans =
     0     0     2
     1     2     0
     5     0     0
```

Functions `sort` and `sum` can also be made to act row-wise, via a second argument. For more on `sort` see Section 24.3.

For complex data, `max` and `min` measure size using the absolute value, like `sort`.

The `diff` function forms differences. Applied to a vector `x` of length `n` it produces the vector `[x(2)-x(1) x(3)-x(2) ... x(n)-x(n-1)]` of length `n-1`. Example:

```
>> x = (1:8).^2
x =
     1     4     9    16    25    36    49    64

>> y = diff(x)
y =
     3     5     7     9    11    13    15

>> z = diff(y)
z =
     2     2     2     2     2
```



In data analysis NaNs are often used to represent “missing values”: data that is not available, which could be the result of the failure of a process being measured (see Section 26.4 for an example). Before carrying out any floating-point computation with the data it is necessary to remove the NaNs, because any computation involving a NaN produces a NaN. This can be done in several ways, all using the function `isnan` (see Section 6.1 for more on `isnan`). For example:

```
>> x = [2 1 NaN -1 6], y = x;
x =
     2     1   NaN    -1     6

>> mean(x)
ans =
     NaN

>> x = x(~isnan(x)), mean(x)
x =
     2     1    -1     6
ans =
     2

>> y(isnan(y)) = [], mean(y)
y =
     2     1    -1     6
ans =
     2
```

*Handled properly,  
empty arrays relieve programmers of the  
nuisance of special cases at beginnings and ends of  
algorithms that construct matrices recursively from submatrices.*

— WILLIAM M. KAHAN (1994)

*Kirk: "You did all this in a day?"  
Carol: "The matrix formed in a day.  
The lifeforms grew later at a substantially accelerated rate."*

— *Star Trek III: The Search For Spock* (Stardate 8130.4)

*I start by looking at a 2 by 2 matrix.  
Sometimes I look at a 4 by 4 matrix.  
That's when things get out of control and too hard.  
Usually 2 by 2 or 3 by 3 is enough, and I look at them,  
and I compute with them, and I try to guess the facts.*

— PAUL R. HALMOS, in *Paul Halmos: Celebrating 50 Years of Mathematics* (1991)

*For the sake of an easy extension of matrix operations,  
we shall introduce one empty matrix of each size ....  
Multiplication of the empty  $0 \times m$ -matrix with any  $m \times n$ -matrix  
is defined to yield the empty  $0 \times n$ -matrix.  
The product of the empty  $m \times 0$ -matrix  
with the empty  $0 \times n$ -matrix, however,  
is defined to be a nonempty matrix,  
namely the zero matrix of size  $m \times n$ .*

— JOSEF STOER and CHRISTOPH WITZGALL,  
*Convexity and Optimization in Finite Dimensions I* (1970)

# Chapter 6

## Operators and Flow Control

### 6.1. Relational and Logical Operators

MATLAB has a logical data type, with the possible values 1, representing true, and 0, representing false. Logicals are produced by relational and logical operators/functions and by the functions `true` and `false`:

```
>> a = true
a =
    logical
     1

>> b = false
b =
    logical
     0

>> c = 1
c =
     1

>> whos
  Name      Size      Bytes  Class  Attributes
  a         1x1         1  logical
  b         1x1         1  logical
  c         1x1         8  double
```

As this example shows, logicals occupy one byte, rather than the eight bytes needed by a `double`.

The relational operators in MATLAB are

<code>==</code>	equal to
<code>~=</code>	not equal to
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>&lt;=</code>	less than or equal to
<code>&gt;=</code>	greater than or equal to

Note that a single `=` denotes assignment and never a test for equality in MATLAB.

Comparisons between scalars produce logical 1 if the relation is true and logical 0 if it is false. Comparisons are also defined between matrices of the same dimension

and between a matrix and a scalar, the result being a matrix of logicals in both cases. For matrix–matrix comparisons corresponding pairs of elements are compared, while for matrix–scalar comparisons the scalar is compared with each matrix element. For example:

```
>> A = [1 2; 3 4]; B = 2*ones(2);

>> A == B
ans =
    2×2 logical array
     0     1
     0     0

>> A > 2
ans =
    2×2 logical array
     0     0
     1     1
```

To test whether arrays *A* and *B* are equal, that is, of the same size with identical elements, the expression `isequal(A,B)` can be used:

```
>> isequal(A,B)
ans =
    logical
     0
```

The function `isequal` is one of many useful logical functions whose names begin with `is`, a selection of which is listed in Table 6.1. See also Table 9.1 for matrix-oriented `is` functions, and for a full list type `doc is*`. For example, `isinf(A)` returns a logical array of the same size as *A* containing true where the elements of *A* are plus or minus `inf` and false where they are not:

```
>> A = [1 inf; -inf NaN];
>> isinf(A)
ans =
    2×2 logical array
     0     1
     1     0
```

The function `isnan` is particularly important because the test `x == NaN` always produces the result 0 (false), even if *x* is a NaN! (A NaN is defined to compare as unequal and unordered with everything.)

Note that an array can be real in the mathematical sense but not real as reported by `isreal`. For `isreal(A)` is true if *A* has *no* imaginary part. Mathematically, *A* is real if every component has *zero* imaginary part. How a mathematically real *A* is formed can determine whether it has an imaginary part or not in MATLAB. The distinction can be seen as follows:

```
>> a = 1;
>> b = complex(1,0);
>> c = 1 + 0i;
```

Table 6.1. *Selected logical is\* functions.*

<code>ischar</code>	Test for char array
<code>isstring</code>	Test for string array
<code>isempty</code>	Test for empty array
<code>isequal</code>	Test if arrays are equal
<code>isequaln</code>	Test if arrays are equal, treating NaNs as equal
<code>isfinite</code>	Detect finite array elements
<code>isfloat</code>	Test for floating-point array (single or double)
<code>isinf</code>	Detect infinite array elements
<code>isinteger</code>	Test for integer array
<code>islogical</code>	Test for logical array
<code>isnan</code>	Detect NaN array elements
<code>isnumeric</code>	Test for numeric array (integer or floating point)
<code>isreal</code>	Test for real array
<code>issorted</code>	Test for sorted vector
<code>isscalar</code>	Test for scalar
<code>iscolumn</code>	Test for column vector
<code>isrow</code>	Test for row vector
<code>isvector</code>	Test for vector
<code>ismatrix</code>	Test for matrix
<code>issparse</code>	Test for sparse matrix

```
>> [a b c]
ans =
    1.0000 + 0.0000i    1.0000 + 0.0000i    1.0000 + 0.0000i

>> whos a b c
Name      Size      Bytes  Class      Attributes

a         1x1         8  double
b         1x1        16  double    complex
c         1x1         8  double

>> [isreal(a), isreal(b), isreal(c)]
ans =
    1x3 logical array
     1     0     1
```

The logical operators in MATLAB are shown in Table 6.2. Like the relational operators, the `&`, `|`, and `~` operators produce matrices of logical zeros and ones when one of the arguments is a matrix. When applied to a vector, the `all` function returns 1 if all the elements of the vector are nonzero and 0 otherwise. The `any` function is defined in the same way, with “any” replacing “all”. Examples:

```
>> x = [-1 1 1]; y = [1 2 -3];

>> x>0 & y>0
```



The second feature of these “double barreled” operators is that they short-circuit the evaluation of the logical expressions, where possible. In the compound expression *expr1* && *expr2*, if *expr1* evaluates to false then *expr2* is not evaluated. Similarly, in *expr1* || *expr2*, if *expr1* evaluates to true then *expr2* is not evaluated. Short-circuiting saves computation, but it also enables warnings and errors to be avoided. For example, a statement beginning

```
if (m >= 0 && m <= 1) && ellipj(u,m) < 0.5
```

avoids calling `ellipj` with an illegal value for the second argument.

The precedence of arithmetic, relational, and logical operators is summarized in Table 6.3 (which is based on the information provided by `help precedence`). For operators of equal precedence MATLAB evaluates from left to right. Precedence can be overridden by using parentheses. Note, in particular, that logical `and` has higher precedence than logical `or`, so a logical expression of the form

```
x | y & z
```

is equivalent to

```
x | (y & z)
```

It is good practice to insert parentheses to make the intention completely clear. For example, the test

```
if 0 < x < 1, disp('Test passed'), end
```

looks as if it is testing whether `x` lies on the interval  $(0, 1)$ . However, rewriting the test in the equivalent form

```
if (0 < x) < 1, disp('Test passed;'), end
```

and noting that  $(0 < x)$  evaluates to 0 or 1 makes it clear that the test is passed when `x` is less than or equal to 0.

For matrices, `all` returns a row vector containing the result of `all` applied to each column. Therefore `all(all(A==B))` is another way of testing equality of the matrices `A` and `B`. The `any` function works in the corresponding way. Thus, for example, `any(any(A==B))` has the value 1 if `A` and `B` have any equal elements and 0 otherwise. Alternatives are `all(A(:)==B(:))` and `any(A(:)==B(:))`.

The `find` command returns the indices corresponding to the nonzero elements of a vector. For example,

```
>> x = [-3 1 0 -inf 0];
>> f = find(x)
f =
     1     2     4
```

The result of `find` can then be used to extract just those elements of the vector:

```
>> x(f)
ans =
    -3     1   -Inf
```

With `x` as above, we can use `find` to obtain the finite elements of `x`,

Table 6.3. *Operator precedence.*

Precedence level	Operator
1 (highest)	Parentheses ()
2	Transpose (.'), power (.^), complex conjugate transpose ('), matrix power (^)
3	Power or matrix power with unary minus (.^-, ^-), unary plus (.^+, ^+), or logical negation (.^~, ^~)
4	Unary plus (+), unary minus (-), logical negation (~)
5	Multiplication (.*), right division (./), left division (.\), matrix multiplication (*), matrix right division (/), matrix left division (\)
6	Addition (+), subtraction (-)
7	Colon operator (:)
8	Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
9	Logical and (&)
10	Logical or ( )
11	Logical short-circuit and (&&)
12 (lowest)	Logical short-circuit or (  )

```
>> x(find(isfinite(x)))
ans =
    -3     1     0     0
```

and to replace negative components of `x` by zero:

```
>> x(find(x < 0)) = 0
x =
     0     1     0     0     0
```

When `find` is applied to a matrix `A`, the index vector corresponds to `A` regarded as a vector of the columns stacked one on top of the other (that is, `A(:)`), and this vector can be used to index into `A`. In the following example we use `find` to set to zero those elements of `A` that are less than the corresponding elements of `B`:

```
>> A = [4 2 16; 12 4 3], B = [12 3 1; 10 -1 7]
A =
     4     2    16
    12     4     3
B =
    12     3     1
    10    -1     7

>> f = find(A<B)
f =
     1
     3
```



6

```
>> A(f) = 0
A =
     0     0    16
    12     4     0
```

An alternative usage of `find` for matrices is `[i,j] = find(A)`, which returns vectors `i` and `j` containing the row and column indices of the nonzero elements.

The results of the MATLAB logical operators and logical functions are logical arrays of 0s and 1s. Logical arrays can also be created by applying the function `logical` to a numeric array; nonzero values other than 1 that are converted to 1 result in a warning message. Logical arrays and numeric arrays can both be used for subscripting, but with an important difference: logical arrays pick out elements where the subscript is true, whereas numeric arrays pick out elements indexed by the subscript. An example should make this distinction clear:

```
>> clear
>> y = [1 2 0 -3 0]
y =
     1     2     0    -3     0

>> i1 = (y ~= 0)
i1 =
    1×5 logical array
     1     1     0     1     0

>> i2 = [1 1 0 1 0]
i2 =
     1     1     0     1     0

>> y(i1)
ans =
     1     2    -3
```

```
>> y(i2)
Subscript indices must either be real positive integers or logicals.
```

```
>> whos i1 i2
Name      Size      Bytes  Class    Attributes

i1        1×5         5  logical
i2        1×5        40  double
```

```
>> isequal(i1,i2)
ans =
    logical
     1
```

```
>> i3 = [1 2 4]; y(i3)
ans =
```

```
1    2    -3
```

Although the numeric array `i2` has the same elements as the logical array `i1` (and compares as equal with it), only `i1` can be used for subscripting. To achieve the required subscripting effect with a numerical array, `i3` must be used.

A call to `find` can sometimes be avoided when its argument is a logical array. In our example on p. 75, `x(find(isfinite(x)))` can be replaced by `x(isfinite(x))`.

Addition and multiplication can be done on logicals, and they can be used in arithmetic expressions containing doubles. The result is always a double:

```
>> a = true; b = false; c = 2*a + b, class(c)
c =
     2
ans =
double
```

However, many other arithmetic operations fail:

```
>> b/a
Undefined operator '/' for input arguments of type 'logical'.
```

## 6.2. Flow Control

MATLAB has five flow control structures: the `if` statement, the `for` loop, the `while` loop, the `switch` statement, and the `try` statement. The simplest form of the `if` statement is

```
if expression
    statements
end
```

where the statements are executed if the elements of *expression* are all nonzero. For example, the following code swaps `x` and `y` if `x` is greater than `y`:

```
if x > y
    temp = y;
    y = x;
    x = temp;
end
```

When an `if` statement is followed on its line by further statements, a comma is needed to separate the `if` from the next statement:

```
if x > 0, x = sqrt(x); end
```

Statements to be executed only if *expression* is false can be placed after `else`, as in the example

```
e = exp(1);
if 2^e > e^2
    disp('2^e is bigger')
else
    disp('e^2 is bigger')
end
```

Finally, one or more further tests can be added with `elseif` (note that there must be no space between `else` and `if`):

```
if isnan(x)
    disp('Not a Number')
elseif isinf(x)
    disp('Plus or minus infinity')
else
    disp('A ''regular'' floating-point number')
end
```

In the third `disp`, `''` prints as a single quote `'`.

The `for` loop is one of the most useful MATLAB constructs, although, as discussed in Section 23.2, experienced programmers who are concerned with producing compact and fast code try to avoid `for` loops wherever possible. The syntax is

```
for variable = expression
    statements
end
```

Usually, *expression* is a vector of the form `i:s:j` (see Section 5.2). The statements are executed with *variable* equal to each element of *expression* in turn. For example, the sum of the first 25 terms of the harmonic series  $1/i$  is computed by

```
>> s = 0;
>> for i = 1:25, s = s + 1/i; end, s
s =
    3.8160
```

Another way to define *expression* is using the square bracket notation:

```
>> for x = [pi/6 pi/4 pi/3], disp([x, sin(x)]), end
    0.5236    0.5000
    0.7854    0.7071
    1.0472    0.8660
```

Multiple `for` loops can of course be nested, in which case indentation helps to improve the readability. The following code forms the 5-by-5 symmetric matrix `A` with  $(i, j)$  element  $i/j$  for  $j \geq i$ :

```
n = 5; A = eye(n);
for j = 2:n
    for i = 1:j-1
        A(i,j) = i/j;
        A(j,i) = i/j;
    end
end
```

The *expression* in the `for` loop can be a matrix, in which case *variable* is assigned the columns of *expression* from first to last. For example, to set `x` to each of the unit vectors in turn, we can write `for x = eye(n), ..., end`.

The `while` loop has the form

```
while expression
    statements
end
```

The *statements* are executed as long as *expression* is true. The following example approximates the smallest nonzero floating-point number:

```
>> x = 1; while x > 0, xmin = x; x = x/2; end, xmin
xmin =
    4.9407e-324
```

A `while` loop can be terminated with the `break` statement, which passes control to the first statement after the corresponding `end`. An infinite loop can be constructed using `while 1, ..., end`, which is useful when it is not convenient to put the exit test at the top of the loop. (Note that, unlike some other languages, MATLAB does not have a “repeat-until” loop.) We can rewrite the previous example less concisely as

```
x = 1;
while 1
    xmin = x;
    x = x/2;
    if x == 0, break, end
end
xmin
```

The `break` statement can also be used to exit a `for` loop. In a nested loop a `break` exits to the loop at the next higher level.

The `continue` statement causes execution of a `for` or `while` loop to pass immediately to the next iteration of the loop, skipping the remaining statements in the loop. As a trivial example,

```
for i=1:10
    if i < 5, continue, end
    disp(i)
end
```

displays the integers 5 to 10. In more complicated loops the `continue` statement can be useful to avoid long-bodied `if` statements.

The `switch` construct consists of “`switch expression`” followed by a list of “`case expression statements`”, optionally ending with “`otherwise statements`” and followed by `end`. The switch expression is evaluated and the statements following the first matching `case` expression are executed. If none of the cases produces a match then the statements following `otherwise` are executed. The next example evaluates the  $p$ -norm of a vector  $x$  (i.e., `norm(x,p)`) for just three values of  $p$ :

```
switch p
    case 1
        y = sum(abs(x));
    case 2
        y = sqrt(x'*x);
    case inf
```

```

        y = max(abs(x));
    otherwise
        error('p must be 1, 2 or inf.')
end

```

(The `error` function is described in Section 14.1.) The expression following `case` can be a list of values enclosed in parentheses (a cell array—see Section 18.7). The switch expression then matches any value in the list:

```

x = input('Enter a real number: ');
switch x
    case {inf,-inf}
        disp('Plus or minus infinity')
    case 0
        disp('Zero')
    otherwise
        disp('Nonzero and finite')
end

```

C programmers should note that the MATLAB `switch` construct behaves differently from that in C: once a MATLAB `case` group expression has been matched and its statements executed, control is passed to the first statement after the `switch`, with no need for `break` statements.

The final control structure is the `try` statement, which has the form

```

try
    statements
catch exception
    statements
end

```

The statements after the `try` are executed and if an error occurs execution jumps immediately to the statements after the `catch`.

Suppose that within a program you want to read in an image file `image.jpg`, and that that image does not exist in the current directory. A statement

```
imread image.jpg
```

generates an error

```

Error using imread (line 349)
File "image.jpg" does not exist.

```

and execution terminates. However, we may want execution of our program to continue. In this circumstance the code

```

try
    imread image.jpg
catch
    disp('Image could not be read')
end

```

displays `Image could not be read` and execution continues after the `end`. Note that an alternative in this example is to check the existence of the file using `exist` (see Section 7.3) before attempting to load it, though this does not guard against a file of the expected name having the wrong format.

The *exception* term allows the action taken in the `catch` section to depend on the nature of the error, based on the information in the object in question. See `doc try` for examples.

*Kirk: "Well, Spock, here we are.  
Thanks to your restored memory, a little bit of good luck,  
we're walking the streets of San Francicso,  
looking for a couple of humpback whales.  
How do you propose to solve this minor problem?"  
Spock: "Simple logic will suffice."  
— Star Trek IV: The Voyage Home (Stardate 8390)*

*Things equally high on the pecking order get evaluated from left to right.  
When in doubt, throw in some parentheses and be sure.  
Only use good quality parentheses with nice round sides.  
— ROGER EMANUEL KAUFMAN, A FORTRAN Coloring Book (1978)*

# Chapter 7

## Program Files

### 7.1. Scripts and Functions

Although you can do many useful computations working entirely at the MATLAB command line, sooner or later you will need to write MATLAB programs: the equivalents of programs, functions, subroutines, and procedures in other programming languages. Collecting together a sequence of commands into a program opens up many possibilities, including

- experimenting with an algorithm by editing a file, rather than retyping a long list of commands,
- making a permanent record of a numerical experiment,
- building up utilities that can be reused at a later date,
- exchanging programs with others.

Many useful programs that have been written by enthusiasts can be found online, for example in File Exchange at MATLAB Central.

MATLAB programs are contained in program files, of which there are four types.

**Scripts** have no input or output arguments and operate on variables in the workspace.

**Live scripts** contain both MATLAB commands and the output that they produce, and are created and viewed in the Live Editor. They are described in Section 16.7.

**Functions** contain a `function` definition line and can accept input arguments and return output arguments, and their internal variables are local to the function (unless declared `global`).

**Classes** contain the definition of a class and the methods defined on it. They are discussed in Chapter 19.

Scripts, functions, and classes have a `.m` extension, while live scripts have a `.mlx` extension.

A script enables you to store a sequence of commands that are to be used repeatedly or will be needed at some future time. A simple example of a script, `marks.m`, was given in Section 2.2. As another example we describe a script for playing “eigenvalue roulette” [43], which is based on counting how many eigenvalues of a random real matrix are real. If the matrix `A` is real and of dimension 8 then the number of real eigenvalues is 0, 2, 4, 6, or 8 (the number must be even, since nonreal eigenvalues appear in complex conjugate pairs). The short script

Listing 7.1. *Script rouldist.*

```

%ROULDIST      Empirical distribution of number of real eigenvalues.

k = 1000;
wheel = zeros(k,1);
for i = 1:k
    A = randn(8);
    % Count number of eigenvalues with imag. part < tolerance.
    wheel(i) = sum(abs(imag(eig(A)))<.0001);
end
histogram(wheel,[0 2 4 6 8]);

```

```

%SPIN
% Counts number of real eigenvalues of random matrix.
A = randn(8); sum(abs(imag(eig(A))) < 0.0001)

```

creates a random normally distributed 8-by-8 matrix and counts how many eigenvalues have imaginary parts with absolute value less than the (somewhat arbitrary) threshold  $10^{-4}$ . The first two lines of this script begin with the % symbol and hence are comment lines. Whenever MATLAB encounters a % it ignores the remainder of the line. This allows you to insert text that makes the script easier for humans to understand. Assuming this script exists as a file `spin.m`, typing `spin` is equivalent to typing the two commands `A = randn(8);` and `sum(abs(imag(eig(A))) < 0.0001)`. This “spins the roulette wheel,” producing one of the five answers 0, 2, 4, 6, and 8. Each call to `spin` produces a different random matrix and hence may give a different answer:

```

>> spin
ans =
    2

>> spin
ans =
    4

```

To get an idea of the probability of each of the five outcomes you can run the script `rouldist` in Listing 7.1. It generates 1000 random matrices and plots a histogram of the distribution of the number of real eigenvalues. Figure 7.1 shows a possible result. (The exact probabilities are known and are given in [43], [44].) Note that to make `rouldist` more readable we have used spaces to indent the `for` loop and inserted a blank line before the first command.

Functions enable you to extend the MATLAB language by writing your own functions that accept and return arguments. They can be used in exactly the same way as existing MATLAB functions such as `sin`, `eye`, `size`, etc.

Listing 7.2 shows a simple function that evaluates the largest element in absolute value of a matrix. This example illustrates a number of features. The first line begins with the keyword `function` followed by the output argument, `y`, and the = symbol. On the right of = comes the function name, `maxentry`, followed by the input argument, `A`, within parentheses. (In general there can be any number of input and



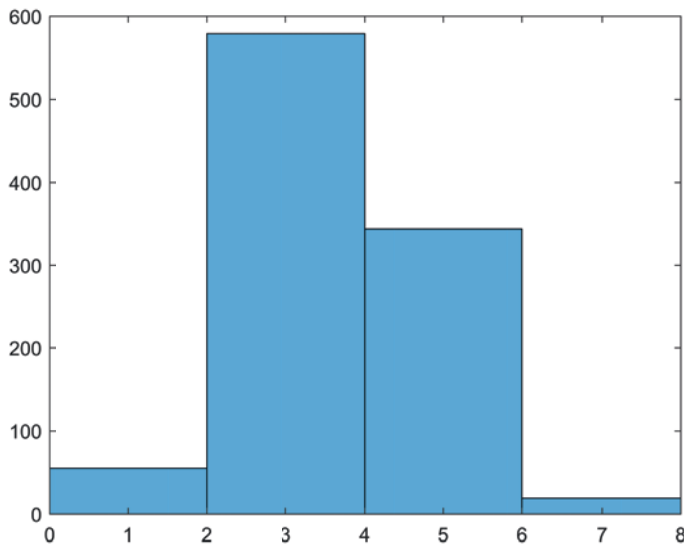


Figure 7.1. *Histogram produced by rouldist.*

output arguments.) The function name must be the same as the name of the `.m` file in which the function is stored—in this case the file must be named `maxentry.m`.

The second line of a function file is called the H1 (help 1) line. It should be a comment line of a special form: a line beginning with a `%` character, followed without any space by the function name in capital letters, followed by one or more spaces and then a brief description. The description should begin with a capital letter, end with a period, and omit the words “the” and “a”. All the comment lines from the first comment line up to the first noncomment line (usually a blank line, for readability of the source code) are displayed when `help function_name` is typed. Therefore these lines should describe the function and its arguments. It is conventional to capitalize function names in these comment lines. For the `maxentry.m` example, we have

```
>> help maxentry
maxentry   Largest absolute value of matrix entries.
maxentry(A) is the maximum of the absolute values
of the entries of A.
```

Note that MATLAB converts the function name to lowercase and it is displayed in bold in the Command Window.

We strongly recommend documenting *all* your function files in this way, however short they may be. It is often useful to record in comment lines the date when the function was first written and to note any subsequent changes that have been made. The `help` command works in a similar manner on script files, displaying the initial sequence of comment lines.

The function `maxentry` is called just like any other MATLAB function:

```
>> maxentry(1:10)
ans =
    10
```

Listing 7.2. *Function maxentry.*

```
function y = maxentry(A)
%MAXENTRY   Largest absolute value of matrix entries.
%   MAXENTRY(A) is the maximum of the absolute values
%   of the entries of A.

y = max(max(abs(A)));
```

```
>> mx = maxentry(magic(4))
mx =
    16
```

The function `flogist` shown in Listing 7.3 illustrates the use of multiple input and output arguments. This function evaluates the scalar logistic function  $x(1 - ax)$  and its derivative with respect to  $x$ . The two output arguments `f` and `fprime` are enclosed in square brackets. When calling a function with multiple input or output arguments it is not necessary to request all the output arguments, but arguments must be dropped starting at the end of the list. If more than one output argument is requested the arguments must be listed within square brackets. Examples of usage are

```
>> f = flogist(2,.1)
f =
    1.6000

>> [f,fprime] = flogist(2,.1)
f =
    1.6000
fprime =
    0.6000
```

A technical point of note in function `flogist` is that array multiplication (`.*`) is used in the statement `f = x.*(1 - a*x)`. So, if a vector or matrix is supplied for `x`, the function is evaluated at each element simultaneously:

```
>> flogist(1:4,2)
ans =
    -1    -6   -15   -28
```

Another function using array multiplication is `cheby` in Listing 7.4, which is used in Chapter 17 to produce Figure 17.9. The  $k$ th Chebyshev polynomial,  $T_k(x)$ , can be defined by the recurrence

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x), \quad \text{for } k \geq 2,$$

with  $T_0(x) = 1$  and  $T_1(x) = x$ . The function `cheby` accepts a vector `x` and an integer `p` and returns a matrix `Y` whose  $i$ th row gives the values of  $T_0(x), T_1(x), \dots, T_{p-1}(x)$  at  $x = x(i)$ .

Listing 7.3. *Function* flogist.

```
function [f,fprime] = flogist(x,a)
%FLOGIST    Logistic function and its derivative.
% [F,FPRIME] = FLOGIST(x,a) evaluates the logistic
% function F(x) = x.*(1 - a*x) and its derivative FPRIME
% at the matrix argument x, where a is a scalar parameter.

f = x.*(1 - a*x);
fprime = 1 - 2*a*x;
```

Listing 7.4. *Function* cheby.

```
function Y = cheby(x,p)
%CHEBY    Chebyshev polynomials.
% Y = CHEBY(x,p) evaluates the first p Chebyshev polynomials
% at the vector x. The k'th column of Y contains the
% Chebyshev polynomial of degree k-1 evaluated at x.

Y = ones(length(x),p);
x = x(:); % Ensure x is a column vector.
if p == 1, return, end

Y(:,2) = x;
for k = 3:p
    Y(:,k) = 2*x.*Y(:,k-1) - Y(:,k-2);
end
```

Note that `cheby` uses the `return` command, which causes an immediate return from the function. It is not necessary (or usual) to put a `return` statement at the end of a function or script, unlike in some other programming languages.

A more complicated function is `sqrtn`, shown in Listing 7.5. Given  $a > 0$ , it implements the Newton iteration for  $\sqrt{a}$ ,

$$x_{k+1} = \frac{1}{2} \left( x_k + \frac{a}{x_k} \right), \quad x_1 = a,$$

printing the progress of the iteration. Output is controlled by the `fprintf` command, which is described in Section 13.2. Examples of usage are

```
>> [x,iter] = sqrtn(2)
k          x_k          rel. change
1:  1.5000000000000000e+00  3.33e-01
2:  1.4166666666666665e+00  5.88e-02
3:  1.4142156862745097e+00  1.73e-03
4:  1.4142135623746899e+00  1.50e-06
5:  1.4142135623730949e+00  1.13e-12
6:  1.4142135623730949e+00  0.00e+00
x =
    1.4142
iter =
     6

>> x = sqrtn(2,1e-4);
k          x_k          rel. change
1:  1.5000000000000000e+00  3.33e-01
2:  1.4166666666666665e+00  5.88e-02
3:  1.4142156862745097e+00  1.73e-03
4:  1.4142135623746899e+00  1.50e-06
```

This function illustrates the use of optional input arguments. The function `nargin` returns the number of input arguments supplied when the function was called and enables default values to be assigned to arguments that have not been specified. In this case, if the call to `sqrtn` does not specify a value for `tol` then `eps` is assigned to `tol`.

An analogous function `nargout` returns the number of output arguments requested. In this example there is no need to check `nargout`, because `iter` is computed by the function whether or not it is requested as an output argument. Some functions gain efficiency by inspecting `nargout` and computing only those output arguments that are requested (for example `svd`, described in Section 9.7). To illustrate, Listing 7.6 shows how the `marks` script on p. 24 can be rewritten as a function. Its usage is illustrated by

```
>> exmark = [12 0 5 28 87 3 56];

>> x_sort = marks2(exmark)
x_sort =
     0     3     5    12    28    56    87
```

Listing 7.5. *Function sqrtn.*

```
function [x,iter] = sqrtn(a,tol)
%SQRN    Square root of a scalar by Newton's method.
%  x = SQRN(a,tol) computes the square root of the scalar
%  a by Newton's method (also known as Heron's method).
%  a is assumed to be nonnegative.
%  tol is a convergence tolerance (default eps).
%  [x,iter] = SQRN(a,tol) returns also the number of
%  iterations iter for convergence.

if nargin < 2, tol = eps; end

x = a;
iter = 0;
xdiff = inf;
fprintf(' k           x_k           rel. change\n')

while xdiff > tol
    iter = iter + 1;
    xold = x;
    x = (x + a/x)/2;
    xdiff = abs(x-xold)/abs(x);
    fprintf('%2.0f: %20.16e %9.2e\n', iter, x, xdiff)
    if iter > 50
        error('Not converged after 50 iterations.')
    end
end
end
```

Listing 7.6. *Function marks2.*

```
function [x_sort,x_mean,x_med,x_std] = marks2(x)
%MARKS2    Statistical analysis of marks vector.
%    Given a vector of marks x,
%    [x_sort,x_mean,x_med,x_std] = MARKS2(x) computes a
%    sorted marks list and the mean, median, and standard deviation
%    of the marks.

x_sort = sort(x);
if nargout > 1, x_mean = mean(x); end
if nargout > 2, x_med  = median(x); end
if nargout > 3, x_std  = std(x); end
```

```
>> [~,~,x_med] = marks2(exmark)
x_sort =
     0     3     5    12    28    56    87
x_mean =
    27.2857
x_med =
    12
```

What if we want to obtain the standard deviation but not the other three statistics? We can discard the unwanted outputs using the tilde symbol:

```
>> [~,~,~,x_med] = marks2(exmark)
x_med =
    32.8010
```

The first three outputs are still computed, but they do not enter the main workspace. Sometimes it is necessary to use a function call of the form

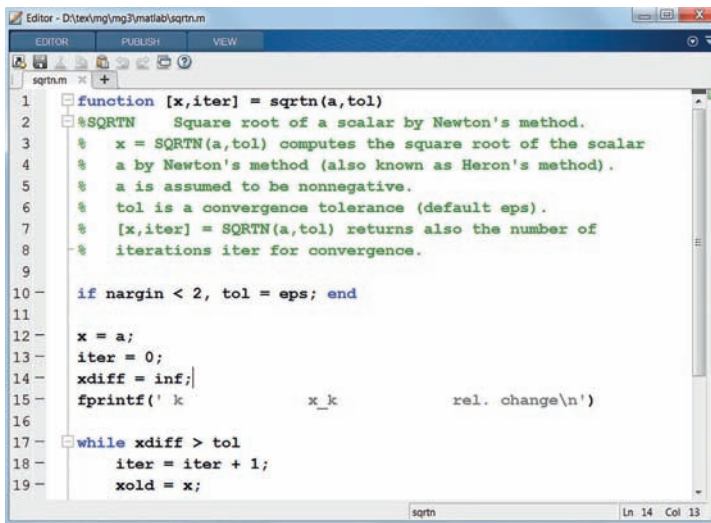
```
[a,b,~] = myfun(...)
```

It may seem strange to ask for a third output argument and then discard it, but this usage can be appropriate for functions whose behavior depends on the number of output arguments (type `edit(condest)` to see an example involving the function `lu`).

## 7.2. Naming and Editing Program Files

Program files share with variables the naming restrictions described on p. 31. In particular, file names are case sensitive. For how to check whether a tentative name already exists, see the next section.

To create and edit program files you have two choices. You can use the built-in MATLAB Editor/Debugger, shown in Figure 7.2. This is invoked by typing `edit` at the command prompt or from the New or Open menu options on the Home tab of the MATLAB Toolstrip. The MATLAB Editor has various features to aid in editing program files, including automatic indentation of loops and `if` structures,

Figure 7.2. *MATLAB Editor/Debugger.*

color syntax highlighting, bracket and quote matching, and the ability to comment out blocks of code and fold sections of code. These and other features can be turned off or customized via the MATLAB Preferences-Editor/Debugger menu. Alternatively, you can use whatever text editor you normally use (if it is a word processor you need to ensure that you save the files in plain text form and with a `.m` extension).

An advantage of using the MATLAB Editor is that it contains a Run button on its toolbar that can be used to run the program being edited and which turns into a Pause button when the code is running. If the Pause button is pressed then the debugger is entered and the user can inspect variables and continue or abort execution (see Section 14.3 for details of the debugger).

A very useful feature is block commenting: a block of code can be commented out (no matter what editor you are using) by surrounding it by two special comment lines:

```

%{
<block of code>
%}

```

Here, `<block of code>` denotes an arbitrary number of lines of code. MATLAB considers all lines between `%{` and `%}` to be comments, even those that are not individually commented out with a leading `%` sign. Block comments can be nested, so that a block comment can be extended without losing the original block comment.

### 7.3. Working with Program Files and the MATLAB Path

Many MATLAB functions are `.m` files residing on the disk, while others are built into the MATLAB interpreter. MATLAB looks for program files in the current directory and then on the search path, which is a list of directories. A program file is available only if it is in the current directory or on the search path. The rules for deciding

which function to call when there is more than one function of a given name in the current scope can be found by searching the documentation for “function precedence order”.

Type `path` to see the current search path. The path can be set and added to with the `path` and `addpath` commands, or from the tool that is invoked by clicking Set Path on the Home tab or by typing `pathtool`.

Several commands can be used to search the path. The `what` command lists the MATLAB files (and other MATLAB-related files) in the current directory, grouped by type; `what dirname` lists the MATLAB files in the directory `dirname` on the path. The command `lookfor keyword` (illustrated on p. 28) searches the path for program files containing `keyword` in their H1 line (the first line of help text). All the comment lines displayed by the `help` command can be searched using `lookfor keyword -all`. Some MATLAB functions use comment lines after the initial block of comment lines to provide further information, such as bibliographic references (an example is `fminsearch`). This information can be accessed using `type` but is not displayed by `help`.

Typing `which foo` displays the pathname of the function `foo` or declares it to be `not found`. This is useful if you want to know in which directory on the path a program file is located. If you suspect there may be more than one program file with a given name on the path you can use `which foo -all` to display all of them.

A script (but not a function) not on the search path can be invoked by typing `run` followed by a statement in which the full pathname to the script is given.

You may list the program file `foo.m` to the screen with `type foo` or `type foo.m`. (If there is a file called `foo` then `type foo` will list `foo` rather than `foo.m`.) Preceding a `type` command with `more on` will cause the listing to be displayed a page at a time; `more off` turns off paging.

Before writing a program file it is important to check whether the name you are planning to give it is the name of an existing program file or built-in function. This can be done in several ways: using `which` as just described, using `type` (e.g., `type lu` produces the response `lu is a built-in function`), using `help`, or using the function `exist`. The command `exist('myname')` tests whether `myname` is a variable in the workspace, a file (with various possible extensions, including `.m`) on the path, or a directory. A result of 0 means no matches were found, while the numbers 1–8 indicate a match; see `help exist` for the precise meaning of these numbers. You should also avoid using MATLAB keywords for program file or variable names. A list of keywords can be obtained with the `iskeyword` function: these are `break`, `case`, `catch`, `classdef`, `continue`, `else`, `elseif`, `end`, `for`, `function`, `global`, `if`, `otherwise`, `parfor`, `persistent`, `return`, `spmd`, `switch`, `try`, `while`.

When a function residing on the path is invoked for the first time it is compiled into memory. MATLAB can usually detect when a function has changed and then automatically recompiles it when it is invoked.

To clear function `fun` from memory, type `clear fun`. To clear all functions type `clear functions`.

## 7.4. Startup

When MATLAB starts it executes the script `matlabrc.m` (located in the directory `toolbox\local` off the MATLAB root). This script sets various defaults and then calls the script `startup.m`, if it exists on the MATLAB search path. The `startup` file



is the place to make your own default settings and to add directories to the MATLAB path. This file is best placed in your MATLAB startup directory (the directory that is initially the current directory in MATLAB). To find how to change the startup directory, search for “MATLAB Startup Folder” in the Help browser. A slightly shortened version of our `startup.m` script is as follows. It uses the function `ispc` to test if MATLAB is running on a PC (there are also analogous functions `ismac` and `isunix`).

```
%STARTUP    Startup file.

if ispc
    prefix = 'd:\';
else % Mac or Unix.
    prefix = '~/';
end

cd([prefix 'matlab'])

% Save original path, in case want to restore standard setup.
path_org = path;

mypaths = {%
'matlab/matrixcomp'
'matlab/misc'
'matlab/book'
'matlab/tools'
'matlab/mats'
'tex/expab/matlab'
'tex/unwinding/matlab'};

for i=1:length(mypaths)
    addpath([prefix mypaths{i}], '-end')
end
clear i mypaths
format compact
```

## 7.5. Command/Function Duality

User-written functions are usually called by giving the function name followed by a list of arguments in parentheses. Yet some built-in MATLAB functions, such as `type` and `what` described in the previous section, are normally called with arguments separated from the function name by spaces. This is not an inconsistency but an illustration of command/function duality. Consider the function

```
function comfun(x,y,z)
%COMFUN    Illustrative function with three string arguments.
disp(x), disp(y), disp(z)
```

We can call it with string arguments in parentheses (functional form) or with the string arguments separated by spaces after the function name (command form):

```

>> comfun('ab','cd','ef')
ab
cd
ef

>> comfun ab cd ef
ab
cd
ef

```

The two invocations are equivalent. Other examples of command/function duality are (with the first in each pair being the most commonly used)

```

format long, format('long')
disp('Hello'), disp Hello
diary mydiary, diary('mydiary')
warning off, warning('off')

```

Note, however, that the command form should be used only for functions that take string arguments. In the example

```

>> mean 2
ans =
    50

```

MATLAB interprets 2 as a string and `mean` is applied to the ASCII value of 2, namely 50. Note also that the command form can be used only when no output argument is requested. Thus `x = mean 2` gives an error.

#### ITERATING MATLAB COMMANDS

Command/function duality has the interesting consequence that many commands can be iterated, in some cases only a certain number of times. For example,

```

and and and
isa isa isa

```

are legal, but two or four iterations of these commands give an error. The `char` function can be iterated an arbitrary number of times:

```

>> char char char char
ans =
char
char
char

```

So can the `menu` command, with interesting results—try it! For more, see [78].

```
>> why
Cleve insisted on it.
>> why
Jack knew it was a good idea.
— MATLAB
```

*Replace repetitive expressions by calls to a common function.*

— BRIAN W. KERNIGHAN and P. J. PLAUGER,  
*The Elements of Programming Style* (1978)

*Much of MATLAB's power is derived from its extensive set of functions. . .*

*Some of the functions are intrinsic,  
or "built-in" to the MATLAB processor itself.*

*Others are available in the library of external M-files distributed with MATLAB. . .  
It is transparent to the user whether a function is intrinsic or contained in an M-file.*

— 386-MATLAB User's Guide (1989)

# Chapter 8

## Graphics

MATLAB has powerful and versatile graphics capabilities. Figures of many types can be generated with relative ease and their “look and feel” is highly customizable. In this chapter we cover the basic use of the most popular MATLAB tools for graphing two- and three-dimensional data; Chapter 17 delves more deeply into the innards of MATLAB graphics. Our philosophy of teaching a useful subset of the MATLAB language, without attempting to be exhaustive, is particularly relevant to this chapter. The final section hints at what we have left unsaid.

Our emphasis in this chapter is on generating graphics at the command line or in programs, but existing figures can also be modified and annotated interactively using the Plot Editor. The Plot Editor can be invoked from the Tools menu and toolbar of the figure window (see `doc plottedit`).

The figures in this chapter—and throughout the book—are the results of saving the figure window generated by the commands shown. We have not postprocessed the MATLAB figures to make them more readable on the printed page. For more on this issue, see Section 8.4. We also note that the limitations of the (CMYK-based) printing process mean that colors can appear somewhat different in print to how they appear on the screen.

Although all the examples in this chapter are concerned with plotting numeric arrays or functions, we note that certain graphics commands work with other data types, too. For example, `plot` works with graph objects (see Chapter 21) and `histogram` works with categorical arrays (see Section 18.4).

### 8.1. Two-Dimensional Graphics

#### 8.1.1. Basic Plots

The `plot` function can be used for simple “join-the-dots”  $x$ - $y$  plots. Typing

```
>> x = [1.5 2.2 3.1 4.6 5.7 6.3 9.4];  
>> y = [2.3 3.9 4.3 7.2 4.5 6.1 1.1];  
>> plot(x,y)
```

produces the left-hand picture in Figure 8.1, where the points `x(i)`, `y(i)` are joined in sequence. MATLAB opens a figure window (unless one has already been opened as a result of a previous command) in which to draw the picture. In this example, default values are used for a number of features, including the ranges for the  $x$ - and  $y$ -axes, the spacing of the axis tick marks, and the color and type of the line used for the plot.

More generally, we could replace `plot(x,y)` with `plot(x,y,string)`, where *string* combines up to three elements that control the color, marker, and line style. For

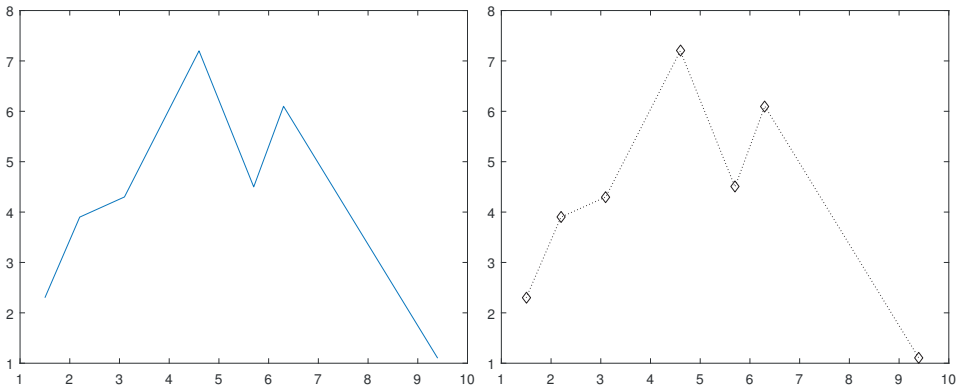


Figure 8.1. *Simple x-y plots. Left: default. Right: nondefault.*



Figure 8.2. *Default color order for lines and markers.*

example, `plot(x,y,'r*--')` specifies that a red asterisk is to be placed at each point  $x(i)$ ,  $y(i)$  and that the points are to be joined by a red dashed line, whereas `plot(x,y,'y+')` specifies a yellow cross marker with no line joining the points. Table 8.1 lists the options available. The right-hand picture in Figure 8.1 was produced with `plot(x,y,'kd:')`, which gives a black dotted line with a diamond marker. The three elements in *string* may appear in any order, so, for example, `plot(x,y,'ms--')` and `plot(x,y,'s--m')` are equivalent.

More than one set of data can be passed to `plot`. For example,

```
plot(x,y,'g-',b,c,'r--')
```

superimposes plots of  $x(i)$ ,  $y(i)$  and  $b(i)$ ,  $c(i)$  with solid green and dashed red line styles, respectively.

If the color is not specified then MATLAB cycles through the default set of seven colors shown in Figure 8.2. Alternatively, the color can be specified by setting the property `Color` to an RGB (red, green, blue) triple `[r,g,b]`, where the coordinates are on the interval `[0,1]`. For example:

```
plot(x,y,'Color',[1 0.75 0.5])
```

The default colors shown in Figure 8.2 do not have names, but their RGB coordinates can be obtained and used as in the example

```
a = get(groot,'defaultAxesColorOrder'); % 7-by-3 matrix.
plot(x,y,'-x','Color',a(4,:)) % Fourth color from default order.
```

The `plot` command also accepts matrix arguments. If  $x$  is an  $m$ -vector and  $Y$  is an  $m$ -by- $n$  matrix, `plot(x,Y)` superimposes the plots created by  $x$  and each column of  $Y$ . Similarly, if  $X$  and  $Y$  are both  $m$ -by- $n$ , `plot(X,Y)` superimposes the plots created by corresponding columns of  $X$  and  $Y$ . If nonreal numbers are supplied to `plot` then imaginary parts are generally ignored. The only exception to this rule

Table 8.1. *Options for the plot command.*

Color		Marker		Line style	
r	Red	o	Circle	-	Solid line (default)
g	Green	*	Asterisk	--	Dashed line
b	Blue	.	Point	:	Dotted line
c	Cyan	+	Plus	-.	Dash-dot line
m	Magenta	x	Cross		
y	Yellow	s	Square		
k	Black	d	Diamond		
w	White	^	Upward triangle		
		v	Downward triangle		
		<	Left triangle		
		>	Right triangle		
		p	Five-point star		
		h	Six-point star		

Table 8.2. *RGB coordinates for the colors in Table 8.1, as used for setting the Color property. The lightness of each color can be reduced by multiplying the color vector by a scalar on the interval [0, 1]. A color vector [x x x] produces gray for  $x \in [0, 1]$ .*

Color	color vector	Color	color vector
Red	[1 0 0]	Cyan	[0 1 1]
Green	[0 1 0]	Magenta	[1 0 1]
Blue	[0 0 1]	Yellow	[1 1 0]
		Black	[0 0 0]
		White	[1 1 1]

arises when `plot` is given a single argument. If `Y` is nonreal, `plot(Y)` is equivalent to `plot(real(Y), imag(Y))`. In the case where `Y` is real, `plot(Y)` plots the columns of `Y` against their index.

COLOR SPACES

In MATLAB colors are specified by a 1-by-3 vector of RGB (red, green, blue) values. The RGB color space is just one of several commonly used color spaces [73]. In fact there are several variants of RGB space that differ in their white points, but this it not something that need concern MATLAB users. Another color space is CMYK, with four coordinates cyan, magenta, yellow, and black, which is the space universally used by color printing devices. The black coordinate is redundant, but it is convenient to have a black ink instead of having to print cyan, magenta, and yellow on top of each other. Relations between RBG and CMYK are given in Table 8.2 and illustrated in Figure 8.3.

You can exert further control by supplying more arguments to `plot`. The properties `LineWidth` (default 0.5 points) and `MarkerSize` (default 6 points) can be specified



Figure 8.3. *Color wheel showing how cyan, magenta, and yellow are obtained by combining red, green, and blue.*

in points, where a point is 1/72 inch. For example, the commands

```
plot(x,y,'LineWidth',2)
plot(x,y,'p','MarkerSize',10)
```

produce a plot with a 2-point line width and a 10-point marker size, respectively. For markers that have a well-defined interior, the `MarkerEdgeColor` and `MarkerFaceColor` can be set to one of the colors in Table 8.1. So, for example,

```
plot(x,y,'o','MarkerEdgeColor','m')
```

gives magenta edges to the circles. The left-hand plot in Figure 8.4 was produced with

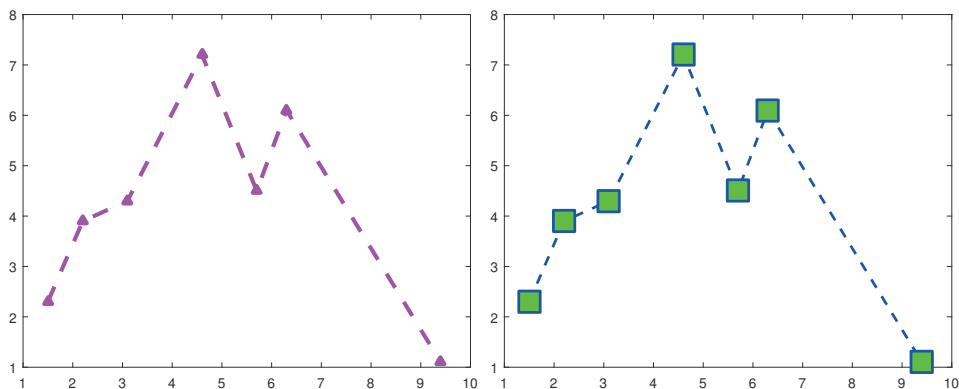
```
plot(x,y,'m--^','LineWidth',3,'MarkerSize',5)
```

and the right-hand plot with

```
plot(x,y,'--bs','MarkerSize',20,'MarkerFaceColor','g','LineWidth',2)
```

Other properties include `FontSize`, in points, and `FontAngle`, which must be either `normal` or `italic`. Default values for these properties are summarized in Table 8.3.

Using `loglog` instead of `plot` causes the axes to be scaled logarithmically. This feature is useful for revealing power-law relationships as straight lines. In the example below we plot  $|1 + h + h^2/2 - \exp(h)|$  against  $h$  for  $h = 10, 1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$ . This quantity behaves like a multiple of  $h^3$  when  $h$  is small, and hence on a log-log scale the values should lie close to a straight line of slope 3. To confirm this, we also plot a dashed reference line with the predicted slope, exploiting the fact that more than one set of data can be passed to the plot commands. The output is shown in Figure 8.5.

Figure 8.4. *Two nondefault x-y plots.*Table 8.3. *Default values for some properties.*

LineWidth	0.5
MarkerSize	6
MarkerEdgeColor	auto
MarkerFaceColor	none
FontSize	10
FontAngle	normal

```

h = 10.^[1:-1:-4];
taylerr = abs((1+h+h.^2/2) - exp(h));
loglog(h,taylerr,'-',h,h.^3,'--')
xlabel('h')
ylabel({'Absolute value','of error'})
title({'Error in quadratic Taylor approximation to exp(h)',...
      'with reference line of slope 3.})
box off

```

In this example, we used `title`, `xlabel`, and `ylabel`. These functions reproduce their input string above the plot and on the  $x$ - and  $y$ -axes, respectively. The multiline  $y$ -axis label and title are created by a cell array of strings, one for each line (see Section 18.7 for details of cell arrays). We also used the command `box off`, which removes the box from the current plot, leaving just the  $x$ - and  $y$ -axes. MATLAB will, of course, complain if nonpositive data is sent to `loglog` (it displays a warning and plots only the positive data). Related functions are `semilogx` and `semilogy`, for which only the  $x$ - or  $y$ -axis, respectively, is logarithmically scaled.

If one plotting command is later followed by another then the new picture will either replace or be superimposed on the old picture, depending on the current `hold` state. Typing `hold on` causes subsequent plots to be superimposed on the current one, whereas `hold off` specifies that each new plot should start afresh. The default status corresponds to `hold off`.



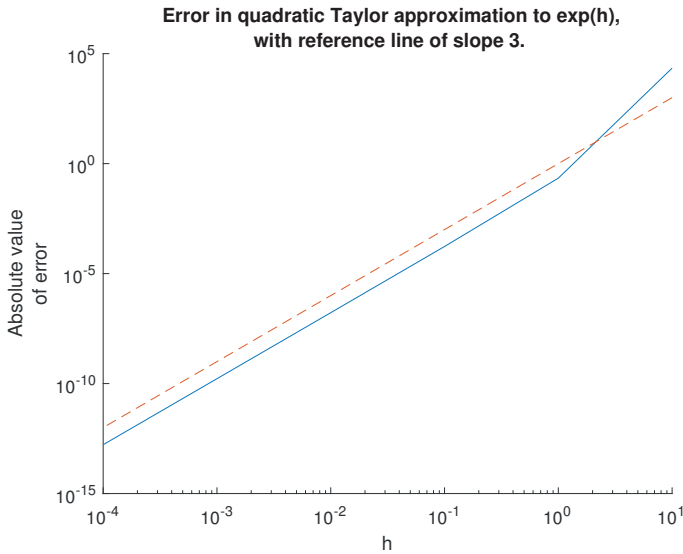


Figure 8.5. *loglog example.*

The command `clf` clears the current figure window, while `close` closes it. It is possible to have several figure windows on the screen. The simplest way to create a new figure window is to type `figure`. The *n*th figure window (where *n* is displayed in the title bar) can be made current and visible, and moved on top of all other figures on the screen, by typing `figure(n)`. The command `shg` is equivalent to `figure(n)`, where the *n*th window is the current one. The command `close all` causes all the figure windows to be closed.

It is possible to zoom in on a particular region of the plot using mouse clicks: see the “zoom in” and “zoom out” icons on the figure toolbar and `help zoom`.

### 8.1.2. Axes and Annotation

Various aspects of the axes of a plot can be controlled with the `axis` command. Some of the options are summarized in Table 8.4. The axes are removed from a plot with `axis off`. The aspect ratio can be set to unity—so that, for example, a circle appears circular rather than elliptical—by typing `axis equal`. The axis box can be made square with `axis square`.

To illustrate, the four plots in Figure 8.6 were produced by

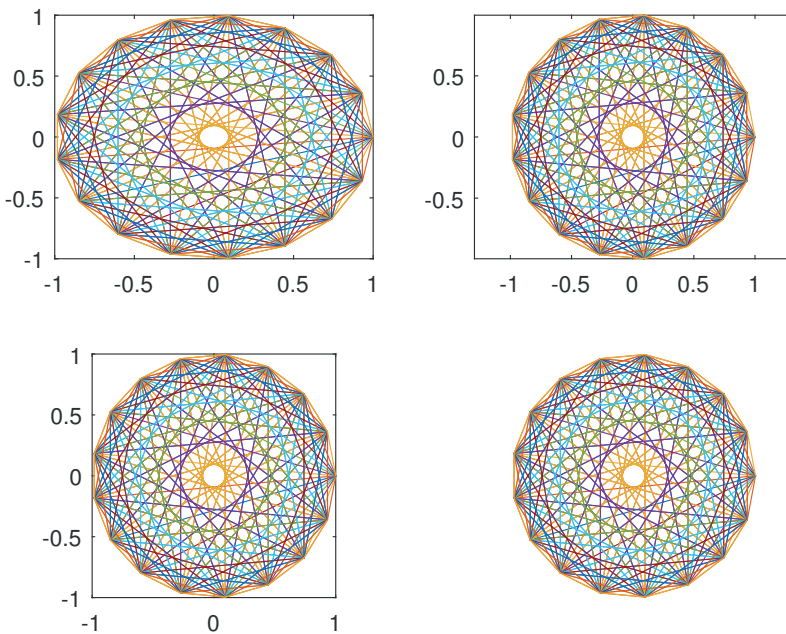
```
plot(fft(eye(17)))           % (1,1) plot
plot(fft(eye(17)), axis equal % (1,2) plot
plot(fft(eye(17)), axis square % (2,1) plot
plot(fft(eye(17)), axis square, axis off % (2,2) plot
```

(The meaning of this interesting picture is described in [123].)

Setting `axis([xmin xmax ymin ymax])` causes the *x*-axis to run from *xmin* to *xmax* and the *y*-axis from *ymin* to *ymax*. To return to the default axis scaling, which MATLAB chooses automatically based on the data being plotted, type `axis auto`. If you want one of the limits to be chosen automatically by MATLAB set it to `-inf` or

Table 8.4. *Some commands for controlling the axes.*

<code>axis([xmin xmax ymin ymax])</code>	Set specified $x$ - and $y$ -axis limits
<code>axis auto</code>	Return to default axis limits
<code>axis equal</code>	Equalize data units on $x$ -, $y$ -, and $z$ -axes
<code>axis off</code>	Remove axes
<code>axis square</code>	Make axis box square (cubic)
<code>axis tight</code>	Set axis limits to range of data
<code>xlim([xmin xmax])</code>	Set specified $x$ -axis limits
<code>ylim([ymin ymax])</code>	Set specified $y$ -axis limits

Figure 8.6. `plot(fft(eye(17)))` with four variations of `axis` (see text).

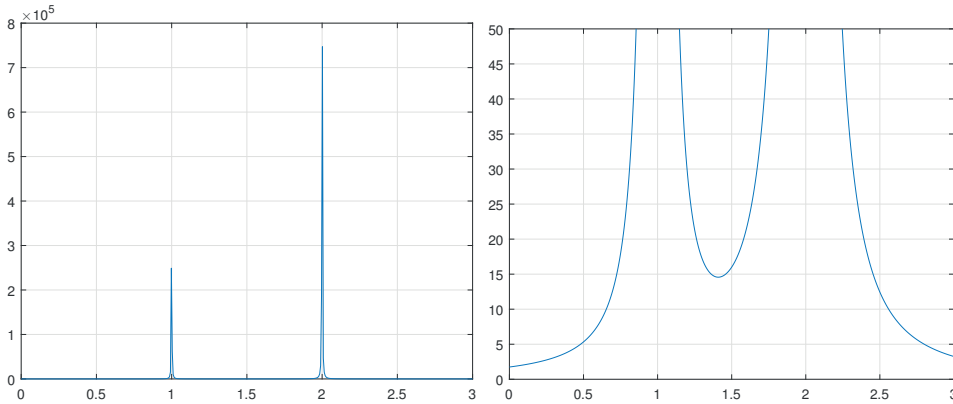


Figure 8.7. Use of `ylim` (right) to change automatic (left) *y*-axis limits.

`inf`, e.g., `axis([-1 1 -inf 0])`. The *x*-axis and *y*-axis limits can be set individually with `xlim([xmin xmax])` and `ylim([ymin ymax])`.

Our next example plots the function  $1/(x-1)^2 + 3/(x-2)^2$  over the interval  $[0, 3]$ :

```
x = linspace(0,3,500);
plot(x,1./(x-1).^2 + 3./(x-2).^2)
grid on
```

The result is shown in the left-hand plot of Figure 8.7. Because of the singularities at  $x = 1, 2$  the plot is uninformative. However, by executing the additional command

```
ylim([0 50])
```

the right-hand plot of Figure 8.7 is produced, which focuses on the interesting part of the first plot. In these two plots we specified `grid on`, which introduces light horizontal and vertical lines that extend from the axis ticks. Unfortunately, these lines tend to become very light when printed, so for better printed results it may be necessary to modify the gridlines with a command such as (here we also choose dotted gridlines)

```
set(gca, 'GridLineStyle', ':', 'Gridalpha', 1.0, 'GridColor', [0 0 0])
```

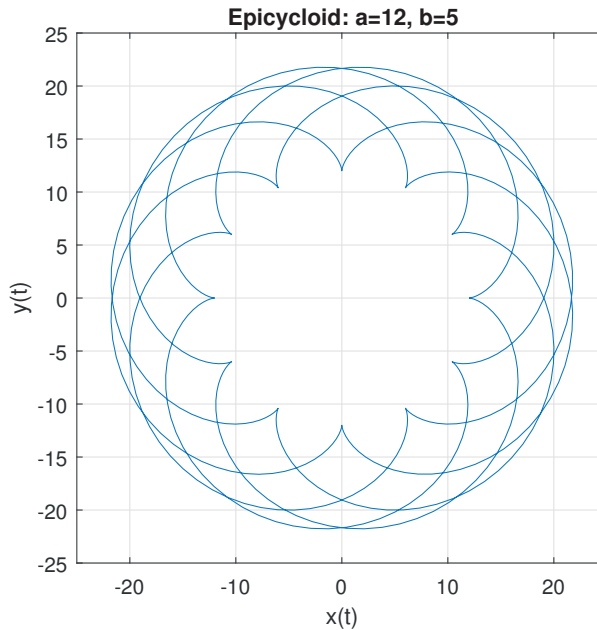
See Figure 17.4 in the Advanced Graphics chapter for an example using these settings.

In the following example we plot the epicycloid

$$\left. \begin{aligned} x(t) &= (a+b)\cos t - b\cos((a/b+1)t) \\ y(t) &= (a+b)\sin t - b\sin((a/b+1)t) \end{aligned} \right\} 0 \leq t \leq 10\pi$$

for  $a = 12$  and  $b = 5$ :

```
a = 12; b = 5;
t = 0:0.05:10*pi;
x = (a+b)*cos(t) - b*cos((a/b+1)*t);
y = (a+b)*sin(t) - b*sin((a/b+1)*t);
```

Figure 8.8. *Epicycloid example.*

```

plot(x,y)
axis equal
axis([-25 25 -25 25])
grid on

title('Epicycloid: a=12, b=5')
xlabel('x(t)'), ylabel('y(t)')

```

The resulting picture appears in Figure 8.8. The `axis` limits were chosen to put some space around the epicycloid.

Next we plot the Legendre polynomials of degrees 1–4 (for the properties of these polynomials, see, for example, [166, Chap. 17]) and use the `legend` function to add a box that explains the line styles. The `legend` function takes as arguments a list of strings and puts each string next to the color/marker/line style information for the corresponding line. The result is shown in Figure 8.9.

```

x = (-1:.01:1)';
p1 = x;
p2 = (3/2)*x.^2 - 1/2;
p3 = (5/2)*x.^3 - (3/2)*x;
p4 = (35/8)*x.^4 - (15/4)*x.^2 + 3/8;

plot(x,[p1 p2 p3 p4])

legend('Degree 1','Degree 2','Degree 3','Degree 4')
xlabel('x')

```

```
ylabel('P_n','Rotation',0)
title('Legendre Polynomials P_n(x)')
```

Note that the strings in the `ylabel` and `title` commands contain a subscript. These commands support a subset of the notation of the typesetting system  $\text{T}_{\text{E}}\text{X}$  to specify Greek letters, mathematical symbols, fonts, and superscripts and subscripts [57], [104], [107], [112]. Table 8.5 lists some of the  $\text{T}_{\text{E}}\text{X}$  notation supported, and a full list can be found under Text Properties (see the entry for Text, Interpreter) in the Help browser, reachable by typing `doc interpreter`. Curly braces can be used to delimit the range of application of the font commands and of subscripts and superscripts. Thus

```
title('\itItalic Normal {\bfBold} \int_{-\infty}^{\infty}')
```

produces a title of the form “*Italic* Normal **Bold**  $\int_{-\infty}^{\infty}$ ”. (Note that, unlike in  $\text{T}_{\text{E}}\text{X}$ , if you leave a space after a font command then that space is printed.) We used the `Rotation` property to rotate the  $y$ -axis label, for better readability. The expressions in the axis labels and title in Figure 8.9 are in upright font, whereas mathematics is normally typeset in italic. This can be achieved by writing, for example, `ylabel('\it P_n','Rotation',0)`.

If you are unfamiliar with  $\text{T}_{\text{E}}\text{X}$  or  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  you may prefer to use `texlabel('string')`, which allows 'string' to be given in the style of a MATLAB expression. Thus the following two commands have identical effect:

```
title('Plot of \alpha(t)^{3/2}+\beta(t)^{12}-\sigma_i')
title(texlabel('Plot of alpha(t)^(3/2)+beta(t)^12-sigma_i'))
```

Now we make a number of refinements to the plot, which lead to the script `legendre_plot` in Listing 8.1, which produces Figure 8.10. By default, the legend box appears in the top right-hand (northeast) corner of the axis area, and this obscures part of the plot in this example. The location of the box can be specified with the syntax `legend('...', 'Location', location)`, where `location` is a string with possible values that include:

'North'	inside plot box near top
'NorthWest'	inside top left
'NorthOutside'	outside plot box near top
'Best'	automatically chosen to give least conflict with data
'BestOutside'	automatically chosen to leave least unused space outside plot

These values can be abbreviated as 'N', 'NW', etc. We choose to put the legend in the bottom right-hand corner. Once the plot has been drawn, the legend box can be repositioned by putting the cursor over it and dragging it using the left mouse button. We also turn off the box around the legend with `legend('boxoff')`. The `legend` function has many other options, which can be seen by typing `doc legend`.

This example uses three other features.

- The `text` command is used to display the three-term recurrence satisfied by the Legendre polynomials. Generally, `text(x,y,'string')` places 'string' at the position whose coordinates are given by `x` and `y`. (A related function `gtext` allows the text location to be determined interactively via the mouse.)
- The `FontSize` property is set in order to adjust the point size and angle of the text produced by the `xlabel`, `ylabel`, `title`, and `text` commands (as Table 8.3 indicates, the default value of `FontSize` is 10).

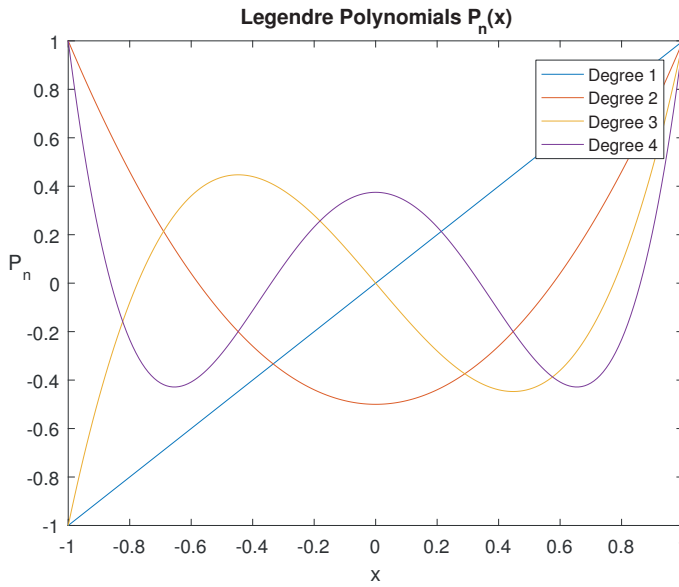


Figure 8.9. Legendre polynomial example, using legend.

Table 8.5. Some of the  $T_E X$  commands supported in text strings.

Greek letters		Selected symbols	
Lowercase		$\approx$	<code>\approx</code>
$\alpha$	<code>\alpha</code>	$\circ$	<code>\circ</code>
$\beta$	<code>\beta</code>	$\geq$	<code>\geq</code>
$\gamma$	<code>\gamma</code>	$\Im$	<code>\Im</code>
$\vdots$	<code>\vdots</code>	$\in$	<code>\in</code>
$\omega$	<code>\omega</code>	$\infty$	<code>\infty</code>
Uppercase		$\int$	<code>\int</code>
$\Gamma$	<code>\Gamma</code>	$\leq$	<code>\leq</code>
$\Delta$	<code>\Delta</code>	$\neq$	<code>\neq</code>
$\Theta$	<code>\Theta</code>	$\otimes$	<code>\otimes</code>
$\vdots$	<code>\vdots</code>	$\partial$	<code>\partial</code>
$\Omega$	<code>\Omega</code>	$\pm$	<code>\pm</code>
		$\Re$	<code>\Re</code>
		$\sim$	<code>\sim</code>
		$\sqrt{\quad}$	<code>\sqrt{\quad}</code>

Fonts	
Normal	<code>\rm</code>
<b>Bold</b>	<code>\bf</code>
<i>Italic</i>	<code>\it</code>

Listing 8.1. *Script* legendre\_plot.

```

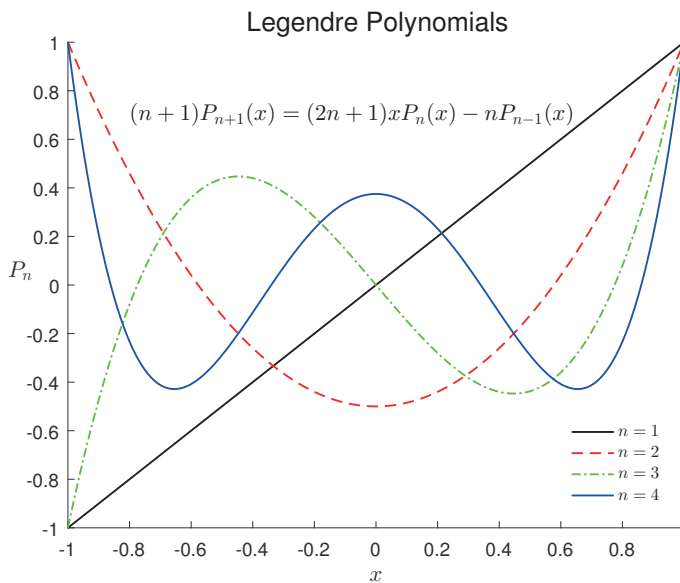
%LEGENDRE_PLOT   Plot Legendre polynomials.

x = -1:.01:1;
p1 = x;
p2 = (3/2)*x.^2 - 1/2;
p3 = (5/2)*x.^3 - (3/2)*x;
p4 = (35/8)*x.^4 - (15/4)*x.^2 + 3/8;

plot(x,p1,'k-',x,p2,'r--',x,p3,'g-.',x,p4,'b-', 'linewidth',1)
box off

legend({'$n=1$', '$n=2$', '$n=3$', '$n=4$'}, 'Location', 'SouthEast', ...
       'Interpreter', 'latex')
legend('boxoff')
xlabel('$x$', 'FontSize', 12, 'Interpreter', 'latex')
ylabel('$P_n$', 'FontSize', 12, 'Interpreter', 'latex', 'Rotation', 0)
title('Legendre Polynomials', 'FontSize', 14, 'FontWeight', 'normal')
text(-.8, .7, '$(n+1)P_{n+1}(x) = (2n+1)x P_n(x) - n P_{n-1}(x)$', ...
     'FontSize', 12, 'Interpreter', 'latex')

```

Figure 8.10. *Legendre polynomial example (revised), using legend.*

- We specified the `xlabel`, `ylabel`, `title`, and `text` command arguments in L<sup>A</sup>T<sub>E</sub>X notation. The L<sup>A</sup>T<sub>E</sub>X interpreter, which supports the mathematical typesetting features of L<sup>A</sup>T<sub>E</sub>X [57], [107], [112], is invoked by setting the `Interpreter` property to `'latex'`. A benefit of using the L<sup>A</sup>T<sub>E</sub>X interpreter is that mathematics is typeset in italic.
- The title has been changed from the default bold weight to normal weight by setting the `FontWeight` property to `'normal'`.

The script `pnorm_plot` in Listing 8.2 makes more extensive use of the L<sup>A</sup>T<sub>E</sub>X interpreter and produces Figure 8.11. Note the use of the cell array options to avoid repeatedly having to type the arguments `'Interpreter'`, `'latex'`, `'FontSize'`, `18`, and the use of the `HorizontalAlignment` property in the `ylabel` to keep the label away from the tick labels.

The `fill` function works in a similar manner to `plot`. Typing `fill(x,y,color)` shades a polygon whose vertices are specified by the points `x(i)`, `y(i)` in the color specified by the RGB triple `color` (see Section 8.1.1). The points are taken in order, and the last vertex is joined to the first.

The script in Listing 8.3 plots a cubic Bezier curve, which is defined by

$$p(u) = (1 - u)^3 \mathbf{P}_1 + 3u(1 - u)^2 \mathbf{P}_2 + 3u^2(1 - u) \mathbf{P}_3 + u^3 \mathbf{P}_4, \quad 0 \leq u \leq 1,$$

where the four control points,  $\mathbf{P}_1$ ,  $\mathbf{P}_2$ ,  $\mathbf{P}_3$ , and  $\mathbf{P}_4$ , have given  $x$  and  $y$  components. We use `fill` to shade the control polygon, that is, the polygon formed by the control points. The matrix `P` stores the control point  $\mathbf{P}_j$  in its  $j$ th column, and `fill(P(1,:),P(2,:),[.8 .8 .8])` shades the control polygon with light gray. The columns of the matrix `Curve` are closely spaced points on the Bezier curve, and the first `plot` command plots the curve. Figure 8.12 gives the resulting picture.

The function `annotation` allows the creation of annotation objects, including lines, various types of arrow, rectangles, and ellipses. While powerful, this function is not particularly easy to use because of the need to specify the location of these objects in normalized coordinates, which represent the bottom left corner of the figure by  $(0,0)$  and the top right corner by  $(1,1)$ . Annotations are most easily created using the interactive tools in the figure window. See `doc annotation` for details.

### 8.1.3. Multiple Plots in a Figure

The `subplot` command allows you to place a number of plots in a grid pattern together on the same figure. Typing `subplot(mnp)` or, equivalently, `subplot(m,n,p)`, splits the figure window into an  $m$ -by- $n$  array of regions, each having its own axes. The current plotting commands will then apply to the  $p$ th of these regions, where the count moves along the first row, and then along the second row, and so on. So, for example, `subplot(425)` splits the figure window into a 4-by-2 matrix of regions and specifies that plotting commands apply to the fifth region, that is, the first region in the third row. If `subplot(427)` appears later, then the region in the  $(4,1)$  position becomes active. Several examples in which `subplot` is used appear below.

For plotting mathematical functions the `fplot` command is useful. It adaptively samples a function at enough points to produce a representative graph. The following example generates the graphs in Figure 8.13.



Listing 8.2. *Function pnorm\_plot.*

```

%PNORM_PLOT    Plot p-norms of a vector.

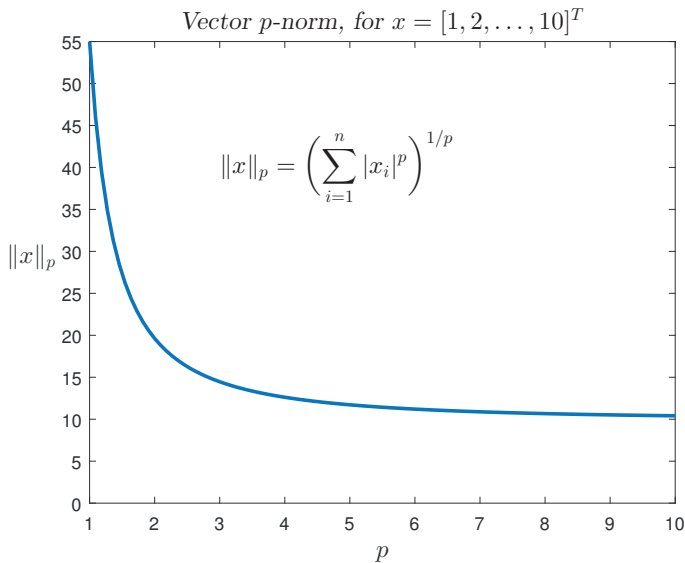
n = 10; m = 100;
x = 1:n;
y = zeros(m,1);

pvals = linspace(1,10,m);
for i = 1:m
    y(i) = norm(x,pvals(i));
end

plot(pvals,y,'LineWidth',2)
ylim([0 inf])
options = {'Interpreter','latex','FontSize',14};
ylabel('$\|x\|_p$',options{:},'Rotation',0,'HorizontalAlignment','right')
xlabel('$p$',options{:})
title(['\slshape Vector $p$-norm, for $x = ' ...
       '[1,2,\dots,' int2str(n) ' ]^T$', options{:})

s = '$$\|x\|_p = \biggl(\sum_{i=1}^n |x_i|^p\biggr)^{1/p}$$';
text(options{:},'String',s,'Position',[3 40])

```

Figure 8.11. *Plot with text produced using the MATLAB L<sup>A</sup>T<sub>E</sub>X interpreter.*

Listing 8.3. *Function* bezier\_plot.

```

%BEZIER_PLOT    Plot bezier curve and control polygon.

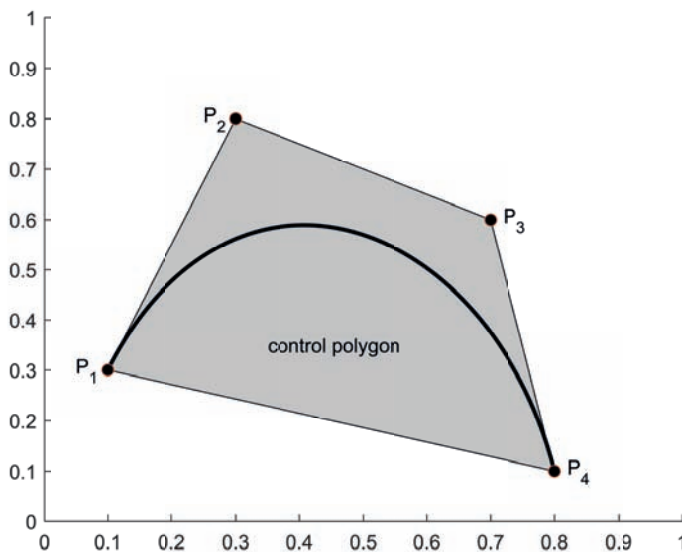
P = [0.1 0.3 0.7 0.8;
      0.3 0.8 0.6 0.1];
axis([0 1 0 1])
hold on

u = 0:.01:1;
umat = [(1-u).^3; 3.*u.*(1-u).^2; 3.*u.^2.*(1-u); u.^3];
Curve = P*umat;
fill(P(1,:),P(2,:),[.8 .8 .8])           % Shaded control polygon.
plot(Curve(1,:),Curve(2,:),'k-', 'linewidth',2) % Bezier curve.

plot(P(1,:),P(2,:), 'o', 'MarkerFaceColor', 'k') % Control points.

text(0.35,0.35,'control polygon')
text(0.05,0.3,'P_1')
text(0.25,0.8,'P_2')
text(0.72,0.6,'P_3')
text(0.82,0.1,'P_4')
hold off

```

Figure 8.12. *Bezier curve and control polygon.*

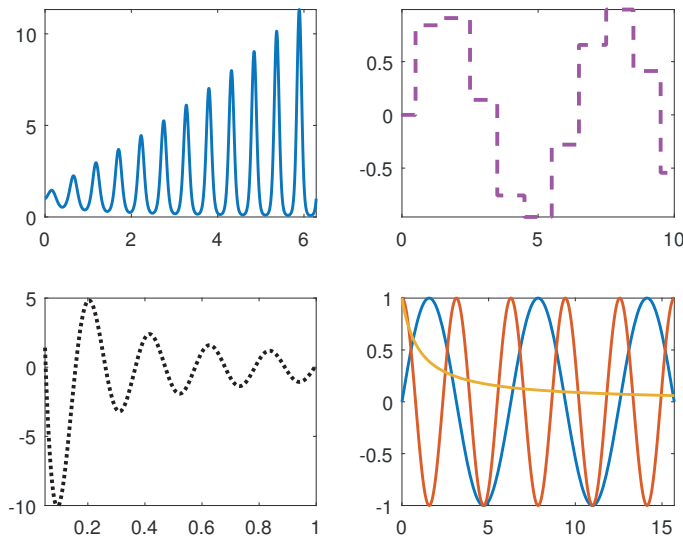


Figure 8.13. Example with subplot and fplot.

```

subplot(221)
fplot(@(x)exp(sqrt(x).*sin(12.*x)),[0 2*pi],'LineWidth',1.5)
subplot(222)
fplot(@(x)sin(round(x)),[0 10],'m--','LineWidth',2)
subplot(223)
fplot(@(x)cos(30.*x)./x,[0.05 1],'k:','LineWidth',2), ylim([-10 5])
subplot(224)
fplot(@(x)[sin(x),cos(2*x),1./(1+x)],[0 5*pi],'LineWidth',1.5),...
      ylim([-1 1])

```

The first call to `fplot` produces a graph of  $\exp(\sqrt{x}\sin 12x)$  over the interval  $0 \leq x \leq 2\pi$ . (See Section 10.2 for an explanation of the anonymous function beginning with `@`.) In the second call, we override the default solid line style by specifying a dashed line with `'--'`. In the third and fourth cases we set limits on the  $y$ -axis with `ylim`. The fourth `fplot` call shows how more than one function can be plotted in the same call; it uses `*` and `/` since `fplot` expects that the functions can be evaluated for vector inputs. The `fplot` function can also detect and label asymptotes; see Figure 11.5 for an example. See `doc fplot` for details of additional capabilities.

It is possible to produce irregular grids of plots by invoking `subplot` with different grid patterns. For example, Figure 8.14 was produced as follows:

```

x = linspace(0,5*pi,100);
subplot(2,2,1)
plot(x,sin(x),'r','LineWidth',2), xlim([0 5*pi])
subplot(2,2,2)
plot(x,round(x),'b','LineWidth',2), xlim([0 5*pi])
subplot(2,1,2)
plot(x,sin(round(x)),'g','LineWidth',2)
xticks(pi*(0:5))
xticklabels({'0','\pi','2\pi','3\pi','4\pi','5\pi'})

```

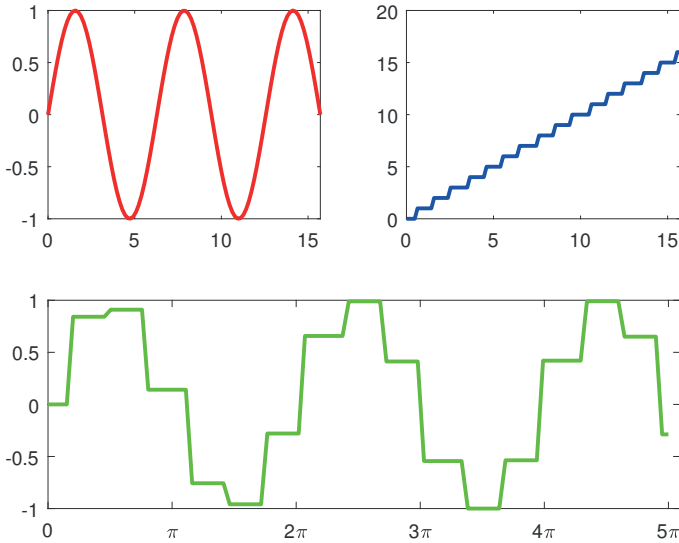


Figure 8.14. *Irregular grid of plots produced with subplot.*

The third argument to `subplot` can be a vector specifying several regions, so we could replace the last line `subplot` command by `subplot(2,2,3:4)`. This example also illustrates how  $x$ -axis tick marks and their labels can be set with the functions `xticks` and `xticklabels`; there are corresponding functions `yticks` and `yticklabels` for the  $y$ -axis. Further functions `xtickformat` and `xtickangle`, and their counterparts `ytickformat` and `ytickangle`, allow the format and the angle of rotation of the labels to be set.

To complete this section, we list in Table 8.6 the most popular 2D plotting functions in MATLAB. Some of these functions are discussed in Section 8.3.

## 8.2. Three-Dimensional Graphics

The function `plot3` is the three-dimensional analogue of `plot`. The following example illustrates the simplest usage: `plot3(x,y,z)` draws a “join-the-dots” curve by taking the points  $x(i)$ ,  $y(i)$ ,  $z(i)$  in order. The result is shown in Figure 8.15.

```
t = -5:.005:5;
x = (1+t.^2).*sin(20*t);
y = (1+t.^2).*cos(20*t);
z = t;

plot3(x,y,z,'LineWidth',1.5)
grid on
FS = 'FontSize';
xlabel('x(t)',FS,14), ylabel('y(t)',FS,14)
zlabel('z(t)',FS,14,'Rotation',0,'HorizontalAlignment','right')
title('\it{plot3 example}',FS,14)
```

This example uses the functions `xlabel`, `ylabel`, and `title`, which were discussed

Table 8.6. 2D plotting functions.

<code>plot</code>	Simple $x$ - $y$ plot
<code>loglog</code>	Plot with logarithmically scaled axes
<code>semilogx</code>	Plot with logarithmically scaled $x$ -axis
<code>semilogy</code>	Plot with logarithmically scaled $y$ -axis
<code>yyaxis</code>	$x$ - $y$ plot with $y$ -axes on left and right
<code>polarplot</code>	Polar coordinates plot
<code>fplot</code>	Function plotter
<code>fill</code>	Polygon fill
<code>area</code>	Filled area graph
<code>bar</code>	Bar graph
<code>barh</code>	Horizontal bar graph
<code>histogram</code>	Histogram
<code>pie</code>	Pie chart
<code>comet</code>	Animated, comet-like, $x$ - $y$ plot
<code>errorbar</code>	Error bar plot
<code>quiver</code>	Quiver (velocity vector) plot
<code>scatter</code>	Scatter plot
<code>stairs</code>	Stairstep plot

in the previous section, and the analogous `zlabel`. Note that we have used the  $\text{\TeX}$  notation `\it` in the `title` command to produce italic text. The color, marker, and line styles for `plot3` can be controlled in the same way as for `plot`. So, for example, `plot3(x,y,z,'rx--')` would use a red dashed line and place a cross at each point. Note that for 3D plots the default is `box off`; specifying `box on` adds a box that bounds the plot.

Two functions are available for plotting contours: `fcontour` for a function and `contour` for a matrix of data. The following example produces contours for the function  $\sin(3y - x^2 + 1) + \cos(2y^2 - 2x)$  over the range  $-2 \leq x \leq 2$  and  $-1 \leq y \leq 1$ ; the result can be seen in Figure 8.16.

```
subplot(311)
fcontour(@(x,y)sin(3*y-x.^2+1)+cos(2*y.^2-2*x),[-2 2 -1 1]);

subplot(312)
f = fcontour(@(x,y)sin(3*y-x.^2+1)+cos(2*y.^2-2*x),[-2 2 -1 1]);
f.Fill = 'on'; colorbar

subplot(313)
x = -2:.01:2; y = -1:.01:1;
[X,Y] = meshgrid(x,y);
Z = sin(3*Y-X.^2+1)+cos(2*Y.^2-2*X);
contour(x,y,Z,20)
```

Note that the contour levels have been chosen automatically. For the second plot we exploit the fact that `fcontour` returns an object that allows various aspects of the contour plot to be customized. Here, we set the regions between contour lines to be

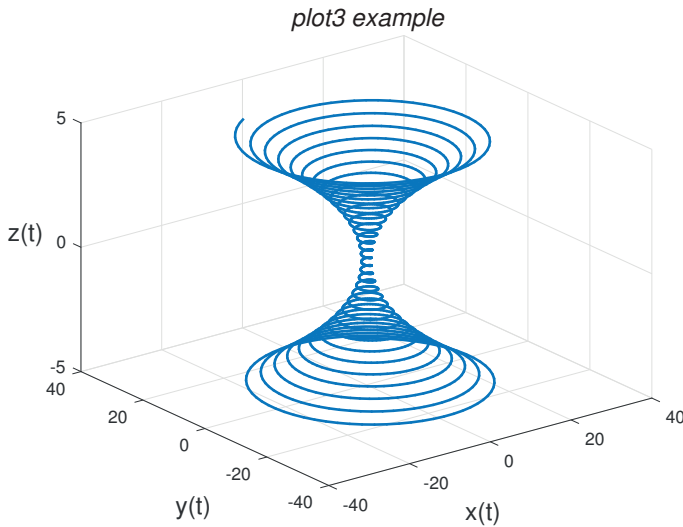


Figure 8.15. 3D plot created with `plot3`.

colored, but we could also have set the contour levels, for example. We also add a `colorbar`.

For the third contour plot in Figure 8.16 we first assign `x = -2:.01:2` and `y = -1:.01:1` to obtain closely spaced points in the appropriate range. We then set `[X,Y] = meshgrid(x,y)`, which produces matrices `X` and `Y` such that each row of `X` is a copy of the vector `x` and each column of `Y` is a copy of the vector `y`. (The function `meshgrid` is extremely useful for setting up data for many of the MATLAB 3D plotting tools.) The matrix `Z` is then generated from array operations on `X` and `Y`, with the result that `Z(i,j)` stores the function value corresponding to `x(j)`, `y(i)`. This is precisely the form required by `contour`. Typing `contour(x,y,Z,20)` tells MATLAB to regard `Z` as defining heights above the  $(x,y)$ -plane with spacing given by `x` and `y`. The final input argument specifies that 20 contour levels are to be used; if this argument is omitted MATLAB automatically chooses the number of contour levels.

The next example illustrates the use of `clabel` to label contours, with the result shown in Figure 8.17.

```
[X,Y] = meshgrid(-3:.05:3, -1.5:.025:1.5);
Z = 4*X.^2 - 2.1*X.^4 + X.^6/3 + X.*Y - 4*Y.^2 + 4*Y.^4;
cvals = [-2:.5:2 2.3 3:5 6:2:10];
[C,h] = contour(X,Y,Z,cvals);
clabel(C,h,cvals([1:2:9 10 11 14 16]))
xlabel('x'), ylabel('y')
title('Six hump camel back function','FontWeight','normal',...
      'FontSize',12)
```

Here, we are using an interesting function having a number of maxima, minima, and saddle points. The default choice of contour levels does not produce an attractive picture, so we specify the levels (chosen by trial and error) in the vector `cvals`. The `clabel` command takes as input the output from `contour` (`C` contains the contour

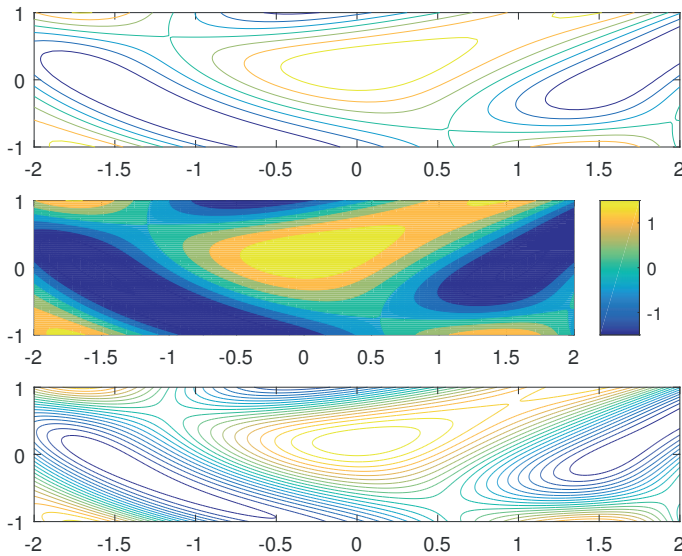


Figure 8.16. Contour plots with `fcontour` (top and middle) and `contour` (bottom).

data and `h` is a graphics object handle) and adds labels to the contour levels specified in its third input argument. Again the contour levels need not be specified, but the default of labeling all contours produces a cluttered plot in this example. An alternative form of `clabel` is `clabel(C,h,'manual')`, which allows you to specify with the mouse the contours to be labeled: click to label a contour and press return to finish. The `h` argument of `clabel` can be omitted, in which case the labels are placed unrotated and close to each contour, with a plus sign marking the contour.

The function `mesh` accepts data in a similar form to `contour` and produces wire-frame surface plots. If `meshc` is used in place of `mesh`, a contour plot is appended below the surface. The example below, which produces Figure 8.18, involves the surface defined by  $\sin(y^2 + x) - \cos(y - x^2)$  for  $0 \leq x, y \leq \pi$ . The first subplot is produced by `mesh(Z)`. Since no  $x, y$  information is supplied to `mesh`, row and column indices are used for the axis ranges. The second subplot shows the effect of `meshc(Z)`. For the third subplot, we use `mesh(x,y,Z)`, so the tick labels on the  $x$ - and  $y$ -axes correspond to the values of  $x$  and  $y$ . We also specify the axis limits with `axis([0 pi 0 pi -5 5])`, which gives  $0 \leq x, y \leq \pi$  and  $-5 \leq z \leq 5$ . For the final subplot, we use `mesh(Z)` again, followed by `hidden off`, which causes hidden lines to be shown.

```
x = 0:.1:pi; y = 0:.1:pi;
[X,Y] = meshgrid(x,y);
Z = sin(Y.^2+X)-cos(Y-X.^2);
subplot(221), mesh(Z)
subplot(222), meshc(Z)
subplot(223), mesh(x,y,Z), axis([0 pi 0 pi -5 5])
subplot(224), mesh(Z), hidden off
```

The function `surf` differs from `mesh` in that it produces a solid filled surface plot, and `surf` adds a contour plot below. In the next example we call `membrane`, which returns the first eigenfunction of an L-shaped membrane. The pictures in the first

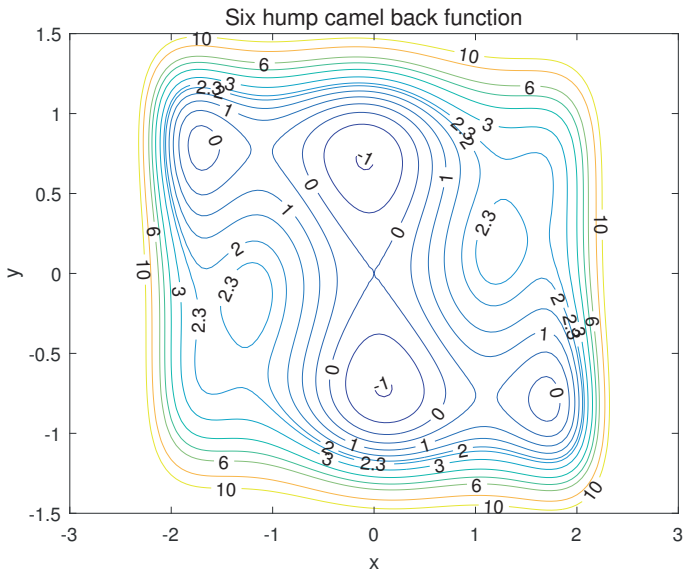


Figure 8.17. *Contour plot labeled using clabel.*

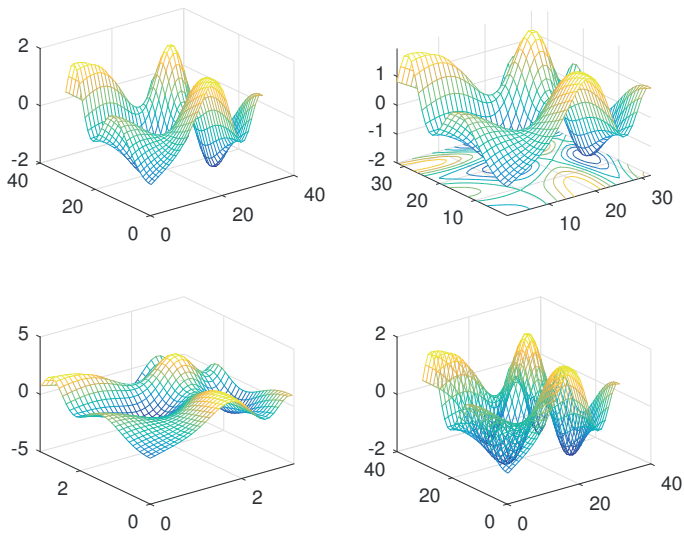


Figure 8.18. *Surface plots with mesh and meshc.*



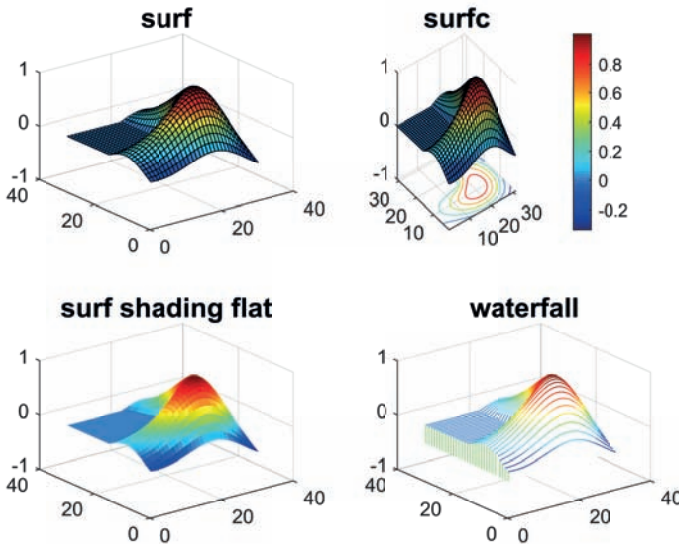


Figure 8.19. *Surface plots with surf, surfc, and waterfall.*

row of Figure 8.19 show the effect of `surf` and `surfc`. The color map for the current figure can be set using `colormap`; here, we set it to `jet`. See doc `colormap` and the Aside on color maps below. The (2,1) plot uses the `shading flat` option to remove the grid lines on the surface; another option is `interp`, which varies the color over each segment by interpolation. The (2,2) plot uses the related function `waterfall`, which is similar to `mesh` but with the wireframes in the column direction removed.

```
Z = membrane; FS = 'FontSize';
colormap(jet)
subplot(221), surf(Z), title('surf',FS,14)
subplot(222), surfc(Z), title('surfc',FS,14), colorbar
subplot(223), surf(Z), shading flat
           title('surf shading flat',FS,14)
subplot(224), waterfall(Z), title('waterfall',FS,14)
```

Ordinarily, the color of a surface represents the height above the  $(x, y)$ -plane. However, `mesh`, `surf`, and related functions may also be used in the form `mesh(x, y, Z, W)`, which bases the colors on the array `W`; this form can be used to display other features of the surface, or to impose an independent coloring pattern.

The `fmesh` and `fsurf` functions can directly plot functions without the need for the use of `meshgrid`. For example, the code

```
fsurf(@(x,y) sin(y.^2+x) - cos(y-x.^2), [0 pi 0 pi])
```

plots Figure 8.20, which shows the same function as Figure 8.18.

The 3D pictures in Figures 8.15 and 8.18–8.20 use the default viewing angle. This can be overridden with the function `view`. Typing `view(a,b)` sets the counterclockwise rotation about the  $z$ -axis to `a` degrees and the vertical elevation to `b` degrees. The default is `view(-37.5,30)`, while `view(2)` is equivalent to `view(0,90)` and gives

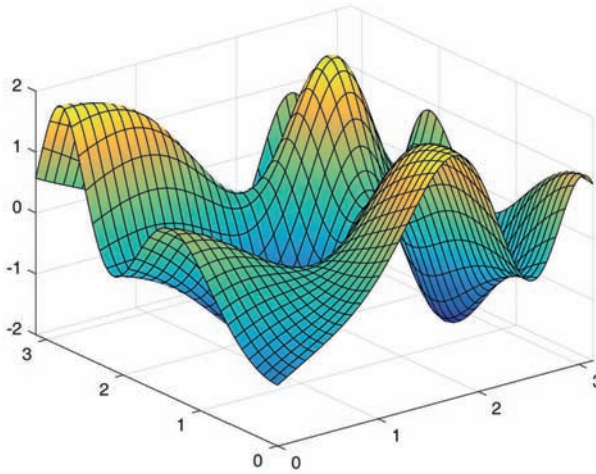


Figure 8.20. *Surface plot with fsurf.*

a 2D view of a surface looking down from above. The `rotate 3D` tool on the toolbar of the figure window enables the mouse to be used to change the angle of view by clicking and dragging within the axis area.

It is possible to view a 2D plot as a 3D one by using the `view` command to specify a viewing angle, or simply by typing `view(3)`. Figure 8.21—a 3D version of Figure 8.6—shows the result of typing

```
plot(fft(eye(17))); view(3); grid
```

#### COLOR MAPS

For many years the default `colormap` in MATLAB was a rainbow color map called `jet`, which runs from blue to red, passing through cyan, green, yellow, and orange. In Release 2014b the default was replaced by a new color map called `parula`: see Figure 8.22. The rainbow color map has been the subject of criticism in recent years, for several reasons [12], [42]. First, it is not perceptually uniform: the perceived rate of change of color is not constant along the map, appearing faster in the yellow and slower in the green. Second, it can be hard to remember the ordering of the colors, making interpretation of the image difficult (even with a color bar at its side). Finally, information is lost when an image in the rainbow color map is printed on a monochrome printer, since different colors map to the same shade of gray. The `parula` map was designed to have a roughly constant gradient of brightness from one end to the other. You can edit color maps in MATLAB using the Colormap Editor, which can be invoked via the command `colormapeditor`.

In the next example we generate a fractal landscape with the recursive function `land` shown in Listing 8.4, which uses a variant of the random midpoint displacement algorithm [139, Sec. 7.6]. (Recursion is discussed further in Section 10.9.) The basic

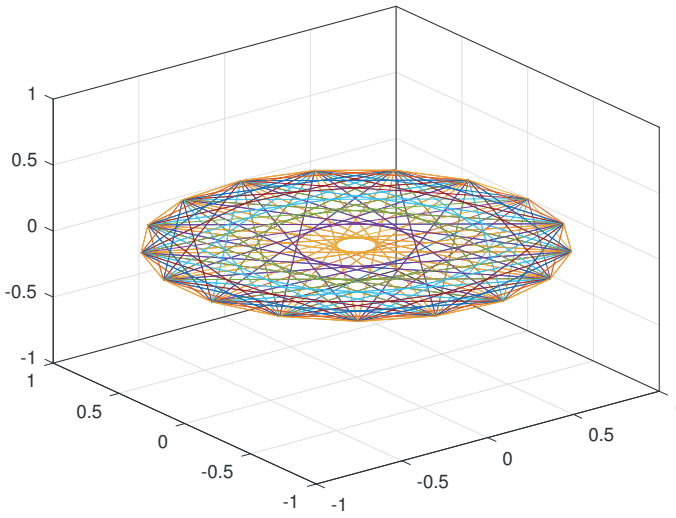


Figure 8.21. *3D view of a 2D plot.*



Figure 8.22. *Color maps jet and parula.*



vertical curtain around the edges of the surface. The first subplot shows the default view of `B`. For the second subplot we impose a “sea level” by raising all heights that are below the average value. The resulting data matrix, `Bisland`, is also plotted with the default view. The third and fourth subplots use `view([-75 40])` and `view([240 55])`, respectively. For these two subplots we also control the axis limits.

```

rng(3)
k = 2^5+1;
A = zeros(k);
A([1 k], [1 k]) = [1 1.25; 1.1 2.0];
B = land(A);

colormap('copper')
subplot(221), meshz(B)
FS = 'FontSize'; title('Default view',FS,12)

Bisland = max(B,mean(mean(B)));
Bmin = min(min(Bisland));
Bmax = max(max(Bisland));
subplot(222), meshz(Bisland)
title('Default view',FS,12)

subplot(223), meshz(Bisland)
view([-75 40])
axis([0 k 0 k Bmin Bmax])
title('view([-75 40])',FS,12)

subplot(224), meshz(Bisland)
view([240 55])
axis([0 k 0 k Bmin Bmax])
title('view([240 55])',FS,12)

```

Table 8.7 summarizes the most popular 3D plotting functions. Section 8.3 discusses some of these functions.

A feature common to all graphics functions is that NaNs are interpreted as “missing data” and are not plotted. For example,

```
plot([1 2 NaN 3 4])
```

draws two disjoint lines and does not connect “2” to “3”, while

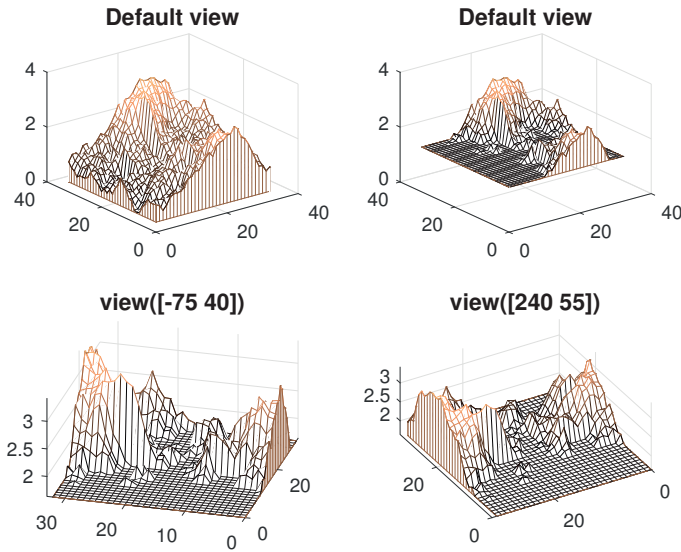
```

A = peaks(80); A(28:52,28:52) = NaN; surfc(A);
% Use last 2/3 of jet colormap.
c = colormap(jet); colormap(c(round(end/3):end,:));

```

produces the `surfc` plot with a hole in the middle shown in Figure 8.24. (The function `peaks` generates a matrix of height values corresponding to a particular function of two variables and is useful for demonstrating 3D plots.)

MATLAB contains in its `demos` directory several functions with names beginning `cplx` for visualizing functions of a complex variable (type `what demos`). Figure 8.25 shows the plot produced by `cplxroot(3)`. In general, `cplxroot(n)` plots the Riemann surface for the function  $z^{1/n}$ .

Figure 8.23. *Fractal landscape views.*Table 8.7. *3D plotting functions.*

<code>plot3</code>	Simple $x$ - $y$ - $z$ plot
<code>fplot3</code>	3D parametric curve
<code>contour</code>	Contour plot
<code>contourf</code>	Filled contour plot
<code>contour3</code>	3D contour plot
<code>mesh</code>	Wireframe surface
<code>meshc</code>	Wireframe surface plus contours
<code>meshz</code>	Wireframe surface with curtain
<code>fmesh</code>	3D meshes, including parametric meshes
<code>surf</code>	Solid surface
<code>surfc</code>	Solid surface plus contours
<code>fsurf</code>	3D surfaces, including parametric surfaces
<code>waterfall</code>	Unidirectional wireframe
<code>bar3</code>	3D bar graph
<code>bar3h</code>	3D horizontal bar graph
<code>pie3</code>	3D pie chart
<code>fill3</code>	Polygon fill
<code>comet3</code>	3D animated, comet-like plot
<code>scatter3</code>	3D scatter plot
<code>stem3</code>	Stem plot

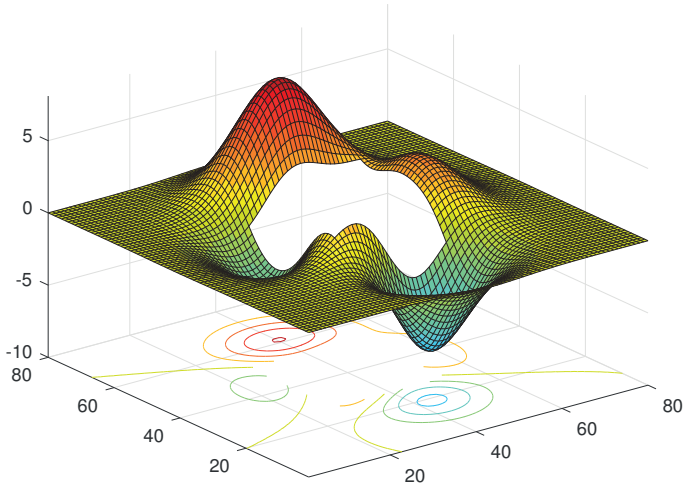


Figure 8.24. `surf` plot of matrix containing NaNs.

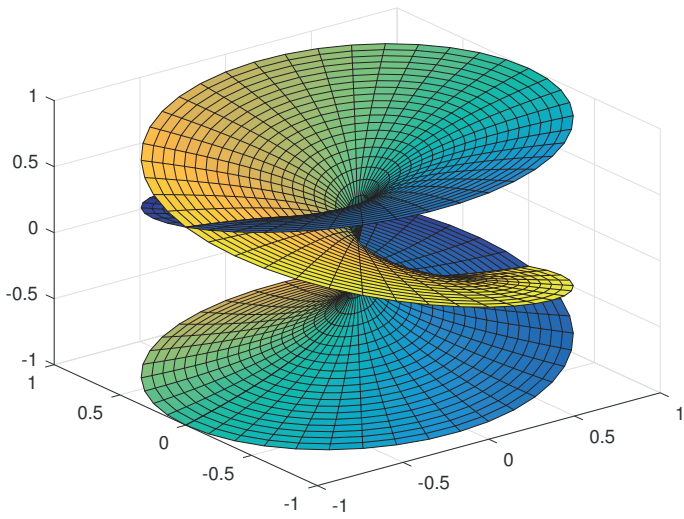


Figure 8.25. Riemann surface for  $z^{1/3}$ .

### 8.3. Specialized Graphs for Displaying Data

In this section we describe some additional functions from Tables 8.6 and 8.7 that are useful for displaying data (as opposed to plotting mathematical functions).

A bar graph consists of bars of equal width whose lengths are proportional to the values in the underlying data. MATLAB has four functions for plotting bar graphs, covering 2D and 3D vertical or horizontal bar graphs, with options to stack or group the bars. The simplest usage of the bar plot functions is with a single  $m$ -by- $n$  matrix input argument. For 2D bar plots elements in a row are clustered together, either in a group of  $n$  bars with the default `'grouped'` argument, or in one bar apportioned among the  $n$  row entries with the `'stacked'` argument.

The following code uses `bar` and `barh` to produce Figure 8.26:

```
Y = [7 6 5
     6 8 1
     4 5 9
     2 3 4
     9 7 2];

subplot(2,2,1)
bar(Y)
title('bar(..., 'grouped')')

subplot(2,2,2)
bar(0:5:20,Y)
title('bar(..., 'grouped')')

subplot(2,2,3)
bar(Y, 'stacked')
title('bar(..., 'stacked')')

subplot(2,2,4)
barh(Y)      % Horizontal bar graph.
title('barh')
```

Note that in the two-argument form `bar(x,Y)` the vector  $x$  provides the  $x$ -axis locations for the bars.

For 3D bar graphs the default arrangement is `'detached'`, with the bars for the elements in each column distributed along the  $y$ -axis. The arguments `'grouped'` and `'stacked'` give 3D views of the corresponding 2D bar plots with the same arguments. With the same data matrix,  $Y$ , Figure 8.27 is produced by

```
subplot(2,2,1)
bar3(Y)
title('bar3(..., 'detached')')

subplot(2,2,2)
bar3(Y, 'grouped')
title('bar3(..., 'grouped')')

subplot(2,2,3)
```



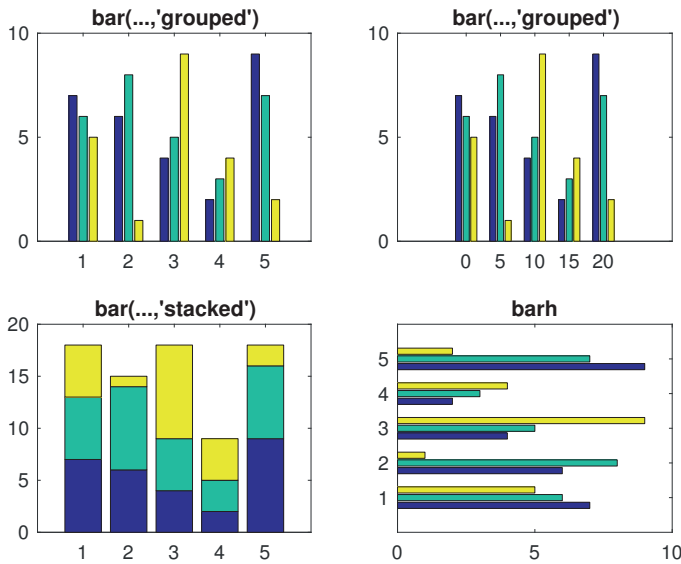


Figure 8.26. 2D bar plots.

```
bar3(Y, 'stacked')
title('bar3(..., 'stacked')')

subplot(2,2,4)
bar3h(Y)
title('bar3h')
```

Note that with the default `'detached'` arrangement some bars are hidden behind others. A satisfactory solution to this problem can sometimes be found by rotating the plot using `view` or the mouse.

Histograms, which show the distribution of data by intervals using bar graphs, are produced by the `histogram` function. The first argument, `y`, to `histogram` is the data vector and the second is either a scalar specifying the number of bins or a vector defining the edges of the bins; if only `y` is supplied then an automatic binning algorithm is used. Further property name–value pairs can be used to customize the histogram. The following code generates Figure 8.28:

```
rng(1)
y = exp(randn(1000,1)/3);
subplot(2,2,1)
histogram(y)
title('Default binning')

subplot(2,2,2)
histogram(y,50)
title('50 bins')

subplot(2,2,3)
h = histogram(y, 'binwidth', 0.25, 'FaceColor', 'green')
```

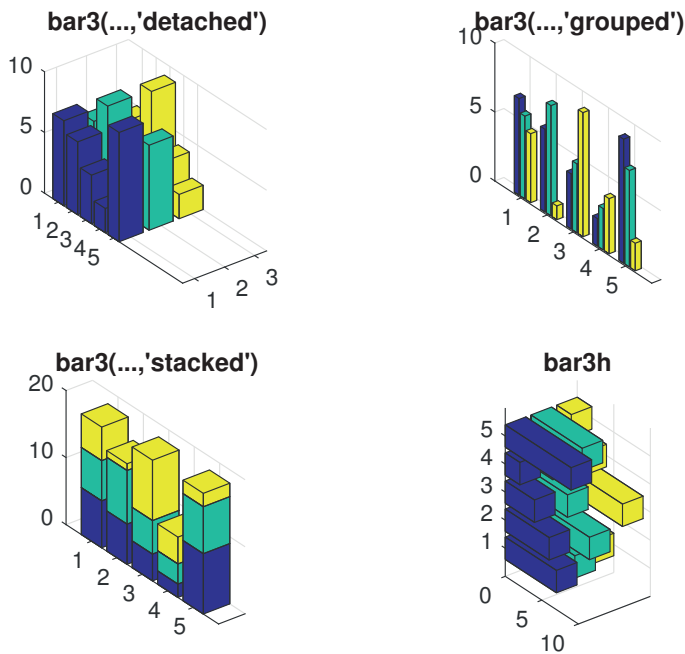


Figure 8.27. 3D bar plots.

```

title('Bin width 0.25')
bin_counts = h.Values

subplot(2,2,4)
histogram(y,16,'normalization','probability',...
          'Orientation','horizontal','FaceColor','red',...
          'EdgeColor','white','LineWidth',1)
title('Probability normalization')

```

The output of the code in the Command Window is

```

h =
    Histogram with properties:

        Data: [1000×1 double]
        Values: [19 164 313 275 123 62 26 9 2 3 3 1]
        NumBins: 12
        BinEdges: [1×13 double]
        BinWidth: 2.5000e-01
        BinLimits: [2.5000e-01 3.2500e+00]
        Normalization: 'count'
        FaceColor: [0 1 0]
        EdgeColor: [0 0 0]

    Show all properties
bin_counts =

```

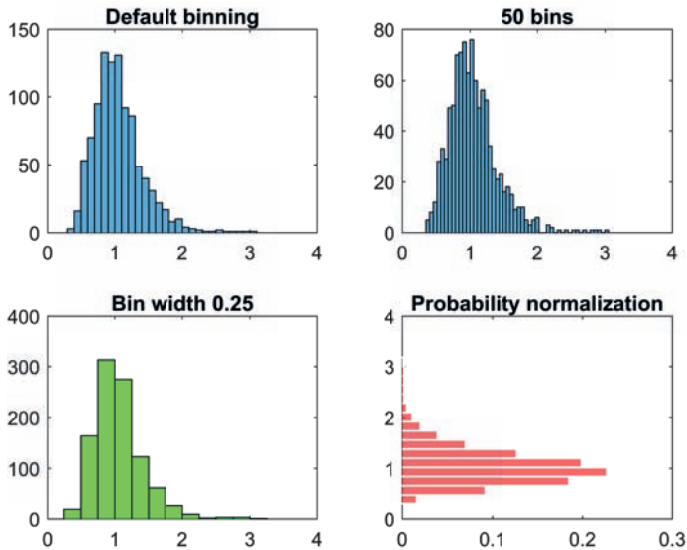


Figure 8.28. Histograms produced with `histogram`, for a 1000-by-1 data vector.

```
19 164 313 275 123 62 26 9 2 3 3 1
```

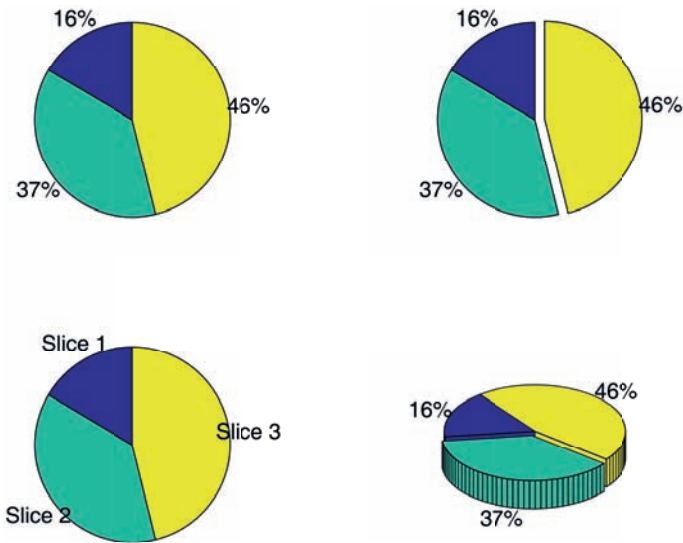
Here, `h` is a Histogram object. Properties of the histogram can be changed after plotting it by changing this object. The object also contains within it information about the histogram that can be inspected; here we printed the bin counts. In the last example we normalized the histogram so that the sum of all the bars is 1. See `doc histogram` for more detail on all these features.

Pie charts can be produced with `pie` and `pie3`. They take a vector argument, `x`, and corresponding to each element `x(i)` they draw a slice with area proportional to `x(i)`. A second argument `explode` can be given, which is a 0-1 vector with a 1 in positions corresponding to slices that are to be offset from the chart. By default, the slices are labeled with the percentage of the total area that they occupy; replacement labels can be specified in a cell array of strings (see Section 18.7). The following code produces Figure 8.29.

```
x = [1.5 3.4 4.2];
subplot(2,2,1), pie(x)
subplot(2,2,2), pie(x,[0 0 1])
subplot(2,2,3), pie(x,{'Slice 1','Slice 2','Slice 3'})
subplot(2,2,4), pie3(x,[0 1 0])
```

The `area` function produces a stacked area plot. With vector arguments, `area` is similar to `plot` except that the area between the `y`-values and 0 (or the level specified by the optional second argument) is filled; for matrix arguments, the plots of the columns are stacked, showing the sum at each `x`-value. The following code produces Figure 8.30.

```
rng(1)
x = [1:12 11:-1:8 10:15]; Y = [x' x']; % Y is 22-by-2.
```

Figure 8.29. *Pie charts.*

```
subplot(2,1,1)
area(Y+randn(size(Y)))
axis tight
```

```
subplot(2,1,2)
Y = Y + 5*randn(size(Y));
area(Y,min(min(Y)))
axis tight
```

As for `histogram`, all the functions in this section can return an object that can be used to manipulate the plot.

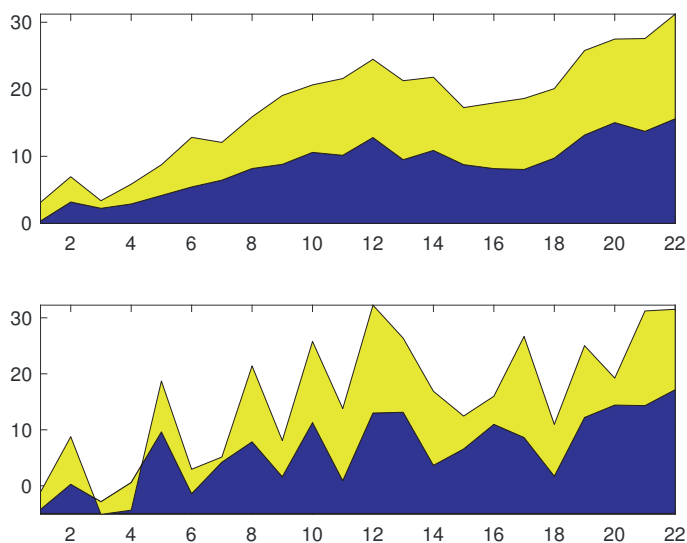
## 8.4. Saving and Printing Figures

If your default printer has been set appropriately, simply typing `print` will send the contents of the current figure window to your printer. An alternative is to use the `print` command to save the figure as a file. For example,

```
print -dpdf myfig.pdf
```

creates a PDF file `myfig.pdf` that can subsequently be printed or included in a document. This file can be incorporated into a `LATEX` document, as in the following outline:

```
\documentclass{article}
\usepackage{graphicx}
...
\begin{document}
...
\begin{center}
```

Figure 8.30. *Area graphs.*

```

\includegraphics[width=8cm]{myfig.pdf}
\end{center}
...
\end{document}

```

See [57], [107], or [112] for more about L<sup>A</sup>T<sub>E</sub>X.

The many options of the `print` command can be viewed with `help print`. The `print` command also has a functional form, illustrated by

```
print('-pdf','myfig.pdf')
```

(an example of command/function duality—see Section 7.5). To illustrate the utility of the functional form, the next example generates a sequence of five figures and saves them to files `fig1.pdf`, ..., `fig5.pdf`:

```

x = linspace(0,2*pi,50);
for i=1:5
    plot(x,sin(i*x))
    print('-dpdf',['fig' int2str(i) '.pdf'])
end

```

The second argument to the `print` command is formed by string concatenation (see Section 18.1), making use of the function `int2str`, which converts its integer argument to a string. Thus when `i=1`, for example, the `print` statement is equivalent to `print('-dpdf2','fig1.pdf')`.

It is important to realize that graphics that look good on the screen may not look so good in print. In particular, if you accept the default values of the various graphics parameters your printed figures could be hard to read. Compare the first two plots in Figure 8.31: in the first we used the default parameters while in the second we increased `LineWidth`, `FontSize`, and `MarkerSize` and also adjusted the

axis tick lengths. To produce visually attractive, readable printed figures it is usually necessary to increase parameters such as these from their default values. This can be done in three ways:

1. By appending modifiers such as `'FontSize',12` to the relevant commands.
2. By resetting the default values for properties prior to creating the MATLAB figure. See Section 17.2 for details of this approach.
3. By changing font size, line thickness, etc., after the figure has been drawn, by setting graphics objects properties, as described in Section 17.4.

Another approach to producing graphics for publication is to convert figures to native L<sup>A</sup>T<sub>E</sub>X figures in the language of the TikZ and PGFPlots packages. A function `matlab2tikz` available on MATLAB Central File Exchange carries out this task and was used to produce the third plot in Figure 8.31.

An alternative to the `print` command is the function `export_fig`, which is not part of MATLAB but is available from the MATLAB Central File Exchange. Like `print`, this function writes a figure to a file in a variety of possible formats, but it aims to produce a more faithful representation of the figure displayed on the screen, overcoming some quirks of `print`. Many of the figures in this book were saved using the following function.

```
function myprint(filename,prnt)
if nargin < 2, prnt = 1; end
if prnt == 1
    set(gcf,'Color','w');
    export_fig(['../figs/' filename '.pdf'])
end
```

The `saveas` command saves a figure to a file in a form that can be reloaded into MATLAB. For example,

```
saveas(gcf,'myfig','fig')
```

saves the current figure as a binary FIG-file, which can be reloaded into MATLAB with `open('myfig.fig')`.

It is also possible to save and print figures from the File menu in the figure window.

## 8.5. On Things Not Treated

We have restricted our treatment in this chapter to high-level graphics functions that deal with common 2D and 3D visualization tasks. The graphics capabilities of MATLAB extend far beyond what is described here. On the one hand, MATLAB provides access to lighting, transparency control, solid model building, texture mapping, and the construction of GUIs. On the other hand, it is possible to control low-level details such as the tick labels and the position and size of the axes, and to produce animation; how to do this is described in Chapter 17 on advanced graphics. You can learn more by reading the MATLAB documentation and by exploring the demonstrations in the `matlab\demos` directory. Try `help demos`, but note that not all files in this directory are documented in the help information.

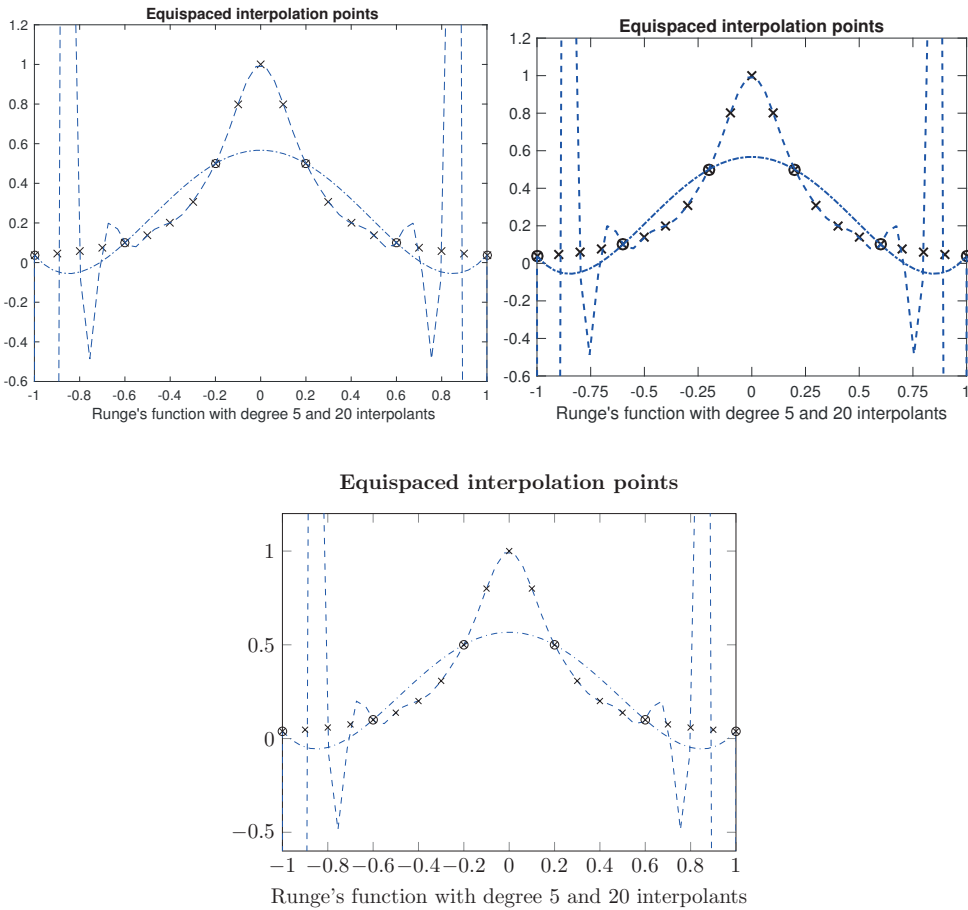


Figure 8.31. Three versions of the same plot. Top left: default parameters. Top right: parameters tuned for the printed page. Bottom: converted to  $\text{\LaTeX}$  tikzpicture environment using matlab2tikz function.



Figure 8.32. *From the 1964 Gatlinburg Conference on Numerical Algebra. From left to right: J. H. Wilkinson, W. J. Givens, G. E. Forsythe, A. S. Householder, P. Henrici, and F. L. Bauer. (Source of photograph: Oak Ridge National Laboratory, U.S. Dept. of Energy.)*

Another area of MATLAB that we have not discussed is image handling and manipulation. If you type `what demos`, you will find that the `demos` directory contains a selection of MAT-files, most of which contain image data. These can be loaded and displayed as in the following example, which produces the image shown in Figure 8.32:

```
load gatlin, image(X); colormap(map), axis off
```

This picture was taken at a meeting in Gatlinburg, Tennessee, in 1964, and shows six major figures in the development of numerical linear algebra and scientific computing (you can find some of their names in Table 5.3).

Before coding graphs in MATLAB you should think carefully about the design, aiming for a result that is uncluttered and clearly conveys the intended message. Good references on graphical design are [11, Chaps. 10, 11], [169], [170], [171].

Finally, for many more examples of creative MATLAB graphics, see the books by D. J. Higham [67] and Trefethen [165], [166].



*“What is the use of a book,” thought Alice,  
“without pictures or conversation?”*

— LEWIS CARROLL, *Alice’s Adventures in Wonderland* (1865)

*The close command closes the current figure window.  
If there is no open figure window MATLAB opens one and then closes it.*

— CLEVE B. MOLER

*A picture is worth a thousand words.*

— ANONYMOUS

*Given their low data-density and  
failure to order numbers along a visual dimension,  
pie charts should never be used.*

— EDWARD R. TUFTE, *The Visual Display of Quantitative Information* (1983)

*It’s kind of scandalous that the world’s calculus books,  
up until recent years, have never had a good picture<sup>4</sup> of a cardioid...  
Nobody ever knew what a cardioid looked like, when I took calculus,  
because the illustrations were done by graphic artists  
who were trying to imitate drawings by previous artists,  
without seeing the real thing.*

— DONALD E. KNUTH, *Digital Typography* (1999)

---

<sup>4</sup>`ezpolar(@(t)1+cos(t))`

# Chapter 9

## Linear Algebra

MATLAB was originally designed for linear algebra computations, so it not surprising that it has a rich set of functions for solving linear equation and eigenvalue problems. Many of the linear algebra functions are based on routines from the LAPACK [3] Fortran library.

Most of the linear algebra functions work for both real and complex matrices. We write  $A^T$  for the transpose of  $A$  and  $A^*$  for the conjugate transpose of  $A$ . Recall that a square matrix  $A$  is Hermitian if  $A^* = A$ , symmetric if  $A^T = A$ , and unitary if  $A^*A = I$ , where  $I$  is the identity matrix. In addition, a matrix is skew-Hermitian if  $A^* = -A$  and skew-symmetric if  $A^T = -A$ .

To avoid clutter, we use the appropriate adjectives for complex matrices. Thus, when the matrix is real, “Hermitian” can be read as “symmetric” and “unitary” can be read as “orthogonal”. For background on numerical linear algebra see [31], [53], [75], [160], [167], [178], and, particularly for the algorithmic aspects, [8], [157], [158].

### 9.1. Matrix Properties

It is useful to be able to check what properties a matrix has, perhaps in order to select a course of action or to verify that an expected property is present at the beginning of a computation. MATLAB has a number of “is functions” for this purpose, which are listed in Table 9.1. Here is an example:

```
>> format shortg
>> A = [1 1i; 1i 0], B = [1 1i; -1i 0]
A =
      1 +      0i      0 +      1i
      0 +      1i      0 +      0i
B =
      1 +      0i      0 +      1i
      0 -      1i      0 +      0i

>> [ishermitian(A), issymmetric(A)]
ans =
  1×2 logical array
   0   1

>> [ishermitian(B), issymmetric(B)]
ans =
  1×2 logical array
   1   0
```

Table 9.1. Logical `is*` functions for matrices.

<code>isbanded</code>	Test for banded matrix
<code>isdiag</code>	Test for diagonal matrix
<code>ishermitian</code>	Test for Hermitian or skew-Hermitian matrix
<code>issymmetric</code>	Test for symmetric or skew-symmetric matrix
<code>istril</code>	Test for lower triangular matrix
<code>istriu</code>	Test for upper triangular matrix

## 9.2. Norms and Condition Numbers

A norm is a scalar measure of the size of a vector or matrix. The  $p$ -norm of an  $n$ -vector  $x$  is defined by

$$\|x\|_p = \left( \sum_{i=1}^n |x_i|^p \right)^{1/p}, \quad 1 \leq p < \infty.$$

For  $p = \infty$  the norm is defined by

$$\|x\|_\infty = \max_{1 \leq i \leq n} |x_i|.$$

The norm function can compute any  $p$ -norm and is invoked as `norm(x,p)`, with default  $p = 2$ . As a special case, for  $p = -\text{inf}$  the quantity  $\min_i |x_i|$  is computed. Example:

```
>> x = 1:4; format
>> [norm(x,1) norm(x,2) norm(x,inf) norm(x,-inf)]
ans =
    10.0000     5.4772     4.0000     1.0000
```

The  $p$ -norm of an  $m$ -by- $n$  matrix is defined by

$$\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}.$$

The 1- and  $\infty$ -norms can be characterized as

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|, \quad \text{“max column sum”},$$

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|, \quad \text{“max row sum”}.$$

The 2-norm of  $A$  can be expressed as the largest singular value of  $A$ , `max(svd(A))` (singular values and the `svd` function are described in Section 9.7). For matrices the norm function is invoked as `norm(A,p)` and supports  $p = 1, 2, \text{inf}$  and  $p = \text{'fro'}$ , the Frobenius norm

$$\|A\|_F = \left( \sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2}.$$

(The norm function is an example of a function with an argument that can vary in type:  $p$  can be a double or a string.) Example:

```

>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> [norm(A,1) norm(A,2) norm(A,inf) norm(A,'fro')]
ans =
    18.0000    16.8481    24.0000    16.8819

```

For cases in which computation of the 2-norm of a matrix is too expensive the function `normest` can be used to obtain an estimate. The call `normest(A,tol)` uses the power method on  $A^*A$  to estimate  $\|A\|_2$  to within a relative error `tol`; the default is `tol = 1e-6`.

For a nonsingular square matrix  $A$ , the quantity  $\kappa(A) = \|A\| \|A^{-1}\| \geq 1$  is called the condition number with respect to inversion. It measures the sensitivity of the solution of a linear system  $Ax = b$  to perturbations in  $A$  and  $b$ . The matrix  $A$  is said to be well conditioned or ill conditioned according to whether  $\kappa(A)$  is small or large, where the meaning of “small” and “large” depends on the context. A condition number of the order of the reciprocal of `eps` is certainly regarded as large, because it implies that  $A$  is within relative distance about `eps` of a singular matrix. The condition number is computed by the `cond` function as `cond(A,p)`. The  $p$ -norm choices `p = 1,2,inf,'fro'` are supported, with default `p = 2`. For `p = 2`, rectangular matrices are allowed, in which case the condition number is defined by  $\kappa_2(A) = \|A\|_2 \|A^+\|_2$ , where  $A^+$  is the pseudo-inverse (see Section 9.4).

Computing the exact condition number is expensive, so MATLAB provides two functions for estimating the 1-norm condition number of a square matrix  $A$ , `rcond` and `condest`. Both functions produce estimates usually of the correct order of magnitude at about one-third the cost of explicitly computing  $A^{-1}$ . Function `rcond` uses the LAPACK condition estimator to estimate the reciprocal of  $\kappa_1(A)$ , producing a result between 0 and 1, with 0 signaling exact singularity. Function `condest` estimates  $\kappa_1(A)$  and also returns an approximate null vector, which is required in some applications. The command `[c,v] = condest(A)` produces a scalar `c` and vector `v` such that  $c \leq \kappa_1(A)$  and  $\text{norm}(A*v,1) = \text{norm}(A,1)*\text{norm}(v,1)/c$ . Example:

```

>> A = gallery('grcar',8);

>> [cond(A,1) 1/rcond(A) condest(A)]
ans =
    7.7778    5.3704    7.7778

>> [cond(A,1) 1/rcond(A) condest(A)]
ans =
    7.7778    5.3704    7.2222

```

As this example illustrates, `condest` does not necessarily return the same result on each invocation, as it makes use of `rand`.

The `condest` function makes use of a function `normest1` that estimates the 1-norm of a matrix. The matrix,  $B$  say, can be given implicitly by a handle to a function that evaluates  $Bx$  or  $B^*x$  given  $x$  (function handles are described in Section 10.1). The `normest1` function, which implements the algorithm of [84], can therefore be

used to estimate  $\|A^{-1}\|_1$  given an LU factorization of  $A$ , as well as more complicated condition number-related quantities.

### 9.3. Linear Equations

The fundamental tool for solving a linear system of equations is the backslash operator `\`. It handles three types of linear system  $Ax = b$ , where the matrix  $A$  and the vector  $b$  are given. The three possible shapes for  $A$  lead to square, overdetermined, and underdetermined systems, as described below. More generally, the backslash operator can be used to solve  $AX = B$ , where  $B$  is a matrix with  $p$  columns; in this case MATLAB solves  $AX(:,j) = B(:,j)$  for  $j = 1:p$ .

#### 9.3.1. Square System

If  $A$  is an  $n$ -by- $n$  nonsingular matrix then `A\b` is the solution  $x$  to  $Ax = b$ , computed by LU factorization with partial pivoting. During the solution process MATLAB computes `rcond(A)`, and it prints a warning message if the result is smaller than about `eps`:

```
>> x = hilb(15)\ones(15,1);
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 1.024999e-18.
```

These warning messages can be turned off using `warning off` (see Section 14.2).

MATLAB recognizes three special forms of square matrices  $A$  in `A\b` and takes advantage of them to reduce the computation.

- Triangular matrices, or permutations of triangular matrices. (The square matrix  $A$  is upper triangular if  $a_{ij} = 0$  for  $i > j$ , and lower triangular if  $a_{ij} = 0$  for  $j > i$ .) The system is solved by substitution.
- Upper Hessenberg matrices. (The square matrix  $A$  is upper Hessenberg if  $a_{ij} = 0$  for  $i > j + 1$ .) The system is solved by LU factorization with partial pivoting, taking advantage of the Hessenberg form.
- Hermitian matrices or Hermitian positive definite matrices. (The Hermitian matrix  $A$  is positive definite if  $x^*Ax > 0$  for all nonzero vectors  $x$ , or, equivalently, if all the eigenvalues are real and positive.) Cholesky factorization is used if the matrix is Hermitian positive definite and `LDL*` factorization if the matrix is merely Hermitian. How does MATLAB know whether the matrix is positive definite? If the matrix has positive diagonal elements MATLAB attempts to Cholesky factorize the matrix. If the Cholesky factorization succeeds it is used to solve the system, otherwise the system is not definite and `LDL*` factorization is used.

In general, `A\B` denotes the solution to  $AX = B$ , and it can also be written in functional form as `mldivide(A,B)`. With the forward slash operator, `/`, `A/B` denotes the solution to  $XA = B$ , and it can be written `mrdivide(A,B)`. The expressions `A\B` and `(B'/A')'` are equivalent, but in practice usually differ due to rounding errors.

When efficiency is important and  $A$  is structured, it may be appropriate to use the `linsolve` function in place of the backslash operator. The syntax is `X =`

`linsolve(A,B,opts)`, which solves  $AX = B$ , or  $A^*X = B$  if `opts.TRANS` has the value `true`. The structure `opts` (see Section 18.7 for details of structures) specifies one of several possible matrix properties, and `linsolve` gains efficiency over backslash by not attempting to determine matrix properties itself. Table 9.2 shows how to set `opts` in order to use `linsolve` to solve some particular types of structured system. The usage is illustrated by the following script:

```
n = 8; rng(1), format short e
B = rand(n,2);

% Upper triangular A.
A = triu(rand(n));
opts = struct('UT',true);
X = linsolve(A,B,opts);
res = norm(B-A*X)

% Give LINSOLVE incorrect opts structure.
opts = struct('LT',true);
X = linsolve(A,B,opts);
res = norm(B-A*X)

% Upper triangular A, solve transposed system.
opts = struct('UT',true,'TRANS',true);
X = linsolve(A,B,opts);
res = norm(B-A'*X)

% Symmetric indefinite A.
A = rand(n); A = A + A';
opts = struct('SYM',true);
X = linsolve(A,B,opts);
res = norm(B-A*X)

% Symmetric positive definite A.
A = rand(n); A = A*A';
opts = struct('SYM',true,'POSDEF',true);
X = linsolve(A,B,opts);
res = norm(B-A*X)
```

The output is

```
res =
  1.5701e-16
res =
  5.7384
res =
  2.2204e-16
res =
  4.0462e-15
res =
  3.0250e-14
```

Table 9.2. *Some examples of how to set the opts structure in linsolve.*

Matrix property	opts structure
Lower triangular	<code>opts = struct('LT', true)</code>
Upper triangular	<code>opts = struct('UT', true)</code>
Upper Hessenberg	<code>opts = struct('UHESS', true)</code>
Symmetric/Hermitian	<code>opts = struct('SYM', true)</code>
Symmetric/Hermitian and positive definite	<code>opts = struct('SYM', true, 'POSDEF', true)</code>
(Conjugate) transpose: <code>linsolve</code> solves $A^*X = B$	<code>opts = struct('TRANSA', true)</code>

The residuals in this example should be of order `eps` for a correct solution. The second call to `linsolve` wrongly specifies the matrix  $A$  as lower triangular. Since `linsolve` performs no checks on the matrix properties it gives an incorrect answer without any warning or error. Hence care is required in the use of this function. The execution time saved by using `linsolve` instead of backslash is highly dependent on the matrix property and the dimension of the matrix. Rectangular systems, as described in the next two subsections, can also be solved by `linsolve`; see the online help for details.

A special type of linear system is the Sylvester equation  $AX + XB = C$ , where  $A$  is  $m$ -by- $m$ ,  $B$  is  $n$ -by- $n$ , and  $C$  and  $X$  are  $m$ -by- $n$ . This equation is solved with the `sylvester` function: `X = sylvester(A,B,C)`. The equation is nonsingular when  $A$  and  $-B$  have no eigenvalue in common.

#### THE BACKSLASH NOTATION

The notation  $A \setminus B$  for  $A^{-1}B$  was suggested in a 1928 paper by Hensel [65], along with  $A/B$  for  $AB^{-1}$ . The suggestion does not appear to have been taken up by anyone else. Cleve Moler independently invented the backslash notation for MATLAB [127] and the term “backslash” is now commonly used to mean solving a matrix equation.

Cayley devised two different notations for the products  $AB^{-1}$  and  $B^{-1}A$ , in the context of groups:

$$\left. \begin{array}{c} A \\ B \end{array} \right|, \quad \left| \begin{array}{c} A \\ B \end{array} \right|$$

[16, p. 71], [22]; and, in an 1860 letter to Sylvester [138, Letter 45],

$$\underset{\sim}{A} \underset{\sim}{B}, \quad \underset{\sim}{A} \underset{\sim}{B}.$$

Given the difficulty of typesetting them, it is not surprising that these notations did not catch on.

### 9.3.2. Overdetermined System

If  $A$  has dimension  $m$ -by- $n$  with  $m > n$  then  $Ax = b$  is an overdetermined system: there are more equations than unknowns. In general, there is no  $x$  satisfying the

system. The vector  $A \setminus b$  is a least-squares solution to the system, that is, it minimizes  $\text{norm}(A*x-b)$  (the 2-norm of the residual) over all vectors  $x$ . If  $A$  has full rank  $n$  there is a unique least-squares solution. If  $A$  has rank  $k$  less than  $n$  then  $A \setminus b$  is a basic solution—one with at most  $k$  nonzero elements ( $k$  is determined, and  $x$  computed, using the QR factorization with column pivoting). In the latter case MATLAB displays a warning message.

A least-squares solution to  $Ax = b$  can also be computed as  $x_{\text{min}} = \text{pinv}(A)*b$ , where the function `pinv` computes the pseudo-inverse (see Section 9.4). In the case where  $A$  is rank-deficient,  $x_{\text{min}}$  is the unique solution of minimal 2-norm.

A vector that minimizes the 2-norm of  $Ax - b$  over all nonnegative vectors  $x$ , for real  $A$  and  $b$ , is computed by `lsqnonneg`. The simplest usage is  $x = \text{lsqnonneg}(A,b)$ , and several other input and output arguments can be specified, including a starting vector for the iterative algorithm that is used. Example:

```
>> A = gallery('lauchli',3,0.25), b = [1 2 4 8]';
A =
    1.0000    1.0000    1.0000
    0.2500         0         0
         0    0.2500         0
         0         0    0.2500

>> x = A \ b;           % Least-squares solution.
>> xn = lsqnonneg(A,b); % Nonnegative least-squares solution.

>> [x xn], [norm(A*x-b) norm(A*xn-b)]
ans =
   -9.9592         0
   -1.9592         0
   14.0408    2.8235
ans =
    7.8571    8.7481
```

### 9.3.3. Underdetermined System

If  $A$  has dimension  $m$ -by- $n$  with  $m < n$  then  $Ax = b$  is an underdetermined system: there are fewer equations than unknowns. The system has either no solution or infinitely many. In the latter case  $A \setminus b$  produces a basic solution. This solution is generally not the solution of minimal 2-norm, which can be computed as  $\text{pinv}(A)*b$ . If the system has no solution (that is, it is inconsistent) then  $A \setminus b$  is a least-squares solution. Here is an example that illustrates the difference between the  $\setminus$  and `pinv` solutions:

```
>> A = [1 1 1; 1 1 -1], b = [3; 1]
A =
     1     1     1
     1     1    -1
b =
     3
     1
```



```

>> x = A\b; y = pinv(A)*b;
>> [x y]
ans =
    2.0000    1.0000
         0    1.0000
    1.0000    1.0000

>> [norm(x) norm(y)]
ans =
    2.2361    1.7321

```

## 9.4. Inverse, Pseudoinverse, and Determinant

The inverse of an  $n$ -by- $n$  matrix  $A$  is a matrix  $X$  satisfying  $AX = XA = I$ , where  $I$  is the identity matrix (`eye(n)`). A matrix without an inverse is called singular. A singular matrix can be characterized in several ways: in particular, its determinant is zero and it has a nonzero null vector, that is, there exists a nonzero vector  $v$  such that  $Av = 0$ .

The matrix inverse is computed by the function `inv`. For example:

```

>> A = pascal(3), X = inv(A)
A =
     1     1     1
     1     2     3
     1     3     6

X =
     3.0000    -3.0000     1.0000
    -3.0000     5.0000    -2.0000
     1.0000    -2.0000     1.0000

>> norm(A*X-eye(3))
ans =
    4.9651e-16

```

The inverse is formed using LU factorization with partial pivoting and the reciprocal condition estimate `rcond` is computed. A warning message is produced if exact singularity is detected or if `rcond` is smaller than about `eps`.

Note that it is rarely necessary to compute the inverse of a matrix. For example, solving a square linear system  $Ax = b$  by `A\b` is 2–3 times faster than by `inv(A)*b` and often produces a smaller residual. It is usually possible to reformulate computations involving a matrix inverse in terms of linear system solving, so that explicit inversion is avoided.

The determinant of a square matrix is computed by the function `.`. It is calculated from the LU factors. Although the computation is affected by rounding errors in general, `det(A)` returns an integer when  $A$  has integer entries:

```

>> A = vander(1:5)
A =
     1     1     1     1     1

```

```

    16     8     4     2     1
    81    27     9     3     1
   256   64    16     4     1
   625  125    25     5     1

```

```

>> det(A)
ans =
    288

```

It is not recommended to test for nearness to singularity using `det`. Instead, `cond`, `rcond`, or `condst` should be used.

The (Moore–Penrose) pseudo-inverse generalizes the notion of inverse to rectangular and rank-deficient matrices  $A$ . It is written  $A^+$  and is computed with `pinv(A)`. The pseudo-inverse  $A^+$  of  $A$  can be characterized as the unique matrix  $X = A^+$  satisfying the four conditions  $AXA = A$ ,  $XAX = X$ ,  $(XA)^* = XA$ , and  $(AX)^* = AX$ . It can also be written explicitly in terms of the singular value decomposition (SVD): if the SVD of  $A$  is given by (9.1) on p. 146 then  $A^+ = V\Sigma^+U^*$ , where  $\Sigma^+$  is  $n$ -by- $m$  diagonal with  $(i, i)$  entry  $1/\sigma_i$  if  $\sigma_i > 0$  and 0 otherwise. To illustrate,

```

>> pinv(ones(3))
ans =
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111

```

and if

```

A =
    0     0     0     0
    0     1     0     0
    0     0     2     0

```

then

```

>> pinv(A)
ans =
    0         0         0
    0    1.0000         0
    0         0    0.5000
    0         0         0

```

## 9.5. LU, LDL\*, and Cholesky Factorizations

An LU factorization of a square matrix  $A$  is a factorization  $A = LU$  in which  $L$  is unit lower triangular (that is, lower triangular with ones on the diagonal) and  $U$  is upper triangular. Not every matrix can be factorized in this way, but when row interchanges are incorporated the factorization always exists. The `lu` function computes an LU factorization with partial pivoting  $PA = LU$ , where  $P$  is a permutation matrix. The call `[L,U,P] = lu(A)` returns the triangular factors and the permutation matrix. The permutations can be returned in the more storage-efficient form of a vector using the call `[L,U,p] = lu(A, 'vector')`, after which `A(p, :) = L*U`. See Section 24.3 for how

to convert between the vector and matrix representations of a permutation. With just two output arguments,  $[L,U] = \text{lu}(A)$  returns  $L = P^T L$ , so  $L$  is a triangular matrix with its rows permuted. Example:

```
>> format short g
>> A = gallery('binomial',3), [L,U] = lu(A)
A =
     1     2     1
     1     0    -1
     1    -2     1
L =
           1           0           0
           1          0.5           1
           1           1           0
U =
     1     2     1
     0    -4     0
     0     0    -2
```

The `lu` function also works for rectangular matrices. If  $A$  is  $m$ -by- $n$  then  $[L,U] = \text{lu}(A)$  produces an  $m$ -by- $n$   $L$  and  $n$ -by- $n$   $U$  if  $m \geq n$  and an  $m$ -by- $m$   $L$  and  $m$ -by- $n$   $U$  if  $m < n$ .

Using  $x = A \setminus b$  to solve a linear system  $Ax = b$  with a square  $A$  is equivalent to LU factorizing the matrix and then solving with the factors:

```
[L,U] = lu(A); x = U \ (L \ b);
```

As noted in Section 9.3.1, MATLAB takes advantage of the fact that  $L$  is a permuted triangular matrix when forming  $L \setminus b$ . An advantage of this two-step approach is that if further linear systems involving  $A$  are to be solved then the LU factors can be reused, with a saving in computation.

A Hermitian matrix can be factorized  $PAP^T = LDL^*$ , where  $P$  is a permutation matrix,  $L$  is unit lower triangular, and  $D$  is block diagonal with diagonal blocks of dimension 1 or 2. Several variants of this factorization exist, corresponding to different ways of choosing  $P$ . The MATLAB function `ldl` uses symmetric rook pivoting [70, Sec. 11.1.3].

```
>> A = gallery('fiedler',3), [L,D,P] = ldl(A)
A =
     0     1     2
     1     0     1
     2     1     0
L =
           1           0           0
           0           1           0
     0.5           0.5           1
D =
     0     2     0
     2     0     0
     0     0    -1
P =
     1     0     0
```

```

0    0    1
0    1    0

```

The appearance of a 2-by-2 block on the diagonal of  $D$ , as in this case with  $D(1:2, 1:2)$ , is a sign that  $A$  is indefinite, that is, has at least one negative eigenvalue.

Any Hermitian positive definite matrix has a Cholesky factorization  $A = R^*R$ , where  $R$  is upper triangular with real, positive diagonal elements. The Cholesky factor is computed by  $R = \text{chol}(A)$ . For example:

```

>> A = pascal(4)
A =
    1    1    1    1
    1    2    3    4
    1    3    6   10
    1    4   10   20

>> R = chol(A)
R =
    1    1    1    1
    0    1    2    3
    0    0    1    3
    0    0    0    1

```

Note that `chol` looks only at the elements in the upper triangle of  $A$  (including the diagonal)—it factorizes the Hermitian matrix agreeing with the upper triangle of  $A$ . An error is produced if  $A$  is not positive definite. The `chol` function can be used to test whether a matrix is positive definite (indeed, this is as good a test as any) using the call  $[R,p] = \text{chol}(A)$ , where the integer  $p$  will be zero if the factorization succeeds and positive otherwise; see `help chol` for more details about `p`.

Function `cholupdate` modifies the Cholesky factorization when the original matrix is subjected to a rank-1 perturbation (either an update,  $+xx^*$ , or a downdate,  $-xx^*$ ).

## 9.6. QR Factorization

A QR factorization of an  $m$ -by- $n$  matrix  $A$  is a factorization  $A = QR$ , where  $Q$  is  $m$ -by- $m$  unitary and  $R$  is  $m$ -by- $n$  upper triangular. This factorization is very useful for the solution of least-squares problems and for constructing an orthonormal basis for the columns of  $A$ . The command  $[Q,R] = \text{qr}(A)$  computes the factorization, while when  $m > n$   $[Q,R] = \text{qr}(A,0)$  produces an “economy size” version in which  $Q$  has only  $n$  columns and  $R$  is  $n$ -by- $n$ . Example:

```

>> format short e, A
A =
    1    0    1
    1   -1    1
    2    0    0

>> [Q,R] = qr(A)
Q =
-4.0825e-01    1.8257e-01   -8.9443e-01
-4.0825e-01   -9.1287e-01   -5.5511e-17

```

```

-8.1650e-01   3.6515e-01   4.4721e-01
R =
-2.4495e+00   4.0825e-01  -8.1650e-01
              0   9.1287e-01  -7.3030e-01
              0           0   -8.9443e-01

```

A QR factorization with column pivoting has the form  $AP = QR$ , where  $P$  is a permutation matrix. The permutation strategy that is used produces a factor  $R$  whose diagonal elements are nonincreasing:  $|r_{11}| \geq |r_{22}| \geq \dots \geq |r_{nn}|$ . Column pivoting is particularly appropriate when  $A$  is suspected of being rank-deficient, as it helps to reveal near rank-deficiency. Roughly speaking, if  $A$  is near a matrix of rank  $r < n$  then the last  $n - r$  diagonal elements of  $R$  will be of order `eps*norm(A)`. A third output argument forces the function `qr` to use column pivoting and return the permutation matrix: `[Q,R,P] = qr(A)`. The syntax `[Q,R,p] = qr(A, 'vector')` causes a permutation vector `p` to be returned such that  $A(:, p) = Q \cdot R$ . If the economy size factorization with column pivoting is requested, via `[Q,R,p] = qr(A, 0)`, then `p` is a permutation vector. Continuing the previous example, we make  $A$  nearly singular and see how column pivoting reveals the near singularity in the last diagonal element of  $R$ :

```

>> A(2,2) = eps
A =
 1.0000e+000           0  1.0000e+000
 1.0000e+000  2.2204e-016  1.0000e+000
 2.0000e+000           0           0

>> [Q,R,P] = qr(A); R, P
R =
-2.4495e+000  -8.1650e-001  -9.0649e-017
              0  -1.1547e+000  -1.2820e-016
              0           0   1.5701e-016

P =
 1   0   0
 0   0   1
 0   1   0

```

Functions `qrdelete`, `qrinsert`, and `qrupdate` modify the QR factorization when a column of the original matrix is deleted or inserted and when a rank-1 perturbation is added.

## 9.7. Singular Value Decomposition

The SVD of an  $m$ -by- $n$  matrix  $A$  has the form

$$A = U \Sigma V^*, \quad (9.1)$$

where  $U$  is an  $m$ -by- $m$  unitary matrix,  $V$  is an  $n$ -by- $n$  unitary matrix, and  $\Sigma$  is a real  $m$ -by- $n$  diagonal matrix with  $(i, i)$  entry  $\sigma_i$ . The singular values  $\sigma_i$  satisfy  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m,n)} \geq 0$ . The SVD is an extremely useful tool [53], [77]. For example, the rank of  $A$  is the number of nonzero singular values and the smallest singular value is the 2-norm distance to the nearest rank-deficient matrix. The complete SVD is

computed using `[U,S,V] = svd(A)`; if only one output argument is specified then a vector of singular values is returned. Example:

```
>> A = reshape(1:9,3,3); format short e
>> svd(A)
ans =
    1.6848e+001    1.0684e+000    5.5431e-016
```

Here, the matrix is singular. The smallest computed singular value is at the level of the unit roundoff rather than zero because of rounding errors.

When  $m > n$  the command `[U,S,V] = svd(A,0)` produces an “economy size” SVD in which  $U$  is  $m$ -by- $n$  with orthonormal columns and  $S$  is  $n$ -by- $n$ . The call `[U,S,V] = svd(X,'econ')` produces the same result as `svd(X,0)` when  $m \geq n$ , but if  $m < n$  it returns an  $n$ -by- $m$   $V$  with orthonormal columns and an  $m$ -by- $m$   $S$ . Example:

```
>> B = gallery('triu',[2 4],-2)
B =
     1     -2     -2     -2
     0      1     -2     -2

>> [U,S,V] = svd(B,'econ')
U =
 -8.1124e-001 -5.8471e-001
 -5.8471e-001  8.1124e-001
S =
 4.1623e+000      0
      0  2.1623e+000
V =
 -1.9490e-001 -2.7041e-001
  2.4933e-001  9.1601e-001
  6.7076e-001 -2.0953e-001
  6.7076e-001 -2.0953e-001
```

Functions `rank`, `null`, and `orth` compute, respectively, the rank, an orthonormal basis for the null space, and an orthonormal basis for the range of their matrix argument. All three base their computation on the SVD, using a tolerance proportional to `eps` to decide when a computed singular value can be regarded as zero. For example, using the previous matrix:

```
>> format, rank(A)
ans =
     2

>> null(A)
ans =
 -0.4082
  0.8165
 -0.4082

>> orth(A)
ans =
 -0.4797    0.7767
```

```

-0.5724    0.0757
-0.6651   -0.6253

```

Another function connected with rank computations is `rref`, which computes the reduced row echelon form. Since the computation of this form is very sensitive to rounding errors, this function is mainly of pedagogical interest.

The generalized SVD of an  $m$ -by- $p$  matrix  $A$  and an  $n$ -by- $p$  matrix  $B$  can be written

$$A = UCX^*, \quad B = VSX^*, \quad C^*C + S^*S = I,$$

where  $U$  and  $V$  are unitary,  $X$  is nonsingular, and  $C$  and  $S$  are real diagonal matrices with nonnegative diagonal elements. The numbers  $C(i, i)/S(i, i)$  are the generalized singular values. This decomposition is computed by `[U,V,X,C,S] = gsvd(A,B)`. See `help gsvd` for more details about the dimensions of the factors.

## 9.8. Eigenvalue Problems

Algebraic eigenvalue problems are straightforward to define, but their efficient and reliable numerical solution is a complicated subject. The MATLAB `eig` function simplifies the solution process by recognizing and taking advantage of the number of input matrices, as well as their structure and the output requested. It automatically chooses among 16 different algorithms or algorithmic variants, corresponding to

- the standard (`eig(A)`) or generalized (`eig(A,B)`) problem,
- real or complex matrices  $A$  and  $B$ ,
- symmetric/Hermitian  $A$  and  $B$  with  $B$  positive definite, or not,
- eigenvectors requested or not.

### 9.8.1. Eigenvalues

The scalar  $\lambda$  and the nonzero vector  $x$  are an eigenvalue and a corresponding eigenvector of the  $n$ -by- $n$  matrix  $A$  if  $Ax = \lambda x$ . The eigenvalues are the  $n$  roots of the degree- $n$  characteristic polynomial  $\det(\lambda I - A)$ . The  $n + 1$  coefficients of this polynomial are computed by `p = poly(A)`:

$$\det(\lambda I - A) = p_1\lambda^n + p_2\lambda^{n-1} + \cdots + p_n\lambda + p_{n+1}.$$

The eigenvalues of  $A$  are computed with the `eig` function: `e = eig(A)` assigns the eigenvalues to the vector `e`. More generally, `[V,D] = eig(A)` computes an  $n$ -by- $n$  diagonal matrix  $D$  and an  $n$ -by- $n$  matrix  $V$  such that  $A*V = V*D$ . Thus  $D$  contains eigenvalues on the diagonal, and the columns of  $V$  are eigenvectors. Not every matrix has  $n$  linearly independent eigenvectors, so the matrix  $V$  returned by `eig` may be singular (or, because of roundoff, nonsingular but very ill conditioned). The matrix in the following example has two eigenvalues 1 and only one eigenvector:

```

>> [V,D] = eig([2 -1; 1 0])
V =
    0.7071    0.7071
    0.7071    0.7071
D =
     1     0
     0     1

```

The scaling of eigenvectors is arbitrary (if  $x$  is an eigenvector then so is any nonzero multiple of  $x$ ). As the last example illustrates, MATLAB normalizes so that each column of  $V$  has unit 2-norm. Note that eigenvalues and eigenvectors can be complex, even for a real (non-Hermitian) matrix.

A Hermitian matrix has real eigenvalues and its eigenvectors can be taken to be mutually orthogonal. For Hermitian matrices MATLAB returns eigenvalues sorted in increasing order and the matrix of eigenvectors is unitary to working precision:

```
>> [V,D] = eig([2 -1; -1 1])
V =
   -0.5257   -0.8507
   -0.8507    0.5257
D =
    0.3820         0
         0    2.6180

>> norm(V'*V-eye(2))
ans =
   2.2204e-016
```

A nonzero vector  $y$  is a left eigenvector of  $A$  corresponding to an eigenvalue  $\lambda$  if  $y^*A = \lambda y^*$ . Left eigenvectors are returned in a third output argument of `eig`. The following example emphasizes that the eigenvalues of a real matrix are not necessarily real:

```
>> A = gallery('leslie',3)
A =
     1     1     1
     1     0     0
     0     1     0

>> [V,D,W] = eig(A)      % Columns of W are left eigenvectors.
V =
  -0.8503 + 0.0000i  -0.1412 - 0.3752i  -0.1412 + 0.3752i
  -0.4623 + 0.0000i  -0.3094 + 0.4471i  -0.3094 - 0.4471i
  -0.2514 + 0.0000i   0.7374 + 0.0000i   0.7374 + 0.0000i
D =
  1.8393 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
  0.0000 + 0.0000i  -0.4196 + 0.6063i   0.0000 + 0.0000i
  0.0000 + 0.0000i   0.0000 + 0.0000i  -0.4196 - 0.6063i
W =
  -0.7071 + 0.0000i  -0.4024 + 0.1719i  -0.4024 - 0.1719i
  -0.5935 + 0.0000i   0.6755 + 0.0000i   0.6755 + 0.0000i
  -0.3844 + 0.0000i   0.1190 - 0.5814i   0.1190 + 0.5814i

>> norm(A*V - V*D,1)    % Residual for right eigenvectors.
ans =
   1.0666e-15
norm(W'*A - D*W',1)    % Residual for left eigenvectors.
ans
=   1.2591e-15
```



In the next example `eig` is applied to the (non-Hermitian) Frank matrix:

```
>> F = gallery('frank',5)
F =
     5     4     3     2     1
     4     4     3     2     1
     0     3     3     2     1
     0     0     2     2     1
     0     0     0     1     1

>> e = eig(F)'
e =
  10.0629    3.5566    1.0000    0.0994    0.2812
```

This matrix has some special properties, one of which we can see by looking at the reciprocals of the eigenvalues:

```
>> 1./e
ans =
   0.0994   0.2812   1.0000  10.0629   3.5566
```

Thus if  $\lambda$  is an eigenvalue then so is  $1/\lambda$ . The reason is that the characteristic polynomial is anti-palindromic:

```
>> poly(F)
ans =
   1.0000  -15.0000   55.0000  -55.0000   15.0000  -1.0000
```

Thus  $\det(F - \lambda I) = -\lambda^5 \det(F - \lambda^{-1}I)$ .

Function `condeig` computes condition numbers for the eigenvalues: a large condition number indicates an eigenvalue that is sensitive to perturbations in the matrix. The following example displays eigenvalues in the first row and condition numbers in the second:

```
>> A = gallery('frank',6);
>> [V,D,s] = condeig(A);
>> [diag(D)'; s']
ans =
  12.9736   5.3832   1.8355   0.5448   0.0771   0.1858
   1.3059   1.3561   2.0412  15.3255  43.5212  56.6954
```

For this matrix the small eigenvalues are slightly ill conditioned.

### 9.8.2. More about Eigenvalue Computations

The function `eig` works in several stages. First, when  $A$  is nonsymmetric, it balances the matrix, that is, it carries out a similarity transformation  $A \leftarrow Y^{-1}AY$ , where  $Y$  is a permutation of a diagonal matrix chosen to give  $A$  rows and columns of approximately equal norm. The motivation for balancing is that it can lead to a more accurate computed eigensystem. However, balancing can worsen rather than improve accuracy (see `doc eig` for an example), so it may be necessary to turn balancing off with `eig(A, 'nobalance')`. Balancing can be carried out independently of `eig` using the `balance` function.

After balancing, `eig` reduces  $A$  to Hessenberg form, then uses the QR algorithm to reach Schur form, after which eigenvectors are computed by substitution if required. The Hessenberg factorization takes the form  $A = QHQ^*$ , where  $H$  is upper Hessenberg and  $Q$  is unitary. If  $A$  is Hermitian then  $H$  is Hermitian and tridiagonal. The Hessenberg factorization is computed by `H = hess(A)` or `[Q,H] = hess(A)`.

The real Schur decomposition of a real  $A$  has the form  $A = QTQ^T$ , where  $T$  is upper quasi-triangular, that is, block triangular with 1-by-1 and 2-by-2 diagonal blocks, and  $Q$  is orthogonal. The (complex) Schur decomposition has the form  $A = QTQ^*$ , where  $T$  is upper triangular and  $Q$  is unitary. If  $A$  is real then `T = schur(A)` and `[Q,T] = schur(A)` produce the real Schur decomposition. If  $A$  is complex then `schur` produces the complex Schur form. The complex Schur form can be obtained for a real matrix with `schur(A, 'complex')` (it differs from the real form only when  $A$  has one or more nonreal eigenvalues).

The Schur decomposition can be reordered (i.e., the eigenvalues placed in a different order on the diagonal blocks of  $T$ ) using the `ordschur` function. The function `ordeig` returns the eigenvalues of a quasi-triangular matrix in the order that they appear along the (block) diagonal; for such matrices it is more efficient than applying `eig`.

If  $A$  is real and symmetric (complex Hermitian), `[V,D] = eig(A)` reduces initially to symmetric (Hermitian) tridiagonal form then iterates to produce a diagonal Schur form, resulting in an orthogonal (unitary)  $V$  and a real, diagonal  $D$ .

### 9.8.3. Generalized Eigenvalues

The generalized eigenvalue problem is defined in terms of two  $n$ -by- $n$  matrices  $A$  and  $B$ :  $\lambda$  is an eigenvalue and  $x \neq 0$  a corresponding eigenvector if  $Ax = \lambda Bx$ . A left eigenvector  $y \neq 0$  corresponding to  $\lambda$  satisfies  $y^*A = \lambda y^*B$ .

The generalized eigenvalues are computed by `e = eig(A,B)`, while the two-output form `[V,D] = eig(A,B)` computes an  $n$ -by- $n$  diagonal matrix  $D$  and an  $n$ -by- $n$  matrix  $V$  of eigenvectors such that  $A*V = B*V*D$ . Left eigenvectors are returned in a third output argument: after `[V,D,W] = eig(A,B)` the columns of  $W$  contain the left eigenvectors and  $W'*A = D*W'*B$ .

The theory of the generalized eigenproblem is more complicated than that of the standard eigenproblem, with the possibility of zero, finitely many, or infinitely many eigenvalues and of eigenvalues that are infinitely large. When  $B$  is singular `eig` may return computed eigenvalues containing NaNs. To illustrate the computation of generalized eigenvalues:

```
>> A = gallery('triu',3), B = magic(3)
A =
     1     -1     -1
     0      1     -1
     0      0      1
B =
     8      1      6
     3      5      7
     4      9      2

>> [V,D] = eig(A,B); V, eivals = diag(D)'
V =
```

```

-1.0000   -1.0000    0.3526
  0.4844   -0.4574    0.3867
  0.2199   -0.2516   -1.0000
eivals =
  0.2751    0.0292   -0.3459

```

When  $A$  is Hermitian and  $B$  is Hermitian positive definite (the Hermitian definite generalized eigenproblem) the eigenvalues are real and  $A$  and  $B$  are simultaneously diagonalizable. In this case `eig` returns real computed eigenvalues sorted in increasing order, with the eigenvectors normalized (up to roundoff) so that  $V'*B*V = \text{eye}(n)$ ; moreover,  $V'*A*V$  is diagonal. The method that `eig` uses (Cholesky factorization of  $B$ , followed by reduction to a standard eigenproblem and solution by the QR algorithm) can be numerically unstable when  $B$  is ill conditioned. You can force `eig` to ignore the structure and solve the problem in the same way as for general  $A$  and  $B$  by invoking it as `eig(A,B,'qz')`; the QZ algorithm (see below) is then used, which has guaranteed numerical stability but does not guarantee real computed eigenvalues. Example:

```

>> A = gallery('fiedler',3), B = gallery('moler',3)
A =
    0     1     2
    1     0     1
    2     1     0
B =
    1    -1    -1
   -1     2     0
   -1     0     3

>> format short g
>> [V,D] = eig(A,B); V, eivals = diag(D)
V =
    0.55335    0.23393    2.3747
    0.15552   -0.57301    1.2835
   -0.36921    0.19163    0.90938
eivals =
   -0.75993   -0.30839    17.068

>> V'*A*V
ans =
   -0.75993   -1.6653e-16   -1.3323e-15
  -1.5959e-16   -0.30839    1.6515e-15
  -1.3323e-15    1.5543e-15    17.068

>> V'*B*V
ans =
         1   -2.2204e-16   -2.2204e-16
  -1.9429e-16         1   -1.6653e-16
  -1.1102e-16   -1.249e-16         1

```

The function `hess` computes the Hessenberg form of  $A$  and  $B$ :  $QAZ = H$ ,  $QBZ = T$ , where  $H$  is upper Hessenberg,  $T$  is upper triangular, and  $Q$  and  $Z$  are unitary.

The syntax is `[H,T,Q,Z] = hess(A,B)`.

The generalized Schur decomposition of a pair of matrices  $A$  and  $B$  has the form

$$QAZ = T, \quad QBZ = S,$$

where  $Q$  and  $Z$  are unitary and  $T$  and  $S$  are upper triangular. The generalized eigenvalues are the ratios  $T(i,i)/S(i,i)$  of the diagonal elements of  $T$  and  $S$ . The generalized real Schur decomposition of real  $A$  and  $B$  has the same form, with  $Q$  and  $Z$  orthogonal and  $T$  and  $S$  upper quasi-triangular. These decompositions are computed by the `qz` function with the command `[T,S,Q,Z,V,W] = qz(A,B)`, where the output arguments  $V$  and  $W$  are matrices of generalized right eigenvectors and left eigenvectors, respectively. The function is named after the QZ algorithm that it implements. By default the (possibly) complex form with upper triangular  $T$  and  $S$  is produced. For real matrices, `{qz(A,B,'real')}` produces the real form and `{qz(A,B,'complex')}` the default complex form. The generalized Schur decomposition can be reordered using the `ordqz` function.

Function `polyeig` solves the polynomial eigenvalue problem  $(\lambda^p A_p + \lambda^{p-1} A_{p-1} + \dots + \lambda A_1 + A_0)x = 0$ , where the  $A_i$  are given square coefficient matrices. The generalized eigenproblem is obtained for  $p = 1$ , with  $A_0 = I$  then giving the standard eigenproblem. The quadratic eigenproblem  $(\lambda^2 A + \lambda B + C)x = 0$  corresponds to  $p = 2$ . If  $A_p$  is  $n$ -by- $n$  and nonsingular then there are  $pn$  eigenvalues. The syntax is `e = polyeig(A0,A1,...,Ap)` or `[X,e] = polyeig(A0,A1,...,Ap)`, with  $e$  a  $pn$ -vector of eigenvalues and  $X$  an  $n$ -by- $pn$  matrix whose columns are the corresponding eigenvectors. Example:

```
>> A = eye(2); B = [20 -10; -10 20]; C = [15 -5; -5 15];

>> [X,e] = polyeig(C,B,A)
X =
    0.7071    0.7071    0.7071    0.7071
   -0.7071    0.7071   -0.7071    0.7071
e =
  -29.3178
   -8.8730
   -0.6822
   -1.1270
```

An optional third output argument of `polyeig` returns condition numbers for the eigenvalues. As this example shows, in the polynomial eigenvalue problem it is possible for distinct eigenvalues to have the same eigenvector.

## 9.9. Iterative Linear Equation and Eigenproblem Solvers

In this section we describe functions that are based on iterative methods and primarily intended for large, possibly sparse problems, for which solution by one of the methods described earlier in the chapter could be prohibitively expensive. Sparse matrices are discussed further in Chapter 15.

Several functions implement iterative methods for solving square linear systems  $Ax = b$ ; see Table 9.3. All apply to general  $A$  except `minres` and `symmlq`, which require  $A$  to be Hermitian, and `pcg`, which requires  $A$  to be Hermitian positive definite. All the methods employ matrix-vector products  $Ax$  and possibly  $A^*x$  and do not

require explicit access to the elements of  $A$ . The functions have identical calling sequences, apart from `gmres` (see below). The simplest usage is `x = solver(A,b)` (where `solver` is one of the functions in Table 9.3). Alternatively, `x = solver(A,b,tol)` specifies a convergence tolerance `tol`, which defaults to `1e-6`. Convergence is declared when an iterate  $x$  satisfies `norm(b-A*x) <= tol*norm(b)`. The argument  $A$  can be a full or sparse matrix, or a handle to a function `afun` such that `afun(x)` returns  $A*x$  and, in the case of `bicg` and `qmr`, such that `afun(x,'transp')` returns  $A'*x$ .

These iterative methods usually need preconditioning if they are to be efficient. All accept further arguments  $M_1$  and  $M_2$  or  $M = M_1M_2$  and effectively solve the preconditioned system

$$M_1^{-1}AM_2^{-1} \cdot M_2x = M_1^{-1}b \quad \text{or} \quad M^{-1}Ax = M^{-1}b.$$

The aim is to choose  $M_1$  and  $M_2$  so that  $M_1^{-1}AM_2^{-1}$  or  $M^{-1}A$  is in some sense close to the identity matrix. Choosing a good preconditioner is a difficult task that usually requires knowledge of the application from which the linear system came. The functions `ilu` and `ichol` compute incomplete factorizations that provide one way of constructing preconditioners (see `doc ilu`, `doc ichol`, and `doc bicg`). For background on iterative linear equation solvers and preconditioning see [9], [54], [94], [145], [177].

To illustrate the usage of the iterative solvers we give an example involving `pcg`, which implements the preconditioned conjugate gradient method. For  $A$  we take a symmetric positive definite finite-element matrix called the Wathen matrix, which has a fixed sparsity pattern and random entries:

```
>> rng(2) % Wathen matrix is random: make experiment reproducible.
>> A = gallery('wathen',12,12); n = length(A)
n =
    481
>> b = ones(n,1);

>> x = pcg(A,b);
pcg stopped at iteration 20 without converging to the desired
tolerance 1e-006 because the maximum number of iterations was
reached.
The iterate returned (number 20) has relative residual 0.064.

>> x = pcg(A,b,1e-6,100);
pcg converged at iteration 84 to a solution with relative residual
8.9e-07.
```

We supplied `pcg` with the matrix and the right-hand side. The conjugate gradient method did not converge to the default tolerance ( $10^{-6}$ ) within the default of 20 iterations, so we tried again with the same tolerance and a new limit of 100 iterations; convergence was then achieved. For this matrix it can be shown that  $M = \text{diag}(\text{diag}(A))$  is a good preconditioner. Supplying this preconditioner as a fifth argument leads to a useful reduction in the number of iterations:

```
>> [x,flag,relres,iter] = pcg(A,b,1e-6,100,diag(diag(A)));
>> flag, relres, iter
flag =
    0
```

Table 9.3. *Iterative linear equation solvers.*

Function	Matrix type	Method
<code>bicg</code>	General	BiConjugate gradient method
<code>bicgstab</code>	General	BiConjugate gradient stabilized method
<code>bicgstabl</code>	General	BiConjugate gradient stabilized method: Bi-CGSTAB( $\ell$ ) with $\ell = 2$
<code>cgs</code>	General	Conjugate gradient squared method
<code>gmres</code>	General	Generalized minimum residual method
<code>lsqr</code>	General	Conjugate gradients on normal equations
<code>minres</code>	Hermitian	Minimum residual method
<code>pcg</code>	Hermitian pos. def.	Preconditioned conjugate gradient method
<code>qmr</code>	General	Quasi-minimal residual method
<code>tfqmr</code>	General	Transpose-free quasi-minimal residual method
<code>symmlq</code>	Hermitian	Symmetric LQ method

```

relres =
    8.2923e-07
iter =
    28

```

Notice that when more than one output argument is requested the messages are suppressed. A zero value of `flag` denotes convergence with relative residual `relres = norm(b-A*x)/norm(b)` after `iter` iterations.

All the other functions in Table 9.3 have the same calling sequence as `pcg` with the exception of `gmres`, which has an extra argument, `restart`, in the third position that specifies at which iteration to restart the method.

Function `eigs` computes a few selected eigenvalues and eigenvectors for the standard eigenvalue problem or for the symmetric definite generalized eigenvalue problem  $Ax = \lambda Bx$  with  $B$  real and symmetric positive definite. This is in contrast to `eig`, which always computes the full eigensystem. Like the iterative linear equation solvers, `eigs` needs just the ability to form matrix–vector products, so  $A$  can be given either as an explicit matrix or as a handle to a function that performs matrix–vector products. In its simplest form, `eigs` can be called in the same way as `eig`, with `[V,D] = eigs(A)`, when it computes the six eigenvalues of largest magnitude and the corresponding eigenvectors. See `doc eigs` for more details and examples of usage. This function is an interface to the ARPACK package [113]. As an example, we form a sparse symmetric matrix and compute its five algebraically largest eigenvalues using `eigs`. For comparison, we also apply `eig`, which requires that the matrix first be converted to a full matrix and always computes all the eigenvalues.

```

>> A = delsq(numgrid('N',70)); % 5-pt finite difference Laplacian.
>> n = length(A)
n =
    4624

>> nnz(A)/n^2 % Percentage of nonzeros
ans =
    0.0011

```

```

>> tic, e_all = eig(full(A)); toc
Elapsed time is 4.940357 seconds.
>> e_all(n:-1:n-4)
ans =
    7.9959    7.9896    7.9896    7.9834    7.9793

>> options.disp = 0; % Turn off intermediate output.
tic, e_big = eigs(A,5,'la',options); toc % la = largest algebraic
Elapsed time is 0.446840 seconds.
e_big
e_big =
    7.9959    7.9896    7.9896    7.9834    7.9793

```

The `tic` and `toc` functions provide an easy way of timing (in seconds) the code that they surround (see Section 23.1 for more details). Clearly, `eigs` is much faster than `eig` in this example, and it also uses much less storage.

A corresponding function `svds` computes a few singular values and singular vectors of an  $m$ -by- $n$  matrix  $A$ .

## 9.10. Functions of a Matrix

As mentioned in Section 5.3, some of the elementary functions defined elementwise for arrays have counterparts defined in the matrix sense, implemented in functions whose names end in `m`. The three main examples are `expm`, `logm`, and `sqrtn`. The exponential of a square matrix  $A$  is defined by

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots.$$

It is computed by `expm`. The logarithm of a matrix is an inverse to the exponential. A nonsingular matrix has infinitely many logarithms. Function `logm` computes the principal logarithm, which, for a nonsingular matrix with no negative real eigenvalues, is the logarithm whose eigenvalues have imaginary parts lying strictly between  $-\pi$  and  $\pi$ .

A square root of a square matrix  $A$  is a matrix  $X$  for which  $X^2 = A$ . Every nonsingular matrix has at least two square roots. Function `sqrtn` computes the principal square root, which, for a nonsingular matrix with no negative real eigenvalues, is the square root with eigenvalues having positive real part.

We give some examples. The matrix

```

A =
    17     8     1     0
     8    18     8     1
     1     8    18     8
     0     1     8    17

```

has a tridiagonal square root:

```

>> format short g, sqrtn(A)
ans =
         4             1  8.8818e-16  -2.7756e-16

```

```

      1          4          1 -2.2204e-16
      8.8818e-16      1          4          1
      -2.7756e-16 -2.2204e-16      1          4

```

The Jordan block

```

>> A = gallery('jordbloc',4,1)
A =
      1      1      0      0
      0      1      1      0
      0      0      1      1
      0      0      0      1

```

has exponential

```

>> X = expm(A)
X =
      2.7183      2.7183      1.3591      0.45305
      0          2.7183      2.7183      1.3591
      0          0          2.7183      2.7183
      0          0          0          2.7183

```

and we can recover the original matrix using `logm`:

```

>> logm(X)
ans =
      1          1 -7.5894e-17  2.7159e-17
      0          1          1 -7.5894e-17
      0          0          1          1
      0          0          0          1

```

The function `funm` computes general matrix functions, using a method based on the Schur decomposition. For the functions `exp`, `log`, `cos`, `sin`, `cosh`, and `sinh`, `funm` can be used as in the following example, which computes  $\cos A$  and  $\sin A$  and checks that  $\cos(A)^2 + \sin(A)^2 = I$ :

```

>> format short, A = gallery('frank',3)
A =
      3      2      1
      2      2      1
      0      1      1

>> X = funm(A,@cos), Y = funm(A,@sin)
X =
      0.3362  -0.2983  -0.1021
     -0.3926   0.6267  -0.1963
      0.1885  -0.3847   0.6345
Y =
     -0.2455  -0.8062  -0.5435
     -0.5255  -0.2635  -0.2628
     -0.5615   0.2987   0.5607

>> residual = norm(X^2 + Y^2 - eye(3),1)

```



```
residual =
    1.3045e-15
```

To compute other functions, say  $f(A)$  in general, it is necessary to write a separate function of the form

```
function fd = fun(x,k)
```

that accepts a vector  $x$  and integer  $k$  and returns the  $k$ th derivative of  $f$  evaluated at  $x$ . (`funm` has such functions built in for the cases listed above.) The derivatives are needed when the matrix has repeated (and, in finite precision, close) eigenvalues.

In some applications it is not  $f(A)$  itself that is required but the action of  $f(A)$  on a vector:  $f(A)b$ . MATLAB does not have any functions for this problem (except, of course, in the special case  $f(x) = 1/x$ ). Codes for computing  $f(A)$  for other functions  $f$  (such as  $A^t$  for  $t \in \mathbb{R}$ ), and for the  $f(A)b$  problem, are available from various sources [82].

For background on matrix functions see [71], [74], [81].

*Nichols: "Transparent aluminum?"*

*Scott: "That's the ticket, laddie."*

*Nichols: "It'd take years just to figure out the dynamics of this matrix."*

*McCoy: "Yes, but you would be rich beyond the dreams of avarice!"*

— *Star Trek IV: The Voyage Home* (Stardate 8390)

*We share a philosophy about linear algebra:*

*we think basis-free,*

*we write basis-free,*

*but when the chips are down we close the office door and*

*compute with matrices like fury.*

— IRVING KAPLANSKY, *Reminiscences [of Paul Halmos]* (1991)

*The matrix of that equation system is negative definite—which is a positive definite system that has been multiplied through by  $-1$ .*

*For all practical geometries the common finite difference Laplacian operator gives rise to these, the best of all possible matrices.*

*Just about any standard solution method will succeed, and many theorems are available for your pleasure.*

— FORMAN S. ACTON, *Numerical Methods That Work* (1970)

*Clearly, inv-abuse is a serious and common problem for MATLAB users.*

— TIMOTHY A. DAVIS, *Algorithm 930: FACTORIZE: An Object-Oriented Linear System Solver for MATLAB* (2013)

# Chapter 10

## More on Functions

### 10.1. Function Handles

Many problems tackled with MATLAB require one function to be passed as an argument to another. The usual mechanism for doing this is through a function handle. A function handle is a MATLAB data type that contains all the information necessary to evaluate a function. It can be created by putting the `@` character before a function name.

To illustrate, if `fun` is a function of the form required by `fplot` then we can write

```
fzplot(@fun)
```

to plot `fun` over the default range `[-5,5]`. Here, `fun` can be a `.m` file or a built-in function:

```
fplot(@sin)
```

A function that accepts another function as an argument will need to evaluate the passed function. Evaluation is achieved simply by treating the function handle as if it were a function name and appending a list of arguments to it. If the function being called takes no input arguments then empty parentheses are required after the function handle name. Consider the function `fd_deriv` in Listing 10.1. This function evaluates the finite difference approximation

$$\frac{f(x+h) - f(x)}{h} \approx f'(x)$$

to the function passed as its first argument. When we type

```
>> fd_deriv(@sqrt,0.1)
ans =
    1.5811
```

the first `f` call in `fd_deriv` is equivalent to `sqrt(x+h)`. We can use our Newton square root function `sqrtn` (Listing 7.5) instead of the built-in square root:

```
>> fd_deriv(@sqrtn,0.1)
k           x_k           rel. change
1:  5.5000000745058064e-001  8.18e-001
% Remaining output from sqrtn omitted.
ans =
    1.5811
```

Listing 10.1. *Function fd\_deriv.*

```

function y = fd_deriv(f,x,h)
%FD_DERIV    Finite difference approximation to derivative.
%  FD_DERIV(f,x,h) is a finite difference approximation
%  to the derivative of function f at x with difference
%  parameter h.  h defaults to sqrt(eps).

if nargin < 3, h = sqrt(eps); end
y = (f(x+h) - f(x))/h;

```

You may come across the older syntax in which a function name is passed in a string, as in `fplot('exp')`. This still works, but is not recommended. Within a function, another function `fun` passed as a string must be evaluated as `feval(fun,1)` (for example), not `fun(1)`. The `feval` syntax also works with function handles. A string representing the name of a function can be converted to a function handle using the function `str2func`.

## 10.2. Anonymous Functions

Anonymous functions provide a way of creating a “one-line” function without writing a program file. For example:

```

>> f = @(x) exp(x)-1
f =
    @(x) exp(x)-1

>> f(2)
ans =
    6.3891

```

Here, `f` is a function handle to the anonymous function on the right of the assignment, which is so-called because it has no name. The `@` character, which constructs the function handle, is followed by a list of input arguments to the function in parentheses, and then by a single MATLAB expression. Like any function handle, `f` can be passed to other functions.

The expression that defines an anonymous function can contain variables not in the argument list. The values of such variables are captured when the function is created and they are held constant throughout the life of the function, as the next example of a three-argument function illustrates:

```

>> alpha = 1;
>> g = @(x,y,z) x^2+y^2-alpha*z^2;
>> g(1,2,3)
ans =
    -4

>> alpha = 0;

```

```
>> g(1,2,3)
ans =
    -4
```

In order for the changed value of `alpha` to be reflected in `g`, the anonymous function must be reconstructed. For another example of this type see Section 11.2.

One of the advantages of anonymous functions can be seen in connection with the example from the previous section in which we invoked `fd_deriv` on the function `sqrtn`. Recall from Listing 7.5 that `sqrtn` has a second input argument that specifies a convergence tolerance. Suppose we wish to use a different tolerance (say `1e-14`) when `sqrtn` is called by `fd_deriv`. This does not seem possible, because `fd_deriv` invokes its input function with only one argument. A way round this difficulty is to set up an anonymous function that calls `sqrtn` with the required convergence tolerance and pass this one-argument function to `fd_deriv`:

```
>> fd_deriv(@(x) sqrtn(x,1e-14), 0.1)
k           x_k           rel. change
1: 5.5000000745058064e-001 8.18e-001
% Remaining output from sqrtn omitted.
ans =
    1.5811
```

Note that here we set up the anonymous function within the call to `fd_deriv`. This technique is very useful when calling the MATLAB “function-function” routines that are described in the next two chapters.

#### LAMBDA

The notion of anonymous functions goes back to the language Lisp (1958), with its lambda expressions, which were derived from the  $\lambda$ -calculus developed by mathematician and logician Alonzo Church in the 1930s. Many modern programming languages support anonymous functions. For comparison, here are three ways in which the function  $2x$  can be represented.

- In the  $\lambda$ -calculus:  $\lambda x.2x$ .
- As an anonymous function in Lisp: `(lambda (x) (* 2 x))`.
- As an anonymous function in MATLAB: `@(x) 2*x`.

## 10.3. Local Functions

A function may contain other functions, called local functions (also known as sub-functions), which appear in any order after the main (or primary) function. Local functions are visible only to the main function and to any other local functions. They typically carry out a task that needs to be separated from the main function but that is unlikely to be needed in other functions, or they may override existing functions of the same names (since local functions take precedence). The use of local functions helps to avoid proliferation of `.m` files.

For an example of a local function see `poly1err`<sup>5</sup> in Listing 10.2, which approximates the maximum error in the linear interpolating polynomial to local function `f` on  $[0, 1]$  based on `n` sample points on the interval:

<sup>5</sup>This function is readily vectorized: see Section 23.2.

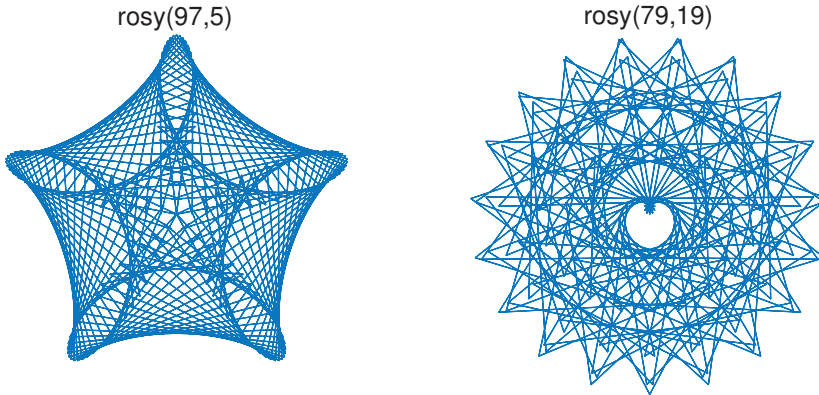


Figure 10.1. *Sample output from rosy.*

```
>> poly1err(5)
ans =
    0.0587

>> poly1err(50)
ans =
    0.0600
```

Alternative ways to code `poly1err` are to define `f` as an anonymous function rather than a local function (as long as  $f$  is given by a single expression) or to make `f` an input argument.

Another example is shown in Listing 10.3, some graphical output from which is displayed in Figure 10.1. It could be argued that the local function `spiro` in this example is unnecessary and that the local function should be “inlined”. Our reason for retaining the local function is to remind us that the formulas underlying `rosy` are a special case of more complicated formulas—a fact that would be less clear after inlining.

Help for a local function is displayed by specifying the main function name followed by “>” and the name of the local function. Thus help for local function `f` of `poly1err` is listed by

```
>> help poly1err>f

f    Function to be interpolated, f(x).
```

A local function can be passed to another function as a function handle. Thus, for example, in the main body of `poly1err` we can write `fplot(@f)` in order to plot the local function `f`.

A script can also contain local functions. Local functions within scripts work in much the same way as they do within functions. They must be located at the end of the script and they have their own workspaces. Listing 10.4 is an example of a script with a local function. It produces the output

```
>> test_solver
```

Listing 10.2. *Function poly1err containing local function f.*

```

function max_err = poly1err(n)
%POLY1ERR   Error in linear interpolating polynomial.
%   POLY1ERR(n) is an approximation based on n sample points
%   to the maximum difference between local function f and its
%   linear interpolating polynomial at 0 and 1.

max_err = 0;
f0 = f(0); f1 = f(1);
for x = linspace(0,1,n)
    p = x*f1 + (x-1)*f0;
    err = abs(f(x)-p);
    max_err = max(max_err,err);
end
end

% Local function.
function y = f(x)
%F   Function to be interpolated, f(x).
y = sin(x);
end

```

Matrix	Rel resid	norm(A,1)	norm(x,1)
spiral	2.3e-17	1.3e+04	2.3e+00
hadamard	4.4e-17	2.4e+01	4.3e+00
invhess	2.2e-17	3.0e+02	2.0e+00
parter	6.4e-17	8.9e+00	8.7e+00

For less trivial examples of local functions see Chapter 12 (including Listing 12.10 for a script with local functions).

## 10.4. Default Input Arguments

As we saw in Section 7.1, by using `nargin` a function can be written so that only the first few of its input arguments need be supplied on a particular call, the rest being set to default values within the function. When a function is designed, its input arguments should therefore be ordered starting with those that must be supplied and continuing with those that can or cannot be specified, in decreasing order of importance. To allow full flexibility a function can be written so that the user can request default values to be defined for arguments that occur before the last argument specified. The idea is for the empty matrix `[]` to be supplied as input argument and for `isempty` to be used to detect an empty input argument.

Consider the following snippet from a function:

```

function f = funarg_ex(A, npts, nangles, step)
if nargin < 2 || isempty(npts), npts = length(A); end
if nargin < 3 || isempty(nangles), nangles = 10; end

```

Listing 10.3. *Function rosy containing local function spiro.*

```

function rosy(a, b)
%ROSY    "Rose" figures.
%   ROSY(a, b) plots the curve
%       x = r*cos(a*theta), y = r*sin(a*theta), where
%       r = sin(a*b*theta) and 0 <= theta <= 2*pi (360 values).
%   Suggestions: ROSY(97, 5); ROSY(43, 4); ROSY(79, n9), n a digit.

%   P. M. Maurer, A rose is a rose..., Amer. Math. Monthly, 94 (1987),
%   pp. 631-645.

if nargin < 2, b = 1; end
if nargin < 1, a = 1; end

c = 0; d = 1; p = a*b;
[x, y] = spiro(a, a, c, d, p, .5);

plot(x,y)
axis square, axis off

% Local function.
function [x, y] = spiro(a, b, c, d, p, k)
h = k*2*pi/180;
t = (0:h:2*pi)';
r = c + d*sin(t*p);
x = r.*cos(a*t);
y = r.*sin(b*t);

```

Listing 10.4. *Script test\_solver containing local function test.*

```

%TEST_SOLVER Test linear equation solver.

n = 24; rng(1)
b = randn(n,1);

fprintf('Matrix   | Rel resid  norm(A,1)  norm(x,1)\n')
fprintf('-----\n')
test(spiral(n),b,'spiral')
test(hadamard(n),b,'hadamard') % n is restricted.
test(gallery('invhess',n),b,'invhess')
test(gallery('parter',n),b,'parter')

function test(A,b,name)
fprintf('%-8s %1s',name, '|')
% Or replace backslash with another solver and relevant statistics.
x = A\b; rel_res = norm(A*x-b,1)/(norm(A,1)*norm(x,1));
fprintf('%8.1e %8.1e %8.1e\n', rel_res, norm(A,1), norm(x,1))
end

```

```
if nargin < 4, step = 0.1; end
```

Here, the array `A` must be supplied, but the other arguments are all optional. Example invocations of the function are:

```
funarg_ex(rand(2),10,20,1e-3)
funarg_ex(rand(2),10)
funarg_ex(rand(2),[],50)
funarg_ex(rand(2),[],[],0.5)
```

When the empty matrix `[]` is passed as the second or third input argument the `isempty` part of the corresponding `if` test ensures that the default value is assigned to the relevant variable.

This technique is used by numerous MATLAB functions (for example, `lsqr` and `normest1`). An alternative to a long list of input arguments, all of which are equally likely to be required to take default values, is to use a structure to specify some of the arguments. This is done by several of the MATLAB numerical methods functions (see Chapters 11 and 12).

## 10.5. Variable Numbers of Arguments

In certain situations a function must accept or return a variable, possibly unlimited, number of input or output arguments. This can be achieved using the `varargin` and `varargout` functions. Suppose we wish to write a function `companb` to form the  $mn$ -by- $mn$  block companion matrix corresponding to the  $n$ -by- $n$  matrices  $A_1, A_2, \dots, A_m$ :

$$C = \begin{bmatrix} -A_1 & -A_2 & \dots & \dots & -A_m \\ I & 0 & & & 0 \\ & I & \ddots & & \vdots \\ & & \ddots & 0 & \vdots \\ & & & I & 0 \end{bmatrix}.$$

We could use a standard function definition such as

```
function C = companb(A_1,A_2,A_3,A_4,A_5)
```

but  $m$  is then limited to 5 and handling the different values of  $m$  between 1 and 5 is tedious. The solution is to use `varargin`, as shown in Listing 10.5. When `varargin` is specified as the input argument list, the input arguments supplied are copied into a cell array called `varargin`. Cell arrays, described in Section 18.7, are a special kind of array in which each element can hold a different type and size of data. The elements of a cell array are accessed using curly braces. Consider the call

```
>> X = ones(2); C = companb(X, 2*X, 3*X)
C =
    -1    -1    -2    -2    -3    -3
    -1    -1    -2    -2    -3    -3
     1     0     0     0     0     0
     0     1     0     0     0     0
     0     0     1     0     0     0
     0     0     0     1     0     0
```



Listing 10.5. *Function companb.*

```

function C = companb(varargin)
%COMPANB    Block companion matrix.
%  C = COMPANB(A_1,A_2,...,A_m) is the block companion matrix
%  corresponding to the n-by-n matrices A_1,A_2,...,A_m.

m = nargin;
n = length(varargin{1});

C = diag(ones(n*(m-1),1),-n);
for j = 1:m
    Aj = varargin{j};
    C(1:n,(j-1)*n+1:j*n) = -Aj;
end

```

If we insert the line

```
varargin
```

at the beginning of `companb` then the above call produces

```

varargin =
  1×3 cell array
    [2×2 double]    [2×2 double]    [2×2 double]

```

Thus `varargin` is a 1-by-3 cell array whose elements are the 2-by-2 matrices passed as arguments to `companb`, and `varargin{j}` is the  $j$ th input matrix,  $A_j$ .

It is not necessary for `varargin` to be the only input argument but it must be the last one, appearing after any named input arguments.

An example using the analogous statement `varargout` for output arguments is shown in Listing 10.6. Here, we use `nargout` to determine how many output arguments have been requested and then create a `varargout` cell array containing the required output. (It is the curly braces on the right-hand side of the assignment statement that make `varargout` a cell array.) To illustrate:

```

>> m1 = moments(1:4)
m1 =
    2.5000

>> [m1,m2,m3] = moments(1:4)
m1 =
    2.5000
m2 =
    7.5000
m3 =
    25

```

Listing 10.6. *Function moments.*

```
function varargout = moments(x)
%MOMENTS    Moments of a vector.
% [m1,m2,...,mk] = MOMENTS(x) returns the first, second, ...,
% k'th moments of the vector x, where the j'th moment
% is sum(x.^j)/length(x).

for j = 1:nargout, varargout(j) = {sum(x.^j)/length(x)}; end
```

## 10.6. Argument Checking and Parsing

MATLAB provides several functions to help in checking the arguments passed to a function. The call `narginchk(minargs,maxargs)` issues an error if the number of arguments passed to the function in which this command appears is less than `minargs` or greater than `maxargs`. Likewise, `nargoutchk(minargs,maxargs)` checks the number of output arguments requested.

Checks on the data types and attributes of the input arguments can be carried out with the function `validateattributes`. The function `arg_checks` in Listing 10.7 illustrates the usage. Here are some examples of calls to that function:

```
>> arg_checks(1,ones(3),2:4)
Error using arg_checks (line 5)
Not enough output arguments.

>> [y,Z] = arg_checks(1,ones(3,'int64'),2:4)
Error using arg_checks
Expected input number 2, A, to be one of these types:

double

Instead its type was int64.
Error in arg_checks (line 10)
validateattributes(A, {'double'}, {'finite', 'square', 'nonnan'},...
```

```
>> y = arg_checks(1,ones(3),4:-1:1)
Error using arg_checks (line 12)
Expected input to be non-decreasing valued.

>> y = arg_checks(1,ones(3),2:4,-1)
Error using arg_checks
Expected input number 4, tol, to be nonnegative.
Error in arg_checks (line 13)
validateattributes(tol, {'double','single'}, {'nonnegative'},...
```

Note that when the final three arguments to `validateattributes` are `mfilename` (a function that returns the name of the currently executing program file), a string representing the name of the argument being tested, and an integer representing the

number of the argument, then the error message includes the name of the function and the name and position of the argument.

For details of all the attributes that can be checked with this function type `doc validateattributes`.

For checking strings the function `validatestring` is useful. It allows a string to be tested against a cell array of strings for a unique case-insensitive partial match. Examples:

```
>> validatestring('Comm',{'Computation','commute','COMB'})
ans =
    1×7 char array
    commute

>> validatestring('com',{'Computation','commute','COMB'})
Expected input to match one of these values:

'Computation', 'commute', 'COMB'
```

The input, `com`, matched more than one valid value

The coding conventions in force, or personal preference, will dictate the balance between brevity of code and checking every input for validity. Even within built-in MATLAB functions the amount of argument checking varies greatly.

For some functions it is convenient to allow input arguments to be specified via name–value pairs, listed in any order, as illustrated by the call

```
myfun(x, 'Tolerance', 1e-4, 'Scaling', 'on', 'Output', 'verbose')
```

MATLAB provides an object called `inputParser` that carries out the nontrivial task of parsing such input arguments, returning the arguments in a structure. The `inputParser` object can optionally do case-sensitive and partial matching of parameter names and provides a way for arguments to be checked for validity. For an example of the use of `inputParser` see the code `quadgk`.

## 10.7. Nested Functions

MATLAB allows one or more functions to be nested wholly inside another. To define the nesting, `end` statements must be placed at the end of the nested functions and the main function (otherwise the functions are local functions), and it is good style to indent the nested functions.

Nested functions have two key properties:

- A nested function has access to the workspaces of all functions within which it is nested.
- A function handle for a nested function stores the information needed to access the nested function *and* the values of any variables in functions containing the nested function (“externally scoped” variables) that are needed to evaluate it.

An example of a nested function is given in `rational_ex` in Listing 10.8, which makes use of `fd_deriv` in Listing 10.1. The example illustrates how a function depending on parameters can be passed to another function. In the body of `rational_ex`

Listing 10.7. *Function arg\_checks.*

```
function [y,Z] = arg_checks(n,A,x,tol)
%ARG_CHECKS    Illustrate argument checking.

narginchk(3,4) % Require 3 to 4 input arguments.
nargoutchk(1,2) % Require 1 to 2 output arguments.

if nargin < 4, tol = sqrt(eps); end

validateattributes(n, {'double', 'single'}, {'scalar', 'positive'})
validateattributes(A, {'double'}, {'finite', 'square', 'nonnan'},...
    mfilename,'A',2)
validateattributes(x, {'numeric'}, {'nondecreasing'});
validateattributes(tol, {'double','single'}, {'nonnegative'},...
    mfilename,'tol',4)

y = n*x.^2; Z = A.^2;
```

a function handle to the nested function `rational` is passed to `fd_deriv`. The variables `a`, `b`, `c`, and `d` from the main function's workspace are available inside `rational`, and their values are encapsulated in the function handle that is passed to `fd_deriv`. Even though these four variables are not in the scope of `fd_deriv`, this function can correctly evaluate `rational` in terms of the values of the variables at the time the function handle was created:

```
>> rational_ex(2)
ans =
    3.0000
```

This example could be rewritten using an anonymous function, as in the example involving `sqrtn` in Section 10.2. However, the anonymous function approach is applicable only when the function is given by a single expression, which is quite limiting. Nested functions have the advantage that they enable a single function (containing nested functions) to be written to solve a complete problem involving a parametrized function; we will use them extensively for this purpose in Chapters 12 and 26.

The precise scoping rules of nested functions can be found in the MATLAB documentation.

## 10.8. Private Functions

A typical MATLAB installation contains hundreds of program files on the user's path, all accessible just by typing the name of the file. While this ease of accessing program files is an advantage, it can lead to clutter and clashes of names, not least due to the presence of "helper functions" that are used by other functions but are not intended to be called directly by the user. Private functions provide an elegant way to avoid these problems. Any functions residing in a directory called `private` are visible only to functions in the parent directory. They can therefore have the same names as functions in other directories. When MATLAB looks for a function it searches local

Listing 10.8. *Function rational\_ex containing a nested function r.*

```

function rational_ex(x)
%RATIONAL_EX    Illustration of nested function.

a = 1; b = 2; c = 1; d = -1;
fd_deriv(@rational,x)

    function r = rational(x)
    % Rational function.
    r = (a+b*x)/(c+d*x);
    end

end

```

functions first, then private functions (relative to the directory in which the function making the call is located), then the current directory, and then the path. Hence if a private function has the same name as a nonprivate function (even a built-in function), the private function will be found first.

Good use of private functions is made by the `gallery` function, which lives in the `matlab\elmat` directory and provides a collection of test matrices (see Section 5.1). The 50 or so matrix-generating functions invoked by `gallery` live in `matlab\elmat\private`, and their names were able to be chosen without fear of clashing with an existing function (perhaps one in a toolbox).

Private directories should not be put on the path.

Help for a private function `fun` can be accessed using `help private\fun`.

## 10.9. Recursive Functions

Functions can be recursive, that is, they can call themselves, as we have seen with function `gasket` in Listing 1.7 and function `land` in Listing 8.4. Recursion is a powerful tool, but not all computations that are described recursively are best programmed this way, as alternative formulations may be faster or use less memory.

The function `koch` in Listing 10.9 uses recursion to draw a Koch curve [139, Sec. 2.4]. The basic construction in `koch` is to replace a line by four shorter lines. The upper left-hand picture in Figure 10.2 shows the four lines that result from applying this construction to a horizontal line. The upper right-hand picture then shows what happens when each of these four lines is processed. The two lower pictures show the next two levels of recursion.

We see that `koch` has three input arguments. The first two, `p1` and `pr`, give the  $(x, y)$  coordinates of the current line and the third, `level`, indicates the level of recursion required. If `level = 0` then a line is drawn; otherwise `koch` calls itself four times with `level` one less and with endpoints that define the four shorter lines.

Figure 10.2 was produced by the following code:

```

p1 = [0;0]; % Left endpoint
pr = [1;0]; % Right endpoint

```

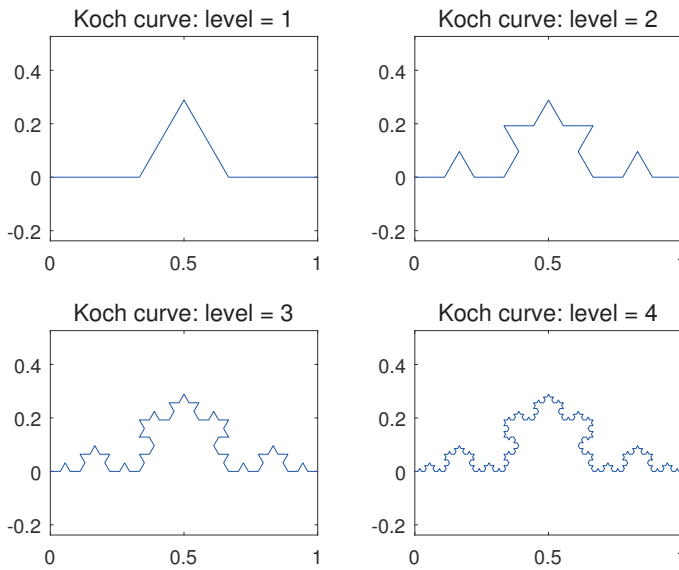


Figure 10.2. *Koch curves created with function koch.*

```

for k = 1:4
    subplot(2,2,k)
    koch(pl,pr,k)
    axis('equal')
    title(['Koch curve: level = ' num2str(k)], 'FontSize',12,...
          'FontWeight','normal')
end
hold off

```

To produce Figure 10.3 we called `koch` with pairs of endpoints equally spaced around the unit circle, so that each edge of the snowflake is a copy of the same Koch curve. The relevant code is

```

level = 4; edges = 7;

for k = 1:edges
    pl = [cos(2*k*pi/edges); sin(2*k*pi/edges)];
    pr = [cos(2*(k+1)*pi/edges); sin(2*(k+1)*pi/edges)];
    koch(pl,pr,level)
end
axis('equal')
title('Koch snowflake', 'FontSize',12, 'FontAngle','italic',...
      'FontWeight','normal')
hold off

```

For another example of recursion see the function `lsys` in Listing 26.6.

Listing 10.9. *Function koch.*

```

function koch(pl,pr,level)
%KOCH   Recursively generated Koch curve.
%   KOCH(pl, pr, level) recursively generates a Koch curve,
%   where pl and pr are the current left and right endpoints and
%   level is the level of recursion.

if level == 0
    plot([pl(1),pr(1)],[pl(2),pr(2)],'b'); % Join pl and pr.
    hold on
else
    A = (sqrt(3)/6)*[0 1; -1 0];           % Rotate/scale matrix.

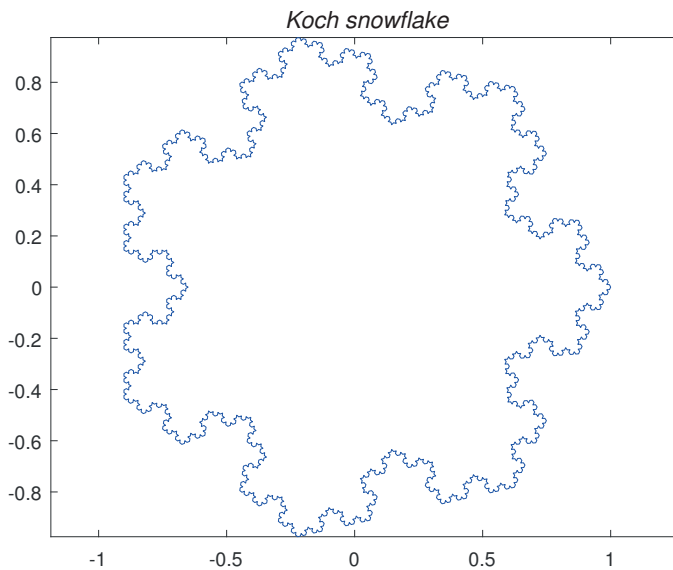
    pmidl = (2*pl + pr)/3;
    koch(pl,pmidl,level-1)                % Left branch.

    ptop = (pl + pr)/2 + A*(pl-pr);
    koch(pmidl,ptop,level-1)              % Left mid branch.

    pmidr = (pl + 2*pr)/3;
    koch(ptop,pmidr,level-1)              % Right mid branch.

    koch(pmidr,pr,level-1)                % Right branch.
end

```

Figure 10.3. *Koch snowflake created with function koch.*

## 10.10. Global and Persistent Variables

Variables within a function are local to that function's workspace. Occasionally it is convenient to create variables that exist in more than one workspace, including, possibly, the main workspace. This can be done using the `global` statement. As an example, suppose we wish to study plots of the function  $f(x) = 1/(a + (x - b)^2)$  on  $[0, 1]$  for various  $a$  and  $b$ . We can define a function

```
function f = myfun(x)
global A B
f = 1./(A + (x-B).^2);
```

At the command line we type

```
>> global A B
>> A = 0.01; B = 0.5;
>> fplot(@myfun,[0 1])
```

The values of `A` and `B` set at the command line are available within `myfun`. New values for `A` and `B` can be assigned and `fplot` called again without editing `myfun.m`. However, in this example it is best to avoid the use of `global` by defining

```
function f = myfun2(x,A,B)
f = 1./(A + (x-B).^2);
```

and then using an anonymous function to set up the required values of `A` and `B`:

```
fplot(@(x)myfun2(x,0.01,0.5),[0 1])
```

Within a function, the `global` declaration should appear before the first occurrence of the relevant variables, ideally at the top of the file. By convention the names of global variables are comprised of capital letters, and ideally the names are long in order to reduce the chance of clashes with other variables.

The use of global variables is not recommended. With the use of anonymous functions and nested functions they can usually be avoided.

Persistent variables are variables local to a function whose values are preserved between calls to the function. They are essentially a more restricted form of global variable. Persistent variables are declared with the `persistent` statement and are initialized to the empty matrix if the variables do not already exist.

One use of persistent variables is to ensure that initialization calculations in a function are carried out only the first time that function is called, as in the following outline:

```
function persistent_ex(varargin)
persistent coeffs
if isempty(coeffs)
    % Form the array coeffs on first call of function.
end
% Rest of function.
```

Another use is to preserve information about the state of a function between calls, assuming it is not desirable to do so through variables in the argument list of the function.



## 10.11. Exemplary Functions in MATLAB

Perhaps the best way to learn how to write functions is by studying well-written examples. An excellent source of examples is MATLAB itself, since all functions that are not built into the interpreter are `.m` files that can be examined. We list below some functions that illustrate particular aspects of MATLAB programming. The source can be viewed with `type function_name`, by loading the file into the Editor/Debugger with `edit function_name` (the editor searches the path for the function, so the pathname need not be given), or by loading the file into your favorite editor (in which case you will need to know the path, which we indicate).

- `datafun/cov`: use of `varargin`.
- `datafun/var`: argument checking.
- `matfun/expm`: argument checking, `switch` construct, local functions.
- `elmat/hadamard`: matrix building.
- `elmat/why`: `switch` construct, local functions.
- `funfun/fminbnd`: argument checking, loop constructs.
- `specfun/nchoosek`, `matfun/private/sqrtm_tri`: recursive function.
- `matfun/gsvd`: local functions.
- `funfun/ode45`: use of `varargin` and `varargout`.
- `sparfun/pcg`: sophisticated argument handling and error checking.

*Lambda is so useful that, like many of lisp's features,  
most modern languages are beginning to import the idea  
from lisp into their own systems.*

— DOUG HOYTE, *Let Over Lambda. 50 Years of Lisp* (2008)

*In this example, ALEVIL is a function name  
being passed to ROOT and MONEY is the ROOT of ALEVIL.*

— ROGER EMANUEL KAUFMAN, *A FORTRAN Coloring Book* (1978)

*Use recursive procedures for recursively-defined data structures.*

— BRIAN W. KERNIGHAN and P. J. PLAUGER,  
*The Elements of Programming Style* (1978)

*Great fleas have little fleas upon their backs to bite 'em,  
And little fleas have lesser fleas and so ad infinitum.  
And the great fleas themselves, in turn, have greater fleas to go on;  
While these again have greater still, and greater still, and so on.*

— AUGUSTUS DEMORGAN

# Chapter 11

## Numerical Methods: Part I

This chapter describes the MATLAB functions for solving problems involving polynomials, nonlinear equations, optimization, and the fast Fourier transform. In many cases a function `fun` must be passed as an argument. The MATLAB functions described in this chapter place various demands on the function that is to be passed, but most require it to return a vector of values when given a vector of inputs.

Several functions described in this chapter make use of structures, which are one of the MATLAB data types: see Section 18.7 for details of structures.

For mathematical background on the methods described in this chapter and the next suitable textbooks are [7], [21], [23], [47], [55], [56], [91], [115], [151], [174].

### 11.1. Polynomials and Data Fitting

MATLAB represents a polynomial

$$p(x) = p_1x^n + p_2x^{n-1} + \cdots + p_nx + p_{n+1}$$

by a row vector  $\mathbf{p} = [p(1) \ p(2) \ \dots \ p(n+1)]$  of the coefficients. (Note that compared with the representation  $\sum_{i=0}^n p_i x^i$  used in many textbooks, the MATLAB vector is reversed and its subscripts are increased by 1.)

Here are three problems related to polynomials:

**Evaluation:** Given the coefficients evaluate the polynomial at one or more points.

**Root finding:** Given the coefficients find the roots (the points at which the polynomial evaluates to zero).

**Data fitting:** Given a set of data  $\{x_i, y_i\}_{i=1}^m$ , find a polynomial that “fits” the data.

The standard technique for evaluating  $p(x)$  is Horner’s method, which corresponds to the nested representation

$$p(x) = \left( \dots \left( (p_1x + p_2)x + p_3 \right) x + \cdots + p_n \right) x + p_{n+1}.$$

Function `polyval` carries out Horner’s method:  $\mathbf{y} = \text{polyval}(\mathbf{p}, \mathbf{x})$ . In this command  $\mathbf{x}$  can be a matrix, in which case the polynomial is evaluated at each element of the matrix (that is, in the array sense). Evaluation of the polynomial  $p$  in the matrix (as opposed to array) sense is defined for a square matrix argument  $X$  by

$$p(X) = p_1X^n + p_2X^{n-1} + \cdots + p_nX + p_{n+1}I.$$

The command  $\mathbf{Y} = \text{polyvalm}(\mathbf{p}, \mathbf{X})$  carries out this evaluation.

The roots (or zeros) of  $p$  are obtained with  $\mathbf{z} = \text{roots}(\mathbf{p})$ . Of course, some of the roots may be complex even if  $p$  is a real polynomial. The function `poly` carries out the converse operation: given a set of roots it constructs a polynomial. Thus if  $\mathbf{z}$  is an  $n$ -vector then  $\mathbf{p} = \text{poly}(\mathbf{z})$  gives the coefficients of the polynomial

$$p_1x^n + p_2x^{n-1} + \cdots + p_nx + p_{n+1} = (x - z_1)(x - z_2)\cdots(x - z_n).$$

(The normalization  $p_1 = 1$  is always used.) The function `poly` also accepts a matrix argument: as explained in Section 9.8.1, for a square matrix  $\mathbf{A}$ ,  $\mathbf{p} = \text{poly}(\mathbf{A})$  returns the coefficients of the characteristic polynomial  $\det(x\mathbf{I} - \mathbf{A})$ .

Function `polyder` computes the coefficients of the derivative of a polynomial, but it does not evaluate the polynomial.

As an example, consider the quadratic  $p(x) = x^2 - x - 1$ . First, we find the roots:

```
>> format short g, p = [1 -1 -1]; z = roots(p)
z =
    -0.61803
     1.618
```

The next command verifies that these are roots, up to roundoff:

```
>> polyval(p,z)
ans =
   -1.1102e-16
    2.2204e-16
```

Next, we observe that a certain 2-by-2 matrix has  $p$  as its characteristic polynomial:

```
>> A = [0 1; 1 1]; cp = poly(A)
cp =
         1         -1        -1
```

The Cayley–Hamilton theorem says that every matrix satisfies its own characteristic polynomial. This is confirmed, modulo roundoff, for our matrix:

```
>> polyvalm(cp,A)
ans =
   1.1102e-16         0
         0   1.1102e-16
```

Polynomials can be multiplied and divided using `conv` and `deconv`, respectively. When a polynomial  $g$  is divided by a polynomial  $h$  there is a quotient  $q$  and a remainder  $r$ :  $g(x) = h(x)q(x) + r(x)$ , where the degree of  $r$  is less than that of  $h$ . The syntax for `deconv` is  $[\mathbf{q}, \mathbf{r}] = \text{deconv}(\mathbf{g}, \mathbf{h})$ . In the following example we divide  $x^3 - 6x^2 + 12x - 8$  by  $x - 2$ , obtaining quotient  $x^2 - 4x + 4$  and zero remainder. Then we reproduce the original polynomial using `conv`:

```
>> g = [1 -6 12 -8]; h = [1 -2];

>> [q,r] = deconv(g,h)
q =
     1     -4     4
r =
```

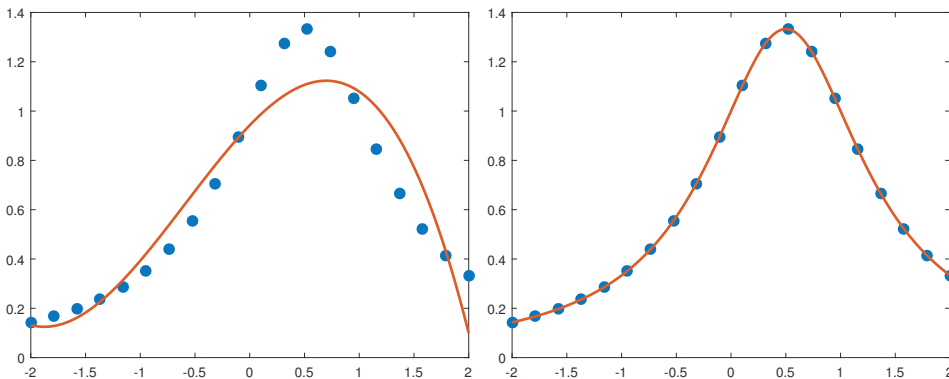


Figure 11.1. *Left: least-squares polynomial fit of degree 3. Right: cubic spline. Data is from  $1/(x + (1 - x)^2)$ .*

```

0      0      0      0

>> conv(h,q)
ans =
1     -6     12     -8

```

The data-fitting problem can be addressed with `polyfit`. Suppose the data  $\{x_i, y_i\}_{i=1}^m$  has distinct  $x_i$ -values, and we wish to find a polynomial  $p$  of degree at most  $n$  such that  $p(x_i) \approx y_i$ ,  $i = 1:m$ . The `polyfit` function computes the least-squares polynomial fit, that is, it determines  $p$  so that  $\sum_{i=1}^m (p(x_i) - y_i)^2$  is minimized. The syntax is `p = polyfit(x,y,n)`. Specifying the degree  $n$  so that  $n \geq m$  produces an interpolating polynomial, that is,  $p(x_i) = y_i$ ,  $i = 1:m$ , so the polynomial fits the data exactly. However, high-degree polynomials can be extremely oscillatory, so small values of  $n$  are generally preferred. The following example computes and plots a least-squares polynomial fit of degree 3. The data comprises the function  $1/(x + (1 - x)^2)$  evaluated at 20 equally spaced points on the interval  $[-2, 2]$ , generated by `linspace`. The resulting plot is that on the left-hand side of Figure 11.1.

```

x = linspace(-2,2,20);
y = 1./(x+(1-x).^2);
p = polyfit(x,y,3);
xx = linspace(-2,2,100);
plot(x,y,'.',xx,polyval(p,xx),'-','MarkerSize',30,'LineWidth',2)

```

The `spline` function can be used if exact data interpolation is required. It fits a cubic spline,  $\text{sp}(x)$ , to the data  $\{x_i, y_i\}_{i=1}^m$ .  $\text{sp}(x)$  has the following properties:

- it is a cubic polynomial between each pair of successive points  $x_i$  and  $x_{i+1}$  (i.e., it is a piecewise cubic polynomial),
- $\text{sp}(x_i) = y_i$ ,  $i = 1:m$  (i.e., `sp` interpolates the data),
- it has continuous first and second derivatives at the points  $x_i$  (i.e., the cubic pieces join up smoothly).

In addition, the extra freedom in the spline is used up by enforcing the not-a-knot end conditions, the meaning of which can be found in the textbooks cited at the start of this section.

Given data vectors  $x$  and  $y$ , the command `yy = spline(x,y,xx)` returns in the vector `yy` the value of the spline at the points given by `xx`. The following code fits a cubic spline to the data in the polynomial example above. The resulting curve is on the right-hand side of Figure 11.1.

```
yy = spline(x,y,xx);
plot(x,y,'.',xx,yy,'-', 'MarkerSize',30,'LineWidth',2)
```

It is also possible to work with the coefficients of the spline curve. The command `pp = spline(x,y)` stores the coefficients in a structure that is interpreted by the `ppval` function, so `plot(x,y,'.',xx,ppval(pp,xx),'--')` would then produce the same plot as in the example above. Low-level manipulation of splines is possible with the functions `mkpp` and `unmkpp`.

MATLAB has another function for piecewise polynomial interpolation: `pchip`. This function produces a piecewise cubic  $p(x)$  whose second derivative is generally not continuous. However, `pchip` has one very special property: it maintains both the shape and the monotonicity of the data. This means that on intervals where the data is monotonic, so is  $p$ , and at points where the data has a local extremum, so does  $p$ . To illustrate, consider this script:

```
x = [-12:4:12];
y = atan(x);
t = [-12:.1:12];
p = pchip(x,y,t);
s = spline(x,y,t);
plot(x,y,'ob',t,p,'r',t,s,'k','LineWidth',1.5)
xlim([-12 12])
legend('data','pchip','spline','Location','NW')
```

The script produces Figure 11.2. The spline is smooth, but oscillates between the first three and last three data points; `pchip` has no oscillations, at the expense of a slight loss of smoothness at the data points.

MATLAB has functions for interpolation in one, two, and more dimensions. For one-dimensional interpolation, the function `interp1` accepts  $x(i), y(i)$  data pairs and a further vector  $xi$ . It fits an interpolant to the data and then returns the values of the interpolant at the points in  $xi$ :

```
yi = interp1(x,y,xi)
```

The vector  $x$  must have monotonically increasing elements. Several types of interpolant are supported, as specified by a fourth input parameter, the main choices for which are

'nearest'	nearest-neighbor interpolation
'linear'	linear interpolation (default)
'spline'	cubic spline interpolation
'pchip'	piecewise cubic Hermite interpolation

Linear interpolation puts a line between adjacent data pairs, while nearest-neighbor interpolation reproduces the  $y$ -value of the nearest  $x$  point. The following example illustrates `interp1`:

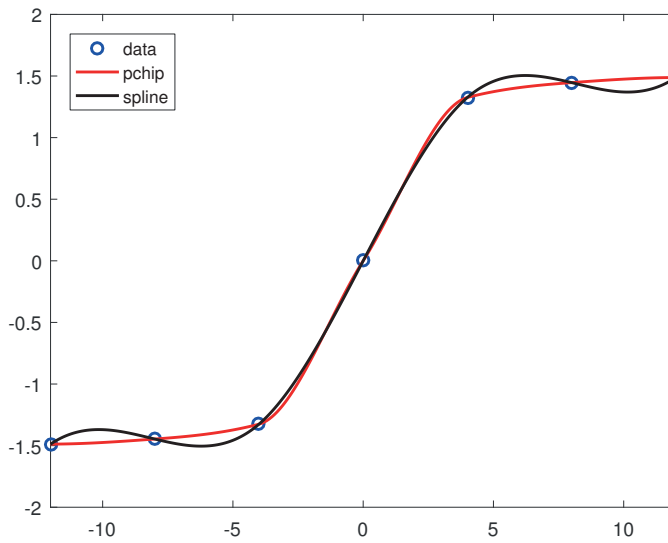


Figure 11.2. *Interpolation with pchip and spline.*

```
x = [0 pi/4 3*pi/8 3*pi/4 pi]; y = sin(x);
xi = linspace(0,pi,40)';
yn = interp1(x,y,xi,'nearest');
yl = interp1(x,y,xi,'linear');
ys = interp1(x,y,xi,'spline');
yp = interp1(x,y,xi,'pchip');
xx = linspace(0,pi,50);
plot(xi,yn,'*', xi,yl,'+', xi,ys,'v', xi,yp,'o')
legend('nearest','linear','spline','pchip')
hold on
plot(xx,sin(xx),'- ',x,y,'.k','MarkerSize',30)
xticks(x)
xticklabels({'0','\pi/4','3\pi/8','3\pi/4','\pi'})
set(gca,'XGrid','on')
axis([-0.25 3.5 -0.1 1.1])
hold off
```

This code samples 5 points from a sine curve on  $[0, \pi]$ , computes interpolants using the four methods above, and evaluates the interpolants at 40 points on the interval. In Figure 11.3 the solid circles plot the  $x(i), y(i)$  data pairs and the symbols plot the interpolants. The graphics commands are discussed in Chapters 8 and 17.

MATLAB has two functions for two-dimensional interpolation: `griddata` and `interp2`. The syntax for `griddata` is

```
ZI = griddata(x,y,z,XI,YI)
```

Here, the vectors  $x$ ,  $y$ , and  $z$  are the data and  $ZI$  is a matrix of interpolated values corresponding to the matrices  $XI$  and  $YI$ , which are usually produced with `meshgrid`. A sixth string argument specifies the method:

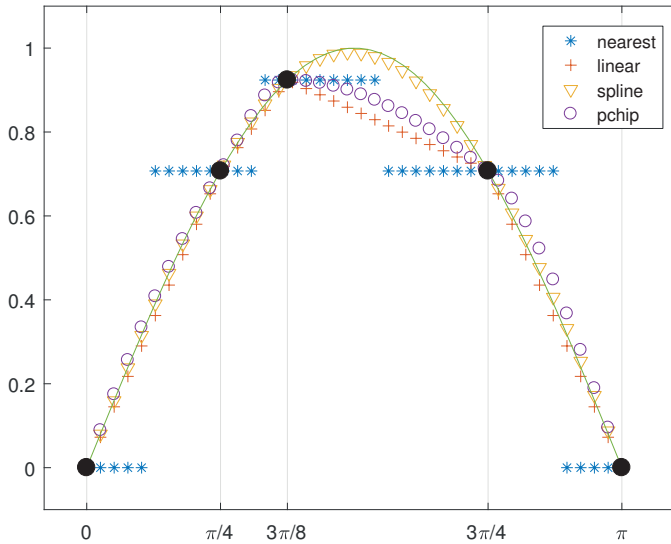


Figure 11.3. *Interpolating a sine curve at five points using interp1.*

'linear'      triangle-based linear interpolation (default)  
 'cubic'      triangle-based cubic interpolation  
 'nearest'    nearest-neighbor interpolation

Function `interp2` has a similar argument list, but it requires `x` and `y` to be monotonic matrices in the form produced by `meshgrid`. Here is an example in which we use `griddata` to interpolate values on a surface:

```
rng(25); x = rand(100,1)*4 - 2; y = rand(100,1)*4 - 2;
z = x.*exp(-x.^2-y.^2);
hi = -2:.1:2;
[XI,YI] = meshgrid(hi);
ZI = griddata(x,y,z,XI,YI);
mesh(XI,YI,ZI), hold
plot3(x,y,z,'o'), hold off
```

The result is shown in Figure 11.4, which plots the original data points as circles and the interpolated surface as a mesh.

Other interpolation functions include `interp3` and `griddata3` for three-dimensional interpolation, `interp $n$`  and `griddatan` for  $n$ -dimensional interpolation, and the functions `scatteredInterpolant` and `griddedInterpolant`.

Further data-fitting capabilities are available in the Statistics and Machine Learning Toolbox.

## 11.2. Nonlinear Equations

MATLAB has routines for finding a zero of a function of one variable (`fzero`) and for minimizing a function of one variable (`fminbnd`) or of  $n$  variables (`fminsearch`). In all

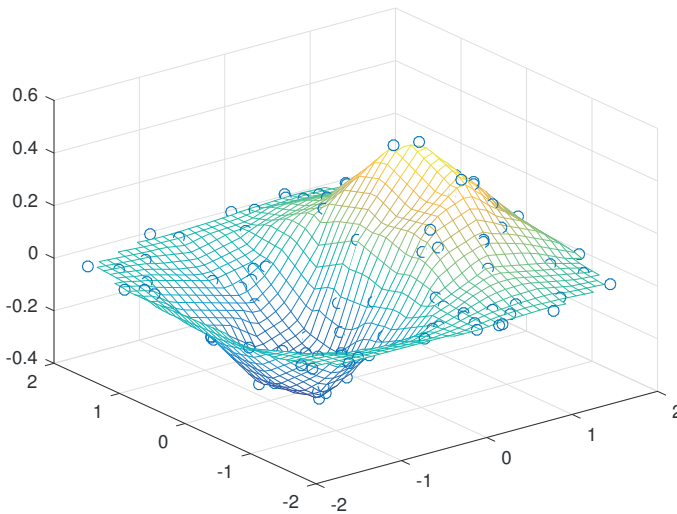


Figure 11.4. *Interpolation with `griddata`.*

cases the function must be real-valued and have real arguments. Unfortunately, there is no provision for directly solving a system of  $n$  nonlinear equations in  $n$  unknowns.<sup>6</sup>

The simplest invocation of `fzero` is `x = fzero(fun, x0)`, with `x0` a scalar, which attempts to find a zero of `fun` near `x0`. The function `fun` should be passed in one of two ways, namely

1. as the handle to an anonymous function: `fzero(@(x)cos(x)-x,x0)`,
2. as the handle to a function: `fzero(@myfun,x0)`, where

```
function f = myfun(x)
f = cos(x)-x;
```

Thus to find a zero near 0, using an anonymous function:

```
>> fzero(@(x)cos(x)-x,0)
ans =
    0.7391
```

More precisely, `fzero` looks for a point where `fun` changes sign, and will not find zeros of even multiplicity. An initial search is carried out starting from `x0` to find an interval on which `fun` changes sign. The function `fun` must return a real scalar when passed a real scalar argument. Failure of `fzero` is signaled by the return of a NaN.

If, instead of being a scalar, `x0` is a 2-vector such that `fun(x0(1))` and `fun(x0(2))` have opposite sign, then `fzero` works on the interval defined by `x0`. Providing a starting interval in this way can be important when the function has a singularity. Consider the example

---

<sup>6</sup>However, an attempt at solving such a system could be made by minimizing the sum of squares of the residual. The Optimization Toolbox contains a nonlinear equation solver. See also the MATLAB codes provided with [96].



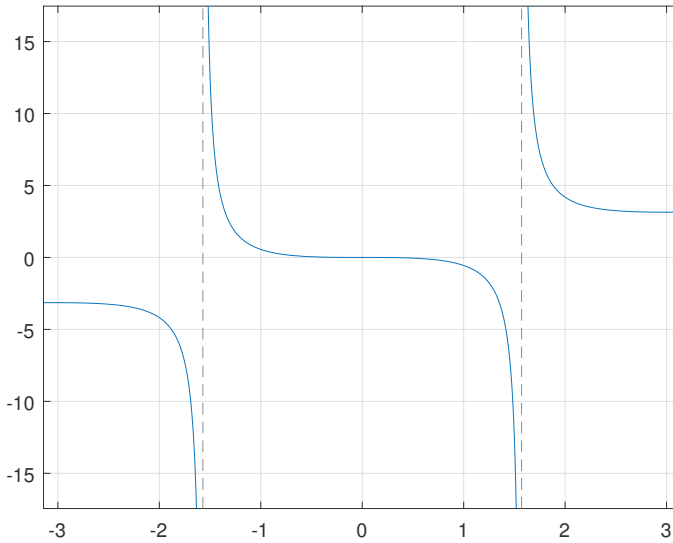


Figure 11.5. Plot produced by `fplot(@(x)x-tan(x),[-pi,pi]), grid`.

```
>> [x, fval] = fzero(@(x)x-tan(x),1)
x =
    1.5708
fval =
    1.2093e+015
```

The second output argument is the function value at  $x$ , the purported zero. Clearly, in this example  $x$  is not a zero but an approximation to the point  $\pi/2$  at which the function has a singularity; see Figure 11.5. To force `fzero` to keep away from singularities we can give it a starting interval that encloses a zero but not a singularity:

```
>> [x, fval] = fzero(@(x)x-tan(x),[-1 1])
x =
    0
fval =
    0
```

The convergence tolerance and the display of output in `fzero` are controlled by a third argument, the structure `options`, which is best defined using the `optimset` function. Four of the fields of the `options` structure are used: `Display` specifies the level of reporting, with values `off` for no output, `iter` for output at each iteration, `final` for just the final output, and `notify` for output only when the iteration fails to converge; `TolX` is a convergence tolerance; `FunValCheck` determines whether function values are checked for complex or NaN values; and `OutputFcn` specifies a user-defined function that is called at each iteration. Example uses are

```
fzero(fun,x0,optimset('Display','iter'))
fzero(fun,x0,optimset('TolX',1e-4))
```

The default corresponds to

```
optimset('Display','notify','TolX',eps,'FunValCheck','off',...
        'OutputFcn',[])
```

Note that the field names passed to `optimset` can be any combination of uppercase and lowercase, and it is sufficient to type just enough characters of the field name to uniquely identify it.

Suppose now that we wish to find a zero of the function  $f(x) = a \sin x + b e^{-x^2/2}$ , where  $a$  and  $b$  are parameters that we wish to vary. Our function is called from within `fzero` with just one argument,  $x$ , so how do we communicate the values of  $a$  and  $b$ ? As we discussed in Section 10.2, this can be achieved using an anonymous function:

```
>> a = 1; b = 2;
>> fzero(@(x)a*sin(x) + b*exp(-x^2/2),0)
ans =
    -1.2274

>> a = 3; b = -1;
>> fzero(@(x)a*sin(x) + b*exp(-x^2/2),0)
ans =
     0.3220
```

Note the importance of reconstructing the anonymous function each time  $a$  and  $b$  change. Compare this example with

```
>> a = 1; b = 2;
>> f = @(x)a*sin(x) + b*exp(-x^2/2)
>> fzero(f,0)
ans =
    -1.2274

>> a = 3; b = -1;
>> fzero(f,0)
ans =
    -1.2274
```

where the second invocation of `fzero` produces the same result as the first, since `f` always uses the values of  $a$  and  $b$  current at the time the anonymous function was constructed. Another approach is to construct the basic function once and for all with

```
>> fun = @(x,a,b)a*sin(x)+b*exp(-x^2/2);
```

and then construct a wrapping anonymous function from it each time:

```
>> a = 1; b = 2;
>> fzero(@(x)fun(x,a,b),0)
ans =
    -1.2274

>> a = 3; b = -1;
>> fzero(@(x)fun(x,a,b),0)
ans =
     0.3220
```

The algorithm used by `fzero`—a combination of the bisection method, the secant method, and inverse quadratic interpolation—is described in [47, Chap. 7].

### 11.3. Optimization

The command `x = fminbnd(fun,x1,x2)` attempts to find a local minimizer `x` of the function of one variable specified by `fun` over the interval `[x1,x2]`. A point  $x$  is a local minimizer of  $f$  if it minimizes  $f$  in an interval around  $x$ . In general, a function can have many local minimizers. MATLAB does not provide a function for the difficult problem of computing a global minimizer (one that minimizes  $f(x)$  over all  $x$ ). Example:

```
>> [x,fval] = fminbnd(@(x)sin(x)-cos(x),-pi,pi)
x =
    -0.7854
fval =
    -1.4142
```

As for `fzero`, options can be specified using a structure `options` set via the `optimset` function. In addition to the fields used by `fzero`, `fminbnd` uses `MaxFunEvals` (the maximum number of function evaluations allowed) and `MaxIter` (the maximum number of iterations allowed). The defaults correspond to

```
optimset('Display','notify','MaxFunEvals',500,'MaxIter',500,...
        'TolX',1e-4,'FunValCheck','off','OutputFcn',[])
```

The algorithm used by `fminbnd`—a combination of golden section search and parabolic interpolation—is described in [47, Chap. 8].

If you wish to maximize a function  $f$  rather than minimize it you can minimize  $-f$ , since  $\max_x f(x) = -\min_x (-f(x))$ .

Function `fminsearch` searches for a local minimum of a real function of  $n$  real variables. The syntax is similar to `fminbnd` except that a starting vector rather than an interval is supplied: `x = fminsearch(fun,x0,options)`. The fields in `options` are those supported by `fminbnd` plus `TolFun`, a termination tolerance on the function value. Both `TolX` and `TolFun` default to `1e-4`. To illustrate the use of `fminsearch` we consider the quadratic function

$$F(x) = x_1^2 + x_2^2 - x_1x_2,$$

which has a minimum at  $x = [0 \ 0]^T$ . Given the function

```
function f = fquad(x)
f = x(1)^2 + x(2)^2 - x(1)*x(2);
```

we can type

```
>> [x,fval] = fminsearch(@fquad,ones(2,1),optimset('Disp','final'))
```

```
Optimization terminated:
the current x satisfies the termination criteria using
OPTIONS.TolX of 1.000000e-004
and F(X) satisfies the convergence criteria using
```

```

OPTIONS.TolFun of 1.000000e-004

x =
    1.0e-004 *
    -0.4582
    -0.4717
fval =
    2.1635e-009

```

Alternatively, we can define  $F$  using an anonymous function:

```
[x,fval] = fminsearch(@(x) x(1)^2+x(2)^2-x(1)*x(2),ones(2,1))
```

Function `fminsearch` is based on the Nelder–Mead simplex algorithm [17, Chap. 8], [95, Sec. 8.1], [142, Sec. 10.4], a direct search method that uses function values but not derivatives. The method can be very slow to converge, or may fail to converge to a local minimum. However, it has the advantage of being insensitive to discontinuities in the function. More sophisticated minimization functions can be found in the Optimization Toolbox.

## 11.4. The Fast Fourier Transform

The discrete Fourier transform of an  $n$ -vector  $x$  is the vector  $y = F_n x$ , where  $F_n$  is an  $n$ -by- $n$  matrix made up of roots of unity and illustrated by

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}, \quad \omega = e^{-2\pi i/4}.$$

$F_n$  is  $\sqrt{n}$  times a unitary matrix. The fast Fourier transform (FFT) is a more efficient way of forming  $y$  than the obvious matrix–vector multiplication. The `fft` function implements the FFT and is called as  $y = \text{fft}(x)$ . The efficiency of `fft` depends on the value of  $n$ ; prime values are bad, highly composite numbers are better, and powers of 2 are best. A second argument can be given to `fft`:  $y = \text{fft}(x, n)$  causes  $x$  to be truncated or padded with zeros to make  $x$  of length  $n$  before the FFT algorithm is applied. The inverse FFT,  $x = F_n^{-1}y = n^{-1}F_n^*y$ , is carried out by the `ifft` function:  $x = \text{ifft}(y)$ . Example:

```

>> y = fft([1 1 -1 -1]')
y =
    0.0000 + 0.0000i
    2.0000 - 2.0000i
    0.0000 + 0.0000i
    2.0000 + 2.0000i

>> x = ifft(y)
x =
     1
     1
    -1
    -1

```

MATLAB also implements higher-dimensional discrete Fourier transforms and their inverses: see functions `fft2`, `fftn`, `ifft2`, and `ifftn`.

To compute FFTs MATLAB uses a package called FFTW (the “Fastest Fourier Transform in the West”) [49]. FFTW is an example of self-adapting numerical software [32], which tunes itself to obtain the best speed on the computational environment in which it is running. The function `fftw` provides control over the tuning process; see the online documentation for details.

#### FFT

James W. Cooley co-authored the classic 1965 publication on the FFT [19]. In *How the FFT gained acceptance* [18] he gives a candid account of rediscoveries, and prediscovers, of the algorithm. His article begins

“The fast Fourier transform (FFT) has had a fascinating history, filled with ironies and enigmas.”

and finishes with the following list of recommendations.

- “Prompt publication of significant achievements is essential.
- Reviews of old literature can be rewarding.
- Communication among mathematicians, numerical analysts, and workers in a wide range of applications can be fruitful.
- Do not publish in neoclassical Latin.”

Table 11.1. *Top ten algorithms. Left: based on The Princeton Companion to Applied Mathematics, in decreasing order. Right: Dongarra and Sullivan's list, in chronological order.*

2015	2000
Newton and quasi-Newton methods	Metropolis algorithm for Monte Carlo
Matrix factorizations (LU, Cholesky, QR)	Simplex method for linear programming
Singular value decomposition, QR and QZ algorithms	Krylov subspace iteration methods
Monte-Carlo methods	The decompositional approach to matrix computations
Fast Fourier transform	The Fortran optimizing compiler
Krylov subspace methods (conjugate gradients, Lanczos, GMRES, minres)	QR algorithm for computing eigenvalues
JPEG	Quicksort algorithm for sorting
PageRank	Fast Fourier transform
Simplex algorithm	Integer relation detection
Kalman filter	Fast multipole method

#### THE TOP TEN ALGORITHMS

In 2000, Dongarra and Sullivan selected the “10 algorithms with the greatest influence on the development and practice of science and engineering in the 20th century” and edited a collection of articles about them [36]. One of us decided to see how the selected algorithms compare with the algorithms having the most page locators in the index of *The Princeton Companion to Applied Mathematics* [80], [83]. The two lists, produced 15 years apart, are shown in Table 11.1. They are remarkably similar, agreeing in six of their entries. Two of the entries that are only in the new list, JPEG and PageRank, were relatively new in 2000, having been developed in 1992 and 1998, respectively.

*Life as we know it would be very different without the FFT.*

— CHARLES F. VAN LOAN, *Computational Frameworks for the Fast Fourier Transform* (1992)

*Do you ever want to kick the computer?  
Does it iterate endlessly on your newest algorithm  
that should have converged in three iterations?  
And does it finally come to a crashing halt  
with the insulting message that you divided by zero?*

*These minor trauma are, in fact,  
the ways the computer manages to kick you and,  
unfortunately, you almost always deserve it!*

*For it is a sad fact that most of us  
can more easily compute than think—  
which might have given rise to that famous definition,  
“Research is when you don’t know what you’re doing.”*

— FORMAN S. ACTON, *Numerical Methods That Work* (1970)

# Chapter 12

## Numerical Methods: Part II

We now move on to describe the capabilities of MATLAB for evaluating integrals and solving ordinary differential equations (both initial-value problems and boundary-value problems), delay-differential equations, and partial differential equations.

Most of the solvers discussed in this chapter support mixed absolute/relative error tests, with tolerances `AbsTol` and `RelTol`, respectively. This means that they test whether an estimate `err` of some measure of the error in the vector `x` is small enough by testing whether, for all `i`,

$$\text{err}(i) \leq \max(\text{AbsTol}, \text{RelTol} * \text{abs}(x(i)))$$

If `AbsTol` is zero this is a pure relative error test, and if `RelTol` is zero it is a pure absolute error test. Since we cannot expect to obtain an answer with more correct significant digits than the 16 or so to which MATLAB works, `RelTol` should be no smaller than about `eps`; and since `x = 0` is a possibility we should also take `AbsTol > 0`. A rough way of interpreting the mixed error test above is that `err(i)` is acceptably small if `x(i)` has as many correct digits as specified by `RelTol` or is smaller than `AbsTol` in absolute value. The default values are listed in Table 12.1. `AbsTol` can be a vector of absolute tolerances, in which case the test is

$$\text{err}(i) \leq \max(\text{AbsTol}(i), \text{RelTol} * \text{abs}(x(i)))$$

Several of the functions described in this chapter employ structures as input and output arguments, in order to group several related pieces of information in one variable. See Section 18.7 for full details of structures.

### 12.1. Numerical Integration

Numerical integration is the approximation of definite integrals  $\int_a^b f(x) dx$ . The main MATLAB function for numerical integration is `integral`. The basic usage is `q = integral(fun,a,b,tol)`, where `fun` specifies the function to be integrated. The function `fun` must accept a vector argument and return a vector of function values. The argument `tol` is an absolute error tolerance, which defaults to  $10^{-6}$ . To approximate  $\int_2^4 x \log x dx$  we can type

Table 12.1. *Default values for absolute and relative error tolerances.*

	Numerical integration	Differential equations
<code>AbsTol</code>	1e-10	1e-6
<code>RelTol</code>	1e-6	1e-3

```
>> integral(@(x)x.*log(x),2,4)
ans =
    6.7041
```

Note the use of array multiplication (`.*`) to make the anonymous function work for vector inputs. The default absolute and relative error tolerances, given in Table 12.1, may be changed by providing name–value pairs, as in the next example:

```
>> f = @(x)sqrt(1 + cos(x).^2); a = 0; b = 48;
>> integral(f,a,b), integral(f,a,b,'Abstol',0,'Reltol',1e-14)
ans =
    58.470469154905132
ans =
    58.470469154899320
```

These two results are correct to about 13 and 16 significant figures, respectively, which is better than the error tolerances would lead us to suspect.

The function `integral` has a number of other features.

- It can handle infinite  $a$  or  $b$  and endpoint singularities.
- For scalar integrands the function `fun` must accept a vector argument and return a vector output. If the name–value pair `'ArrayValued',true` is supplied then the integrand is assumed to return an array (vector, matrix, or multidimensional array) for scalar inputs. This is intended for integrating array functions.
- Integration waypoints—points at which the integrand must be evaluated—can be specified as a pair `'Waypoints',v`, where  $v$  is a vector of real or complex numbers. If the function or its derivative has discontinuities on the interval of integration then it is helpful to take these as waypoints.
- The limits of integration and the waypoints can be complex numbers, in which case the integration is performed over a sequence of straight-line paths in the complex plane.

Here are some examples of these features. The first example uses the `ArrayValued` option.

```
>> f = @(x)([cos(x).^2, sin(x).^2]); a = 0; b = 1/3;
>> q = integral(f,a,b,'ArrayValued',true); % Integrate the vector f.
>> sum(q) - (b - a) % Should be zero.
ans =
    5.5511e-17
```

The next example uses waypoints to speed up the evaluation of an integral with discontinuities in the derivative of the integrand:

```
>> format long
>> f = @(x)(abs(x-1/sqrt(3)) + abs(x-1/sqrt(2))); a = -1; b = 2;
>> tic, q = integral(f,a,b,'RelTol',1e-12), toc
q =
    4.548876283013526
Elapsed time is 0.004175 seconds.
```



```
>> tic, q = integral(f,a,b,'RelTol',1e-12,...
                    'Waypoints',[1/sqrt(3),1/sqrt(2)]), toc
q =
    4.548876282957160
Elapsed time is 0.001549 seconds.
```

The final example is a complex integral. We specify equal starting and ending points 1 for the integration, with the waypoints directing the integration along the edges of the unit square centered on the origin. Cauchy's residue theorem tells us that the result should be  $2\pi i$ :

```
>> integral(@(z) cos(z)./z,1,1,'Waypoints',[1+i,-1+i,-1-i,1-i])
ans =
    0.0000 + 6.2832i
```

If the integrand has singularities on the range of integration it is recommended to split the range into subintervals so that the singularities appear at the ends of the subintervals, then integrate separately over each subinterval, and add the results.

It should be kept in mind that a variety of analytic approaches are available to convert an integral over an infinite range to one over a finite range or to remove singularities. These include change of variable, integration by parts, and analytic treatment of the integral over part of the range. See numerical analysis textbooks for details, for example, [7, Sec. 5.6], [23, Sec. 7.4.3], and [151, Sec. 5.4].

The `integral` function uses global adaptive quadrature based on a Gauss–Kronrod (7, 15) pair of integration rules. It breaks the range of integration into subintervals and applies the basic integration rule over each subinterval. It chooses the subintervals according to the local behavior of the integrand, placing the smallest ones where the integrand is changing most rapidly. Warning messages are produced if the subintervals become very small or if an excessive number of function evaluations is used, either of which could indicate that the integrand has a singularity.

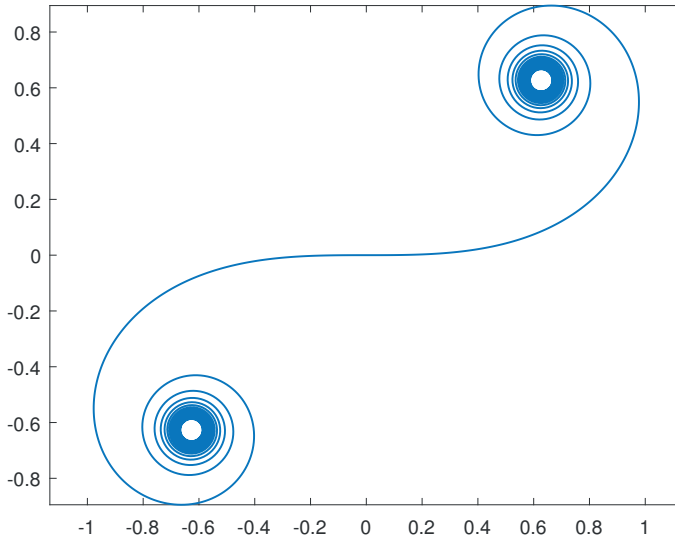
Functions `quad`, based on Simpson's rule, and `quadl`, based on the 4-point Gauss–Lobatto rule together with a 7-point Kronrod extension, were for many years the main MATLAB functions for numerical integration, but they have been superseded and are marked as “will be removed in a future release”. Function `quadgk` has a very similar specification to `integral` but lacks the `ArrayValued` option. See [128] for some historical perspective on the `quad*` functions.

For another example we take the Fresnel integrals

$$x(t) = \int_0^t \cos u^2 \, du, \quad y(t) = \int_0^t \sin u^2 \, du.$$

Plotting  $x(t)$  against  $y(t)$  produces a spiral [60, Sec. 2.6]. The following code plots the spiral by sampling at 2001 equally spaced points  $t$  on the interval  $[-4\pi, 4\pi]$ ; the result is shown in Figure 12.1. For efficiency we exploit symmetry and avoid repeatedly integrating from 0 to  $t$  by integrating over each subinterval and then evaluating the cumulative sums using `cumsum`:

```
n = 1000;
x = zeros(1,n); y = x;
t = linspace(0,4*pi,n+1);
for i = 1:n
```

Figure 12.1. *Fresnel spiral.*

```

x(i) = integral(@(x) cos(x.^2),t(i),t(i+1));
y(i) = integral(@(x) sin(x.^2),t(i),t(i+1));
end
x = cumsum(x); y = cumsum(y);
plot([-x(end:-1:1) 0 x], [-y(end:-1:1) 0 y], 'LineWidth',1)
axis equal

```

Another numerical integration function is `trapz`, which applies the repeated trapezium rule. It differs from `integral` in that its input comprises vectors of  $x_i$ - and  $f(x_i)$ -values rather than a function representing the integrand  $f$ ; therefore it is not adaptive. Example:

```

>> x = linspace(0,2*pi,10);
>> f = sin(x).^2./sqrt(1+cos(x).^2);
>> trapz(x,f)
ans =
    2.8478

```

In this example the error in the computed integral is of the order  $10^{-7}$ , which is much smaller than the standard error expression for the repeated trapezium rule would suggest. The reason is that we are integrating a periodic function over a whole number of periods and the repeated trapezium rule is known to be highly accurate in this situation [7, Sec. 5.4], [151, p. 182], [168]. In general, though, provided that a function is available to evaluate the integrand at arbitrary points, `integral` and `quadgk` are preferable to `trapz`.

Double integrals can be evaluated with `integral2`. To illustrate, suppose we wish to approximate the integral

$$\int_4^6 \int_0^1 (y^2 e^x + x \cos y) dx dy.$$

We type

```
>> integral2(@(x,y)(y.^2.*exp(x) + x.*cos(y)),0,1,4,6)
ans =
    87.2983
```

The function passed to `dblquad` must accept a vector `x` and a scalar `y` and return a vector as output. Additional arguments to `dblquad` can be used to specify the tolerance and the method of integration.

An analogous function `integral3` evaluates triple integrals.

## 12.2. Ordinary Differential Equations

MATLAB has a range of functions for solving initial-value ordinary differential equations (ODEs). These mathematical problems have the form

$$\frac{d}{dt}y(t) = f(t, y(t)), \quad y(t_0) = y_0, \quad (12.1)$$

where  $t$  is a real scalar,  $y(t)$  is an unknown  $m$ -vector, and the given function  $f$  of  $t$  and  $y$  is also an  $m$ -vector. To be concrete, we regard  $t$  as representing time. The function  $f$  defines the ODE, and the initial condition  $y(t_0) = y_0$  then defines an initial-value problem. The simplest way to solve such a problem is to write a function that evaluates  $f$  and then calls one of the MATLAB ODE solvers. The minimum information that the solver must be given is the function name, the range of  $t$ -values over which the solution is required, and the initial condition  $y_0$ . However, the MATLAB ODE solvers allow for extra (optional) input and output arguments that make it possible to specify more about the mathematical problem and how it is to be solved. Each of the solvers is designed to be efficient in specific circumstances, but all are essentially interchangeable. In the next subsection we develop examples that illustrate the use of `ode45`. This function implements an adaptive Runge–Kutta algorithm and is typically the most efficient solver for the classes of ODEs that concern MATLAB users. The full range of ODE solving functions is discussed in Section 12.2.3 and is listed in Table 12.2 on p. 208. The functions follow a naming convention: all names begin `ode` and are followed by digits denoting the orders of the underlying integration formulas, with a final “s”, “t”, or “tb” denoting a function intended for stiff problems, and a final “i” denoting a function intended for fully implicit systems.

### 12.2.1. Examples with Ode45

In order to solve the scalar ( $m = 1$ ) ODE

$$\frac{d}{dt}y(t) = -y(t) - 5e^{-t} \sin 5t, \quad y(0) = 1,$$

for  $0 \leq t \leq 3$  with `ode45`, we create in the file `myf.m` the function

```
function yprime = myf(t,y)
%MYF ODE example function.
% YPRIME = MYF(t,y) evaluates derivative.

yprime = -y - 5*exp(-t)*sin(5*t);
```

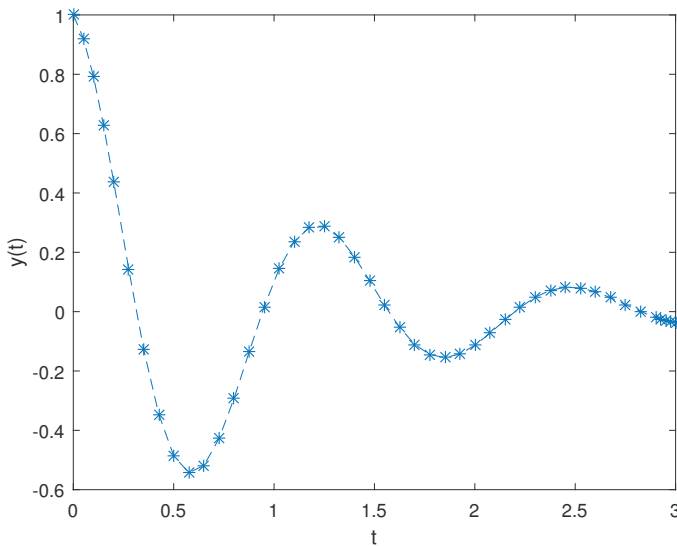


Figure 12.2. *Scalar ODE example.*

and then type

```
tspan = [0 3]; yzero = 1;
[t,y] = ode45(@myf,tspan,yzero);
plot(t,y,'*--')
xlabel t, ylabel y(t)
```

This produces the plot in Figure 12.2. (Note that here we have exploited command/function duality in setting the  $x$ - and  $y$ -axis labels—see Section 7.5.) The input arguments to `ode45` are the function `myf`, the 2-vector `tspan` that specifies the time interval, and the initial condition `yzero`. Two output arguments `t` and `y` are returned. The `t`-values are ordered in the range  $[0, 3]$  and `y(i)` approximates the solution at time `t(i)`. So `t(1) = 0` and `t(end) = 3`, with the points `t(2:end-1)` chosen automatically by `ode45` in much the same way that the adaptive quadrature routines choose their subintervals—the points are more closely spaced in regions where the solution is rapidly varying.

The solution to the scalar ODE above is  $y(t) = e^{-t} \cos 5t$ , so we may check the maximum error in the `ode45` approximation:

```
>> max(abs(y - exp(-t).*cos(5*t)))
ans =
    2.8991e-04
```

If more than two time values are specified, then `ode45` returns the solution at these times only, suppressing any solution values that may have been computed for intervening times:

```
>> tspan2 = 0:4;
>> [t2,y2] = ode45(@myf,tspan2,yzero);
>> disp([t2 y2])
```

```

      0      1.0000
    1.0000    0.1043
    2.0000   -0.1136
    3.0000   -0.0378
    4.0000    0.0075

```

Requesting the solution at specific times in this way has little effect on the computational cost of the integration. A decreasing list of times is allowed, so that integration is backward in time:

```

>> tspan3 = [0 -0.5 -1];
>> [t3,y3] = ode45(@myf,tspan3,yzero);
>> disp([t3 y3])
      0      1.0000
 -0.5000   -1.3209
 -1.0000    0.7711

```

At any point in the  $(t, y)$ -plane the differential equation (12.1) gives the gradient of the solution  $y(t)$ . Therefore it defines a *vector field* (or *direction field*) through which all solutions “navigate”. We can picture the vector field by plotting at each point  $(t, y)$  an arrow whose slope is  $f(t, y)$ . The following code uses the `quiver` function to plot the vector field for the differential equation at the start of this section:

```

n = 16;
tpts = linspace(0,3,n); ypts = linspace(-1,1,n);
[t,y] = meshgrid(tpts,ypts);
pt = ones(size(y));
py = -y-5*exp(-t).*sin(5*t);
quiver(t,y,pt,py,1.5);
title('dy/dt = -y-5e^{-t}sin(5t)', 'FontWeight', 'normal');
xlabel('t'), ylabel('y', 'Rotation', 0)
xlim([0 3.15]), ylim([-1.25 1.1]) % Tune axis limits.

```

The command `quiver(x,y,u,v,scale)` plots arrows with components  $(u, v)$  at the locations  $(x, y)$ , producing arrows whose length is `scale` times the 2-norm of the  $[u(i), v(i)]$  vectors. Figure 12.3 shows the resulting graph.

Higher-order ODEs can be solved if they are first rewritten as a larger system of first-order ODEs [56, Sec. 1.2], [148, Chap. 1]. For example, the simple pendulum equation [161, Sec. 6.7] has the form

$$\frac{d^2}{dt^2}\theta(t) + \sin\theta(t) = 0.$$

Defining  $y_1(t) = \theta(t)$  and  $y_2(t) = d\theta(t)/dt$ , we may rewrite this equation as the two first-order equations

$$\begin{aligned}\frac{d}{dt}y_1(t) &= y_2(t), \\ \frac{d}{dt}y_2(t) &= -\sin y_1(t).\end{aligned}$$

This information can be encoded for use by `ode45` in the function `pend` as follows:

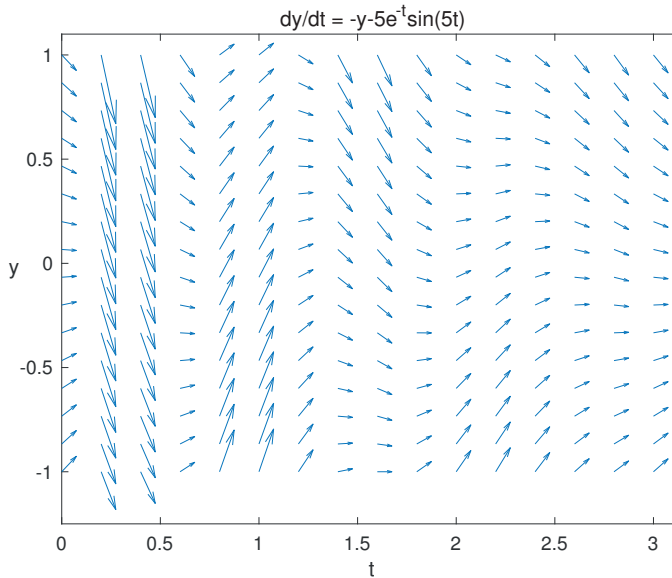


Figure 12.3. *Vector field for scalar ODE example.*

```
function yprime = pend(t,y)
%PEND Simple pendulum.
% yprime = PEND(t,y).
```

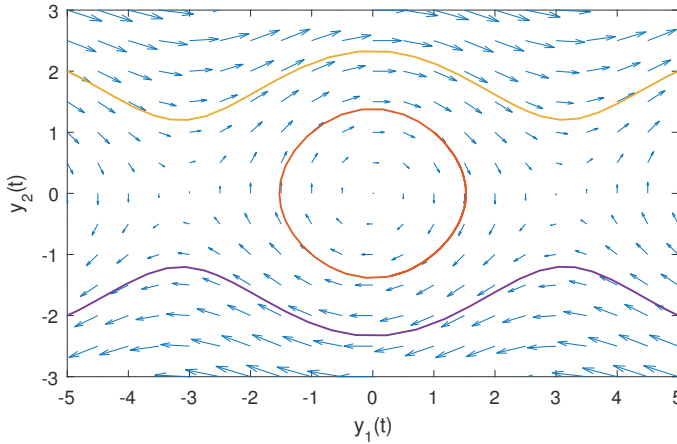
```
yprime = [y(2); -sin(y(1))];
```

The following commands compute solutions over  $0 \leq t \leq 10$  for three different initial conditions. Since we are solving a system of  $m = 2$  equations, in the output  $[\mathbf{t}, \mathbf{y}]$  from `ode45` the  $i$ th row of the matrix  $\mathbf{y}$  approximates  $(y_1(t), y_2(t))$  at time  $t = \mathbf{t}(i)$ .

```
tspan = [0 10];
yazero = [1; 1]; ybzero = [-5; 2]; yczero = [5; -2];
[ta,ya] = ode45(@pend,tspan,yazero);
[tb,yb] = ode45(@pend,tspan,ybzero);
[tc,yc] = ode45(@pend,tspan,yczero);
```

To produce phase plane plots, that is, plots of  $y_1(t)$  against  $y_2(t)$ , we simply plot the first column of the numerical solution against the second. The commands below generate phase plane plots of the solutions `ya`, `yb`, and `yc` computed above, and make use of `quiver` to superimpose a vector field. The resulting picture is shown in Figure 12.4.

```
[y1,y2] = meshgrid(-5:.5:5,-3:.5:3);
Dy1Dt = y2; Dy2Dt = -sin(y1);
quiver(y1,y2,Dy1Dt,Dy2Dt)
hold on
plot(ya(:,1),ya(:,2),yb(:,1),yb(:,2),yc(:,1),yc(:,2),'LineWidth',1)
axis equal, axis([-5 5 -3 3])
xlabel y_1(t), ylabel y_2(t), hold off
```

Figure 12.4. *Pendulum phase plane solutions.*

The pendulum ODE preserves energy: any solution keeps  $y_2(t)^2/2 - \cos y_1(t)$  constant for all  $t$ . We can check that this is approximately true for `yc` as follows:

```
>> Ec = .5*yc(:,2).^2 - cos(yc(:,1));
>> max(abs(Ec(1)-Ec))
ans =
    0.0263
```

The general form of a call to `ode45` is

```
[t,y] = ode45(@fun,tspan,yzero,options);
```

The optional argument `options` is a structure that controls many features of the solver and can be set via the `odeset` function. In our next example we create a structure `options` by the assignment

```
options = odeset('AbsTol',1e-7,'RelTol',1e-4);
```

Passing this structure as an input argument to `ode45` causes the absolute and relative error tolerances to be set to  $10^{-7}$  and  $10^{-4}$ , respectively. (The default values are  $10^{-6}$  and  $10^{-3}$ , as given in Table 12.1; see the start of this chapter and `doc odeset` for more details about the tolerances.) These tolerances apply on a local, step-by-step, basis and it is not generally the case that the overall error is kept within these limits. However, under reasonable assumptions about the ODE, it can be shown that decreasing the tolerances by some factor, say 100, will decrease the overall error by a similar factor, so the error is usually roughly proportional to the tolerances. See [148, Chap. 7] for further details about error control in ODE solvers.

We now consider the Rössler system [161, Secs. 10.6, 12.3],

$$\begin{aligned}\frac{d}{dt}y_1(t) &= -y_2(t) - y_3(t), \\ \frac{d}{dt}y_2(t) &= y_1(t) + ay_2(t), \\ \frac{d}{dt}y_3(t) &= b + y_3(t)(y_1(t) - c),\end{aligned}$$

Listing 12.1. *Function rossler\_ex.*

```

function rossler_ex
%ROSSLER_EX    Run Rossler example.

tspan = [0 100]; yzero = [1; 1; 1];
options = odeset('AbsTol',1e-7,'RelTol',1e-4);

a = 0.2; b = 0.2; c = 2.5;
[t,y] = ode45(@rossler,tspan,yzero,options);
subplot(221), plot3(y(:,1),y(:,2),y(:,3)), mytitle, zlabel y_3(t), grid
subplot(223), plot(y(:,1),y(:,2)), mytitle

c = 5;
[t,y] = ode45(@rossler,tspan,yzero,options);
a = get(groot,'defaultAxesColorOrder'); c2 = {'Color',a(2,:)};
subplot(222), plot3(y(:,1),y(:,2),y(:,3),c2{:})
mytitle, zlabel y_3(t), grid
subplot(224), plot(y(:,1),y(:,2),c2{:}), mytitle

% ----- Nested functions -----
function yprime = rossler(t,y)
%ROSSLER    Rossler system, parameterized.
yprime = [-y(2)-y(3); y(1)+a*y(2); b+y(3)*(y(1)-c)];
end

function mytitle
title(sprintf('c = %2.1f',c))
xlabel y_1(t), ylabel y_2(t)
end

end

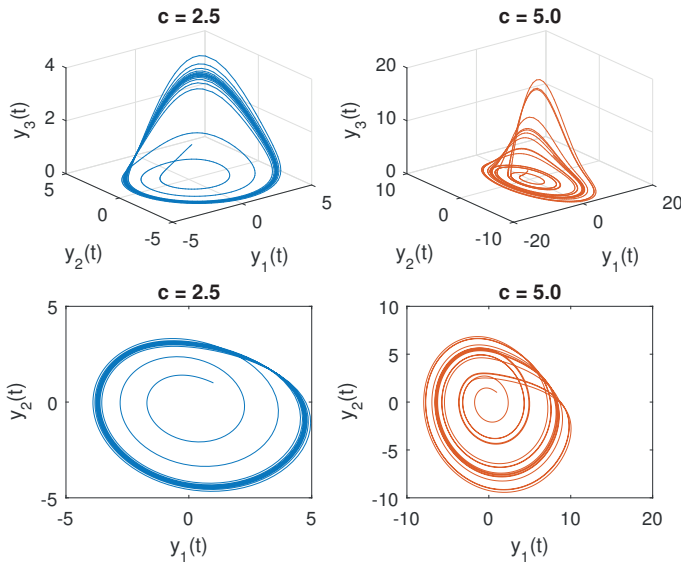
```

where  $a$ ,  $b$ , and  $c$  are parameters. The function `rossler_ex` in Listing 12.1 solves the Rössler system over  $0 \leq t \leq 100$  with initial condition  $y(0) = [1 \ 1 \ 1]^T$  for  $(a, b, c) = (0.2, 0.2, 2.5)$  and  $(a, b, c) = (0.2, 0.2, 5)$ . The ODE is defined in the nested function `rossler`, in order that the parameters  $a$ ,  $b$ , and  $c$ , defined in the main function, are captured in the function handle `@rossler` that is passed to `ode45`. The nested function `mytitle` is used to avoid repetition of title commands. For more on nested functions, see Section 10.7. Figure 12.5 shows the results. The 221 subplot gives the 3D phase space solution for  $c = 2.5$  and the 223 subplot gives the 2D projection onto the  $(y_1, y_2)$ -plane. The 222 and 224 subplots give the corresponding pictures for  $c = 5$ .

Function `rossler_ex` illustrates how a complete problem specification and solution can be encapsulated in a single function (which does not need to have any input or output arguments) by making use of nested functions, local functions, and function handles. (Note that nested functions are needed only when the ODE is parameter dependent.)

The ODE solvers may also be called with a single output argument. Specifying



Figure 12.5. *Rössler system phase space solutions.*

```
sol = ode45(@fun,tspan,yzero,options);
```

causes the solution to be returned in a structure `sol`. The fields `sol.x` and `sol.y` are equivalent to the output arguments `t` and `y`, respectively, that arise from the call `[t,y] = ode45(@fun,tspan,yzero,options)`. Hence, the field `sol.x` is a row vector containing the  $t$ -values chosen by `ode45`, and the field `sol.y` is an array whose  $i$ th column `sol.y(:,i)` contains the solution at the points `sol.x(i)`. A utility function `deval` is available that, given `sol`, will evaluate the solution (and, optionally, the derivative) at any set of intermediate  $t$  values. So, if `trange` is a vector of points between `sol.x(1)` and `sol.x(end)`, then `ysol = deval(sol,trange)` will return an array `ysol` whose  $i$ th column corresponds to the solution at `trange(i)`. Adding a third input argument, `ysol = deval(sol,trange,idx)`, restricts the output to solution components specified by the array `idx`. For example, `deval(sol,trange,[1,3])` picks out the first and third solution components. We may add a second output argument: `[ysol,ypsol] = deval(sol,trange,idx)` returns an array `ypsol` containing the corresponding approximations to the first derivative of the solution.

The use of the name `sol.x`, rather than `sol.t`, for the array containing values of the independent variable arose because the structure output format was first introduced with the boundary-value problem solver, `bvp4c` (see Section 12.3).

To illustrate the use of `deval`, we look at the task of plotting  $y_1(t)$  against  $y_1(t-\tau)$ , for some fixed  $\tau$ . This operation arises in *attractor reconstruction* [161, Sec. 12.4], where an attempt is made to recover dynamics in complete phase space from a single solution component. Using the `[t,y] = ode45(...)` mode would be inconvenient in this case, because if a point  $t^*$  appears in the array `t` it is not generally true that  $t^* - \tau$  also appears. The function `rossler_ex2` in Listing 12.2 uses two calls to `deval` and produces Figure 12.6.

Listing 12.2. *Function rossler\_attract2.*

```

function rossler_attract
%ROSSLER_ATTRACT    Attractor reconstruction for Rossler system.

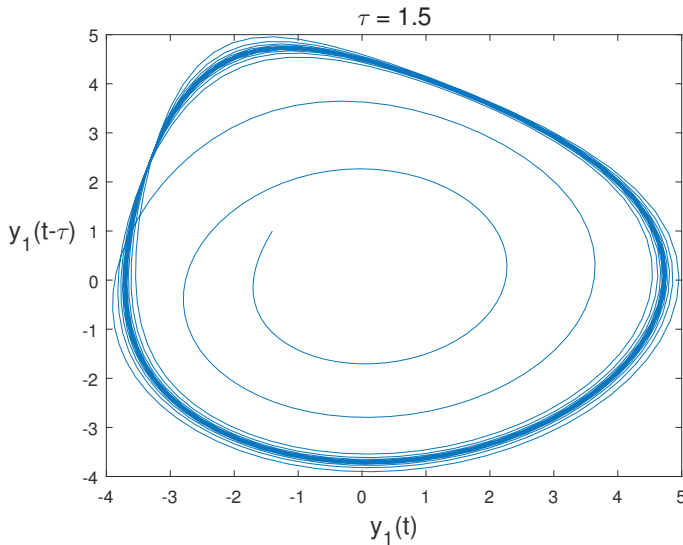
tspan = [0 100]; yzero = [1; 1; 1];
options = odeset('AbsTol',1e-7,'RelTol',1e-4);

a = 0.2; b = 0.2; c = 2.5;
sol = ode45(@rossler,tspan,yzero,options);
tau = 1.5;
t = linspace(tau,100,1000);
y = deval(sol,t,1);
ylag = deval(sol,t-tau,1);
plot(y,ylag), title('\tau = 1.5','FontSize',14)
xlabel('y_1(t)','FontSize',14)
ylabel('y_1(t-\tau)','FontSize',14,'Rotation',0,...
      'HorizontalAlignment','right')

function yprime = rossler(t,y)
%ROSSLER    Rossler system, parametrized.
yprime = [-y(2)-y(3); y(1)+a*y(2); b+y(3)*(y(1)-c)];
end

end

```

Figure 12.6. *Attractor reconstruction using deval.*

### 12.2.2. Case Study: Pursuit Problem with Event Location

Next we consider a pursuit problem [24, Chap. 5]. Suppose that a rabbit follows a predefined path  $(r_1(t), r_2(t))$  in the plane and that a fox chases the rabbit in such a way that (a) at each moment the tangent of the fox's path points toward the rabbit and (b) the speed of the fox is some constant  $k$  times the speed of the rabbit. Then the path  $(y_1(t), y_2(t))$  of the fox is determined by the ODE

$$\begin{aligned}\frac{d}{dt}y_1(t) &= s(t)(r_1(t) - y_1(t)), \\ \frac{d}{dt}y_2(t) &= s(t)(r_2(t) - y_2(t)),\end{aligned}$$

where

$$s(t) = \frac{k\sqrt{\left(\frac{d}{dt}r_1(t)\right)^2 + \left(\frac{d}{dt}r_2(t)\right)^2}}{\sqrt{(r_1(t) - y_1(t))^2 + (r_2(t) - y_2(t))^2}}.$$

Note that this ODE system becomes ill-defined if the fox approaches the rabbit. We let the rabbit follow an outward spiral,

$$\begin{bmatrix} r_1(t) \\ r_2(t) \end{bmatrix} = \sqrt{1+t} \begin{bmatrix} \cos t \\ \sin t \end{bmatrix},$$

and start the fox at  $y_1(0) = 3$ ,  $y_2(0) = 0$ . The function `fox1` in Listing 12.3 implements the ODE, with  $k$  set to 0.75. The `error` function (see Section 14.1) has been used so that execution terminates with an error message if the denominator of  $s(t)$  in the ODE becomes too small. The script below calls `fox1` to produce Figure 12.7. Initial conditions are denoted by circles, and the dashed and solid lines show the phase plane paths of the rabbit and the fox, respectively:

```
tspan = [0 10]; yzero = [3; 0];
LW = 'LineWidth';
[tfox,yfox] = ode45(@fox1,tspan,yzero);
plot(yfox(:,1),yfox(:,2),LW,1.5), hold on
plot(sqrt(1+tfox).*cos(tfox),sqrt(1+tfox).*sin(tfox),'--',LW,1.5)
plot([3 1],[0 0],'o','MarkerFaceColor','k');
axis equal, axis([-3.5 3.5 -2.5 3.1])
legend('Fox','Rabbit'), hold off
```

The implementation above is unsatisfactory for  $k > 1$ , that is, when the fox is faster than the rabbit. In this case, if the rabbit is caught within the specified time interval then no solution is displayed. It would be more natural to ask `ode45` to return with the computed solution if the fox and rabbit become close. Function `fox_rabbit` in Listing 12.4 does this by using the ODE solvers' event location facility, producing Figure 12.8. We have allowed  $k$  to be a parameter and set  $k = 1.1$ . The initial condition and the rabbit's path are as for Figure 12.7.

We use `odeset` to set the event location property to the handle of the local function `fox2_events`. This function has the three output arguments `value`, `isterminal`, and `direction`. It is the responsibility of `ode45` to use `fox2_events` to check whether any component passes through zero by monitoring the quantity returned in `value`. In our example `value` is a scalar, corresponding to the distance between the rabbit and the fox, minus a threshold of  $10^{-4}$ . Hence, `ode45` checks if the fox has approached

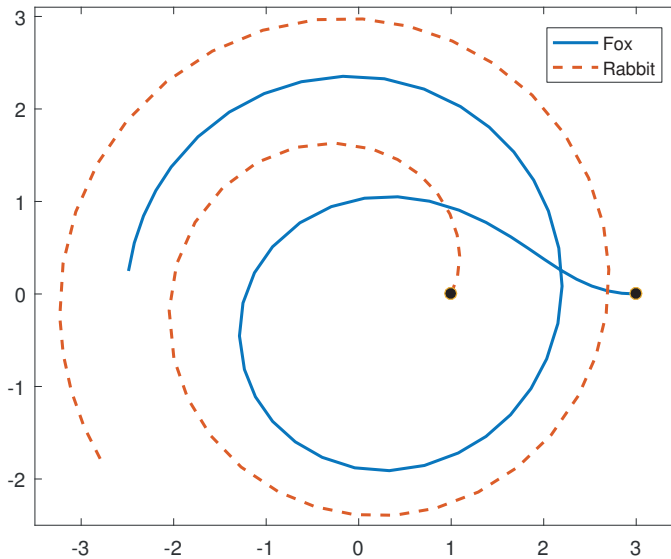
Listing 12.3. *Function fox1.*

```

function yprime = fox1(t,y)
%FOX1  Fox-rabbit pursuit simulation.
%  yprime = FOX1(t,y).

k = 0.75;
r = sqrt(1+t)*[cos(t); sin(t)];
r_p =(0.5/sqrt(1+t))*[cos(t)-2*(1+t)*sin(t);sin(t)+2*(1+t)*cos(t)];
dist = norm(r-y);
if dist > 1e-4
    factor = k*norm(r_p)/dist;
    yprime = factor*(r-y);
else
    error('ODE model ill-defined.')
end

```

Figure 12.7. *Pursuit example.*

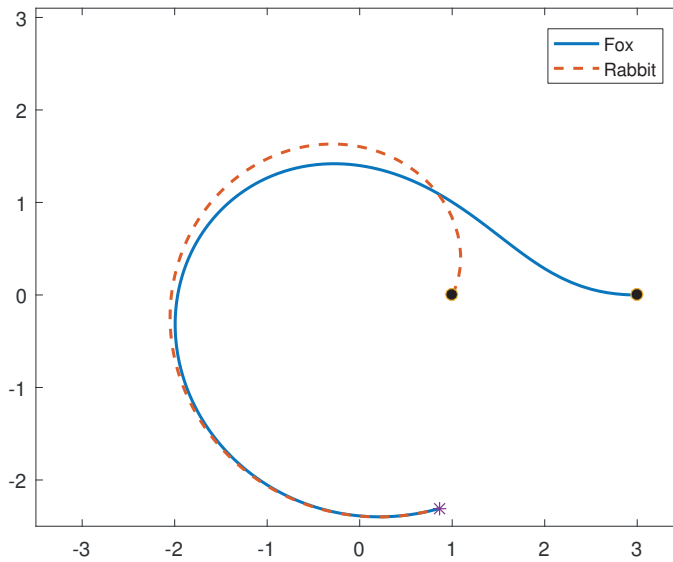


Figure 12.8. *Pursuit example, with capture.*

within distance  $10^{-4}$  of the rabbit. We set `direction = -1`, which signifies that `value` must be decreasing through zero in order for the event to be considered. The alternative choice `direction = 1` tells MATLAB to consider only crossings where `value` is increasing, and `direction = 0` allows for any type of zero. Since we set `isterminal = 1`, integration will cease when a suitable zero crossing is detected. With the other option, `isterminal = 0`, the event is recorded and the integration continues.

The output arguments from `ode45` are `[tfox,yfox,te,ye,ie]`. Here, `tfox` and `yfox` are the usual solution approximations, so `yfox(i,:)` approximates  $y(t)$  at time  $t = tfox(i)$ . The arguments `te` and `ye` record those  $t$  and  $y$  values at which the event(s) were recorded and, for vector-valued events, `ie` specifies which component of the event occurred each time. (If no events are detected then `te`, `ye`, and `ie` are returned as empty matrices.) In our example, we have

```
>> te, ye
te =
    5.0710
ye =
    0.8646   -2.3073
```

showing that the rabbit was captured after 5.07 time units at the point  $(0.86, -2.31)$ .

Listing 12.4. *Function fox\_rabbit.*

```

function fox_rabbit
%FOX_RABBIT    Fox-rabbit pursuit simulation.
%    Uses relative speed parameter, k.

k = 1.1;
tspan = [0 10]; yzero = [3; 0];
options = odeset('RelTol',1e-6,'AbsTol',1e-6,'Events',@fox2_events);
[tfox,yfox,te,ye,ie] = ode45(@fox2,tspan,yzero,options);
LW = 'LineWidth';
plot(yfox(:,1),yfox(:,2),LW,1.5), hold on
plot(sqrt(1+tfox).*cos(tfox),sqrt(1+tfox).*sin(tfox),'--',LW,1.5)
plot([3 1],[0 0],'o','MarkerFaceColor','k')
plot(yfox(end,1),yfox(end,2),'*')
axis equal, axis([-3.5 3.5 -2.5 3.1])
legend('Fox','Rabbit'), hold off

    function yprime = fox2(t,y)
    %FOX2    Fox-rabbit pursuit simulation ODE.

    r = sqrt(1+t)*[cos(t); sin(t)];
    r_p = (0.5/sqrt(1+t)) * [cos(t)-2*(1+t)*sin(t); sin(t)+2*(1+t)*cos(t)];
    dist = max(norm(r-y),1e-6);
    factor = k*norm(r_p)/dist;
    yprime = factor*(r-y);

    end

end

function [value,isterminal,direction] = fox2_events(t,y)
%FOX2_EVENTS    Events function for fox2.
%    Locate when fox is close to rabbit.

r = sqrt(1+t)*[cos(t); sin(t)];
value = norm(r-y) - 1e-4;    % Fox close to rabbit.
isterminal = 1;            % Stop integration.
direction = -1;            % Value must be decreasing through zero.

end

```

### 12.2.3. Stiff Problems, Differential-Algebraic Equations, and the Choice of Solver

The Robertson ODE system

$$\begin{aligned}\frac{d}{dt}y_1(t) &= -0.04y_1(t) + 10^4y_2(t)y_3(t), \\ \frac{d}{dt}y_2(t) &= 0.04y_1(t) - 10^4y_2(t)y_3(t) - 3 \times 10^7y_2(t)^2, \\ \frac{d}{dt}y_3(t) &= 3 \times 10^7y_2(t)^2\end{aligned}$$

models a reaction between three chemicals [61, p. 3], [148, p. 418]. We set the system up as the function `chem`:

```
function yprime = chem(t,y)
%CHEM    Robertson's chemical reaction model.
%    yprime = CHEM(t,y).

yprime = [-0.04*y(1) + 1e4*y(2)*y(3);
          0.04*y(1) - 1e4*y(2)*y(3) - 3e7*y(2)^2;
          3e7*y(2)^2];
```

The script file below solves this ODE for  $0 \leq t \leq 3$  with initial condition  $[1; 0; 0]$ , first using `ode45` and then using another solver, `ode15s`, which is based on implicit linear multistep methods. (Implicit means that a nonlinear algebraic equation must be solved at each step.) The results for  $y_2(t)$  are plotted in Figure 12.9.

```
tspan = [0 3]; yzero = [1; 0; 0];
[ta,ya] = ode45(@chem,tspan,yzero);
subplot(121), plot(ta,ya(:,2),'-*')
ax = axis; ax(1) = -0.2; axis(ax) % Make initial transient clearer.
xlabel('t'), ylabel('y_2(t)', 'Rotation', 0), title('ode45')
[tb,yb] = ode15s(@chem,tspan,yzero);
subplot(122), plot(tb,yb(:,2),'-*'), axis(ax)
xlabel('t'), ylabel('y_2(t)', 'Rotation', 0), title('ode15s')
```

We see from Figure 12.9 that the solutions agree to within a small absolute tolerance (note the scale factor  $10^{-5}$  for the  $y$ -axis labels). However, the left-hand solution from `ode45` has been returned at many more time values than the right-hand solution from `ode15s` and seems to be less smooth. To emphasize these points, Figure 12.10 plots `ode45`'s  $y_2(t)$  for  $2.0 \leq t \leq 2.1$ . We see that the  $t$ -values are densely packed, and spurious oscillations are present at the level of the default absolute error tolerance,  $10^{-6}$ . The Robertson problem is a classic example of a *stiff* ODE; see [61] or [148, Chap. 8] for full discussions about stiffness and its effects. Stiff ODEs arise in a number of application areas, including the modeling of chemical reactions and electrical circuits. Semi-discretized time-dependent partial differential equations are also a common source of stiffness (we give an example below). Many solvers behave inefficiently on stiff ODEs: they take an unnecessarily large number of intermediate steps in order to complete the integration and hence make an unnecessarily large number of calls to the ODE function (in this case, `chem`). We can obtain statistics on the computational cost of the integration by setting

```
options = odeset('Stats','on');
```

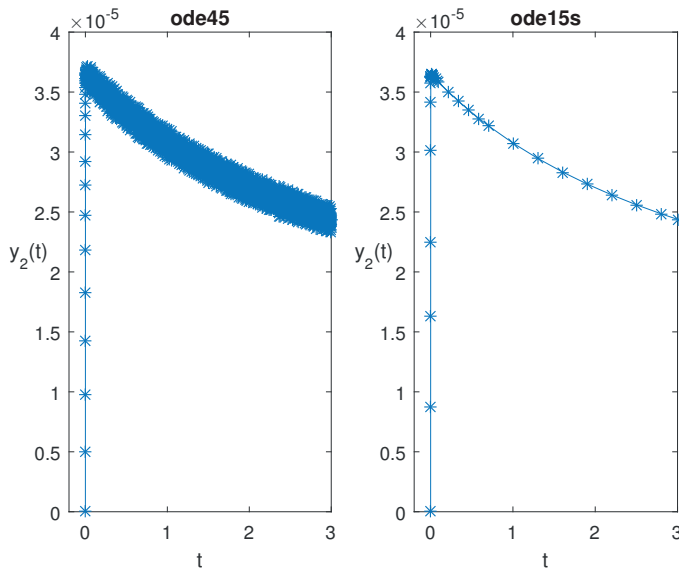


Figure 12.9. Chemical reaction solutions. Left: ode45. Right: ode15s.

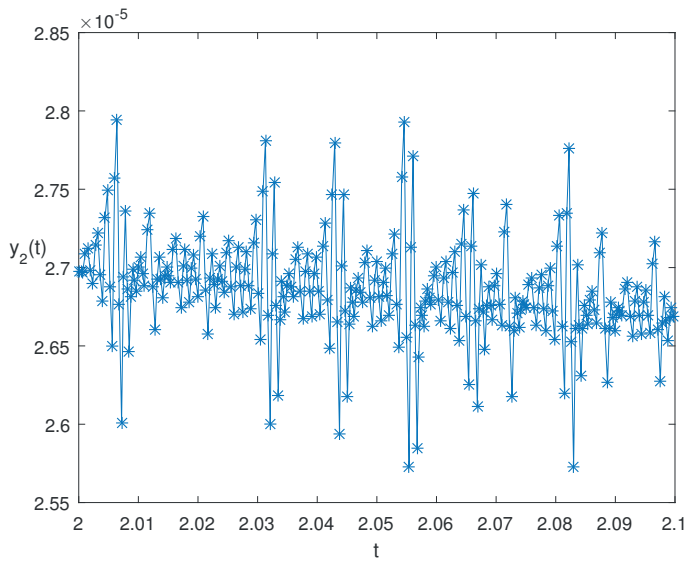


Figure 12.10. Zoom of chemical reaction solution from ode45.



and providing `options` as an input argument:

```
[ta,ya] = ode45(@chem,tspan,yzero,options);
```

On completion of the run of `ode45`, the following statistics are then printed:

```
2052 successful steps
440 failed attempts
14953 function evaluations
```

Using the same `options` argument with `ode15s` gives

```
33 successful steps
5 failed attempts
73 function evaluations
2 partial derivatives
13 LU decompositions
63 solutions of linear systems
```

The behavior of `ode45` typifies what happens when an adaptive algorithm designed for nonstiff ODEs operates in the presence of stiffness. The solver does not break down or compute an inaccurate solution, but it does behave nonsmoothly and extremely inefficiently in comparison with solvers that are customized for stiff problems. This is one reason why MATLAB provides a suite of ODE solvers.

Note that in the computation above, we have

```
>> disp([length(ta), length(tb)])
      8209          34
```

showing that `ode45` returned output at almost 250 times as many points as `ode15s`. However, the statistics show that `ode45` took 2051 steps, only about 62 times as many as `ode15s`. The explanation is that by default `ode45` uses interpolation to return four solution values at equally spaced points over each “natural” step. The default interpolation level can be overridden via the `Refine` option with `odeset`.

A full list of the MATLAB ODE solvers is given in Table 12.2. The authors of these solvers, Shampine and Reichelt, discuss some of the theoretical and practical issues that arose during their development in [154]. The functions are designed to be interchangeable in basic use. So, for example, the illustrations in the previous subsection continue to work if `ode45` is replaced by any of the other solvers. The functions mainly differ in (a) their efficiency on different problem types and (b) their capacity for accepting information about the problem in connection with Jacobians and mass matrices. With regard to efficiency, Shampine and Reichelt write in [154]:

The experiments reported here and others we have made suggest that except in special circumstances, `ode45` should be the code tried first. If there is reason to believe the problem to be stiff, or if the problem turns out to be unexpectedly difficult for `ode45`, the `ode15s` code should be tried.

The stiff solvers in Table 12.2 use information about the Jacobian matrix,  $\partial f_i / \partial y_j$ , at various points along the solution. By default, they automatically generate approximate Jacobians using finite differences. An option can be set via `odeset` to specify the sparsity pattern of the Jacobian, which aids construction of the finite difference



Listing 12.5. *Function* rcd.

```

function rcd
%RCD Stiff ODE from method of lines on reaction-convection-diffusion PDE.

N = 38; a = 1; b = 5e-2;
tspan = [0 2]; space = [1:N]/(N+1);

y0 = 0.5*(1+cos(2*pi*space));
y0 = y0(:);
options = odeset('Jacobian',@jacobian);
options = odeset(options,'RelTol',1e-3,'AbsTol',1e-3);

[t,y] = ode15s(@f,tspan,y0,options);
e = ones(size(t)); U = [e y e];
waterfall([0:1/(N+1):1],t,U)
colormap hsv
xlabel('space','FontSize',12), ylabel('time','FontSize',12)

% ----- Nested functions -----
function dydt = f(t,y)
%F    Differential equation.

r1 = -a*(N+1)/2;
r2 = b*(N+1)^2;
up = [y(2:N);0]; down = [0;y(1:N-1)];
e1 = [1;zeros(N-1,1)]; eN = [zeros(N-1,1);1];

dydt = r1*(up-down) + r2*(-2*y+up+down) + (r2-r1)*e1 + ...
      (r2+r1)*eN + y.*(1-y);
end

function dfdy = jacobian(t,y)
%JACOBIAN    Jacobian matrix.

r1 = -a*(N+1)/2;
r2 = b*(N+1)^2;
u = (r2-r1)*ones(N,1);
v = (-2*r2+1)*ones(N,1) - 2*y;
w = (r2+r1)*ones(N,1);

dfdy = spdiags([u v w],[-1 0 1],N,N);
end

end

```

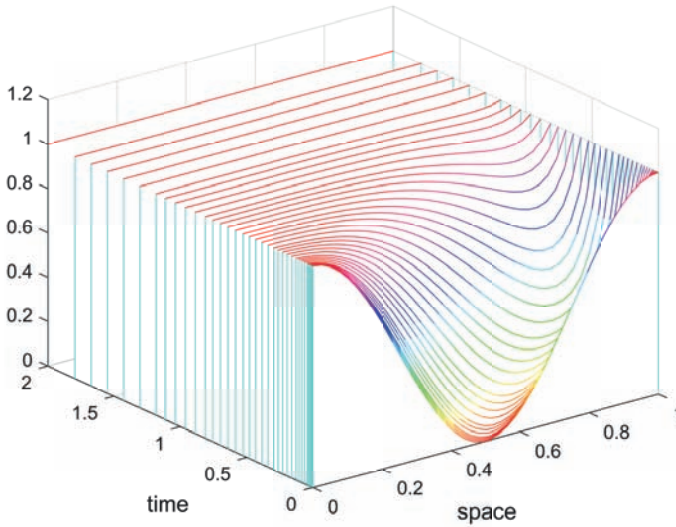


Figure 12.11. *Stiff ODE example, with Jacobian information supplied.*

$y$  to account for the PDE boundary conditions. The plot produced by `rcd` is shown in Figure 12.11.

The ODE solvers can be applied to problems of the form

$$M(t, y(t)) \frac{d}{dt} y(t) = f(t, y(t)), \quad y(t_0) = y_0,$$

where the *mass matrix*,  $M(t, y(t))$ , is square and nonsingular. (The `ode23s` solver applies only when  $M$  is independent of  $t$  and  $y(t)$ .) Mass matrices arise naturally when semi-discretization is performed with a finite-element method. A mass matrix can be specified in a similar manner to a Jacobian, via `odeset`. The `ode15s` and `ode23t` functions can solve certain problems where  $M$  is singular—more precisely, they can be used if the resulting differential-algebraic equation (DAE) is of index 1 and  $y_0$  is sufficiently close to being consistent. DAEs are a class of problems that contain algebraic, as well as differential, constraints on the variables; see [5], [14], or [109] for details.

We now discuss a DAE example: the Chemical Akzo Nobel problem of [164]. This index-1 system has the form

$$M \frac{d}{dt} y(t) = f(y(t)),$$

with  $y(t) \in \mathbb{R}^6$ ,  $0 \leq t \leq 180$ , and

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad f(y) = \begin{bmatrix} -2r_1 + r_2 - r_3 - r_4 \\ -0.5r_1 - r_4 - 0.5r_5 + F_{\text{in}} \\ r_1 - r_2 + r_3 \\ -r_2 + r_3 - 2r_4 \\ r_2 - r_3 + r_5 \\ K_s y_1 y_4 - y_6 \end{bmatrix},$$

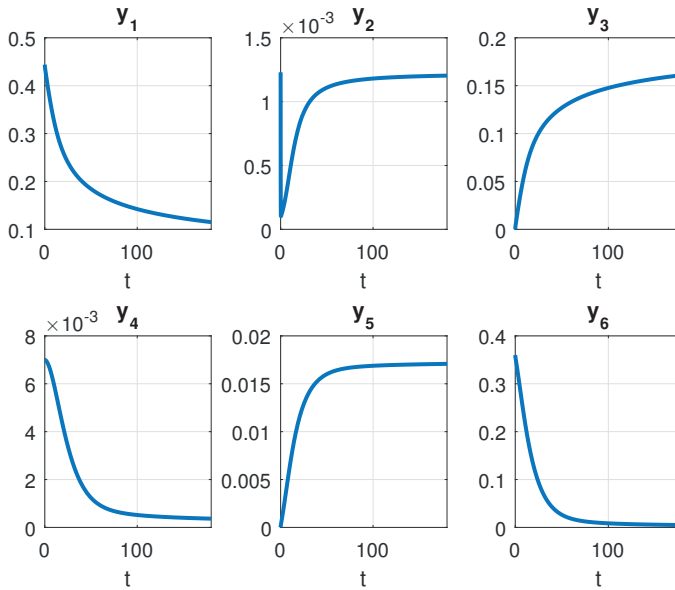


Figure 12.12. DAE solution components from `chemakzo` in Listing 12.6.

where the auxiliary variables are defined as

$$\begin{aligned} r_1 &= k_1 y_1^4 \sqrt{y_2}, & r_2 &= k_2 y_3 y_4, \\ r_3 &= k_2 y_1 y_5 / K, & r_4 &= k_3 y_1 y_4^2, \\ r_5 &= k_4 y_6^2 \sqrt{y_2}, & F_{\text{in}} &= k l A (p(\text{CO}_2) / H - y_2). \end{aligned}$$

Constants in the system take values  $k_1 = 18.7$ ,  $k_2 = 0.58$ ,  $k_3 = 0.09$ ,  $k_4 = 0.42$ ,  $K = 34.4$ ,  $k l A = 3.3$ ,  $K_s = 115.83$ ,  $p(\text{CO}_2) = 0.9$ , and  $H = 737$ , and the initial condition is

$$y_0 = [0.444, 0.00123, 0, 0.007, 0, K_s y_1(0) y_4(0)]^T.$$

Details about the mathematical modeling and chemistry issues behind this problem can be found in [164].

The function `chemakzo` in Listing 12.6 solves this DAE using `ode15s`, which can solve DAEs of index 1. We use `odeset` to set up an `options` structure that specifies the mass matrix and reports it as singular. In our case, because the mass matrix  $M$  is constant, we are able to specify it as the value of the `Mass` property. More generally, for a mass matrix that depends on  $t$  and  $y$ , a suitable function, `mymass(t,y)`, say, must be set up and the `Mass` property set to `@mymass`.

After solving the DAE, `chemakzo` plots the six solution components, as shown in Figure 12.12. These agree to visual accuracy with the solution plots in [164]. As a further check, we compare the solution at  $t = 180$  with the reference solution supplied in [164] and find that the norm of the difference is `yerr = 2.0626e-6`.

The solver `ode15i` is designed to handle general index-1 DAEs that may be written in the fully implicit form

$$F\left(t, y(t), \frac{dy(t)}{dt}\right) = 0,$$

Listing 12.6. *Function chemakzo.*

```

function chemakzo
%CHEMAKZO    Chemical Akzo Nobel problem.
%    Index 1 DAE describing a chemical process.

M = eye(6); M(6,6) = 0;
options = odeset('Mass',M,'MassSingular','yes');

tspan = [0 180];
Ks = 115.83;
y0 = [0.444; 0.00123; 0; 0.007; 0; Ks*0.444*0.007];
[t,y] = ode15s(@chem_rhs,tspan,y0,options);

for i = 1:6
    subplot(2,3,i)
    plot(t,y(:,i),'LineWidth',2), grid on
    title(['y_',int2str(i)]), xlabel('t'), xlim([0 180])
end

% Reference solution at t = 180
yref = [0.1150794920661702;    0.1203831471567715e-2
        0.1611562887407974;    0.3656156421249283e-3
        0.1708010885264404e-1; 0.4873531310307455e-2]';

yerr = norm(y(end,:) - yref)

% ----- Nested function -----
function rhs = chem_rhs(t,y)
%CHEM_RHS    Right-hand side of DAE

if y(2) < 0, error('Negative y(2) in DAE function. '), end

k1 = 18.7; k2 = 0.58; k3 = 0.09; k4 = 0.42;
K = 34.4; k1A = 3.3; pCO2 = 0.9; H = 737;

r1 = k1*(y(1)^4)*sqrt(y(2));
r2 = k2*y(3)*y(4);
r3 = k2*y(1)*y(5)/K;
r4 = k3*y(1)*(y(4)^2);
r5 = k4*(y(6)^2)*sqrt(y(2));
Fin = k1A*(pCO2/H - y(2));

rhs = [-2*r1 + r2 - r3 - r4;
       -0.5*r1 - r4 - 0.5*r5 + Fin;
       r1 - r2 + r3;
       -r2 + r3 - 2*r4;
       r2 - r3 + r5;
       Ks*y(1)*y(4)-y(6)];
end

end

```

where  $F$  is a given nonlinear function and suitable initial conditions are supplied. A separate function `decic` is available to compute consistent initial conditions for these problems. The reference [149] describes `ode15i` and `decic` and gives examples of their use.

The ODE solvers offer other features that you may find useful. Type `help odeset` to see the full range of properties that can be controlled through the `options` structure. The function `odeget` extracts property values from the `options` structure. The MATLAB ODE solvers are well documented and are supported by a rich variety of example files, some of which we list below. In each case, `help filename` gives an informative description of the file, `type filename` lists the contents of the file, and typing `filename` runs a demonstration. The examples can also be accessed via the `odeexamples` function.

`burgersode`, `rigidode`: nonstiff ODEs.

`brussode`, `hb1ode`, `kneode`, `vdode`: stiff ODEs.

`ballode`: event location problem.

`orbitode`: problem involving event location and the use of an output function (`odephas2`) to process the solution as the integration proceeds.

`fem1ode`, `fem2ode`, `batonode`: ODEs with mass matrices.

`amp1dae`, `hb1dae`, `ihb1dae`, `iburgersode`: DAEs.

#### STIFFNESS

Although `ode15s` has a higher computational cost per subinterval than `ode45`, we saw that `ode15s` was much the more efficient integrator in the Robertson stiff ODE example. Stiffness concerns the stability of numerical methods—the way that errors propagate in time—and is a well-developed field with a rich set of theoretical results. But most researchers agree that it is more useful to illustrate stiffness through practical examples than to attempt a rigorous definition. Indeed, Gear and Skeel [50] point out that Gear—who pioneered the backward differentiation formulas used, in a slightly modified form, by `ode15s`—was motivated by a concrete challenge. In 1966 he was shown a small ODE system from chemical kinetics and told “You people will never be able to handle these types of problems with your digital computers.”

### 12.3. Boundary-Value Problems

The function `bvp4c` uses a collocation method to solve systems of ODEs in two-point boundary-value form. These systems may be written

$$\frac{d}{dx}y(x) = f(x, y(x), p), \quad g(y(a), y(b), p) = 0.$$

Here, as for the initial-value problem in the previous section,  $y(x)$  is an unknown  $m$ -vector and  $f$  is a given function of  $x$  and  $y$  that also produces an  $m$ -vector. The vector  $p$ , which may be absent, is an unknown vector of parameters to be determined. The

solution is required over the range  $a \leq x \leq b$  and the given function  $g$  specifies the boundary conditions. Note that the independent variable was labeled  $t$  in the previous section and is now labeled  $x$ . This is consistent with the MATLAB documentation and reflects the fact that two-point boundary-value problems (BVPs) usually arise over an interval of space rather than time. Generally, BVPs are more computationally challenging than initial-value problems. They may have no solution, and it is common for more than one solution to exist. For these reasons, `bvp4c` requires an initial guess to be supplied for the solution. The initial guess and the final solution are stored in structures. We introduce `bvp4c` through a simple example before giving more details.

A scalar BVP describing the cross-sectional shape of a water droplet on a flat surface is given by (see [141])

$$\frac{d^2}{dx^2}h(x) + (1 - h(x)) \left( 1 + \left( \frac{d}{dx}h(x) \right)^2 \right)^{3/2} = 0, \quad h(-1) = 0, \quad h(1) = 0.$$

Here,  $h(x)$  measures the height of the droplet at point  $x$ . We set  $y_1(x) = h(x)$  and  $y_2(x) = dh(x)/dx$  and rewrite the equation as a system of two first-order equations:

$$\begin{aligned} \frac{d}{dx}y_1(x) &= y_2(x), \\ \frac{d}{dx}y_2(x) &= (y_1(x) - 1) (1 + y_2(x)^2)^{3/2}. \end{aligned}$$

This system is represented by the function

```
function yprime = drop(x,y)
%DROP    ODE/BVP water droplet example.
%   prime = DROP(x,y) evaluates derivative.

yprime = [y(2); (y(1)-1)*((1+y(2)^2)^(3/2))];
```

The boundary conditions are specified via a residual function. This function returns zero when evaluated at the boundary values. Our boundary conditions  $y_1(-1) = y_1(1) = 0$  can be encoded in the following function:

```
function res = dropbc(ya,yb)
%DROPBC    ODE/BVP water droplet boundary conditions.
%   res = DROPBC(ya,yb) evaluates residual.

res = [ya(1); yb(1)];
```

As an initial guess for the solution, we use  $y_1(x) = \sqrt{1 - x^2}$  and  $y_2(x) = -x/(0.1 + \sqrt{1 - x^2})$ . This information is set up by the function `dropinit`:

```
function yinit = dropinit(x)
%DROPINIT    ODE/BVP water droplet initial guess.
%   yinit = DROPINIT(x) evaluates initial guess at x.

yinit = [sqrt(1-x^2); -x/(0.1+sqrt(1+x^2))];
```

The following code solves the BVP and produces Figure 12.13:



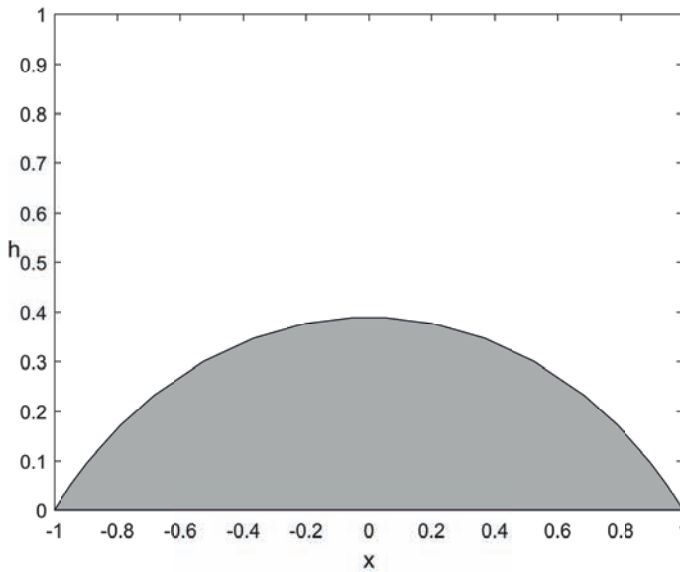


Figure 12.13. *Water droplet BVP solved by bvp4c.*

```
solinit = bvpinit(linspace(-1,1,20),@dropinit);
sol = bvp4c(@drop,@dropbc,solinit);
fill(sol.x,sol.y(1,:),[0.7 0.7 0.7])
axis([-1 1 0 1])
xlabel('x','FontSize',12)
ylabel('h','Rotation',0,'FontSize',12)
```

Here, the call to `bvpinit` sets up the structure `solinit`, which contains the data produced by evaluating `dropinit` at 20 equally spaced values between  $-1$  and  $1$ . We then call `bvp4c`, which returns the solution in the structure `sol`. The `fill` command fills the curve that the solution makes in the  $(x, y_1)$ -plane.

In general, `bvp4c` can be called in the form

```
sol = bvp4c(@odefun,@bcfun,solinit,options);
```

Here, `odefun` evaluates the differential equations and `bcfun` gives the residual for the boundary conditions. The function `odefun` has the general form

```
yprime = odefun(x,y)
```

and `bcfun` has the general form

```
res = bcfun(ya,yb)
```

Both functions must return column vectors. The initial guess structure `solinit` has two required fields: `solinit.x` contains the  $x$ -values at which the initial guess is supplied, ordered from left to right with `solinit.x(1)` and `solinit.x(end)` giving  $a$  and  $b$ , respectively. Correspondingly, `solinit.y(:,i)` gives the initial guess for the solution at the point `solinit.x(i)`. This structure also allows a guess for a vector of unknown parameters to be specified, as we will see in function `skiprun`

below. The helper function `bvpinit` can be used to create the initial guess structure, as in the example above. The remaining arguments for `bvp4c` are optional. The `options` structure allows various properties of the collocation algorithm to be altered from their default values, including the error tolerances and the maximum number of meshpoints allowed. The function `bvpset`, which is similar to `odeset`, can be used to create the required structure (see `doc bvpset` for details).

The output argument `sol` is a structure that contains the numerical solution. The field `sol.x` gives the array of  $x$ -values at which the solution has been computed. (These points are chosen automatically by `bvp4c`.) The approximate solution at `sol.x(i)` is given by `sol.y(:,i)`. Similarly, an approximate value of the first derivative of the solution at `sol.x(i)` is given by `sol.yp(:,i)`. Unlike for the ODE solvers, the only form in which output from `bvp4c` can be obtained is a structure (and the same is true for the delay-differential equations (DDE) and PDE solvers described in the next two sections).

Note that the structures `solinit` and `sol` above can be given any names, but the field names `x`, `y`, and `yp` must be used.

The function `deval`, described on p. 199, may be used to provide the solution and its derivative at general  $x$ -values.

Our next example treats a differential equation depending on a parameter and emphasizes that nonlinear BVPs can have nonunique solutions. The equation

$$\frac{d^2}{dx^2}\theta(x) + \lambda \sin\theta(x) \cos\theta(x) = 0, \quad \theta(-1) = 0, \theta(1) = 0,$$

arises in liquid crystal theory [114]. Here,  $\theta(x)$  quantifies the average local molecular orientation, and the constant parameter  $\lambda > 0$  is a measure of an applied magnetic field. If  $\lambda$  is small then the only solution to this problem is the trivial one,  $\theta(x) \equiv 0$ . However, for  $\lambda > \pi^2/4 \approx 2.467$  a solution with  $\theta(x) > 0$  for  $-1 < x < 1$  exists, and  $-\theta(x)$  is then also a solution. (Physically, a distorted state of the material may arise if the magnetic field is sufficiently strong.) For the positive solution the midpoint value,  $\theta(0)$ , increases monotonically with  $\lambda$  and approaches  $\pi/2$  as  $\lambda$  tends to infinity. Writing  $y_1(x) = \theta(x)$  and  $y_2(x) = d\theta(x)/dx$  the ODE becomes

$$\begin{aligned} \frac{d}{dx}y_1(x) &= y_2(x), \\ \frac{d}{dx}y_2(x) &= -\lambda \sin y_1(x) \cos y_1(x). \end{aligned}$$

The function `lcrun` in Listing 12.7 solves the BVP for parameter values  $\lambda = 2.4, 2.5, 3,$  and  $10$ , producing Figure 12.14. In this example, as for some of the ODE problems in the previous section, we have written a function `lcrun` that has no input or output arguments and created `lc` as a nested function and `lcbc` and `lcinit` as local functions of `lcrun`. This allows us to solve the BVP with a single function. The nested function `lc` evaluates the right-hand side of the ODE. The boundary conditions, which are the same as those in the previous example, are coded in `lcbc`. Note that making `lc` a nested function ensures that the parameter `lambda` is known to `lc` when it is called by `bvp4c`. As an initial guess for  $\lambda = 10$ , we use  $y_1(x) = \sin((x+1)\pi/2)$  and  $y_2(x) = \pi \cos((x+1)\pi/2)/2$ , which is set up by `lcinit`. For the remaining three  $\lambda$ -values we use the solution for the previous  $\lambda$  as the starting guess for the next; this is known as continuation in the parameter  $\lambda$ , and it is a valuable (perhaps necessary) technique when solving hard problems. From Figure 12.14 we see that `bvp4c` has found

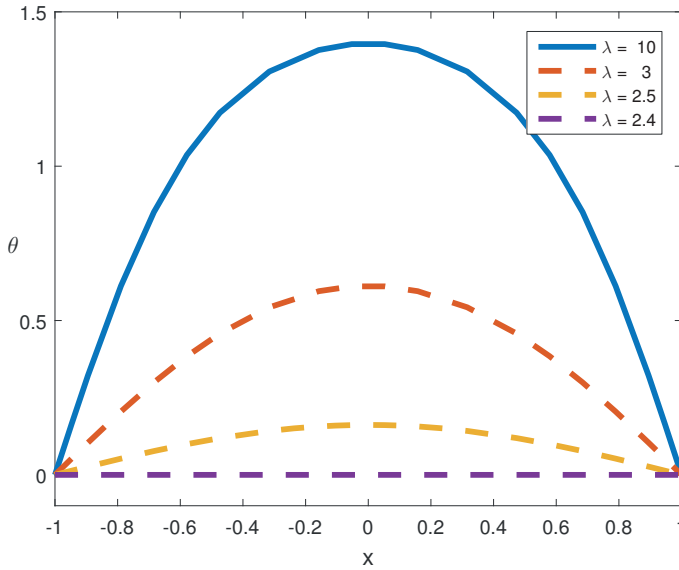


Figure 12.14. *Liquid crystal BVP solved by bvp4c.*

the nontrivial positive solution for each of the three `lambda`-values beyond  $\pi^2/4$ . For continuation to work in this example we need to take the  $\lambda$ -values in decreasing order; the increasing order leads to the trivial solution each time.

Our final example involves the equation

$$\frac{d^2}{dx^2}y(x) + \mu y(x) = 0,$$

with boundary conditions

$$y(0) = 0, \quad \left(\frac{d}{dx}y(x)\right)_{x=0} = 1, \quad \left(y(x) + \frac{d}{dx}y(x)\right)_{x=1} = 0.$$

This equation models the displacement of a skipping rope that is fixed at  $x = 0$ , has elastic support at  $x = 1$ , and rotates with uniform angular velocity about its equilibrium position along the  $x$ -axis [85, Sec. 5.2]. This BVP is an *eigenvalue problem*: we must find a value of the parameter  $\mu$  for which a solution exists. (We can regard the two conditions at  $x = 0$  as defining an initial-value problem; we must then find a value of  $\mu$  for which the solution matches the boundary condition at  $x = 1$ .) We can use `bvp4c` to solve this eigenvalue problem if we supply a guess for the unknown parameter  $\mu$  as well as a guess for the corresponding solution  $y(x)$ . This is done in the function `skiprun` in Listing 12.8. As a first-order system, the differential equation may be written

$$\begin{aligned} \frac{d}{dx}y_1(x) &= y_2(x), \\ \frac{d}{dx}y_2(x) &= -\mu y_1(x). \end{aligned}$$

This system is encoded in the local function `skip` and the boundary conditions in `skipbc`. Our initial guess for the solution is  $y_1(x) = \sin x$ ,  $y_2(x) = \cos x$ , specified in

Listing 12.7. *Function lcrun.*

```

function lcrun
%LCRUN    Liquid crystal BVP.
%   Solves the liquid crystal BVP for four different lambda values.

lambda_vals = [2.4, 2.5, 3, 10];
lambda_vals = lambda_vals(end:-1:1); % Necessary order for continuation.

solinit = bvpinit(linspace(-1,1,20),@lcinit);
lambda = lambda_vals(1); sola = bvp4c(@lc,@lcbc,solinit);
lambda = lambda_vals(2); solb = bvp4c(@lc,@lcbc,sola);
lambda = lambda_vals(3); solc = bvp4c(@lc,@lcbc,solb);
lambda = lambda_vals(4); sold = bvp4c(@lc,@lcbc,solc);
plot(sola.x,sola.y(1,:), '- ', 'LineWidth',3), hold on
plot(solb.x,solb.y(1,:), '-- ', 'LineWidth',3)
plot(solc.x,solc.y(1,:), '-- ', 'LineWidth',3)
plot(sold.x,sold.y(1,:), '-- ', 'LineWidth',3), hold off
legend([repmat('\lambda = ',4,1) num2str(lambda_vals')])
xlabel('x', 'FontSize',12)
ylabel('\theta', 'Rotation',0, 'FontSize',12)
ylim([-0.1 1.5])

    function yprime = lc(x,y)
    %LC    ODE/BVP liquid crystal system.
    yprime = [y(2); -lambda*sin(y(1))*cos(y(1))];
    end

end

function res = lcbc(ya,yb)
%LCBC    ODE/BVP liquid crystal boundary conditions.
res = [ya(1); yb(1)];
end

function yinit = lcinit(x)
%LCINIT    ODE/BVP liquid crystal initial guess.
yinit = [sin(0.5*(x+1)*pi); 0.5*pi*cos(0.5*(x+1)*pi)];
end

```

Listing 12.8. *Function skiprun.*

```

function sol = skiprun
%SKIPRUN    Skipping rope BVP/eigenvalue example.

solinit = bvpinit(linspace(0,1,10),@skipinit,5);
sol = bvp4c(@skip,@skipbc,solinit);
plot(sol.x,sol.y(1,:),'-', sol.x,sol.yp(1,:),'--', 'LineWidth',3)
xlabel('x','FontSize',12)
legend('y_1','y_2')

% ----- Local functions -----
function yprime = skip(x,y,mu)
%SKIP    ODE/BVP skipping rope example.
% yprime = SKIP(x,y,mu) evaluates derivative.
yprime = [y(2); -mu*y(1)];

function res = skipbc(ya,yb,mu)
%SKIPBC    ODE/BVP skipping rope boundary conditions.
% res = skipbc(ya,yb,mu) evaluates residual.
res = [ya(1); ya(2)-1; yb(1)+yb(2)];

function yinit = skipinit(x)
%SKIPINIT    ODE/BVP skipping rope initial guess.
% yinit = SKIPINIT(x) evaluates initial guess at X.
yinit = [sin(x); cos(x)];

```

`skipinit`. Note that the input argument 5 is added in the call to `bvpinit`. This is our guess for  $\mu$ , and it is stored in the `parameters` field of the structure `solinit` and hence passed to `bvp4c`. Figure 12.15 shows the solution computed by `bvp4c`. The computed value for  $\mu$  is returned in the `parameters` field of the structure `sol`. We have

```

>> sol = skiprun
sol =
      x: [1×10 double]
      y: [2×10 double]
     yp: [2×10 double]
 solver: 'bvp4c'
parameters: 4.1159e+000

```

It is known that this BVP has eigenvalues given by  $\mu = \gamma^2$ , where  $\gamma$  is a solution of  $\tan \gamma + \gamma = 0$ . Using `fzero` to locate a  $\gamma$ -value near 2, we can check the accuracy of the computed  $\mu$  as follows:

```

>> gam = fzero(@(x)tan(x)+x,2); mu = gam^2;
>> error = abs(sol.parameters - mu)
error =
    2.9343e-005

```

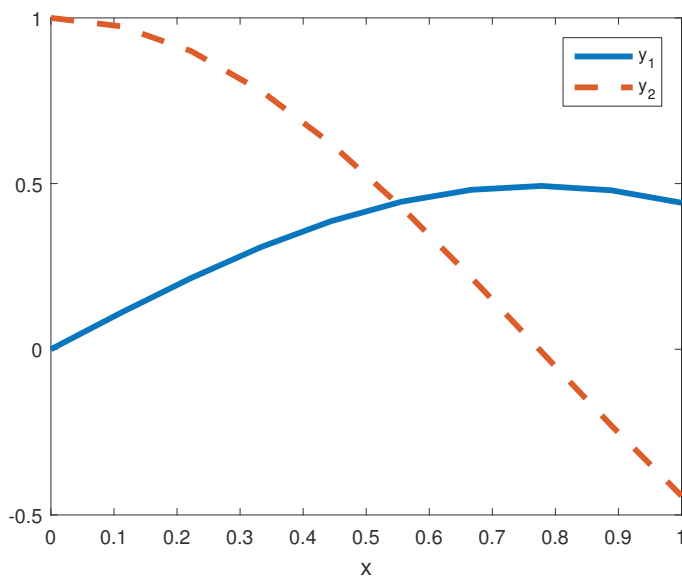


Figure 12.15. *Skipping rope eigenvalue BVP solved by `bvp4c`.*

The tutorial [98] gives a range of examples that illustrate the versatility of `bvp4c`. The examples deal with a number of issues, including

- changing the error tolerances,
- evaluating the solution at any point in the range  $[a, b]$ ,
- choosing appropriate initial guesses by continuation,
- dealing with singularities,
- solving problems with periodic boundary conditions,
- solving problems over an infinite interval,
- solving multipoint BVPs (where non-endpoint conditions are specified for the solution).

Further information can also be obtained from the help for the functions `bvp4c`, `bvpget`, `bvpinit`, `bvpval`, `bvpset`; from [100]; and from the following example files:

**twobvp:** solves a BVP that has exactly two solutions;

**mat4bvp:** finds the fourth eigenvalue of Mathieu's equation;

**shockbvp:** solves a difficult BVP with a shock layer;

**threebvp:** solves a three-point BVP.

The function `bvp4c` can also solve a class of boundary-value problems over  $0 \leq x \leq b$  that have a singularity at  $x = 0$ ; for details see the online help and [150].

Finally, we note that MATLAB has another function `bvp5c` that has the same syntax as `bvp4c` but differs from it in that it controls the error in the solution rather than the residual of the differential equation.

## 12.4. Delay-Differential Equations

The function `dde23` solves systems of DDEs of the form

$$\frac{d}{dt}y(t) = f(t, y(t), y(t - \tau_1), y(t - \tau_2), \dots, y(t - \tau_k))$$

for  $t > t_0$ , where  $\tau_1, \tau_2, \dots, \tau_k$  are positive constants known as delays or lags. DDEs differ from ODEs in that the right-hand side function,  $f$ , depends on the solution value at earlier points  $t - \tau_i$  as well as at the current point  $t$ . Instead of an initial condition, an *initial function*,  $S(t)$ , must be specified such that  $y(t) = S(t)$  for  $t \leq t_0$ . Using `dde23` is similar to using one of the MATLAB ODE solvers in the mode where the solution is returned as a structure.

A DDE system for predator–prey populations is [119, (3.11)]

$$\begin{aligned} \frac{d}{dt}y_1(t) &= y_1(t) \left( 2 \left( 1 - \frac{y_1(t)}{50} \right) - \frac{y_2(t)}{y_1(t) + 40} \right) - h, \\ \frac{d}{dt}y_2(t) &= y_2(t) \left( -3 + \frac{6y_1(t - \tau)}{y_1(t - \tau) + 40} \right). \end{aligned}$$

Here,  $y_1(t)$  and  $y_2(t)$  denote the densities of the prey and predator populations, respectively, at time  $t$ . The delay parameter  $\tau$  accounts for either (a) a gestation period in the predators or (b) a reaction time in the predators, and the parameter  $h$  represents a harvesting rate for the prey. We will take  $\tau = 9$ , a constant initial function  $S(t) = [35, 10]^T$ , and try two different harvesting rates,  $h = 10$  and  $h = 15$ . The analysis and computation in [119] show that for  $h = 15$  the solution should evolve to the equilibrium state  $y_1(t) \equiv 40$ ,  $y_2(t) \equiv 2$ , whereas for  $h = 10$  a limit cycle is present.

This DDE is solved for  $h = 10$  and  $h = 15$  by function `harvest` in Listing 12.9, which produces the pictures in Figure 12.16. The first two input arguments for `dde23` are `@f`, the handle of the function that evaluates the right-hand side of the DDE, and `tau`, which specifies the size of the delay. In general, `tau` is a  $k$ -vector when there are  $k$  lags. Next, `ic = [35; 10]` specifies the initial function to be a constant. The vector `tspan` gives the range for the independent variable. In the nested function `f`, `Z` is a column vector that represents  $y(t - \tau)$ .

As for the ODE solvers, the output argument `sol` is a structure, with the field `sol.x` giving an array of  $t$ -values for which the field `sol.y` is a corresponding array of solution values. A third field, `sol.yprime`, provides the first derivative of the solution. Figure 12.16 confirms the expected difference in behavior between  $h = 10$  and  $h = 15$ .

A general call to `dde23` takes the form

```
sol = dde23(@ddefun,delays,history,tspan,options);
```

The function `ddefun` has input arguments `(t, y, Z)` and returns a column vector giving the right-hand side of the DDE. The input argument `Z` is an array whose  $j$ th column corresponds to  $y(t - \tau_j)$ . The second input argument to `dde23`, `delays`, is a row vector that defines the delays; so `delays(j)` is  $\tau_j$ . Any number of distinct, positive delays may be used. (It is possible to use `dde23` to solve an ODE system—that is, a DDE with no delays. In this case it is best to pass the empty array `[]` as the delay argument.) The function `history` has input argument `t` and returns the value  $S(t)$  as a column vector. As we saw in the example above, in the commonly arising case where the initial function is constant it is permissible to supply a vector as the `history` argument, rather than a function name. The input argument `tspan` is used

Listing 12.9. *Function harvest.*

```

function harvest
%HARVEST    Predator-prey model with delay and harvesting.

tau = 9;
ic = [35; 10];
tspan = [0 250];

h = 10;
sol = dde23(@f,tau,ic,tspan);
subplot(2,1,1)
plot(sol.x,sol.y(1,:), 'r-', sol.x,sol.y(2,:), 'b--', 'LineWidth',2)
legend('y_1','y_2','Location','East')
title('h = 10'), xlabel t, ylabel('y','Rotation',0)

h = 15;
sol = dde23(@f,tau,ic,tspan);
subplot(2,1,2)
plot(sol.x,sol.y(1,:), 'r-', sol.x,sol.y(2,:), 'b--', 'LineWidth',2)
legend('y_1','y_2','Location','East')
title('h = 15'), xlabel t, ylabel('y','Rotation',0)

function v = f(t,y,Z)
%F    Harvest differential equation.
v = [y(1)*(2*(1-y(1)/50) - y(2)/(y(1)+40)) - h
     y(2)*(-3 + 6*Z(1)/(Z(1)+40))];
end

end

```



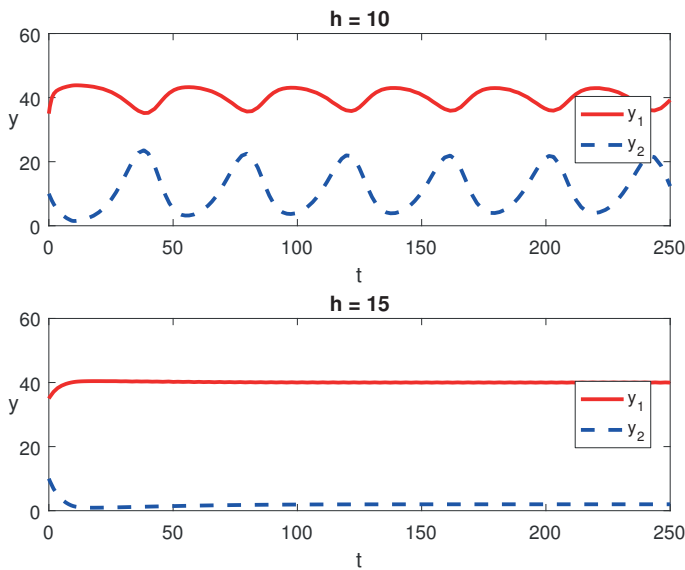


Figure 12.16. *Predator–prey model with delay and harvesting.*

to specify the points at which a solution is required. It has similar functionality to the corresponding argument of the ODE solvers. Similarly, the `options` argument, which may be set by calls to `dde23`, allows error tolerances and event location requirements to be specified, as in the ODE case. Additionally, points where low-order derivatives of the solution are known to have discontinuities may be specified via `options`. Such information may improve the accuracy and efficiency of `dde23`.

As in the ODE case described on p. 198, the output argument `sol` may be fed into the function `deval` in order to evaluate the solution and its derivative at specified  $t$ -values.

Our second DDE system example is

$$\begin{aligned}\frac{d}{dt}y_1(t) &= -y_1(t) + 2 \tanh(y_2(t - \tau_2)), \\ \frac{d}{dt}y_2(t) &= -y_2(t) - 1.5 \tanh(y_1(t - \tau_1)),\end{aligned}$$

which models a network of two neurons. Using an initial function of the form  $S(t) = 0.1(\sin(t/10), \cos(t/10))^T$ , we will examine two pairs of delays: first  $\tau_1 = 0.2$  and  $\tau_2 = 0.5$  and then  $\tau_1 = 0.325$  and  $\tau_2 = 0.525$ . In the first case, the system should damp down to zero and in the second case a limit cycle should be apparent (see [181] for details). This DDE is solved by the script `neural` in Listing 12.10, which produces Figure 12.17. Notice that `neural` is a script containing local functions, in contrast to the earlier codes in this chapter.

The tutorial [99] discusses `dde23` more comprehensively and provides downloadable code that illustrates its use, including examples that require event location and discontinuity handling. The theory and algorithmics behind `dde23` are covered in [152] and [155].

MATLAB also has two other functions for solving DDEs: `ddesd` solves equations

Listing 12.10. *Script neural.*

```

%NEURAL    Neural network model with delays.

tspan = [0 40];
sol = dde23(@f,[0.2,0.5],@history,tspan);
subplot(2,2,1)
plot(sol.x,sol.y(1,:), 'r-', sol.x,sol.y(2,:), 'b--', 'LineWidth',2)
legend('y_1','y_2')
title('\tau_1 = 0.2, \tau_2 = 0.5','FontSize',12)
xlabel t, ylabel('y','Rotation',0), ylim([-0.2,0.2])

subplot(2,2,3)
plot(sol.y(1,:),sol.y(2,:), 'r-')
xlabel y_1, ylabel('y_2','Rotation',0)
xlim([-0.2,0.2]), ylim([-0.1,0.1])

sol = dde23(@f,[0.325,0.525],@history,tspan);
subplot(2,2,2)
plot(sol.x,sol.y(1,:), 'r-', sol.x,sol.y(2,:), 'b--', 'LineWidth',2)
legend('y_1','y_2')
title('\tau_1 = 0.325, \tau_2 = 0.525','FontSize',12)
xlabel t, ylabel('y','Rotation',0), ylim([-0.2,0.2])

subplot(2,2,4)
plot(sol.y(1,:),sol.y(2,:), 'r-')
xlabel y_1, ylabel('y_2','Rotation',0)
xlim([-0.2,0.2]), ylim([-0.1,0.1])

function v = f(t,y,Z)
%F    Neural network differential equation.
ylag1 = Z(:,1);
ylag2 = Z(:,2);
v = [-y(1) + 2*tanh(ylag2(2))
     -y(2) - 1.5*tanh(ylag1(1))];
end

function v = history(t)
%HISTORY    Initial function for neural network model
v = 0.1*[sin(t/10);cos(t/10)];
end

```

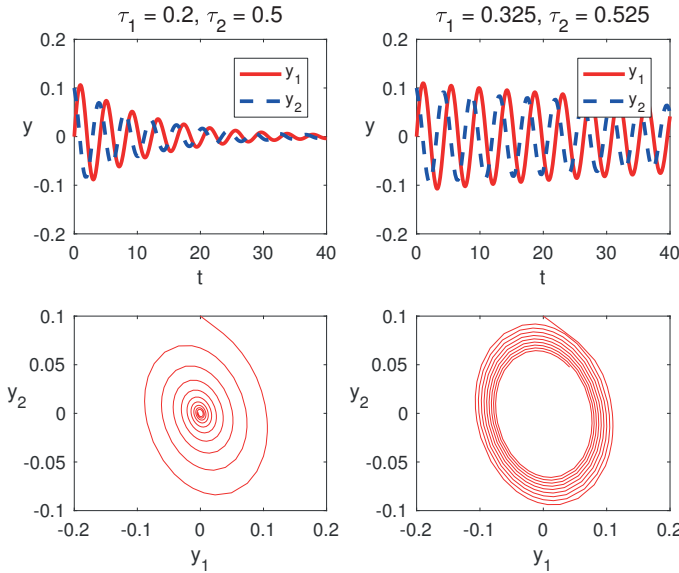


Figure 12.17. *Neural network DDE.*

with general delays and `ddensd` solves DDEs of neutral type, which involve delays in the derivative  $y'$  as well as in  $y$ .

### 12.5. Partial Differential Equations

The `pdepe` function solves a class of parabolic/elliptic PDE systems. These systems involve a vector-valued unknown function  $u$  that depends on a scalar space variable,  $x$ , and a scalar time variable,  $t$ . The general class to which `pdepe` applies has the form

$$c \left( x, t, u, \frac{\partial u}{\partial x} \right) \frac{\partial u}{\partial t} = x^{-m} \frac{\partial}{\partial x} \left( x^m f \left( x, t, u, \frac{\partial u}{\partial x} \right) \right) + s \left( x, t, u, \frac{\partial u}{\partial x} \right),$$

where  $a \leq x \leq b$  and  $t_0 \leq t \leq t_f$ . The integer  $m$  can be 0, 1, or 2, corresponding to slab, cylindrical, and spherical symmetry, respectively. The function  $c$  is a diagonal matrix and the flux and source functions  $f$  and  $s$  are vector valued. Initial and boundary conditions must be supplied in the following form. For  $a \leq x \leq b$  and  $t = t_0$  the solution must satisfy  $u(x, t_0) = u_0(x)$  for a specified function  $u_0$ . For  $x = a$  and  $t_0 \leq t \leq t_f$  the solution must satisfy

$$p_a(x, t, u) + q_a(x, t) f \left( x, t, u, \frac{\partial u}{\partial x} \right) = 0$$

for specified functions  $p_a$  and  $q_a$ . Similarly, for  $x = b$  and  $t_0 \leq t \leq t_f$ ,

$$p_b(x, t, u) + q_b(x, t) f \left( x, t, u, \frac{\partial u}{\partial x} \right) = 0$$

must hold for specified functions  $p_b$  and  $q_b$ . Certain other restrictions are placed on the class of problems that can be solved by `pdepe`; see `doc pdepe` for details.

A call to `pdepe` has the general form

```
sol = pdepe(m,@pdefun,@pdeic,@pdebc,xmesh,tspan,options);
```

which is similar to the syntax for `bvp4c`. The input argument `m` can take the values 0, 1, or 2, as described above. The function `pdefun` has the form

```
function [c,f,s] = pdefun(x,t,u,DuDx)
```

It accepts the space and time variables together with vectors `u` and `DuDx` that approximate the solution  $u$  and the partial derivative  $\partial u/\partial x$ , and it returns vectors containing the diagonal of the matrix  $c$  and the flux and source functions  $f$  and  $s$ . Initial conditions are encoded in the function `pdeic`, which takes the form

```
function u0 = pdeic(x)
```

The function `pdebc` of the form

```
function [pa,qa,pb,qb] = pdebc(xa,ua,xb,ub,t)
```

evaluates  $p_a$ ,  $q_a$ ,  $p_b$ , and  $q_b$  for the boundary conditions at  $x_a = a$  and  $x_b = b$ . The vector `xmesh` in the argument list of `pdepe` is a set of points in  $[a, b]$  with `xmesh(1) = a` and `xmesh(end) = b`, ordered so that `xmesh(i) < xmesh(i+1)`. This defines the  $x$ -values at which the numerical solution is computed. The algorithm uses a second-order spatial discretization method based on the `xmesh`-values. Hence the choice of `xmesh` has a strong influence on the accuracy and cost of the numerical solution. Closely spaced `xmesh` points should be used in regions where the solution is likely to vary rapidly with respect to  $x$ . The vector `tspan` specifies the time points in  $[t_0, t_f]$  where the solution is to be returned, with `tspan(1) = t_0`, `tspan(end) = t_f`, and `tspan(i) < tspan(i+1)`. The time integration in `pdepe` is performed by `ode15s` and the actual timestep values are chosen dynamically—the `tspan` points simply determine where the solution is returned and have little impact on the cost or accuracy. The default properties of `ode15s` can be overridden via the optional input argument `options`, which can be created with the `odeset` function (see Section 12.2.1). Altering the defaults is not usually necessary so we do not discuss this further.

The output argument `sol` is a three-dimensional array such that `sol(j,k,i)` is the approximation to the  $i$ th component of  $u$  at the point  $t = \text{tspan}(j)$ ,  $x = \text{xmesh}(k)$ . A postprocessing function `pdeval` is available for computing  $u$  and  $\partial u/\partial x$  at points that are not in `xmesh`.

To illustrate the use of `pdepe`, we begin with the Black–Scholes PDE, famous for modeling derivative prices in financial mathematics. In transformed and dimensionless form [183, Sec. 5.4], using parameter values from [133, Chap. 13], we have

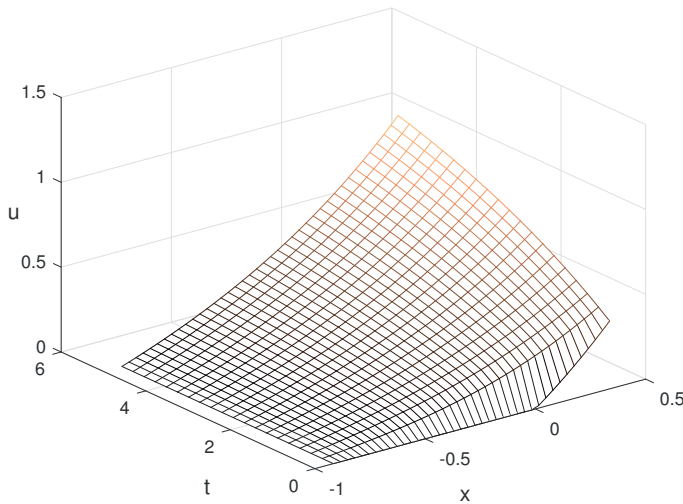
$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + (k-1)\frac{\partial u}{\partial x} - ku, \quad a \leq x \leq b, \quad t_0 \leq t \leq t_f,$$

where  $k = r/(\sigma^2/2)$ ,  $r = 0.065$ ,  $\sigma = 0.8$ ,  $a = \log(2/5)$ ,  $b = \log(7/5)$ ,  $t_0 = 0$ ,  $t_f = 5$ , with initial condition

$$u(x, 0) = \max(\exp(x) - 1, 0)$$

and boundary conditions

$$u(a, t) = 0, \quad u(b, t) = \frac{7 - 5\exp(-kt)}{5}.$$

Figure 12.18. *Black-Scholes solution with pdepe.*

This is of the general form allowed by `pdepe` with  $m = 0$  and

$$c(x, t, u) = 1, \quad f\left(x, t, u, \frac{\partial u}{\partial x}\right) = \frac{\partial u}{\partial x}, \quad s\left(x, t, u, \frac{\partial u}{\partial x}\right) = (k - 1)\frac{\partial u}{\partial x} - ku.$$

At  $x = a$  the boundary conditions have  $p(x, t, u) = u$  and  $q(x, t, u) = 0$ , and at  $x = b$  they have  $p(x, t, u) = u - (7 - 5 \exp(-kt))/5$  and  $q(x, t, u) = 0$ . The function `bs` in Listing 12.11 implements the problem. Here, we have used `linspace` to generate 40 equally spaced  $x$ -values between  $a$  and  $b$  for the spatial mesh and 20 equally spaced  $t$ -values between  $t_0$  and  $t_f$  for the output times. The nested function `bspde` defines the PDE in terms of `c`, `f`, and `s` and `bsic` specifies the initial condition. Similarly, in nested function `bsbc` the boundary conditions at  $x = a$  and  $x = b$  are returned in `pa`, `qa`, `pb`, and `qb`. We use the 3D plotting function `mesh` to display the solution. Figure 12.18 shows the resulting picture.

Next, we look at a system of two reaction–diffusion equations of a type that arises in mathematical biology [89, Chap. 12]:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{1}{2} \frac{\partial^2 u}{\partial x^2} + \frac{1}{1 + v^2}, \\ \frac{\partial v}{\partial t} &= \frac{1}{2} \frac{\partial^2 v}{\partial x^2} + \frac{1}{1 + u^2} \end{aligned}$$

for  $0 \leq x \leq 1$  and  $0 \leq t \leq 0.2$ . Our initial conditions are

$$u(x, 0) = 1 + \frac{1}{2} \cos(2\pi x), \quad v(x, 0) = 1 - \frac{1}{2} \cos(2\pi x),$$

and our boundary conditions are

$$\frac{\partial u}{\partial x}(0, t) = \frac{\partial u}{\partial x}(1, t) = \frac{\partial v}{\partial x}(0, t) = \frac{\partial v}{\partial x}(1, t) = 0.$$

Listing 12.11. *Function bs.*

```

function bs
%BS    Black-Scholes PDE.
%    Solves the transformed Black-Scholes equation.

m = 0;
r = 0.065;
sigma = 0.8;
k = r/(0.5*sigma^2);
a = log(2/5);
b = log(7/5);
t0 = 0;
tf = 5;

xmesh = linspace(a,b,40);
tspan = linspace(t0,tf,20);

sol = pdepe(m,@bspde,@bsic,@bsbc,xmesh,tspan);
u = sol(:,:,1);

mesh(xmesh,tspan,u)
xlabel('x','FontSize',12)
ylabel('t','FontSize',12)
zlabel('u','FontSize',12,'Rotation',0)
colormap copper

    function [c,f,s] = bspde(x,t,u,DuDx)
    %BSPDE    Black-Scholes PDE.
    c = 1;
    f = DuDx;
    s = (k-1)*DuDx-k*u;
    end

    function u0 = bsic(x)
    %BSIC    Initial condition at t = t0.
    u0 = max(exp(x)-1,0);
    end

    function [pa,qa,pb,qb] = bsbc(xa,ua,xb,ub,t)
    %BSBC    Boundary conditions at x = a and x = b.
    pa = ua;
    qa = 0;
    pb = ub - (7 - 5*exp(-k*t))/5;
    qb = 0;
    end

end

```

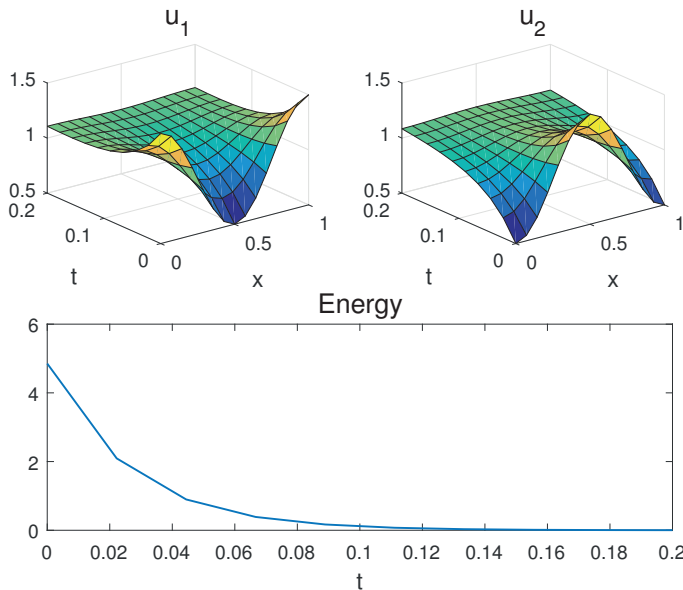


Figure 12.19. Reaction-diffusion system solution with `pdepe`.

To put this into the framework of `pdepe` we write  $(u, v)$  as  $(u_1, u_2)$  and express the PDE as

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \frac{\partial}{\partial t} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x} \begin{bmatrix} \frac{1}{2} \partial u_1 / \partial x \\ \frac{1}{2} \partial u_2 / \partial x \end{bmatrix} + \begin{bmatrix} 1 / (1 + u_2^2) \\ 1 / (1 + u_1^2) \end{bmatrix}.$$

The function `mbio1` in Listing 12.12 solves the PDE system. Note that the output arguments `c`, `f`, `s`, `pa`, `qa`, `pb`, and `qb` in the local functions `mbpde` and `mbbc` are 2-by-1 arrays, because there are two PDEs in the system. The solutions plotted with `surf` can be seen in the upper part of Figure 12.19. It follows from [89, Ex. 12.5] that the energy

$$E(t) = \frac{1}{2} \int_0^1 \left[ \left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial x} \right)^2 \right] dx$$

decays exponentially to zero as  $t \rightarrow \infty$ . To verify this fact numerically, we use simple finite differences and quadrature in `mbio1` to approximate the energy integral. (Alternatively, the function `pdeval` could be used to obtain approximations to  $\partial u / \partial x$  and  $\partial v / \partial x$ .) The resulting plot of  $E(t)$  is given in the lower part of Figure 12.19.

Further examples of `pdepe` in use can be found in `doc pdepe`. We note that `pdepe` is designed to solve a subclass of small systems of parabolic and elliptic PDEs to modest accuracy. If your PDE is not suitable for `pdepe` then the Partial Differential Equation Toolbox might be appropriate.

Listing 12.12. *Function* mbiol.

```

function mbiol
%MBIOL    Reaction-diffusion system from mathematical biology.
%    Solves the PDE and tests the energy decay condition.

m = 0;
xmesh = linspace(0,1,15);
tspan = linspace(0,0.2,10);
sol = pdepe(m,@mbpde,@mbic,@mbbc,xmesh,tspan);
u1 = sol(:,:,1);
u2 = sol(:,:,2);

subplot(221)
surf(xmesh,tspan,u1)
xlabel('x','FontSize',12)
ylabel('t','FontSize',12)
title('u_1','FontSize',14,'FontWeight','normal')

subplot(222)
surf(xmesh,tspan,u2)
xlabel('x','FontSize',12)
ylabel('t','FontSize',12)
title('u_2','FontSize',14,'FontWeight','normal')

% Estimate energy integral.
dx = xmesh(2) - xmesh(1); % Constant spacing.
energy = 0.5*sum( (diff(u1,1,2)).^2 + (diff(u2,1,2)).^2, 2)/dx;
subplot(212)
plot(tspan,energy,'LineWidth',1)
xlabel('t','FontSize',12)
title('Energy','FontSize',14,'FontWeight','normal')

% ----- Local functions -----
function [c,f,s] = mbpde(x,t,u,DuDx)
c = [1; 1];
f = DuDx/2;
s = [1/(1+u(2)^2); 1/(1+u(1)^2)];

function u0 = mbic(x);
u0 = [1+0.5*cos(2*pi*x); 1-0.5*cos(2*pi*x)];

function [pa,qa,pb,qb] = mbbc(xa,ua,xb,ub,t)
pa = [0; 0];
qa = [1; 1];
pb = [0; 0];
qb = [1; 1];

```



*Multidimensional integrals are another whole multidimensional bag of worms.*

— WILLIAM H. PRESS, SAUL A. TEUKOLSKY,  
WILLIAM T. VETTERLING, and BRIAN P. FLANNERY,  
*Numerical Recipes in FORTRAN* (1992)

*Perhaps the crudest way to evaluate  $\int_y^x f(u)du$   
is to plot the graph of  $f(u)$  on uniformly squared paper  
and then count the squares that lie inside the desired area.*

*This method gives numerical integration its other name:  
numerical quadrature.*

*Another way, suitable for chemists,  
is to plot the graph on paper of uniform density,  
cut out the area in question, and weigh it.*

— WILLIAM M. KAHAN, *Handheld Calculator Evaluates Integrals* (1980)

*The options vector is optional.*

— LAWRENCE F. SHAMPINE and MARK W. REICHELTL,  
*The MATLAB ODE Suite* (1997)

*Stiff equations are problems for which explicit methods don't work.*

— E. HAIRER AND G. WANNER, *Solving Ordinary Differential Equations II* (1996)

*Just about any BVP can be formulated for solution with `bvp4c`.*

— LAWRENCE F. SHAMPINE, JACEK KIERZENKA, and MARK W. REICHELTL,  
*Solving Boundary Value Problems for Ordinary  
Differential Equations in MATLAB with `bvp4c`* (2000)

# Chapter 13

## Input and Output

In this chapter we discuss how to obtain input from the user, how to display information on the screen, and how to read and write text files. Note that textual output can be captured into a file (perhaps for subsequent printing) using the `diary` command, as described on p. 32. How to print and save figures is discussed in Section 8.4.

### 13.1. User Input

User input can be obtained with the `input` function, which displays a prompt and waits for a user response:

```
>> x = input('Starting point: ')
Starting point: 0.5
x =
    0.5000
```

Here, the user has responded by typing “0.5”, which is assigned to `x`. The input is interpreted as a string when an argument `'s'` is appended:

```
>> mytitle = input('Title for plot: ','s')
Title for plot: Experiment 2
mytitle =
Experiment 2
```

The function `ginput` collects data via mouse clicks. The command

```
[x,y] = ginput(n)
```

returns in the vectors `x` and `y` the coordinates of the next `n` mouse clicks from the current figure window. Input can be terminated before the `n`th mouse click by pressing the return key. One use of `ginput` is to find the approximate location of points on a graph. For example, with Figure 8.10 (on p. 108) in the current figure window, you might type `[x,y] = ginput(1)` and click on one of the places where the curves intersect. As another example, the first two lines of `bezier_plot` in Listing 8.3 can be replaced by

```
axis([0 1 0 1])
[x,y] = ginput(4);
P = [x';y'];
```

Now the control points are determined by the user’s mouse clicks.

The `pause` command suspends execution until a key is pressed, while `pause(n)` waits for `n` seconds before continuing. A typical usage is to interpose `pause` commands

within a sequence of `plot` commands. In the past it was also used in conjunction with the `echo` command in scripts intended for demonstration, though this usage has been superseded by demonstrations shown in the Help browser.

## 13.2. Output to the Screen

The results of MATLAB computations are displayed on the screen whenever a semicolon is omitted after an assignment, and the format of the output can be varied using the `format` command. But much greater control over the output is available with the use of several functions.

The `disp` function displays the value of a variable, according to the current `format`, without first printing the variable name and “=”. If its argument is a string, `disp` displays the string. Example:

```
>> disp('Here is a 3-by-3 magic square'), disp(magic(3))
Here is a 3-by-3 magic square
     8     1     6
     3     5     7
     4     9     2
```

More sophisticated formatting can be done with the `fprintf` function. The syntax is `fprintf(format, list-of-expressions)`, where `format` is a string that specifies the precise output format for each expression in the list. In the example

```
>> fprintf('%6.3f\n', pi)
3.142
```

the `%` character denotes the start of a format specifier requesting a field width of 6 with three digits after the decimal point, and `\n` denotes a new line (without which subsequent output would continue on the same line). If the specified field width is not large enough MATLAB expands it as necessary:

```
>> fprintf('%6.3f\n', pi^10)
93648.047
```

The fixed-point notation produced by `f` is suitable for displaying integers (using `%n.0f`) and when a fixed number of decimal places are required, such as when displaying dollars and cents (using `%n.2f`). If `f` is replaced by `e` then the digit after the period denotes one less than the total number of significant digits to display in exponential notation (there will always be one digit before the decimal point):

```
>> fprintf('%12.3e\n', pi)
3.142e+00
```

When choosing the field width remember that for a negative number a minus sign occupies one position:

```
>> fprintf('%5.2f\n%5.2f\n', exp(1), -exp(1))
 2.72
-2.72
```

A minus sign just after the `%` character causes the field to be left justified:

```
>> fprintf('%5.0f\n%5.0f\n',9,103)
  9
 103

>> fprintf('%-5.0f\n%-5.0f\n',9,103)
  9
 103
```

The format string can contain characters to be printed literally, as the following example shows:

```
>> iter = 11; m = 5; rng(1); U = orth(randn(m)) + 1e-10;

>> fprintf('iter = %2.0f\n', iter)
iter = 11

>> fprintf('norm(U'*U-I) = %11.4e\n', norm(U'*U - eye(m)))
norm(U'*U-I) = 5.2325e-10
```

Note that, within a string, `'` represents a single quote.

To print `%` and `\` use `\%` and `\\` in the format string. Another useful format specifier is `g`, which uses whichever of `e` and `f` produces the shorter result:

```
>> fprintf('%g %g\n', exp(1), exp(20))
2.71828 4.85165e+08
```

Various other specifiers and special characters are supported by `fprintf`, which behaves similarly to the C function `printf` (see `doc fprintf`).

If more numbers are supplied to be printed than there are format specifiers in the `fprintf` statement then the format specifiers are reused, with elements being taken from a matrix down the first column, then down the second column, and so on. This feature can be used to avoid a loop. Example:

```
>> A = [30 40 60 70];
>> fprintf('%g miles/hour = %g kilometers/hour\n', [A; 8*A/5])
30 miles/hour = 48 kilometers/hour
40 miles/hour = 64 kilometers/hour
60 miles/hour = 96 kilometers/hour
70 miles/hour = 112 kilometers/hour
```

To print a string variable use the `s` format specifier. This example makes use of a cell array (see Section 18.7):

```
>> data = {'Alan Turing', 1912, 1954};
>> fprintf('%s (%4.0f-%4.0f)\n', data{1:3})
Alan Turing (1912-1954)
```

The function `sprintf` is analogous to `fprintf` but returns its output as a string. It is useful for producing labels for plots. A simpler to use but less versatile alternative is `num2str`: `num2str(x,n)` converts `x` to a string with `n` significant digits, with `n` defaulting to 4. For converting integers to strings, `int2str` can be used. Here are three examples, the first of which uses the format specifier `u` for an unsigned integer and the second and third of which make use of string concatenation (see Section 18.1):

```

>> n = 16;
>> err_msg = sprintf('Must supply a %u-by-%u matrix', n, n)
err_msg =
Must supply a 16-by-16 matrix

>> disp(['Pi is given to 6 significant figures by ' num2str(pi,6)])
Pi is given to 6 significant figures by 3.14159

>> i = 3;
>> title_str = ['Result of experiment ' int2str(i)]
title_str =
Result of experiment 3

```

### 13.3. File Input and Output

A number of functions are provided for reading and writing binary and formatted text files. These include functions for reading and writing spreadsheets (`xlsread`, `xlswrite`) and tables (`readtable`, `writetable`—tables are discussed in Section 18.6). Type `help iofun` to see the complete list.

We show by example how to write data to a formatted text file and then read it back in. Before operating on a file it must be opened with the `fopen` function, whose first argument is the filename and whose second argument is a file permission, which has several possible values including `'r'` for read and `'w'` for write. A file identifier is returned by `fopen`; it is used in subsequent read and write statements to specify the file. Data is written using the `fprintf` function, which takes as its first argument the file identifier. Thus the code

```

A = [30 40 60 70];
fid = fopen('myoutput','w');
fprintf(fid,'%g miles/hour = %g kilometers/hour\n', [A; 8*A/5]);
fclose(fid);

```

creates a file `myoutput` containing

```

30 miles/hour = 48 kilometers/hour
40 miles/hour = 64 kilometers/hour
60 miles/hour = 96 kilometers/hour
70 miles/hour = 112 kilometers/hour

```

Note that we included a semicolon after `fprintf` and `fclose` because we wanted to suppress the screen output of these statements, which is the number of bytes written and the status of the close operation, respectively. The file can be read in as follows:

```

>> fid = fopen('myoutput','r');
>> X = fscanf(fid,'%g miles/hour = %g kilometers/hour')
X =
    30
    48
    40
    64
    60

```

```

    96
    70
    112
>> fclose(fid);

```

The `fscanf` function reads data formatted according to the specified format string, which in this example says, “read a general floating-point number (`%g`), skip over the string `'miles/hour = '`, read another general floating-point number and skip over the string `'kilometers/hour'`. The format string is recycled until the entire file has been read and the output is returned in a vector. We can convert the vector to the original matrix format using

```

>> X = reshape(X,2,4)'
X =
    30    48
    40    64
    60    96
    70   112

```

Alternatively, a matrix of the required shape can be obtained directly:

```

>> X = fscanf(fid,'%g miles/hour = %g kilometers/hour',[2 inf])'
X =
    30    48
    40    64
    60    96
    70   112

```

The third argument to `fscanf` specifies the dimensions of the output matrix, which is filled column by column. We specify `inf` for the number of columns, to allow for any number of lines in the file, and transpose to recover the original format.

Another way to read a text file is with the powerful `textscan` function, which uses a different syntax than `fscanf` for the format string and returns the output in a cell array:

```

>> fid = fopen('myoutput','r');
>> C = textscan(fid,'%f miles/hour = %f kilometers/hour')
C =
    [4×1 double]    [4×1 double]
>> fclose(fid);

>> C{1}'
ans =
    30    40    60    70
>> C{2}'
ans =
    48    64    96   112

```

`textscan` has a number of configurable parameters and is recommended for reading large files.

Binary files are created and read using the functions `fread` and `fwrite`. See the online help for details of their usage.

MATLAB has the capability to read and write XML files, Excel spreadsheets, zip files, gzip files, and tar files. Type `help iofun` for a list of the relevant functions.

### 13.4. Fine Tuning the Display of Arrays

Researchers often need to incorporate the results of MATLAB computations into tables in a document. Copying and pasting raw output from the Command Window is not a good idea, as it may not provide the numbers in the required format and it is error prone and wastes time if computations need to be redone. It is much better to have MATLAB print the output formatted exactly as required. Listing 13.1 shows a function `print_matrix` that does this job. It offers options to print to a file, to remove unnecessary zeros from the exponent field, to print a matrix in the form of a  $\text{\LaTeX}$  tabular environment, and to format each column to a given field width (no warning is given if this results in truncation of a number). Here are some examples:

```
>> x = [0 1.999 pi*1e5 3.959*1e20 -5.4555*1e100];
>> A = [x; 1./(x+eps)]
A =
           0   1.9990e+00   3.1416e+05   3.9590e+20  -5.4555e+100
  4.5036e+15   5.0025e-01   3.1831e-06   2.5259e-21  -1.8330e-101

>> print_matrix(A)
  0.00e+00   2.00e+00   3.14e+05   3.96e+20  -5.46e+100
  4.50e+15   5.00e-01   3.18e-06   2.53e-21  -1.83e-101

>> print_matrix(A,{'%6.2e','%3.1f','%7.3e','%4.1e','%g'})
  0.00e+00   2.0   3.142e+05   4.0e+20  -5.4555e+100
  4.50e+15   0.5   3.183e-06   2.5e-21  -1.83301e-101

>> print_matrix(A,'%4.1e',[],9,1,1) % LaTeX form, remove +, zeros.
  0.0       & 2.0       & 3.1e5     & 4.0e20    & -5.5e100  \\
  4.5e15    & 5.0e-1    & 3.2e-6   & 2.5e-21   & -1.8e-101  \\
```

The `print_matrix` function was used to produce Table 26.1. It can readily be adapted to particular needs.

Listing 13.1. *Script print\_matrix.*

```

function print_matrix(x, fmt, file, fw, ltx, ex)
%PRINT_MATRIX    Print formatted matrix to Command Window or file.
% PRINT_MATRIX(X, fmt, file, fw, ltx, ex) prints the matrix X according
% to the fprintf-style format specified by the string fmt.
% Default: fmt = '%9.2e'.
% If fmt is a cell array of strings then its j'th entry will be used
% to format the j'th column, with cyclic re-use of entries as necessary.
% fw: fieldwidth of every number (default: automatic).
% file: filename for output or empty for output to Command Window.
% ltx = 1 for LaTeX tabular format, else ltx = 0 (default).
% ex = 1 for delete plus and leading zeros in exponent,
% else ex = 0 (default).

if nargin < 2 | isempty(fmt), fmt = '%9.2e'; end
if nargin < 3, file = []; end
if nargin < 4, fw = []; end
if nargin < 5 | isempty(ltx), ltx = 0; end
if nargin < 6, ex = 0; end
if isstr(fmt), fmt = {fmt}; end % Ensure fmt is cell array.

[m,n] = size(x);
if isempty(file)
    fid = 1; % Output to screen.
else
    fid = fopen(file,'w');
end

% Conversions for removing plus and leading zeros in exponents.
exps = {'e+' 'e'; 'e00' ' ' ' '; 'e0' 'e'; 'e-00' 'e-' ; 'e-0' 'e-'};

for i=1:m
    for j=1:n
        fmtj = fmt{1+mod(j-1,length(fmt))}; % Format string for j'th col.
        str = sprintf([fmtj ' '], x(i,j));
        if ex
            for k=1:length(exps)
                str = strrep(str,exps{k,1},exps{k,2});
            end
        end
        if fw
            str = [str char(32*ones(1,fw))];
            str = str(1:fw);
        end
        fprintf(fid,'%s ', str)
        if ltx && j<n, fprintf(fid,'% '); end
    end
    if ltx, fprintf(fid,'\\'); end
    fprintf(fid,'\n');
end
if ~isempty(file), fclose(fid); end

```



*Make input easy to prepare and output self-explanatory.*

— BRIAN W. KERNIGHAN and P. J. PLAUGER,  
*The Elements of Programming Style* (1978)

*Output is almost like input but it's not input, it's output.  
To correlate the output with the input and  
to verify that the input was put in correctly,  
it's a good idea to output the input along with the output.*

— ROGER EMANUEL KAUFMAN, *A FORTRAN Coloring Book* (1978)

*On two occasions, I have been asked [by the members of Parliament],  
"Pray, Mr. Babbage,  
if you put into the machine wrong figures,  
will the right answers come out?"...  
I am not able rightly to apprehend  
the kind of confusion of ideas  
that could provoke such a question.*

— CHARLES BABBAGE, *Passages from the Life of a Philosopher* (1864)

# Chapter 14

## Troubleshooting

### 14.1. Errors and Assertions

Errors in MATLAB are of two types: syntax errors and runtime errors. A syntax error is illustrated by

```
>> for i=1#10, x(i) = 1/i; end
    for i=1#10, x(i) = 1/i; end
        |
Error: The input character is not valid in MATLAB statements or
expressions.
```

Here, a # has been typed instead of a colon and the error message pinpoints where the problem occurs. If an error occurs in a code then the name of the code and the line on which the error occurred are shown.

A runtime error occurs with the script `fib` in Listing 14.1. The loop should begin at `i = 3` to avoid referencing `x(0)`. When we run the script, MATLAB produces an informative error message:

```
>> fib
Subscript indices must either be real positive integers or logicals.
Error in fib (line 4)
    x(i) = x(i-1) + x(i-2);
```

In the Command Window, the phrase `line 4` is an underlined hyperlink to line 4 of the script `fib`. If you click on the hyperlink then `fib.m` is opened in the MATLAB Editor/Debugger (see Section 7.2) and the cursor is placed on line 4.

When an error occurs in a nested sequence of calls, the history of the calls is shown in the error message. The first “Error in” line is the one describing the code in which the error is located.

MATLAB error messages are sometimes rather unhelpful and occasionally misleading. It is perhaps inevitable with such a powerful language that an error message does not always make it immediately clear what has gone wrong. We give a few examples of error messages generated for reasons that are perhaps not obvious.

- **At least one END is missing: the statement may begin here.** The message is produced by the following code, which is one way of implementing the sign function (`sign`):

```
if x > 0
    f = 1;
else if x == 0
    f = 0;
```

Listing 14.1. *Script fib that generates a runtime error.*

```
%FIB    Fibonacci numbers.
x = ones(50,1);
for i = 2:50
    x(i) = x(i-1) + x(i-2);
end
```

```
else
    f = -1;
end
```

The problem is an unwanted space between `else` and `if`. MATLAB (correctly) interprets the `if` after the `else` as starting a new `if` statement and then complains when it runs out of `ends` to match the `ifs`.

- **Undefined function or variable.** Several commands, such as `clear`, `load`, and `global`, take a list of arguments separated by spaces. If a comma is used in the list it is interpreted as separating statements, not arguments. For example, the command `clear a,b` clears `a` and prints `b`, so if `b` is undefined the above error message is produced.

- **Inputs must be a scalar and a square matrix.** This message is produced when an attempt is made to exponentiate a nonsquare matrix, and it can be puzzling. For example, it is generated by the expression `(1:5)^3`, which was presumably meant to be an elementwise cubing operation and thus should be expressed as `(1:5).^3`.

Many functions check for error conditions, issuing an error message and terminating when one occurs. For example:

```
>> mod(3,sqrt(-2))
??? Error using ==> mod
Arguments must be real.
```

In a code file this behavior can be achieved with the `error` command: the line

```
if ~isreal(arg2), error('Arguments must be real. '), end
```

produces the result just shown when `arg2` is not real. An invocation of `error` can also give arguments and use a format conversion string, just as with `fprintf` (see Section 13.2). For example, assuming that `A = zeros(3)`:

```
>> error('Matrix dimension is %g, but should be even.', length(A))
??? Matrix dimension is 3, but should be even.
```

The most recent error message can be recalled with the `lasterr` function.

Errors support message identifiers that identify the error. Identifiers are more commonly used with warnings, and they are described in the next section.

The `assert` function is similar to the `error` function but takes a logical expression `cond` as argument: if `cond` is false then the statement `assert(cond)` terminates execution and prints **Assertion failed** in the Command Window, but if `cond` is true execution passes to the next statement. With a second argument, `assert(cond,msg)`, the string `msg` is displayed as the error message. Examples:

```
>> assert(cond(hilb(6)) < 1e5)
Assertion failed.

>> x = [1 4 3 5 7];
>> assert(isequal(x,sort(x)), 'Vector x must be sorted. ')
Vector x must be sorted.
```

The example using `error` above can be rewritten as

```
>> assert(isreal(arg2), 'Arguments must be real.')
```

Assertions are useful in program development for verifying that a program is performing correctly [10]. They can be used to check conditions that are known to be true for the underlying algorithm, such as ranges of variables at particular points in the code or loop invariants (relations between variables that should be true on every iteration of a loop).

## 14.2. Warnings

The function `warning`, like `error`, displays its string argument, but execution continues instead of stopping. The reason for using `warning` rather than displaying a string with `disp` (for example) is that the display of warning messages can be controlled via certain special string arguments to `warning`. In particular, `warning('off')` or `warning off` turns off the display of all warning messages and `warning('on')` or `warning on` turns them all back on again.

MATLAB allows message identifiers to be attached to warnings in order to identify the source of the warning and to allow warnings to be turned off and on individually. For example:

```
>> nchoosek(100,80)
Warning: Result may not be exact. Coefficient is greater than
9.007199e+15 and is only accurate to 15 digits
ans =
    5.3598e+20
```

To suppress this warning, we need to find its identifier, which can be obtained as the second output argument of `lastwarn`:

```
>> [warnmsg,msg_id] = lastwarn
warnmsg =
Result may not be exact. Coefficient is greater than 9.007199e+15
and is only accurate to 15 digits
msg_id =
MATLAB:nchoosek:LargeCoefficient
```

We can turn this warning off, without affecting the status of any other warnings, with

```
>> warning off MATLAB:nchoosek:LargeCoefficient
>> % Or warning('off',msg_id)
>> nchoosek(100,80)
ans =
    5.3598e+20
```

The identifiers of warnings can be automatically displayed by setting `warning on verbose`:

```
>> warning on verbose
>> inv([1 1; 1 1])
Warning: Matrix is singular to working precision.
(Type "warning off MATLAB:singularMatrix" to suppress this warning.)
ans =
     Inf     Inf
     Inf     Inf
```

A quick way to turn off a warning that has just been invoked is to type `warning off last`.

The status of all warnings can be viewed:

```
>> warning query all
The default warning state is 'on'. Warnings not set to the
default are
```

State Warning Identifier

```
off MATLAB:COPYFILE:SHFileOperationErrorID
off MATLAB:Debugger:BreakpointSuppressed
off MATLAB:JavaComponentThreading
off MATLAB:JavaEDTAutoDelegation
...
off MATLAB:nchoosek:LargeCoefficient
...
```

Here, we have truncated the list. Apart from the `nchoosek` warning, all the other warnings are in the state they were in when our MATLAB session started.

A common warning message is `Matrix is close to singular or badly scaled`, which signals an attempt to invert a nearly singular matrix; see the example on p. 138. While this warning is welcome, in that it may indicate a bug in your code or unexpected behavior of an algorithm, it is sometimes unwanted. It can be turned off with `warning('off','MATLAB:nearlySingularMatrix')`.

You can define your own warnings with identifiers using the syntax

```
warning('msg_id','warnmsg')
```

(or by making `'warnmsg'` a format string and following it by a list of arguments). Here, `'msg_id'` is a string comprising a component field, which might identify a product or a toolbox, followed by a mnemonic field relating to the message. The actual warning message is in the string `'warnmsg'`. For example, the function `fd_deriv` in Listing 10.1 is likely to return inaccurate results if `h` is close to `eps`, so we could append to the function the lines

```
if h <= 1e-14
    warning('MATLABGuide:fd_deriv:RoundingErrorMayDominate',...
           'Difference h of order eps may produce inaccurate ', ...
           'result.')
end
```

If we do so, then the following behavior is observed:

```
>> fd_deriv(@exp,2,1e-15);
Warning: Difference h of order eps may produce inaccurate result.
> In fd_deriv at 11
```

If you change the `warning` state in a code it is good practice to save the old state and restore it before the end of the code, as in the following example:

```
warns = warning; % or "warns = warning('query','all')"
warning('off') % or "warning off all"
...
warning(warns)
```

### 14.3. Debugging

Debugging MATLAB codes is in principle no different to debugging any other type of computer program, but several facilities are available to ease the task. When a code runs but does not perform as expected it is often helpful to print out the values of key variables, which can be done by removing semicolons from assignment statements or adding statements consisting of the relevant variable names.

When it is necessary to inspect several variables and the relations between them the `keyboard` statement is invaluable. When a `keyboard` statement is encountered in a code execution halts and a command line with the special prompt `K>>` appears. Any MATLAB command can be executed and variables in the workspace can be inspected or changed. When keyboard mode is invoked from within a function the visible workspace is that of the function. The command `dbup` changes the workspace to that of the calling function or the main workspace; `dbdown` reverses the effect of `dbup`. Typing `return` followed by the return key causes execution of the code to be resumed. The `dbcont` command has the same effect. Alternatively, the `dbquit` command quits keyboard mode and terminates the code.

Another way to invoke keyboard mode is via the debugger. Typing

```
dbstop in foo at 5
```

sets a breakpoint at line 5 of `foo.m`; this causes subsequent execution of `foo.m` to stop just before line 5 and keyboard mode to be entered. The command

```
dbstop in foo at 3 if i==5
```

sets a conditional breakpoint that causes execution to stop only if the given expression evaluates to true, that is, if  $i = 5$ . A listing of `foo.m` with line numbers is obtained with `dbtype foo`. Breakpoints are cleared using the `dbclean` command. Breakpoints can also be set from the MATLAB Editor/Debugger.

We illustrate the use of the debugger on the script `fib` discussed in the last section (Listing 14.1). Here, we set a breakpoint (on a runtime error) and then inspect the value of the loop index when the error occurs:

```
>> dbstop if error
>> fib
Subscript indices must either be real positive integers or logicals.
Error in fib (line 4)
```

```

        x(i) = x(i-1) + x(i-2);
4       x(i) = x(i-1) + x(i-2);
K>> i
i =
     2
K>> dbquit

```

The MATLAB debugger is a powerful tool with several other features that are described in the online documentation. In addition to the command line interface to the debugger illustrated above, an Editor/Debugger window is available that provides a visual interface (see Section 7.2).

A useful tip for debugging is to execute

```
clear all
```

and one of

```
clf
close all
```

before executing the code with which you are having trouble. The first command clears variables and functions from memory. This is useful when, for example, you are working with scripts because it is possible for existing variables to cause unexpected behavior or to mask the fact that a variable is accessed before being initialized in the script. The other commands are useful for clearing the effects of previous graphics operations.

## 14.4. Pitfalls

Here are some suggestions to help avoid pitfalls particular to MATLAB.

- If you use functions `i` or `j` for the imaginary unit, make sure that they have not previously been overridden by variables of the same name (`clear i` or `clear j` clears the variable and reverts to the functional form). In general it is not advisable to choose variable names that are the names of MATLAB functions. For example, if you assign

```
>> rand = 1;
```

then subsequent attempts to use the `rand` function generate an error:

```
>> A = rand(3)
Index exceeds matrix dimensions.
```

In fact, MATLAB is still aware of the function `rand`, but the variable takes precedence, as can be seen from

```
>> which -all rand
rand is a variable.
C:\MATLAB2016b\toolbox\matlab\randfun\@RandStream\rand.m
    % Shadowed RandStream method
built-in (C:\MATLAB2016b\toolbox\matlab\randfun\rand)
    % Shadowed
```

The function can be reinstated by clearing the variable:

```
>> clear rand
>> rand
ans =
    0.6068
```

- Confusing behavior can sometimes result from the fact that `max`, `min`, and `sort` behave differently for real and for complex data—in the complex case they work with the absolute values of the data. For example, suppose we compute the following 4-vector, which should be real but has a tiny nonzero imaginary part due to rounding errors:

```
e =
-4.7986e+00 + 1.6448e-14i
 4.9552e+00 + 1.4972e-14i
 1.3982e+00 + 4.5330e-15i
-6.6605e-02 + 6.8292e-15i
```

To find the most negative element we need to use `min(real(e))` rather than `min(e)`:

```
>> min(e)
ans =
-6.6605e-02 + 6.8292e-15i

>> min(real(e))
ans =
-4.7986e+00
```

- As noted in the Aside on p. 56, mathematical formulas and descriptions of algorithms often index vectors and matrices so that their subscripts start at 0. Since subscripts of MATLAB arrays start at 1, translation of subscripts is necessary when implementing such formulas and algorithms in MATLAB.



*The road to wisdom?  
Well, it's plain and simple to express:  
Err  
and err  
and err again  
but less  
and less  
and less.*

— PIET HEIN, *Grooks* (1966)

*Beware of bugs in the above code;  
I have only proved it correct, not tried it.*

— DONALD E. KNUTH<sup>7</sup> (1977)

*Test programs at their boundary values.*

— BRIAN W. KERNIGHAN and P. J. PLAUGER,  
*The Elements of Programming Style* (1978)

*By June 1949 people had begun to realize that  
it was not so easy to get a program right as had at one time appeared. . .*

*The realization came over me with full force that  
a good part of the remainder of my life was going to be spent in  
finding errors in my own programs.*

— MAURICE WILKES, *Memoirs of a Computer Pioneer* (1985)

---

<sup>7</sup>See <http://www-cs-faculty.stanford.edu/~uno/faq.html>

# Chapter 15

## Sparse Matrices

A sparse matrix is one with a large percentage of zero elements. When dealing with large, sparse matrices it is desirable to take advantage of the sparsity by storing and operating only on the nonzeros. MATLAB arrays of dimension up to 2 can have a `sparse` attribute, in which case just the nonzero entries of the array together with their row and column indices are stored. Currently, sparse arrays are supported only for the `double` data type. In this chapter we will use the term “sparse matrix” for a two-dimensional `double` array having the `sparse` attribute and the term “full matrix” for such an array having the (default) `full` attribute.

More details on sparse matrix methods can be found in [27] (of which Chapter 10 is devoted to MATLAB) and [29].

### 15.1. Sparse Matrix Generation

Sparse matrices can be created in various ways, several of which involve the `sparse` function. Given a `t`-vector `s` of matrix entries and `t`-vectors `i` and `j` of indices, the command `A = sparse(i, j, s)` defines a sparse matrix `A` of dimension `max(i)`-by-`max(j)` with `A(i(k), j(k)) = s(k)` for `k=1:t` and all other elements zero. Example:

```
>> A = sparse([1 2 2 4 4], [3 1 4 2 4], 1:5)
A =
    (2,1)      2
    (4,2)      4
    (1,3)      1
    (2,4)      3
    (4,4)      5
```

MATLAB displays a sparse matrix by listing the nonzero entries preceded by their indices, sorted by columns. If an index `i(k), j(k)` is supplied more than once then the corresponding entries are added:

```
>> sparse([1 2 2 4 1], [3 1 4 2 3], 1:5)
ans =
    (2,1)      2
    (4,2)      4
    (1,3)      6
    (2,4)      3
```

A sparse matrix can be converted to a full one using the `full` function:

```
>> B = full(A)
B =
```

```

0    0    1    0
2    0    0    3
0    0    0    0
0    4    0    5

```

Conversely, a full matrix **B** is converted to the sparse storage format by **A = sparse(B)**. The number of nonzeros in a sparse (or full) matrix is returned by **nnz**:

```

>> nnz(A)
ans =
    5

```

After defining **A** and **B**, we can use the **whos** command to check the amount of storage used:

```

>> whos
Name      Size      Bytes  Class      Attributes

A         4x4       120    double     sparse
B         4x4       128    double
ans       1x1        8      double

```

The matrix **B** comprises 16 double-precision numbers of 8 bytes each, making a total of 128 bytes. The storage required for a sparse **n**-by-**n** matrix with **nnz** nonzeros is  $16*\text{nnz} + 8*(\text{n}+1)$  bytes, which includes the **nnz** double-precision numbers plus some 4-byte integers. The same formula applies to **m**-by-**n** matrices, since the number of rows does not affect the required storage.

The **sparse** function accepts three extra arguments. The command

```
A = sparse(i,j,s,m,n)
```

constructs an **m**-by-**n** sparse matrix; the last two arguments are necessary when the last row or column of **A** is all zero. The command

```
A = sparse(i,j,s,m,n,nzmax)
```

allocates space for **nzmax** nonzeros, which is useful if extra nonzeros, not in **s**, are to be introduced later, for example when **A** is generated column by column.

A sparse matrix of zeros is produced by **sparse(m,n)** (both arguments must be specified), which is an abbreviation for **sparse([], [], [], m, n, 0)**.

The sparse identity matrix is produced by **speye(n)** or **speye(m,n)**, while the command **spones(A)** produces a matrix with the same sparsity pattern as **A** and with ones in the nonzero positions.

The arguments that **sparse** would need to reconstruct an existing matrix **A** via **sparse(i,j,s,m,n)** can be obtained using

```
[i,j,s] = find(A);
[m,n] = size(A);
```

If just **s** is required, then **s = nonzeros(A)** can be used. The number of storage locations allocated for nonzeros in **A** can be obtained with **nzmax(A)**. The inequality  $\text{nnz}(A) \leq \text{nzmax}(A)$  always holds.

The function **spdiags** is an analogue of **diag** for sparse matrices. The command **A = spdiags(B,d,m,n)** creates an **m**-by-**n** matrix **A** whose diagonals indexed by **d** are taken from the columns of **B**. This function is best understood by looking at examples. Given

```

B =
    1     2     0
    1     2     3
    0     2     3
    0     2     3
d =
   -2     0     1

```

we can define

```

>> A = spdiags(B,d,4,4)
A =
(1,1)     2
(3,1)     1
(1,2)     3
(2,2)     2
(4,2)     1
(2,3)     3
(3,3)     2
(3,4)     3
(4,4)     2

```

```

>> full(A)
ans =
    2     3     0     0
    0     2     3     0
    1     0     2     3
    0     1     0     2

```

Note that the subdiagonals are taken from the leading parts of the columns of **B** and the superdiagonals from the trailing parts. Diagonals can be extracted with `spdiags`: `[B,d] = spdiags(A)` recovers **B** and **d** above. The next example sets up a particular tridiagonal matrix:

```

>> n = 5; e = ones(n,1);
>> A = spdiags([-e 4*e -e],[-1 0 1],n,n);
>> full(A)
ans =
    4    -1     0     0     0
   -1     4    -1     0     0
    0    -1     4    -1     0
    0     0    -1     4    -1
    0     0     0    -1     4

```

Random sparse matrices are generated with `sprand` and `sprandn`. The command `A = sprand(S)` generates a matrix with the same sparsity pattern as **S** and with nonzero entries uniformly distributed on  $[0, 1]$ . Alternatively, `A = sprand(m,n,density)` generates an **m**-by-**n** matrix of a random sparsity pattern containing approximately `density*m*n` nonzero entries uniformly distributed on  $[0, 1]$ . With four input arguments, `A = sprand(m,n,density,rc)` produces a matrix for which the reciprocal of the condition number is about `rc`. The syntax for `sprandn` is the same, but random

numbers from the normal (0,1) distribution are produced. The function `sprandsym` is similar to `sprandn`, but produces symmetric matrices.

An invaluable command for visualizing sparse matrices is `spy`, which plots the sparsity pattern with a dot representing a nonzero; see the plots in the next section.

A sparse array can be distinguished from a full one using the logical function `issparse` (there is no “`isfull`” function).

## 15.2. Linear Algebra

MATLAB is able to solve sparse linear equation, eigenvalue, and singular value problems, taking advantage of sparsity as it does so.

As for full matrices (see Section 9.3.1), the backslash operator `\` can be used to solve linear systems. The effect of  $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$  when  $\mathbf{A}$  is sparse is roughly as follows (for full details, type `doc mldivide`). If  $\mathbf{A}$  is square and banded then a banded solver is used. If  $\mathbf{A}$  is a permutation of a triangular matrix substitution is used. If  $\mathbf{A}$  is a Hermitian positive definite matrix a Cholesky factorization with a minimum-degree reordering is used. For a square matrix with no special properties, a sparse LU factorization is computed using UMFPACK [25], [26] with a reordering produced by `amd` (approximate minimum-degree permutation) or `colamd` (column approximate minimum-degree permutation), depending on the sparsity pattern of the matrix. If  $\mathbf{A}$  is rectangular then QR factorization is used; a rank-deficiency test is performed based on the diagonal elements of the triangular factor.

The importance of avoiding matrix inversion where possible was explained in Section 9.4 in the context of full matrices. Matrix inversion is deprecated even more strongly for sparse matrices, because the inverse of a matrix containing many zeros usually has far fewer zeros—often none. Hence although the `inv` function will invert a sparse matrix (and return the result in the sparse format), it should almost never be used to do so.

To compute or estimate the condition number of a sparse matrix `condest` should be used (see Section 9.2), as `cond` and `rcond` are designed only for full matrices.

The `chol` function for Cholesky factorization behaves in a similar way for sparse matrices as for full matrices, but the computations are done using sparse data structures.

For real, sparse symmetric indefinite  $\mathbf{A}$  the command `[L,D,P] = ld1(A,thresh)` computes the factorization  $P^T A P = LDL^T$ , where  $P$  is a permutation matrix,  $L$  is unit lower triangular, and  $D$  is block diagonal with diagonal blocks of dimension 1 or 2. The pivot threshold `thresh`, which defaults to 0.01, must lie in the interval  $[0, 0.5]$ . The computation is carried out with MA57 from the HSL library [40]. If a fourth output argument, `S`, is given, then  $P^T S A S P = LDL^T$ , where  $S$  is a diagonal scaling matrix.

The `lu` function for LU factorization has some extra options not present in the full case. With up to three output arguments, the same mathematical factorization is produced as in the full case. A second input argument `thresh`, as in `lu(A,thresh)`, sets a pivoting threshold, which must lie between 0 and 1. The pivoting strategy requires the pivot element to have magnitude at least `thresh` times the magnitude of the largest element below the diagonal in the pivot column. The default is 1, corresponding to partial pivoting, and a threshold of 0 forces no pivoting.

With a fourth output argument `lu` uses UMFPACK. The general syntax is

$$[L,U,P,Q] = \text{lu}(A,\text{thresh})$$

Here, a factorization  $PAQ = LU$  is produced where  $L$  is unit lower triangular,  $U$  is upper triangular, and  $P$  and  $Q$  are permutation matrices. The row permutations in  $P$  are used for numerical stability and the column permutations in  $Q$  are used to reduce fill-in (a zero element becoming nonzero). The threshold `thresh` has the same meaning as when there are fewer than four output arguments, but its default is now 0.1 (it can also now be a 1-by-2 vector: see `doc lu` for the meaning in this case). A fifth output argument can be given to invoke diagonal row scaling.

Since `lu` (with two output arguments) and `chol` do not pivot for sparsity (that is, they do not use row or column interchanges in order to try to reduce the cost of the factorizations), it is advisable to consider reordering the matrix before factorizing it. A full discussion of reordering algorithms is beyond the scope of this book, but we give some examples.

We illustrate reorderings with the Wathen matrix:

```
A = gallery('wathen',8,8);
subplot(121), spy(A), subplot(122), spy(chol(A))
```

The `spy` plots of  $A$  and its Cholesky factor are shown in Figure 15.1. Now we reorder the matrix using the symmetric reverse Cuthill–McKee permutation and refactorize:

```
r = symrcm(A);
subplot(121), spy(A(r,r)), subplot(122), spy(chol(A(r,r)))
```

Note that all the reordering functions return an integer permutation vector rather than a permutation matrix (see Section 24.3 for more on permutation vectors and matrices). The `spy` plots are shown in Figure 15.2. Finally, we try the symmetric approximate minimum-degree ordering:

```
m = symamd(A);
subplot(121), spy(A(m,m)), subplot(122), spy(chol(A(m,m)))
```

The `spy` plots are shown in Figure 15.3. For this matrix the minimum-degree ordering leads to the sparsest Cholesky factor: the one with the least nonzeros.

For LU factorization, possible reorderings include

```
p = colamd(A); p = colperm(A);
```

after which  $A(:,p)$  is factorized.

In the QR factorization  $[Q,R] = \text{qr}(A)$  of a sparse rectangular matrix  $A$  the orthogonal factor  $Q$  can be much less sparse than  $A$ , so it is usual to try to avoid explicitly forming  $Q$ . When given a sparse matrix and one output argument, the `qr` function returns just the upper triangular factor  $R$ :  $R = \text{qr}(A)$ . When called as  $[C,R] = \text{qr}(A,B)$ , the matrix  $C = Q^T * B$  is returned along with  $R$ . This enables an overdetermined system  $Ax = b$  to be solved in the least-squares sense by

```
[c,R] = qr(A,b);
x = R\c;
```

The backslash operator ( $A \setminus b$ ) uses this method for rectangular  $A$ .

The iterative linear system solvers in Table 9.3 are also designed to handle large sparse systems. See Section 9.9 for details of how to use them. Sparse eigenvalue and singular value problems can be solved using `eigs` and `svds`, which are also described in Section 9.9. For a real, sparse symmetric matrix the eigenvalues (only) can be

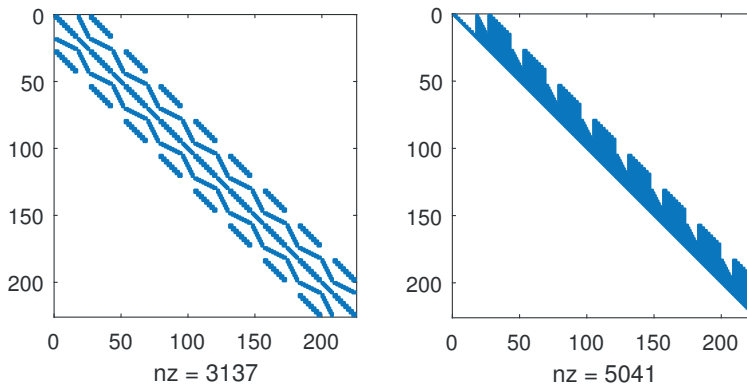


Figure 15.1. *Wathen matrix (left) and its Cholesky factor (right).*

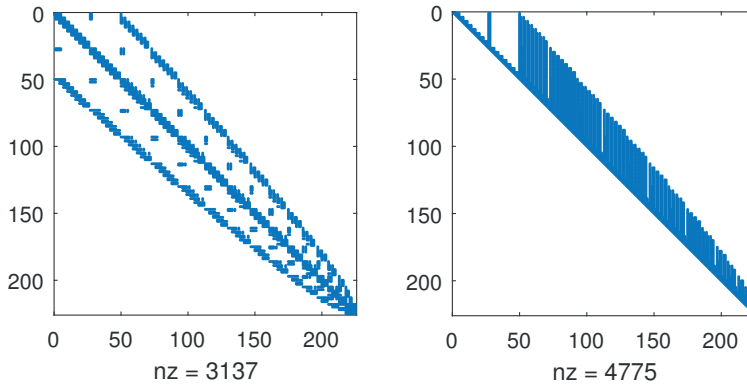


Figure 15.2. *Wathen matrix (left) and its Cholesky factor (right) with symmetric reverse Cuthill-McKee ordering (symrcm).*

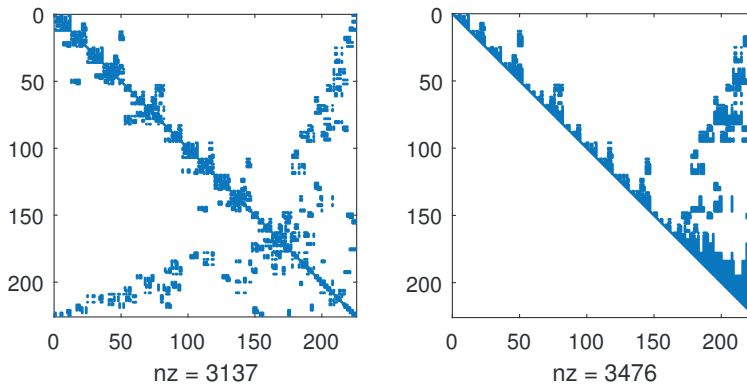


Figure 15.3. *Wathen matrix (left) and its Cholesky factor (right) with symmetric minimum-degree ordering (symamd).*

computed by `eig`, but for all other types of matrix (including complex Hermitian) it is necessary to use `eigs`.

The function `spparms` helps determine or change the internal workings of some of the sparse factorization functions. Typing `spparms` by itself prints the current settings:

```
>> spparms
No SParse MONItor output.
mmd: threshold = 1.1 * mindegree + 1,
      using approximate degrees in A'*A,
      supernode amalgamation every 3 stages,
      row reduction every 3 stages,
      withhold rows at least 50% dense in colmmd.
Minimum-degree orderings used with v4 chol, lu, and qr in \ and /.
Approximate minimum-degree orderings used with CHOLMOD and UMFPACK
in \ and /.
Pivot tolerance of 0.1 used by UMFPACK in \ and /.
Backslash uses band solver if band density is > 0.5
UMFPACK used for lu in \ and /.
Symmetric pivot tolerance of 0.001 used by UMFPACK in \ and /.
Pivot tolerance of 0.01 used by MA57 in \ and /.
```

These parameters should not normally need to be adjusted.

*The MATLAB statement  $x = A \setminus b$  for a sparse matrix  $A$  is a simple one-character interface to perhaps over 120 000 lines of high-quality software for sparse direct methods.*

— TIMOTHY A. DAVIS, SIVASANKARAN RAJAMANICKAM,  
and WISSAM M. SID-LAKHDAR,  
*A Survey of Direct Methods for Sparse Linear Systems* (2016)

*How much of the matrix must be zero for it to be considered sparse depends on the computation to be performed, the pattern of the nonzeros, and even the architecture of the computer.*

*Generally, we say that a matrix is sparse if there is an advantage in exploiting its zeros.*

— I. S. DUFF, A. M. ERISMAN, and J. K. REID,  
*Direct Methods for Sparse Matrices* (1986)

*Sparse matrices are created explicitly rather than automatically. If you don't need them, you won't see them mysteriously appear.*

— *The MATLAB EXPO: An Introduction to MATLAB, SIMULINK and the MATLAB Application Toolboxes* (1993)

*An objective of a good sparse matrix algorithm should be: The time required for a sparse matrix operation should be proportional to the number of arithmetic operations on nonzero quantities.*

*We call this the "time is proportional to flops" rule; it is a fundamental tenet of our design.*

— JOHN R. GILBERT, CLEVE B. MOLER, and ROBERT S. SCHREIBER,  
*Sparse Matrices in MATLAB: Design and Implementation* (1992)



# Chapter 16

## More on Coding

### 16.1. Elements of Coding Style

As you use MATLAB you will build up your own collection of program files. Some may be short scripts that are intended to be used only once, but others will be of potential use in future work. Based on our experience with MATLAB we offer some guidelines on making program files easy to use, understand, and maintain.

In Chapter 7 we explained the structure of the leading comment lines of a function, including the H1 line. Adhering to this format and fully documenting the function in the leading comment lines is vital if you are to be able to reuse and perhaps modify the function some time after writing it. A further benefit is that writing the comment lines forces you to think carefully about the design of the function, including the number and ordering of the input and output arguments.

We recommend following the practice of MATLAB itself and using the active voice and the present tense in comment lines. For example, `help integral` produces

```
Q = integral(FUN,A,B) approximates the integral of function FUN ...
```

rather than “is an approximation to the integral of” or “will approximate the integral of”.

It is helpful to include in the leading comment lines an example of how the function is used, in a form that can be cut and pasted into the command line. MATLAB functions that provide such examples include `fplot`, `fzero`, `meshgrid`, and `integral`.

You may want to include a pointer to other related functions. This can be done by including a final leading comment line of the form

```
% See also fun1, fun2, fun3
```

Assuming that `fun1`, `fun2`, and `fun3` are functions on the MATLAB path or in the current directory, the `help` command displays these function names as hyperlinks to the help for the corresponding functions.

In formatting the code, it is advisable to follow the example of the functions provided with MATLAB: that is, to

- indent so that comment lines after the H1 line start in column 5;
- put spaces around logical operators and = in assignment statements;
- use one statement per line (with exceptions such as a short `if`, or a related sequence of assignments);
- indent to emphasize `if`, `for`, `switch`, and `while` structures (as provided automatically by the MATLAB Editor/Debugger—see Section 7.2);

- use variable names beginning with capital letters for matrices.

Compare the code segment

```
if stopit(4)==1
% Right-angled simplex based on coordinate axes.
alpha=norm(x0,inf)*ones(n+1,1);
for j=2:n+1, V(:,j)=x0+alpha(j)*V(:,j); end
end
```

with the more readable

```
if stopit(4) == 1
% Right-angled simplex based on coordinate axes.
alpha = norm(x0,inf)*ones(n+1,1);
for j = 2:n+1
    V(:,j) = x0 + alpha(j)*V(:,j);
end
end
```

In this book we usually follow these rules, occasionally breaking them to save space.

A rough guide to choosing variable names is that the length and complexity of a name should be proportional to the variable's scope (the region in which it is used). Loop index variables are typically one character long because they have local scope and are easily recognized. Constants used throughout a code file merit longer, more descriptive names. For long variable names comprising two or more words joined together, two commonly used styles are

- camel case: `stepSizeLimit` (lower camel case) or `StepSizeLimit` (upper camel case);
- pothole case (or snake case): `step_size_limit`.

A MATLAB function that helps in choosing variable names, especially in an automated way, is `matlab.lang.makeValidName`. See the help for the function for some interesting examples of its use.

## 16.2. Cleaning Up

It is good practice for your functions to leave the MATLAB environment in the same state it was in when the files began. If the MATLAB path, warning states, or graphics defaults are changed they should be restored before exit. A general clean up technique that can be used is illustrated by the code

```
warns = warning('query','all');
temp = onCleanup(@()warning(warns));
warning('off','all');
```

In Section 14.2 we suggested issuing the command `warning(warns)` at the end of the code, after assigning `warns` as in this example. The use of `onCleanup`, which is designed for functions rather than scripts, causes the anonymous function that is its argument to be executed once the variable `temp` is destroyed, that is, upon termination of the function. The key point is that even if termination is due to an error or an exit

forced by the user hitting Ctrl-c the `onCleanup` action will be invoked. For a quick illustration of the function in action, try the following commands in the Command Window:

```
>> beep      % Check speaker volume is turned up.
>> temp = onCleanup(@()beep)
temp =
    onCleanup with properties:

        task: @()beep
>> clear temp % Causes a beep.
```

For various other examples of the use of this clean up technique see `doc onCleanup`.

### 16.3. Checking and Comparing Code Files

MATLAB provides some useful tools for automatically checking and comparing code files. The `checkcode` function reads a code file and produces a report of potential errors and problems, and also makes suggestions for improving the efficiency and maintainability of the code. The function `badfun` in Listing 16.1 is perfectly legal MATLAB:

```
>> badfun(1,2);
x =
    2.2361
```

However, it contains several weaknesses that `checkcode` detects:

```
>> checkcode badfun
L 1 (C 13): The function return value 'y' might be unset.
L 1 (C 18-22): Function name 'badfu' is known to MATLAB by its
              file name: 'badfun'.
L 4 (C 15): Use || instead of | as the OR operator in (scalar)
              conditional statements.
L 4 (C 29): The value assigned to variable 'c' might be unused.
L 6 (C 3): Terminate statement with semicolon to suppress output
              (in functions).
```

The output refers to lines (L) and columns (C). The third reported problem refers to the fact that `||` is preferred for tests between scalars (see Section 6.1). As this example shows, `checkcode` is good at detecting variables that are never assigned or used, which is useful because these problems can be hard to spot in longer codes. The MATLAB Editor automatically highlights code to indicate some of the information produced by `checkcode`; hover the cursor over the highlighted code to see the corresponding message.

Another feature of `checkcode` is that with the `'-cyc'` argument it computes the cyclomatic complexity, or McCabe complexity, of each function in a code file [76], [120]. Here, we check the complexity of the `integral2` function:

```
>> checkcode integral2 -cyc
L 1 (C 14-22): The McCabe complexity of 'integral2' is 16.
L 118 (C 16-23): The McCabe complexity of 'outclass' is 1.
L 127 (C 14-26): The McCabe complexity of 'interiorPoint' is 9.
```

Listing 16.1. *Script badfun.*

```
function [x,y] = badfu(a,b,c)
%BADFUN    Function on which to illustrate checkcode.

if nargin < 3 | isempty(c), c = 1; end
x = sqrt(a^2+b^2)
```

#### CODE COMPLEXITY

A simple measure of the complexity of a MATLAB function is the number of executable lines of source code. The cyclomatic complexity of a function is defined in terms of a directed graph built from the code and turns out to be equal to one plus the number of predicates (logical tests). Smaller values of this measure are thought to correspond to code that is less likely to contain bugs and to be easier to test and maintain. One way to reduce the measure is to split a function into separate, simpler functions.

The companion function `mlintrpt` runs `checkcode` on all the code files in the current directory and reports the results in the MATLAB Web browser, with hypertext links to the file names and line numbers. It can also be called on a single code file. This function can also be invoked from the Reports menu of the dropdown list in the Current Folder browser.

Two code files can be compared with the `visdiff` function, which produces a report in the MATLAB Web browser containing listings of the two code files with differences marked.

## 16.4. Profiling

MATLAB has a profiler that reports, for a given sequence of computations, how much time is spent in each line of each code file and how many times each line is executed, how many times each function is called, and the results of running `checkcode` (see Section 16.3). Profiling has several uses.

- Identifying “hot spots”: those parts of a computation that dominate the execution time. If you wish to optimize the code then you should concentrate on the hot spots.
- Spotting inefficiencies, such as code that can be taken outside a loop.
- Revealing lines in a code that are never executed. This enables you to spot unnecessary code and to check whether your test data fully exercises the code.

To illustrate the use of the MATLAB Profiler, we apply it to the `membrane` function (used on p. 116):

```
profile on
A = membrane(1,250);
profile viewer
profile off
```

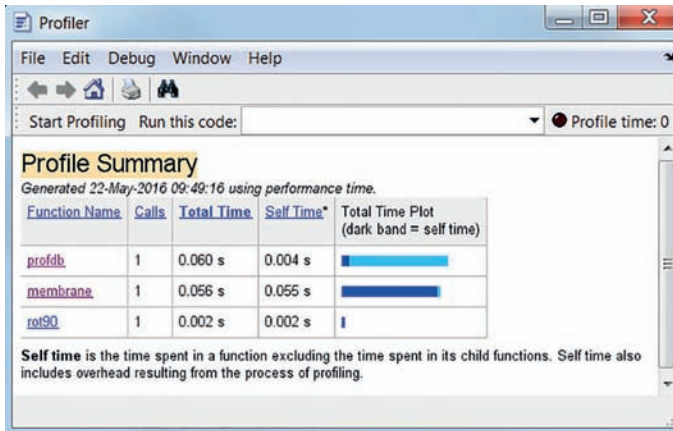


Figure 16.1. profile viewer report for membrane example.

The `profile viewer` command generates an HTML report that is displayed in the Profiler window. Figure 16.1 shows the result. Actually, this is the result from the second run of this code. It is always a good idea to discard the first run, since it includes compilation overheads and MATLAB or system caching overheads. Clicking on the `membrane` link in Figure 16.1 produces Figure 16.2, which shows only the first part of a long report. The profile reveals that `membrane` spends most of its time evaluating Bessel functions. Note also the section toward the bottom of Figure 16.2 headed “Coverage results”. This reveals how many lines of the code were and were not executed. This is useful information, as in testing it is desirable that as much code as possible is exercised by the tests.

Next, consider the script `ops` in Listing 16.2. We profiled the script in order to compare the relative costs of the elementary operations `+`, `-`, `*`, `/` and the elementary functions `sqrt`, `exp`, `sin`, `tan`.

```
profile on
ops
profile viewer
profile off
```

The report is shown in Figure 16.3. The exponential and trigonometric functions are much more costly than the other operations.

## 16.5. P-Code

The `pcode` command creates P-code files, which have a `.p` extension, from code files. A P-code file is functionally equivalent to, and runs at the same speed as, the code file from which it was produced, but it is obfuscated: it is a binary file that is not readable when viewed in an editor. If both P-code and the original code file are present then the P-code takes precedence.

The original code file cannot be reconstructed from the P-code file. In particular, all comments are lost.

P-code is useful when you want to distribute code to others but do not want the recipient to see the source code or to be able to change it. An alternative way to

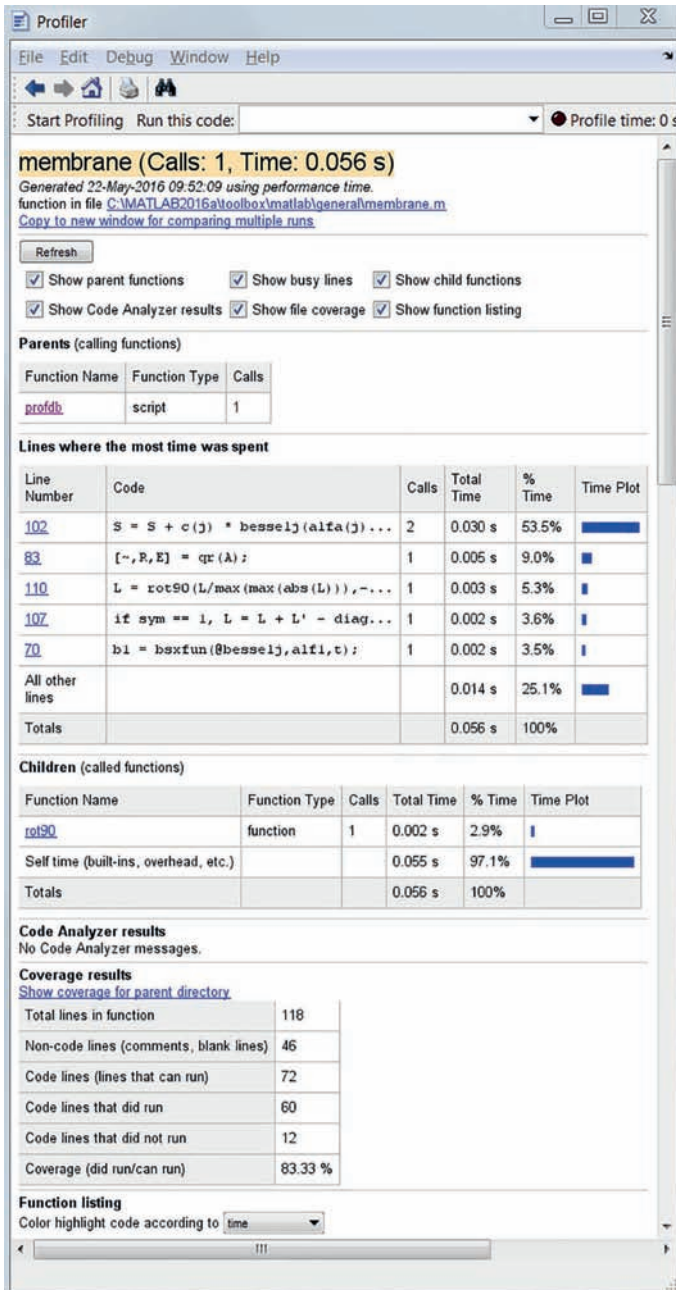


Figure 16.2. More from profile viewer report for membrane example.

Listing 16.2. *Script ops.*

```
%OPS Profile this file to check costs of various elementary ops and funs.

rng(1)
n = 500;
a = 100*rand(n);
b = randn(n);

for i = 1:100
    a+b;
    a-b;
    a.*b;
    a./b;
    sqrt(a);
    exp(a);
    sin(a);
    tan(a);
end
```

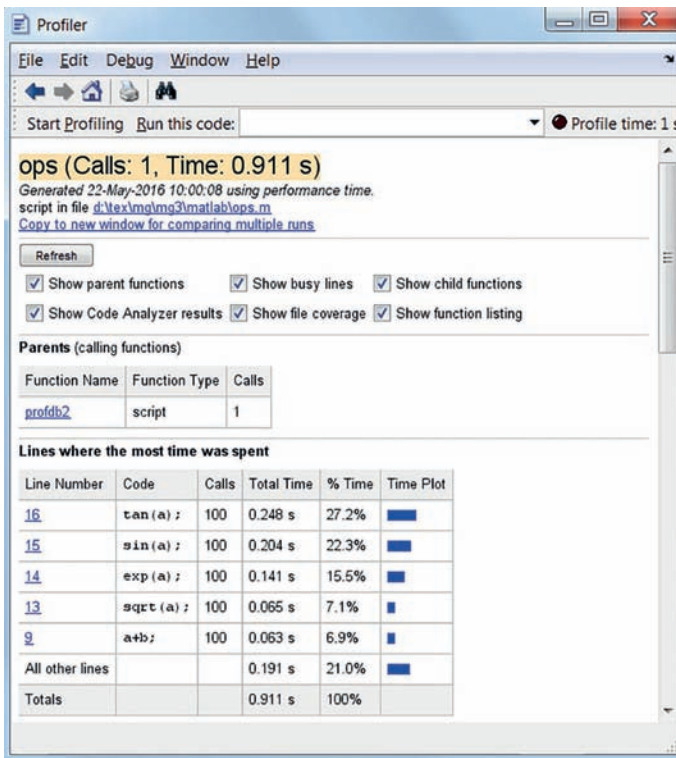


Figure 16.3. profile viewer report for ops example.

protect your source code is to compile it with the MATLAB Compiler and distribute the executable code (which runs independently of MATLAB).

## 16.6. Source Control

It is good practice to use a source control (or version control) system to store snapshots of your code files as you develop them. Such a system allows you to go back to earlier versions of files, perhaps because you realize you have introduced errors and wish to revert to a working version; to annotate what changes were made with each snapshot, thus documenting your progress; and to use branches to manage different lines of development. Source control does not replace backup procedures but does provide a way to recover from unintended changes, such as deletion or overwriting of a file.

MATLAB provides an interface to source control that integrates with the open source Git and Subversion (SVN) systems. It is accessed from the Current Folder browser. However, if you are confident in working at the command line we recommend handling source control outside MATLAB, since you can then do all your source control (MATLAB,  $\LaTeX$ , plain text files, etc.) from the same interface. You do need, however, to install the source control software yourself. As a simple example, using Git at the Windows command line we set up source control for this book in the following way:

```
cd \matlab_guide_3ed
git init
git add *.tex matlab\*.m
git commit -m "First commit."
...
git commit -a -m "Message for second commit."
```

Documentation for Git and SVN is readily available on the web.

## 16.7. Live Editor

The Live Editor is an interactive environment for editing and running MATLAB code that allows you to see your results together with the code that produced them. It supports narrative text, with formatting and  $\LaTeX$  typesetting of equations. The Live Editor works with live scripts, which are files with a `.mlx` extension. You can create a live script from the New menu option on the Home tab of the MATLAB Toolstrip, or by typing (for example) `edit myscript.mlx` at the MATLAB prompt.

Live scripts are broken into sections, each of which can be evaluated and the output inserted at the end of the section. The toolbar on the Live Editor provides tools for inserting code, text, equations, images, and hyperlinks, and for executing either the whole script or sections of it.

A live script can be exported as a regular `.m` file, containing the MATLAB code with the text in comment lines, or as an HTML or PDF file. In the HTML and PDF formats the document closely resembles what is seen in the Live Editor, but the output is frozen in that the code cannot be rerun. These export options are useful for sharing documents on the web and with people who do not have MATLAB, and in particular they are useful in teaching, both for distributing course material and as a way for students to hand in their work.



Listing 16.3. *Script* calculus.m.

```

%% Functions of Several Variables
%% Continuity
% Consider the function
%
% $$$F(x) = F(x_1,x_2) = \frac{x_1x_2}{x_1^2+x_2^2}.$$$
%
% It is continuous away from the origin, but what happens at the origin,
% where $$$ is not defined? We can examine the function graphically, using

fsurf(@(x_1,x_2) x_1.*x_2./(x_1.^2+x_2.^2), [-1 1])
%%
% The surface has large curvature and it needs to be rotated to see what
% is happening at the origin (click the image in the Live Editor to open up a
% figure window, select the Rotate 3D icon, then rotate the image using
% the mouse); in doing so, a sharp drop is observed. To gain more
% insight we examine the contours:

fcontour(@(x_1,x_2) x_1.*x_2./(x_1.^2+x_2.^2), [-1 1])
%%
% Contours join points $(x_1,x_2)$ of equal height $F(x_1,x_2)$. It is clear
% from the contours that $$$ attains many different values close to the origin,
% and we therefore expect that $$$ is not continuous at the origin, no matter
% how $F(0,0)$ is defined.

```

Figure 16.4 shows the first part of a live script as it appears in the Live Editor. Listing 16.3 shows how the same live script looks when exported as a .m file, and Figure 16.5 shows the first page of the exported PDF file.

To a large extent, live scripts remove the need to use the `publish` command that was introduced earlier in the history of MATLAB. The `publish` command takes as argument a code file and exports it to HTML, PDF, Word (.doc), L<sup>A</sup>T<sub>E</sub>X, PowerPoint (.ppt), or XML format. If the code file has not been specially formatted, the output consists of the code file followed by its output. If the code file is broken into cells, each of which begins with a line consisting solely of two percent signs, then the exported document contains code interspersed with text and output. When a live script is exported as a .m file the same cell formatting is generated, as in Figure 16.3. The benefit of the Live Editor is that it provides a WYSIWYG (“what you see is what you get”) interface, so that the user does not need to remember the formatting requirements.

Trefethen’s book [166] was produced using the `publish` command, and the .m files (one per chapter) can be downloaded from the book’s web page.

A kind of inverse to exporting or publishing is the `grabcode` function. Given the name of an exported or published HTML file it extracts the MATLAB code contained in it and opens it in the MATLAB Editor.

## 16.8. Creating a Toolbox

If you develop a number of functions with a common theme that you wish to group together, and possibly distribute to others, you should consider creating a toolbox. A toolbox is simply a collection of MATLAB codes living in the same directory, along

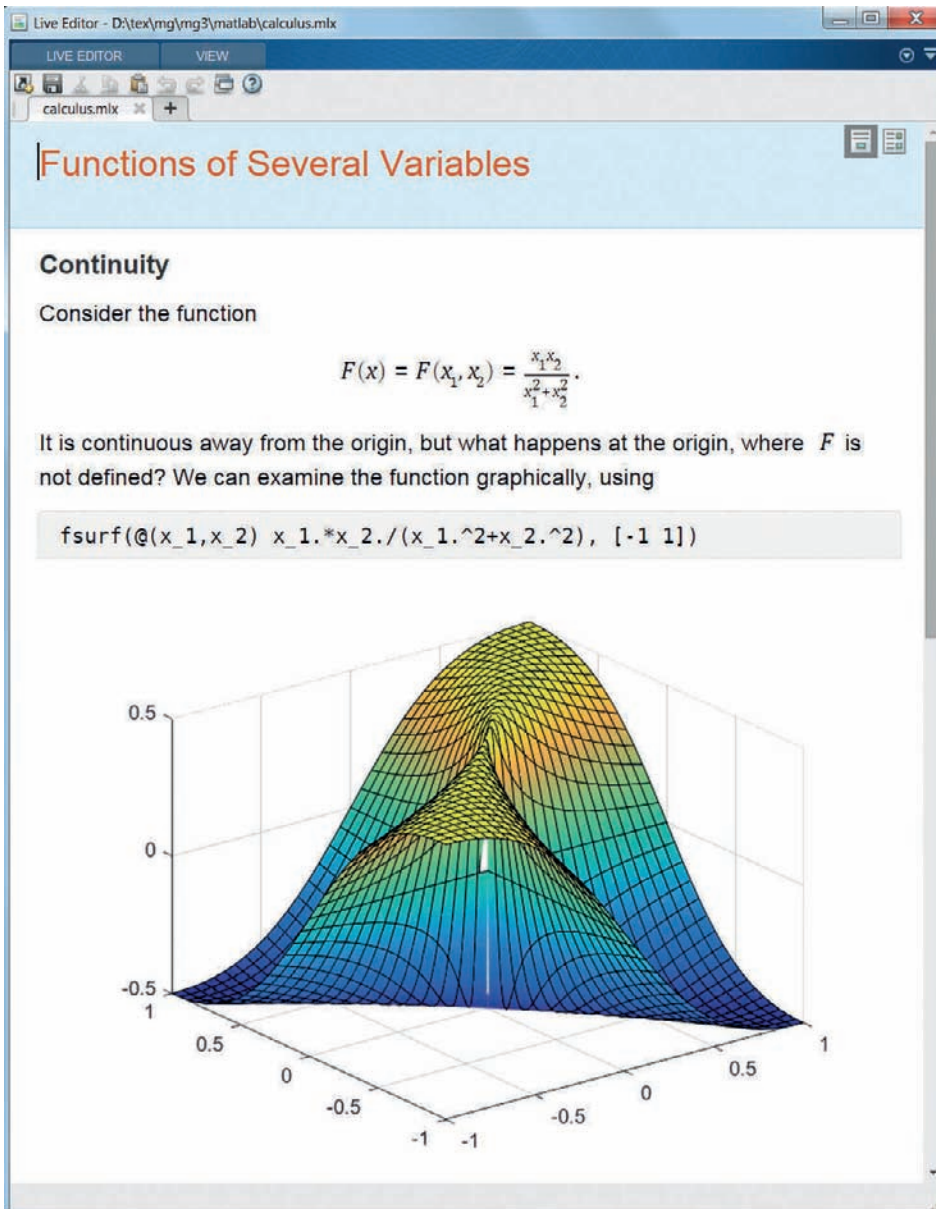


Figure 16.4. *Calculus example in the Live Editor.*

## Functions of Several Variables

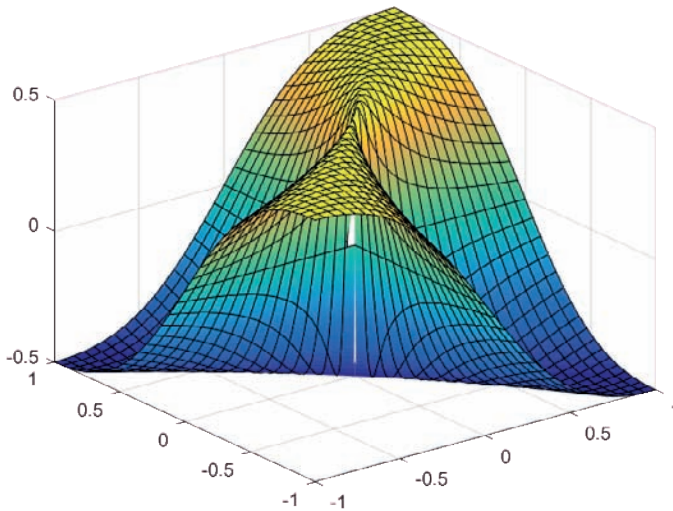
### Continuity

Consider the function

$$F(x) = F(x_1, x_2) = \frac{x_1 x_2}{x_1^2 + x_2^2}.$$

It is continuous away from the origin, but what happens at the origin, where  $F$  is not defined? We can examine the function graphically, using

```
fsurf(@(x_1,x_2) x_1.*x_2./(x_1.^2+x_2.^2), [-1 1])
```



The surface has large curvature and it needs to be rotated to see what is happening at the origin (click the image in the Live Editor to open up a figure window, select the Rotate 3D icon, then rotate the image using the mouse); in doing so, a sharp drop is observed. To gain more insight we examine the contours:

```
fcontour(@(x_1,x_2) x_1.*x_2./(x_1.^2+x_2.^2), [-1 1])
```

Figure 16.5. *First page of PDF file exported from the live script in Figure 16.4.*

with the special files `Contents.m` and, possibly, `readme.m`. The file `Contents.m` is a script of comments containing the names of the code files in the toolbox with a short description of each, and its leading lines give the version and copyright information for the toolbox.

For example, we could create a toolbox in a directory `mytoolbox` on the MATLAB path, with a `Contents.m` file beginning as follows:

```
% Mytoolbox.
% Version 1.1           5-Mar-2017
% Copyright (c) 2017 by A. N. Other
%
% myfun1 - My first useful function.
% myfun2 - My second useful function.
```

Provided the precise format of the first two lines is followed, typing `ver mytoolbox` lists some version information about MATLAB followed by

```
Mytoolbox           Version 1.1
```

Type `doc mytoolbox` or `help mytoolbox` to view `Contents.m`.

A `Contents.m` file can be generated automatically—and an existing file checked against the H1 lines of the code files in the directory—with the `contentsrpt` function. An alternative invocation is by selecting Contents Report from the Reports menu in the dropdown list of the Current Folder browser.

An automated way of packaging a toolbox is provided by Package Toolbox in the Add-Ons menu of the Home tab. A dialog box asks you to fill in information about the toolbox and then a file with a `.mltbx` extension is created. The toolbox can be installed by double-clicking that file.

## 16.9. Distributing Code Files

When you give someone else a code `myfun.m` that you have written, you also need to give them all the codes that it calls that are not provided with MATLAB and you need to tell them which toolboxes (if any) are required. This information can be determined by typing

```
[Mfiles,tlbxs] = matlab.codetools.requiredFilesAndProducts('myfun')
```

which returns a list of the codes that are called by `myfun` or by a function called by `myfun`, and so on, along with a list of required toolboxes. Note that the input to this function must be a string, not a function handle. Another way to obtain this information is with the `inmem` command, which lists all codes that have been parsed into memory. If you begin by clearing all functions (`clear functions`), run the code in question, and then invoke `inmem`, you can deduce which codes have been called. Finally, `deprpt` lists dependencies for all the codes in the current directory, showing the results in a MATLAB Web browser.

All the tools mentioned in this section can also be invoked from the Reports menu of the dropdown list in the Current Folder browser.

An automated way of packaging a function and its dependencies into an App, to appear on the Apps tab, is with Package App in the Apps tab (callable from the

Command Window with function `matlab.apputil.package`). This creates a single file with an `.mlappinstall` extension that allows easy installation.

An excellent way to make your codes available, whether as a toolbox, as individual codes, or as a packaged app, is via MATLAB Central File Exchange. To post files there you will need a MathWorks account. Click on the “Submit a File” button and you can create a submission comprising one or more codes.

An alternative is to put the files in a GitHub repository and give File Exchange the name of the repository; File Exchange will then copy the files across and will update them when the GitHub repository changes. GitHub is a website that hosts Git repositories. The codes from this book are maintained in a Git repository hosted on GitHub, which can be reached via the book’s web page, whose address is given in the Preface.

## 16.10. Unit Tests

Unit testing is a testing approach in which individual units of code (in MATLAB these are typically functions) are tested separately. Ideally, the testing is automated so that the tests can be run repeatedly as code is developed. MATLAB has a powerful unit testing framework for creating and running suites of tests. It can also run a suite of tests in parallel using the Parallel Computing Toolbox.

We give just one simple example of unit testing. Script `test_acos` in Listing 16.4 tests the MATLAB `acos` function to see whether it correctly returns values of the principal inverse cosine. The test script is broken up into sections separated by lines beginning with `%;`; each section contains independent tests that may call on shared variables set up in the opening lines up to the first line that begins `%%`. The script makes use of the `assert` function described in Section 14.1. Most of the tests are self-explanatory given the definition of the principal inverse cosine [4]. The tolerance in the final test includes a term that accounts for the growth of errors in the evaluation of the test and is justified in [30].

We could simply run `test_acos` and see whether it runs to completion or terminates with `Assertion failed`. The problem is that it will stop on the first failed assertion, whereas it is more useful if all the tests are run and a list of failed assertions is produced. This is achieved using the `runtests` function:

```
>> results = runtests('test_acos.m')
Running test_acos
.....
Done test_acos
-----

results =
    1×5 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
```

```
5 Passed, 0 Failed, 0 Incomplete.
0.02791 seconds testing time.
```

All the tests have passed. We now introduce a test that must fail by adding

```
assert(acos(1) == 1, 'This acos(1) test should fail!')
```

at the end of the Branch points block. Here, we have used the second argument of `assert` to provide an error message. The output is now

```
>> results = runtests('test_acos.m')
Running test_acos
...
=====
Error occurred in test_acos/BranchPoints and it did not run to completion.

-----
Error ID:
-----
''

-----
Error Details:
-----
Error using test_acos (line 33)
This acos(1) test should fail!
=====
..
Done test_acos
-----

Failure Summary:

Name                Failed  Incomplete  Reason(s)
=====
test_acos/BranchPoints  X        X        Errored.

results =
1x5 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
4 Passed, 1 Failed, 1 Incomplete.
0.033807 seconds testing time.
```

The offending block is identified by the name that follows the `%%` (if no name is given, MATLAB will introduce its own numbering of tests, which can be difficult to match to the block in question).

Listing 16.4. *Script test\_acos.*

```
%TEST_ACOS Test acos function.
% Test that f(z) = acos(z) is the principal arc cosine function.

% Setup
rng(1)
n = 1e4;
x1 = linspace(-1,1,n);
x2 = linspace(-1,-1e4,n);

%% Interval 1
% For real x on [-1,1], f(x) must be real and between -pi and pi.
y = acos(x1);
assert(all(isreal(y)))
assert(all(y >= -pi))
assert(all(y <= pi))

%% Interval 2
% Test x on branch cut [-inf,-1].
y = acos(x2);
assert(all(real(y) == pi))
assert(all(imag(y) <= 0))

%% Interval 3
% Test x on branch cut [1,inf].
y = acos(-x2);
assert(all(real(y) == 0))
assert(all(imag(y) >= 0))

%% Branch points
% Test values at branch points.
assert(acos(1) == 0)
assert(acos(-1) == pi)

%% Complex random arguments
x = randn(n,1) + 1i*randn(n,1);
acosx = acos(x);
x1 = cos(acosx);
assert(all( abs(x-x1) <= 2*eps*(abs(x) + abs(acosx.*sin(acosx))) ))
```

This example gives just a hint of the capabilities of the unit test framework, which stem from the powerful object-oriented features that are not used here. See `doc testing frameworks` for the details.

A performance testing framework is available for automating the testing of the speed of codes. See the function `runperf` and `doc testing frameworks`.

*If we wish to count lines of code,  
we should not regard them as lines produced but as lines spent.*

— EDSGER W. DIJKSTRA, *On the Cruelty of Really Teaching Computing Science* (1998)

*Readability of code is now my first priority.  
It's more important than being fast, almost as important as being correct,  
but I think being readable is actually the most likely way of making it correct.*

— DOUGLAS CROCKFORD, in *Coders At Work* (2009)

*Experienced MATLAB quality engineers . . . routinely write test cases with  
empty input matrices or NaN and Inf values. . .  
I anticipate possible coding errors for  
constant-valued images or those that have a single row or column.*

— STEVEN L. EDDINS, *Automated Software Testing for MATLAB* (2009)

*I've become convinced that all compilers written from now on  
should be designed to provide all programmers with feedback indicating  
what parts of their programs are costing the most.*

— DONALD E. KNUTH, *Structured Programming with go to Statements* (1974)

*Instrument your programs.  
Measure before making "efficiency" changes.*

— BRIAN W. KERNIGHAN and P. J. PLAUGER,  
*The Elements of Programming Style* (1978)

*Arnold was unhappily aware that the complete Jurassic Park program contained  
more than half a million lines of code,  
most of it undocumented, without explanation.*

— MICHAEL CRICHTON, *Jurassic Park* (1990)



# Chapter 17

## Advanced Graphics

The graphics functions described in Chapter 8 can produce a wide range of output and are sufficient to satisfy the needs of many MATLAB users. These functions are part of an object-oriented graphics system that provides full control over the way MATLAB displays data. A knowledge of graphics objects is useful if you want to fine-tune the appearance of your plots, and it enables you to produce displays that are not possible with the existing functions. This chapter provides an introduction to graphics objects. More information can be found in the MATLAB documentation.

### 17.1. Objects, Handles, and Properties

Graphs are built from objects organized in a hierarchy, as shown in Figure 17.1. The Root object corresponds to the whole screen. Underneath the Root are Figure objects, corresponding to figure windows, and under them are Axes objects. An Axes object is a region of the figure window with a coordinate system, in which other objects such as lines and text are displayed.

Each specific instance of an object has a unique identifier called a handle. To illustrate, consider the simple example

```
>> h = plot(1:10,'o-')
```

This produces the left-hand plot in Figure 17.2. Now we look at the handle, `h`:

```
>> h
h =
  Line with properties:
      Color: [0 0.4470 0.7410]
  LineStyle: '-'
  LineWidth: 0.5000
      Marker: 'o'
  MarkerSize: 6
  MarkerFaceColor: 'none'
      XData: [1 2 3 4 5 6 7 8 9 10]
      YData: [1 2 3 4 5 6 7 8 9 10]
      ZData: [1×0 double]
```

Show all properties

In the Command Window, all properties in the last line is a hyperlink that, when clicked, shows a number of additional properties. Another way to obtain the list of all properties is to type `get(h)`. To see what type of variable `h` is, we look at its class:

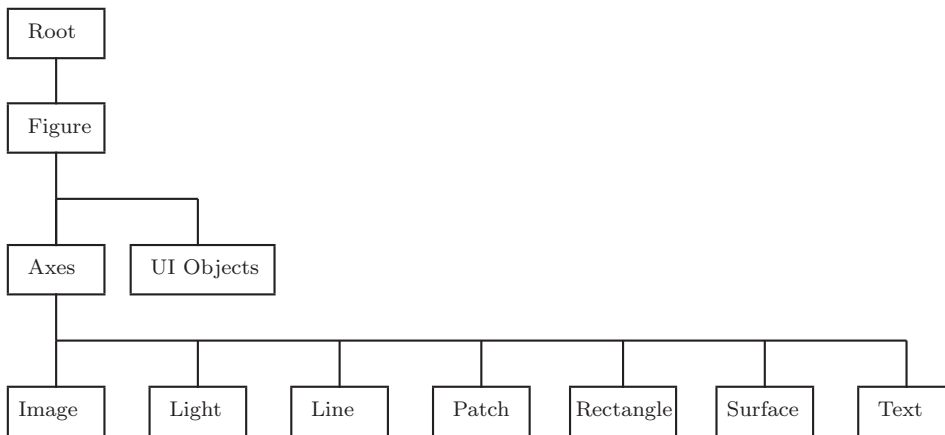


Figure 17.1. *Hierarchical structure of graphics objects (simplified).*

```
>> class(h)
matlab.graphics.chart.primitive.Line
```

This full class name shows that `h` is a handle to an object referencing a line. We can confirm that `h` is a handle to a graphics object using `isgraphics`:

```
>> isgraphics(h)
ans =
    logical
     1
```

If we close the graphics window, the variable `h` still exists but no longer references a valid graphics object:

```
>> close
>> isgraphics(h)
ans =
    logical
     0
```

A handle provides access to the various properties of an object that govern its appearance, which can be assigned or viewed using the dot notation. We now repeat the above plot and change some of the line properties:

```
h = plot(1:10,'o-');
h.LineWidth = 1.5;
h.MarkerSize = 8;
h.MarkerFaceColor = 'red';
```

The result is in the right-hand plot in Figure 17.2.

Tab completion works with graphics object properties: if you type `h` followed by a period and hit tab then a window pops up with a list of completions.

The previous plot could have been obtained by providing the appropriate property name–value pairs in the `plot` statement, as we did in Chapter 8. One advantage of

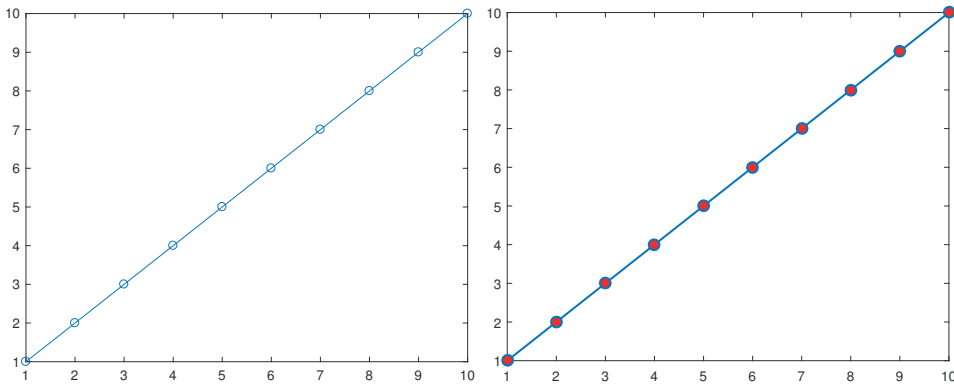


Figure 17.2. *Left: original. Right: modified via the line object.*

the latter approach is that property names are case insensitive ('LineWidth' and 'linewidth' have the same effect), whereas with the dot notation the exact case must be used (`h.LineWidth`). However, the object system is much more powerful, as we will see.

Another way to assign properties to a graphics object is with the `set` command (which predates the more recent dot notation). For example, the line width assignment in the previous example could equally well be written

```
set(h, 'LineWidth', 1.5);
```

A useful feature of `set` is that it can return all possible values for a property:

```
>> set(h, 'LineStyle');
    '-'
    '--'
    ':'
    '-.'
    'none'
```

Now consider the following code, which produces Figure 17.3:

```
x = linspace(0, 2*pi, 35);

a1 = subplot(2,1,1);           % Axes object.
l1 = plot(x, sin(x), 'x');     % Line object.
title('Sine curve')

a2 = subplot(2,1,2);           % Axes object.
l2 = plot(x, cos(x).*sin(x));  % Line object.
tx2 = xlabel('x'); ty2 = ylabel('y'); % Text objects.
```

When the following code is executed it modifies properties of objects to produce Figure 17.4:

```
a1.Box = 'off';               % box off.
a1.XTick = [];
```

```

a1.YAxisLocation = 'right';
a1.TickDir = 'out';
l1.Marker = '<';
a1.YGrid = 'on';
% Set dotted gridlines and make them more prominent:
a1.GridLineStyle = ':';
a1.GridColor = 0*[1 1 1];
a1.GridAlpha = 1;
a1.TitleFontSizeMultiplier = 1;

a2.Position = [0.2 0.15 0.65 0.35];
a2.XLim = [0 2*pi]; % xlim([0 2*pi]).
a2.FontSize = 14;
a2.XTick = [0 pi/8 pi/4 pi/2 pi 2*pi];
% Or xticks(...) for previous line and xticklabels(..) for the next.
a2.XTickLabel = {'0', '\pi/8', '\pi/4', '\pi/2', '\pi', '2\pi'};
a2.XScale = 'log';
a2.LabelFontSizeMultiplier = 1.5; % For x- and y-axis labels.
l2.LineWidth = 6;
tx2.FontAngle = 'italic'; ty2.Rotation = 0;
ty2.FontAngle = 'italic';

```

Some of the effects of these `set` commands can be produced using commands discussed in Chapter 8, as indicated in the comments, or by appending property name–value pairs to argument lists of `plot` and `text`. For example, `box off` can be used in place of `a1.Box = 'off'` provided that the first Axes object is current. However, certain effects, such as changing the tick direction, can be conveniently achieved only by using the objects.

The properties altered here are mostly self-explanatory. We mention some exceptions. The `Position` property of Axes is specified by a vector of the form `[left bottom width height]`, where `left` and `bottom` are the distances from the left edge and bottom edge, respectively, of the figure window to the bottom left corner of the Axes rectangle, and `width` and `height` define the dimensions of the rectangle. The units of measurement are defined by the `Units` property, whose default is `normalized`, which maps the lower left corner of the figure window to  $(0, 0)$  and the upper right corner to  $(1.0, 1.0)$ . By default, titles are displayed in bold and at a font size 10% bigger than the current `FontSize` setting; the `TitleFontSizeMultiplier` property of the Axes object is a scale factor that controls this relation and here we change it from its default value of 1.1. Similarly, the `LabelFontSizeMultiplier` property of the Axes object controls the scaling of the fonts used for the  $x$ -,  $y$ -, and  $z$ -axis labels.

An object can be deleted by passing its handle to the `delete` function. Thus `delete(l1)` removes the sine curve from the top plot in Figure 17.4 and `delete(tx2)` removes the  $x$ -axis label from the bottom plot.

Generally, if you plan to change the properties of an object after creating it then you should save the handle when you create it, as in the example above. However, the handles of the current Axes, Figure, and object can be retrieved using `gca`, `gcf`, and `gco`, respectively. In the following example we check the current and possible values of the `FontWeight` property for the current Axes and then change the property to bold:

```
>> get(gca, 'FontWeight')
```

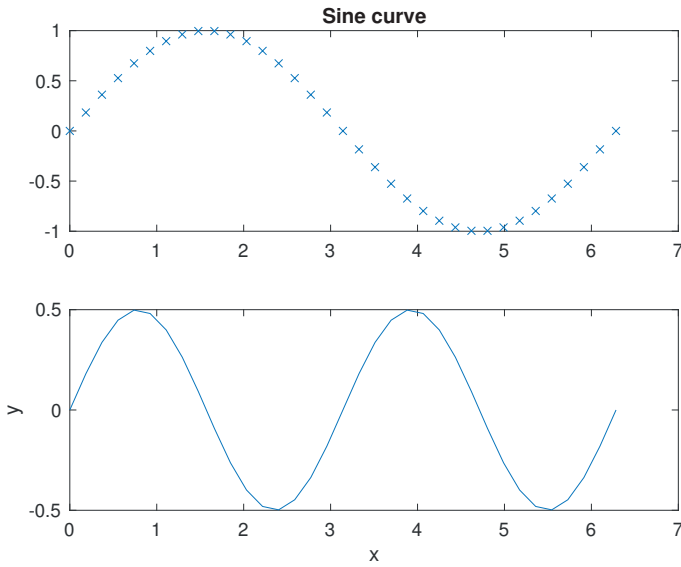


Figure 17.3. *Straightforward use of subplot.*

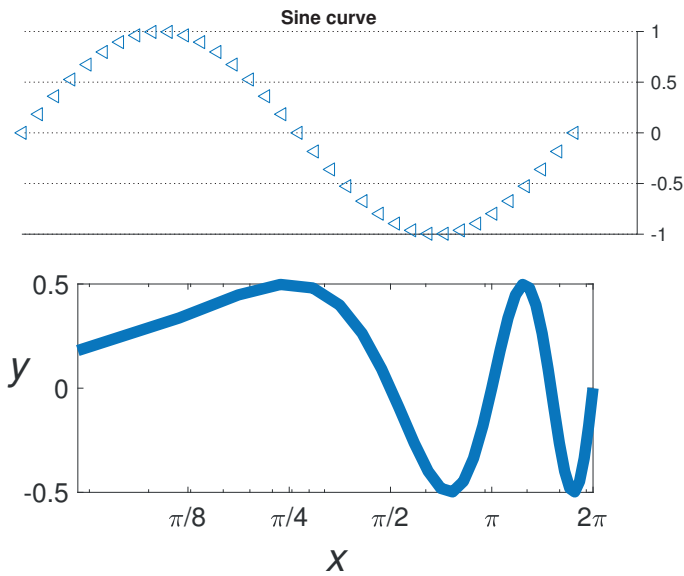


Figure 17.4. *Modified version of Figure 17.3 postprocessed by modifying object properties.*

```

ans =
normal

>> set(gca,'FontWeight')
    2×1 cell array
    'normal'
    'bold'

>> set(gca,'FontWeight','bold')

```

The “current object” whose handle is returned by `gco` is the object last clicked on with the mouse. Thus if we want to change the marker to '\*' for the curve in the upper plot of Figure 17.4 we can click on the curve and then type

```
>> set(gco,'Marker','*')
```

In addition to setting graphics properties from the command line or in codes it is possible to set them interactively. A particular graphic object can be edited by first enabling plot editing—by clicking on the Edit Plot icon in the figure window toolbar, or selecting Tools-Edit Plot, or typing `plottedit` in the Command Window—and then double-clicking on the object. Experimenting with plot edit mode is an excellent way to learn about graphics objects.

The importance of the hierarchical nature of the graphics system is not completely apparent in the simple examples described above. A particular object, say the Root, contains the handles of all its children, which makes it possible to traverse the tree structure, using `get(h,'Children')`, `get(h,'Parent')`, and the `findobj` and `findall` functions. Furthermore, it is possible to set default values for properties, and if these are set on a particular Axes, for example, they are inherited by all the children of that Axes. Some of these aspects are described in the following sections.

The GUI tools are beyond the scope of this book (type `help uitools` for a list of the relevant functions). However, we mention one GUI function that is of broad interest: `waitbar` displays a graphical bar in a window that can be used to show the progress of a computation. Its usage is illustrated by (see also Listing 1.4)

```

h = waitbar(0,'Computing...')
for j = 1:n
    % Some computation ...
    waitbar(j/n) % Set bar to show fraction j/n complete.
end
close(h)

```

## 17.2. Root and Default Properties

The handle of the Root object is returned by `groot`. The assignable root properties can be determined using `get(groot)`.

Typing `get(groot,'Factory')` returns in a structure all the factory-defined values of all user-settable properties, of which there are over 1600. The default value for the object property *ObjectTypePropertyName* can be obtained with the command `get(groot,'DefaultObjectTypePropertyName')`. For example:

```
>> get(groot, 'DefaultLineMarkerSize')
ans =
     6
```

Typing `set(groot, 'DefaultObjectTypePropertyName')` sets the default value for *ObjectTypePropertyName*, and since the root handle is specified this default applies to all levels in the hierarchy. This command is useful for setting default property values at the start of a session. For example, before giving a presentation with a data projector we might type

```
set(groot, 'defaulttextfontsize', 14)
set(groot, 'defaultaxesfontsize', 14)
set(groot, 'defaultlinemarkersize', 10)
set(groot, 'defaultlinelinewidth', 2)
set(groot, 'defaulttextfontweight', 'bold')
set(groot, 'defaultaxesfontweight', 'bold')
```

in order to make the MATLAB graphics more readable to the audience. (It obviously makes sense to create a code file containing these commands.) On the other hand, before generating PDF figures for inclusion in a paper we might type

```
set(groot, 'defaulttextfontsize', 12)
set(groot, 'defaultaxesfontsize', 12)
set(groot, 'defaultlinemarkersize', 8)
set(groot, 'defaultlinelinewidth', 1)
```

The advantage of these commands is that they make it unnecessary to append modifiers such as `'FontSize', 12` to every `title`, `xlabel`, and so on.

The factory settings can be restored with commands of the form

```
set(groot, 'defaultlinelinewidth', 'factory')
```

To reset all properties to the factory values type

```
reset(groot)
```

### 17.3. Animation

Two types of animation are possible in MATLAB. A sequence of figures can be saved and then replayed as a movie, and an animated plot can be produced by manipulating the `XData`, `YData`, and `ZData` properties of objects. We give one example of each type.

To create a movie, you draw the figures one at a time, use the `getframe` function to save each one as a pixel snapshot in a structure, and then invoke the `movie` function to replay the figures. Here is an example:

```
clear % Remove existing variables.
Z = peaks; surf(Z)
axis tight manual % Manual to freeze axis between frames.
set(gca, 'nextplot', 'replacechildren')
disp('Creating the movie...')
n = 25;
F(n+1) = struct('cdata', [], 'colormap', []); % Preallocate struct.
```

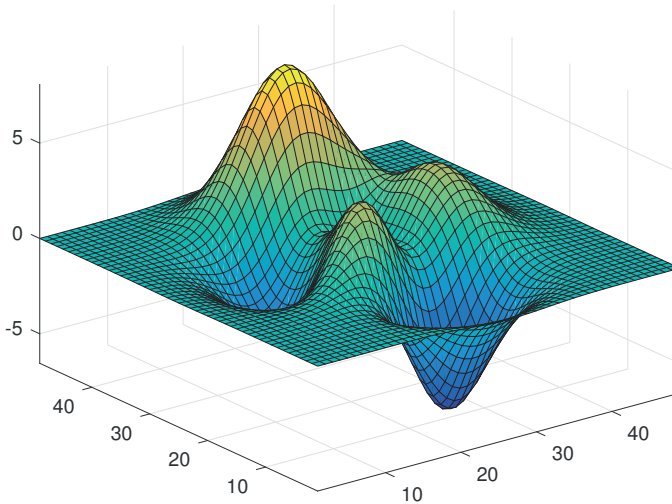


Figure 17.5. *One frame from a movie.*

```

for j = 1:n+1
    surf(cos(2*pi*(j-1)/n).*Z,Z)
    F(j) = getframe;
end
disp('Playing the movie 2 times ...')
movie(F,2)

```

Figure 17.5 shows one intermediate frame from the movie. The `set` command causes all `surf` plots after the first to leave the Axes properties, such as `axis tight` `manual` and the grid lines, unaltered. The movie is replayed `p` times with `movie(F,p)`.

The code above can be modified to create a movie viewable outside MATLAB as follows. Before the `for` loop, set up a `VideoWriter` object, then open the file, with

```

v = VideoWriter('mymovie','MPEG-4');
open(v)

```

Now write frames using

```
writeVideo(v,F(j))
```

within the loop. Finally, close the file with

```
close(v)
```

at the end of the code. The file `mymovie.mp4` is created in the current directory. The `VideoWriter` function supports a variety of output formats.

The second type of animation is most easily obtained using the functions `comet` and `comet3`. They behave like limited versions of `plot` and `plot3`, differing in that the plot is traced out by a “comet” consisting of a head (a circle), a body (in one color), and a tail (in another color). For example, try

```

x = linspace(-2,2,500);
y = exp(x).*sin(1./x);
comet(x,y)

```



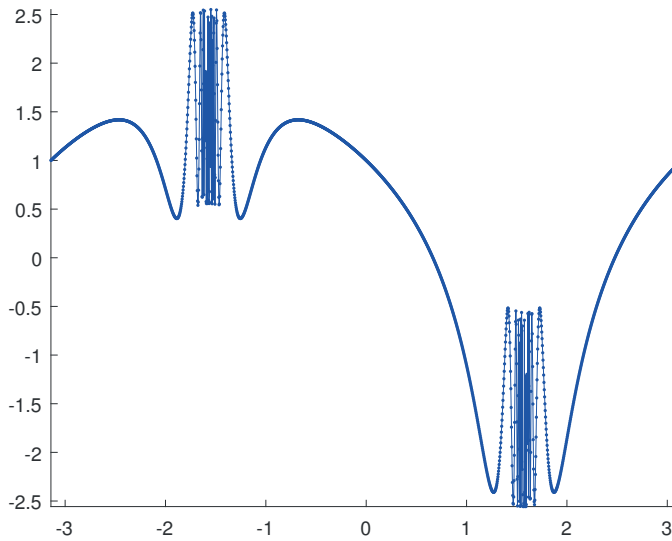


Figure 17.6. *Animated figure upon completion.*

We give a simple example to illustrate the principle used by `comet`. This example can be adapted for use in situations in which the data must be plotted as it is generated, as when solving a differential equation, for example (see the MATLAB demonstration function `lorenz`, mentioned on p. 12):

```
x = linspace(-pi,pi,2000);
y = cos(tan(x)) - tan(sin(x));
h = animatedline('Marker','.', 'Color','b','MarkerSize',5);
axis([min(x) max(x) min(y) max(y)])
for k = 1:length(x)
    addpoints(h,x(k),y(k))
    drawnow
end
```

This code creates an `AnimatedLine` object then adds points to it using `addpoints` within the loop. The points accumulate to produce the final plot shown in Figure 17.6. The `drawnow` command is necessary in order for the figure to be updated on the screen after each call to `addpoints`, and it has options that control the speed of the updates.

## 17.4. Examples

In this section we give some practical examples of how to manipulate graphics objects in order to produce customized graphics.

Suppose that you wish to use a nonstandard font size (say, 16 point) throughout a Figure object. Explicitly setting the `FontSize` property for each Text object and each Axes is tedious. Instead, after creating the figure, you can type

```
h = findall(gcf,'type','text'); set(h,'FontSize',16)
h = findall(gcf,'type','axes'); set(h,'FontSize',16)
```

Note that using `findobj` in the first line would not produce any change to the `xlabel`, `ylabel`, or `title`. The reason is that these text objects are created with the `HandleVisibility` property set to `off`, which makes them invisible to `findobj`, but not to `findall`. (Look at the code with `type findall` to gain some insight.) For this reason, however, `findall` should be used with caution as it may expose to view handles that have intentionally been hidden by an application, so manipulating the corresponding objects could produce strange results.

The automatic choices of tick marks and axis limits are not always the most appropriate. The upper plot in Figure 17.7 shows the relative distance from IEEE single-precision numbers  $x \in [1, 16]$  to the next larger floating-point number. The tick marks on the  $x$ -axis do not emphasize the important fact that interesting changes happen at a power of 2. The lower plot in Figure 17.7 (which is essentially [70, Fig. 2.1]) differs from the upper one in that the following commands were appended:

```
set(gca,'xtick',[1 2 4 8 16])
set(gca,'TickLength',[0.02 0.025])
set(gca,'FontSize', 12)
set(findall(gcf,'type','line'),'LineWidth',1.25)
```

The first `set` command specifies the location of the ticks on the  $x$ -axis and the second increases the length of the ticks (to 0.02 for 2D plots and 0.025 for 3D plots, in units normalized relative to the longest of the visible  $x$ -,  $y$ -, or  $z$ -axis lines). The third command sets a 12-point font size for the tick labels and  $x$ -axis label. The final command increases the thickness of the line; this would be better done using a handle to the line if that is available.

The next example illustrates the use of a cell array (see Section 18.7) to specify the `YTickLabel` data and the `YDir` property to reverse the order of the  $y$ -axis values. The script file in Listing 17.1 produces Figure 17.8, which shows the most frequently used words of four letters or more, and their frequencies of occurrence, in a draft of this book.

The script `cheb3plot` in Listing 17.2 plots seven Chebyshev polynomials in three dimensions, producing Figure 17.9 (this plot reproduces part of one in [46, Fig. A-1]). The script uses the `cheby` function from Listing 7.4. Note the use of the `DataAspectRatio` Axes property to set the aspect ratios of the axes, and the adjustment of the  $x$ - and  $y$ -axis labels, which by default are placed rather noncentrally.

Two different axes can be superimposed, using the left  $y$ -axis for one set of data and the right  $y$ -axis for another. The script `garden` in Listing 17.3 uses the `yyaxis` function to set up and select between two axes, and produces Figure 17.10. The comments in the code explain how it works.

The final example illustrates how diagrams, as opposed to plots of data or functions, can be generated. The script `sqrt_ex` in Listing 17.4 produces Figure 17.11. It uses the `line` function, which is a low-level routine that creates a line object in the current Axes. Several of the higher-level graphics routines make use of `line`. Somewhat oxymoronically, the script also uses the `rectangle` function to draw a circle. The `Position` property of `rectangle` is a vector `[x y w h]` that specifies a rectangle of width `w` and height `h` with bottom left corner at the point `x, y`, all in Axes data units. The `Curvature` property determines the curvature of the sides of the rectangle, with extremes `[0 0]` for square sides and `[1 1]` for an ellipse. The `HorizontalAlignment` and `VerticalAlignment` text properties have been used to help position the text.

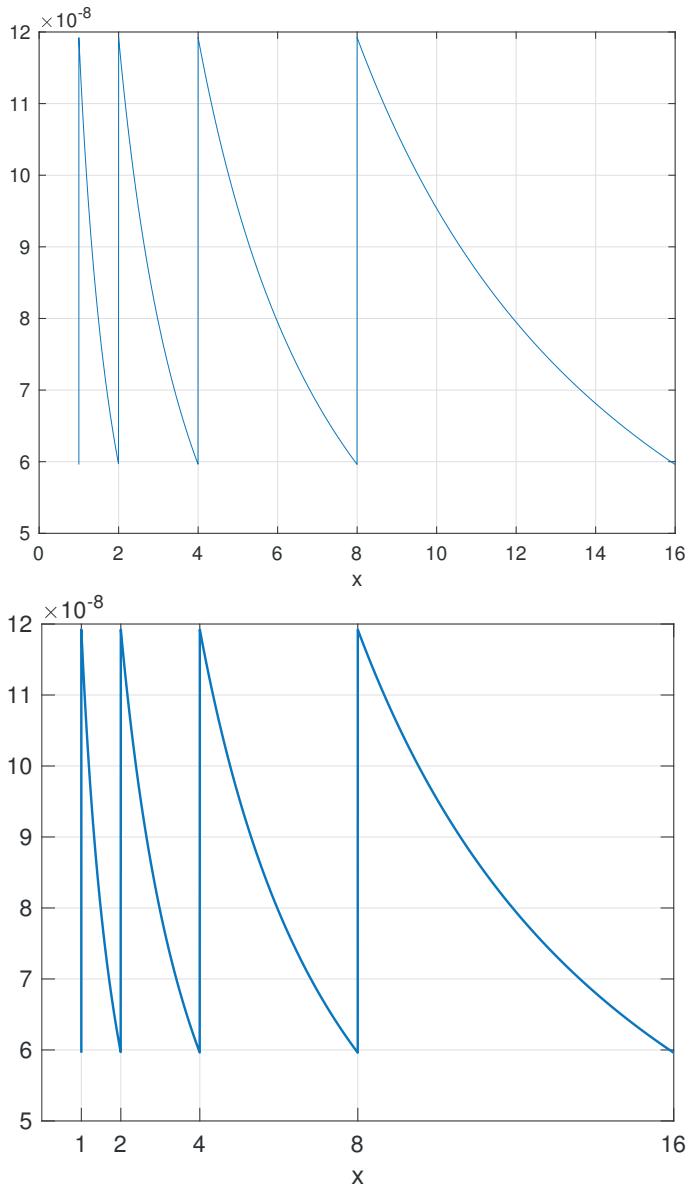


Figure 17.7. Plot with default (upper) and modified (lower) settings.

Listing 17.1. *Script wfreq.*

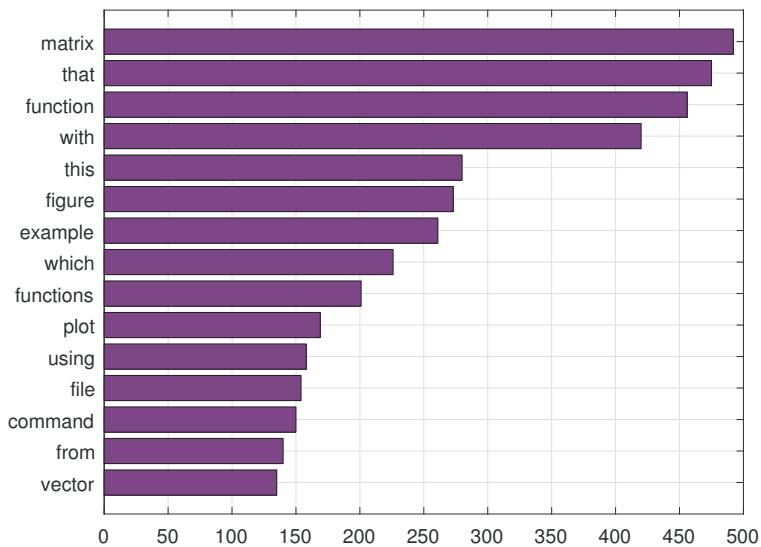
```

%WFREQ

% Cell array z stores the data:
z = {492, 'matrix'
     475, 'that'
     456, 'function'
     420, 'with'
     280, 'this'
     273, 'figure'
     261, 'example'
     226, 'which'
     201, 'functions'
     169, 'plot'
     158, 'using'
     154, 'file'
     150, 'command'
     140, 'from'
     135, 'vector'};

% Draw bar graph of first column of z.  cat converts to column vector.
h = barh(cat(1,z{:},1));
n = length(z);
h.FaceColor = [.5 .25 .5]; % RGB values giving a purple.
set(gca,'YTickLabel',z(:,2))
set(gca,'YDir','reverse') % Reverse order of y-values.
ylim([0 n+1])
grid

```

Figure 17.8. *Word frequency bar chart created by script wfreq.*

Listing 17.2. *Script cheb3plot.*

```

%CHEB3PLOT

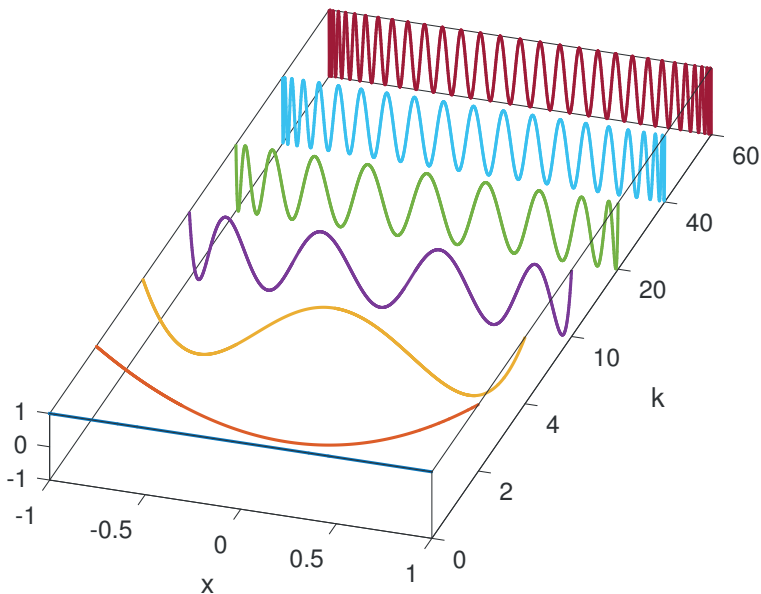
y = linspace(-1,1,1500)';
Z = cheby(y,61);

k = [0 2 4 10 20 40 60];
z = Z(:,k+1);

for j = 1:length(k)
    plot3(j*ones(size(y)),y,z(:,j),'LineWidth',1.5);
    hold on
end
hold off
box on
set(gca,'DataAspectRatio',[1 0.75 4]) % Change shape of box.
view(-72,28)
set(gca,'XTickLabel',k)
set(gca,'YDir','reverse') % Needed because our x-axis is MATLAB y-axis.
set(gca,'BoxStyle','full') % Put box around all sides.

% Labels, with adjustment of position.
hx = xlabel('k'); hx.Position = hx.Position + [1.5 0.1 0];
hy = ylabel('x'); hy.Position = hy.Position + [0 -0.25 -0.25];
set(gca,'FontSize',12)

```

Figure 17.9. *Selected Chebyshev polynomials  $T_k(x)$  on  $[-1, 1]$ , created by script cheb3plot.*

Listing 17.3. *Script garden.*

```

%GARDEN

months = {'June','July','August','September','October'}; % Rows.
vegetables = {'Lettuce','Green Beans','Potatoes',...
              'Swiss Chard','Pumpkins'}; % Columns.
Y = [0.4 0.3 0.0 0.0 0.0
     0.6 0.4 0.0 0.0 0.0
     0.7 0.8 0.3 0.2 0.0
     0.6 0.5 0.9 0.4 1.1
     0.4 0.4 0.7 0.6 1.6];

t = [19.5 21 24 21 18]; % Temperature.

% Make sure y-axes are black; by default they use axes color order.
fig = figure;
set(fig,'defaultAxesColorOrder',[0 0 0; 0 0 0]);

yyaxis left
b = bar(Y,'stacked');
ylabel('Yield (kg)'), ylim([0 4])

%           RGB values
b(1).FaceColor = [0.5 1.0 0.25]; % Light green (tuned for printing).
b(2).FaceColor = [0 0.6 0]; % Mid green.
b(3).FaceColor = [0.9 0.9 0]; % Mid yellow.
b(4).FaceColor = [0.75 0 0]; % Mid red.
b(5).FaceColor = [1 0.5 0]; % Orange.

set(gca,'XTickLabel',months)

% Create a second axis at same location as first and plot to it.
yyaxis right
p = plot(t,'Marker','square','MarkerSize',10,'LineStyle','-','...
        'LineWidth',2,'MarkerFaceColor',[.6 .6 .6]);

ylabel('Degrees (Celsius)')
t = title('Fran''s vegetable garden','FontSize',14);
% Raise title slightly to avoid clash with marker.
t.Position = t.Position + [0 0.05 0];

% Make second y-axis tick marks line up with those of first.
h2 = gca;
ylimits = h2.YLim;
yinc = (ylimits(2)-ylimits(1))/4;
h2.YTick = [ylimits(1):yinc:ylimits(2)];

% Give legend the Axes handles and place top left.
legend([b,p],vegetables{:},'Temperature','Location','NW')

```

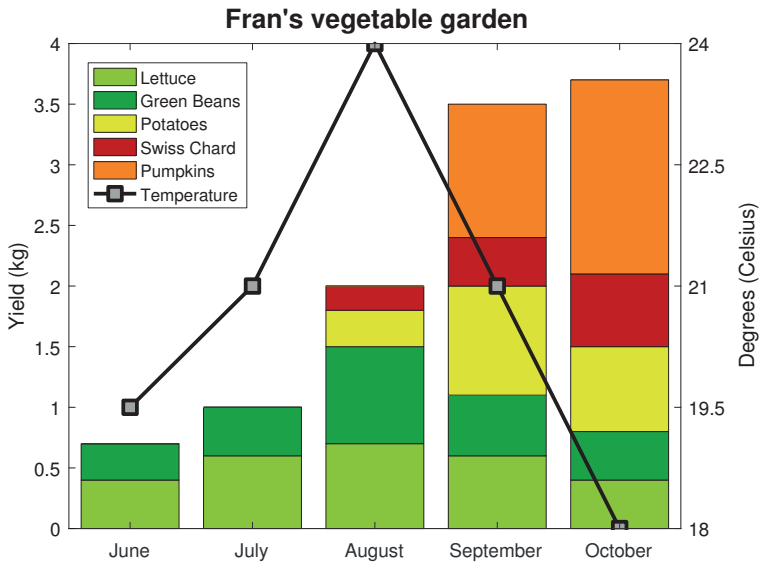


Figure 17.10. Example with superimposed axes created by script garden.

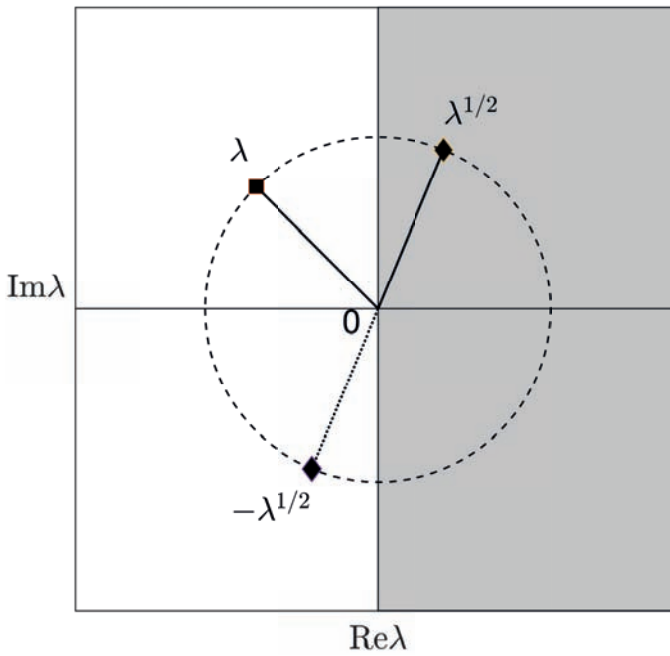


Figure 17.11. Diagram created by sqrt\_ex.

Listing 17.4. *Script sqrt\_ex.*

```

%SQRT_EX
% Script plotting a point on the unit circle and its two square roots,
% with the right half-plane shaded.

z = -1 + 1i; z = z/abs(z);           % Point z on unit circle.
s = sqrt(z);

% Create Axes with specified range.
a = 1.75;
h = axes('XLim',[-a a], 'YLim',[-a a]);

fill([0 a a 0],[-a -a a a],[.8 .8 .8]) % Shade right half-plane.
hold on

options1 = {'MarkerSize',8,'MarkerFaceColor','black'};
options2 = {'Color','k','LineWidth',1};
plot(z,'s',options1{:})
line([0 real(z)],[0 imag(z)],options2{:})
plot(s,'d',options1{:})
line([0 real(s)],[0 imag(s)],options2{:})
plot(-s,'d',options1{:})
line([0 -real(s)],[0 -imag(s)],'LineStyle',':',options2{:})

% Unit circle.
rectangle('Position',[-1,-1,2,2],'Curvature',[1,1],'LineStyle','--',...
          'LineWidth',0.75)
axis square tight

% Draw x- and y-axes through origin.
plot([-a a], [0 0], '-k'), plot([0 0], [-a a], '-k')
set(h,'XTick',[], 'YTick',[])

options = {'Interpreter','latex'};
xlabel('\mathrm{Re} \lambda',options{:})
ylabel('\mathrm{Im} \lambda','Rotation',0,'HorizontalAlignment',...
       'right', options{:})

text(real(z)-0.1,imag(z)+0.2,'\lambda','HorizontalAlignment','center',...
     options{:})
text(-0.1,0.05,'0','HorizontalAlignment','right','VerticalAlignment','top')
text(real(s),imag(s)+0.2,'\lambda^{1/2}',options{:})
text(-real(s),-imag(s)-0.25,'\lambda^{1/2}', ...
     'HorizontalAlignment','right',options{:})
hold off

% Reset FontSize for all text.
g = findall(gca,'type','text'); set(g,'FontSize',14)

```



*Words with most meanings in the Oxford English Dictionary:*

1. set
- ⋮
6. get

— RUSSELL ASH, *The Top 10 of Everything* (1994)

*Handle Graphics ...*

*allows you to display your data and then  
"reach in" and manipulate any part of the image you've created,  
whether that means changing a color, a line style, or a font.*

— *The MATLAB EXPO: An Introduction to MATLAB,  
SIMULINK and the MATLAB Application Toolboxes* (1993)

*The best designs ...*

*are intriguing and curiosity-provoking,  
drawing the viewer into the wonder of the data,  
sometimes by narrative power,  
sometimes by immense detail,  
and sometimes by elegant presentation of simple but interesting data.*

— EDWARD R. TUFTE, *The Visual Display of Quantitative Information* (1983)

*Did we really want to clutter the text with  
endless formatting and Handle Graphics commands such as  
fontsize, markersize, subplot, and pbspect,  
which have nothing to do with the mathematics?*

*In the end I decided that yes, we did.*

*I want you to be able to download these programs  
and get beautiful results immediately.*

— LLOYD N. TREFETHEN, *Spectral Methods in MATLAB* (2000)

# Chapter 18

## Other Data Types and Multidimensional Arrays

So far we have used several MATLAB data types (or classes): `double`, `single`, `int*`, `uint*`, `logical`, and `function_handle`. In this chapter we describe ten further data types: `char`, `string`, `categorical`, `datetime`, `duration`, `calendarDuration`, `table`, `timetable`, `struct`, and `cell`. Several MATLAB data types are, in general, multidimensional arrays; we describe multidimensional arrays in the second section. Figure 18.1 shows a picture of what the MATLAB documentation describes as the “fundamental data types”.

If you want to determine the data type of an object you can use the `class` function, which provides essentially the same information as the last column of the output from `whos`. For example,

```
>> class(pi)
ans =
double

>> class(@sin)
ans =
function_handle

>> class(true)
ans =
logical
```

You can also use the `isa` function to test whether a variable is of a particular class:

```
>> isa(rand(2), 'double')
ans =
    logical
     1

>> isa(eye(2), 'logical')
ans =
    logical
     0
```

As well as being any of the classes mentioned above, the second argument of `isa` can be

- `float`, representing `double` and `single`;

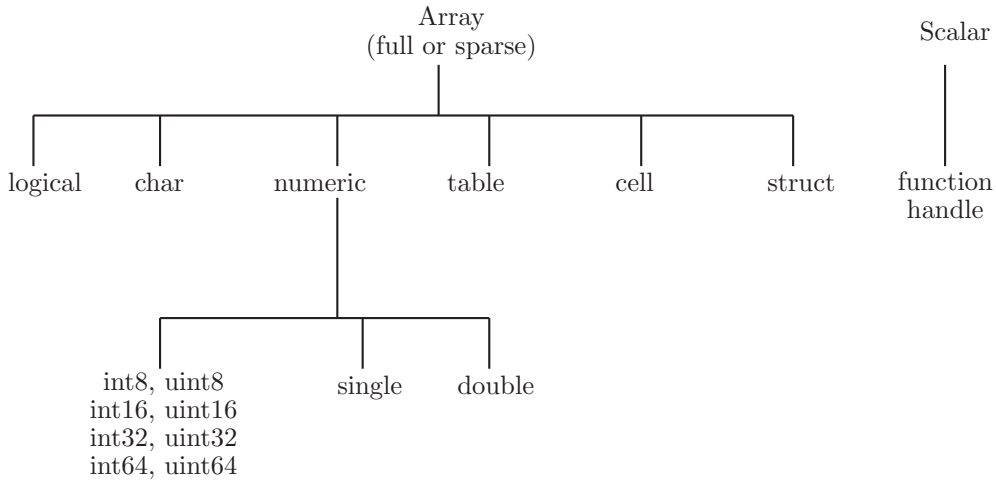


Figure 18.1. *Hierarchy of fundamental MATLAB data types.*

- **integer**, representing signed or unsigned integer types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, and `uint64`;
- **numeric**, representing all the types included under `float` and `integer`.

Functions `isfloat`, `isinteger`, and `isnumeric` can also be used to test for these three respective groups of types.

## 18.1. Character Vectors and Arrays

A character vector (**char** vector) is a vector of characters represented internally in MATLAB by numbers from 0 to 65,535. Consider the following example:

```

>> s = 'ABCabc'
ABCabc

>> sd = double(s)
sd =
    65    66    67    97    98    99

>> s2 = char(sd)
ABCabc

>> whos
  Name      Size      Bytes  Class      Attributes
  s         1x6         12   char
  s2        1x6         12   char
  sd        1x6         48   double
  
```

We see that a **char** vector can be specified by placing characters between single quotes or by applying the `char` function to a vector of positive integers. Each character in a

`char` vector occupies 2 bytes. Values up to 127 correspond to ASCII characters, the rest being Unicode characters. For example, here we produce the letter “x” and the Unicode multiplication symbol:

```
>> [char(120) char(215)]
ans =
x×
```

#### STRINGS

MATLAB has always had strings: sequences of characters delineated by single quotes and stored as vectors of integers. The notion of string is embedded in the language though the names of functions such as `str2double` and `num2str`. Recently, the language has expanded to provide more flexible handling of strings, so that nowadays there is a `string` class, discussed in the next section, and the traditional MATLAB string is now called a character vector.

Character vectors are indexed just like any other array:

```
>> s(6:-1:4)
cba
```

Character vectors can also be created by formatting the values of numeric variables, using `int2str`, `num2str`, or `sprintf`, as described in Section 13.2.

MATLAB has several functions for working with character vectors. Function `strcat` concatenates two character vectors into one longer vector. It removes trailing spaces but leaves leading spaces:

```
>> strcat('Hello ',' world')
Hello world
```

A similar effect can be achieved using the square bracket notation:

```
>> ['Hello ' 'world']
Hello world
```

Both approaches provide a convenient way to generate character vectors whose construction does not fit on a single line of code. For example,

```
fprintf(['This collection of test problems '...
        'was released on %s and contains %d problems.\n'],...
        v.date, v.problemcount)
```

Two character vectors can be compared using `strcmp`: `strcmp(s,t)` returns 1 (true) if `s` and `t` are identical and 0 (false) otherwise. Function `strcmpi` does likewise but treats uppercase and lowercase letters as equivalent. Note the difference between using `strcmp` and the relational operator `==`:

```
>> strcmp('Matlab8','Matlab9')
ans =
    logical
         0
```

```
>> 'Matlab8' == 'Matlab9'
ans =
    1×7 logical array
     1     1     1     1     1     1     0
```

The relational operator can be used only to compare character vectors of equal length and it returns a logical vector showing which characters match. To test whether one vector is contained in another use `strfind`. `strfind(s,t)` returns a vector of starting indices of locations where character vector `t` appears in `s`:

```
>> strfind('abcd','bc')
ans =
     2

>> strfind('abacad','a')
ans =
     1     3     5
```

Note that `strfind` is recommended to be used in preference to an older function `findstr`, but the syntaxes are subtly different because `findstr(s,t)` looks for the occurrences of the shorter of `s` and `t` in the longer of the two.

A character vector can be tested for with logical function `ischar`.

Function `eval` executes a character vector containing any MATLAB expression. Suppose we want to set up matrices `A1`, `A2`, `A3`, `A4`, the `p`th of which is `A - p*eye(n)`. Instead of writing four assignment statements this can be done in a loop using `eval`:

```
for p = 1:4
    eval(['A', int2str(p), ' = A - p*eye(n)'])
end
```

When `p = 2`, for example, the argument to `eval` is the string `'A2 = A - p*eye(n)'` and `eval` executes the assignment.

Character vectors of the same length can be combined into character arrays. If the vectors to be combined are not of the same length they can be padded with spaces as necessary, and the padding can be done automatically with the `char` function:

```
>> subjects = ['Chemistry';'Physics']
Dimensions of matrices being concatenated are not consistent.

>> subjects = ['Chemistry';'Physics  ']
subjects =
Chemistry
Physics

>> subjects = char('Chemistry','Physics')
subjects =
Chemistry
Physics
```

For more functions relating to character arrays see `help strfun`.

## THE CHAR MAZE

The `char` function provides access to a wondrous variety of over 65,000 Unicode characters. This one-line script exploits two of them:

```
while 1, fprintf('%s\n',char(rand(1,80)+9585.5)); pause(.2), end
```

Try it! This is our MATLAB version of a classic 1980s code for the Commodore 64 (see [79], [130]).

## 18.2. String Arrays

A member of the MATLAB `string` class is an array for which each entry contains a character vector. Here, we set up a 1-by-1 string array:

```
>> s = string('This is a string')
s =
    string

    "This is a string"

>> whos s
Name      Size      Bytes  Class    Attributes

s         1×1         166   string
```

String arrays can be formed in several ways, in particular using the `string` function, which can take arguments in various forms and convert them to strings. Building on the example at the end of the previous section, we can write

```
>> subjects = string({'Mathematics';'Computer Science';...
                    'Chemistry'; 'Physics'})
subjects =
    4×1 string array

    "Mathematics"
    "Computer Science"
    "Chemistry"
    "Physics"

>> [isstring(subjects) ischar(subjects)]
ans =
    1×2 logical array
    1    0
```

Unlike character vectors, strings can be compared with the relational operator `==`:

```
>> subjects(1) == subjects(2)
ans =
    logical
    0
```

They can also be joined with the + operator:

```
>> subjects(1) + ' ' + subjects(3)
ans =
    string

    "Mathematics Chemistry"
```

Many functions are available for working with strings, including some of those mentioned in the previous section:

```
>> contains(subjects, 'Science')
ans =
    4×1 logical array
     0
     1
     0
     0
```

```
>> split(subjects(2)) % Split string at whitespace characters.
ans =
    2×1 string array

    "Computer"
    "Science"
```

```
>> join(ans) % Join strings together, with space between.
ans =
    string

    "Computer Science"
```

```
>> reverse(subjects)' % Reverse character order in each element.
ans =
    1×4 string array

    "scitamehtaM"    "ecneicS retupmoC"    "yrtsimehC"    "scisyhP"
```

```
>> subjects = replace(subjects, 'Computer', 'Computational')
subjects =
    4×1 string array

    "Mathematics"
    "Computational Science"
    "Chemistry"
    "Physics"
```

```
>> s = sort(subjects); s(3)
ans =
    Mathematics
```

These examples give just a brief indication of the power of string arrays for handling and analyzing textual data. At the time of writing strings are new to MATLAB and as time goes on we expect them to become more fully developed and integrated.

### 18.3. Multidimensional Arrays

Full (but not sparse) arrays of type `double`, `single`, `int*`, `uint*`, `char`, `logical`, `cell`, and `struct` can have more than two dimensions. Multidimensional arrays are defined and manipulated using natural generalizations of the techniques for matrices. For example, we can set up a 3-by-2-by-2 array of random normal numbers as follows:

```
>> rng(1), A = randn(3,2,2)
A(:,:,1) =
   -0.6490   -1.1096
    1.1812   -0.8456
   -0.7585   -0.5727
A(:,:,2) =
   -0.5587    0.5864
    0.1784   -0.8519
   -0.1969    0.8003

>> whos
  Name      Size      Bytes  Class  Attributes

  A         3x2x2      96     double
```

Notice that MATLAB displays this three-dimensional array a two-dimensional slice at a time. Functions `rand`, `randn`, `zeros`, and `ones` all accept an argument list of the form  $(n_1, n_2, \dots, n_p)$  or  $([n_1, n_2, \dots, n_p])$  in order to set up an array of dimension  $n_1$ -by- $n_2$ -...-by- $n_p$ . An existing two-dimensional array can have its dimensionality extended by assigning to elements in a higher dimension; MATLAB automatically increases the dimensions:

```
>> B = [1 2 3; 4 5 6];
>> B(:,:,2) = ones(2,3)
B(:,:,1) =
     1     2     3
     4     5     6
B(:,:,2) =
     1     1     1
     1     1     1
```

The number of dimensions can be queried using `ndims`, and the `size` function returns the number of elements in each dimension:

```
>> ndims(B)
ans =
     3

>> size(B)
ans =
     2     3     2
```



Table 18.1. *Multidimensional array functions.*

<code>cat</code>	Concatenate arrays
<code>ndims</code>	Number of dimensions
<code>ndgrid</code>	Generate arrays for multidimensional functions and interpolation
<code>permute</code>	Permute array dimensions
<code>ipermute</code>	Inverse permute array dimensions
<code>shiftdim</code>	Shift dimensions
<code>circshift</code>	Circularly shift elements
<code>squeeze</code>	Remove singleton dimensions

To build a multidimensional array by listing elements in one statement use the `cat` function, whose first argument specifies the dimension along which to concatenate the arrays comprising its remaining arguments:

```
>> C = cat(3,[1 2 3; 0 -1 -2],[-5 -3 -1; 10 5 0])
C(:,:,1) =
     1     2     3
     0    -1    -2
C(:,:,2) =
    -5    -3    -1
    10     5     0
```

Functions that operate in an elementwise sense can be applied to multidimensional arrays, as can arithmetic, logical, and relational operators. Thus, for example, `B-ones(size(B))`, `B.*B`, `exp(B)`, `2.^B`, and `B > 0` all return the expected results. The data analysis functions in Table 5.7 all operate along the first nonsingleton dimension by default and accept an extra argument `dim` that specifies the dimension over which they are to operate. For `B` as above, compare

```
>> sum(B)
ans(:,:,1) =
     5     7     9
ans(:,:,2) =
     2     2     2

>> sum(B,3)
ans =
     2     3     4
     5     6     7
```

The transpose operator and the linear algebra operations such as `diag`, `inv`, `eig`, and `\` are undefined for arrays of dimension greater than 2; they can be applied to two-dimensional sections only.

Table 18.1 lists some functions designed specifically for manipulating multidimensional arrays.

## 18.4. Categorical Arrays

Categorical arrays contain elements drawn from a finite set of categories. They are particularly useful when the number of elements is much greater than the number of categories, and in this case they can save on storage, as each category name is stored only once.

When the categories can be represented by character vectors, categorical arrays allow comparisons to be done with the logical operator `eq (==)` instead of `strcmp`. Moreover, the categories can be defined as mathematically ordered, enabling relational operations on their elements.

Suppose we know ten optimal points for some function and wish to record the nature of each point. Here, we form a cell array and then, since there are only three possible types, 'max', 'min', and 'saddle', we convert it to a categorical array:

```
>> opt_type = {'max','min','max','saddle','min','saddle','max','max'};
>> opt_type = categorical(opt_type)
opt_type =
    max    min    max    saddle    min    saddle    max
    max
```

Alternatively, we could rename the categories on (or, as here, after) creation:

```
>> opt_type = categorical(opt_type,{'max','min','saddle'},...
                          {'maximum','minimum','saddle point'})
opt_type =
Columns 1 through 5
    maximum    minimum    maximum    saddle point    minimum
Columns 6 through 8
    saddle point    maximum    maximum

>> categories(opt_type)
ans =
3×1 cell array
    'maximum'
    'minimum'
    'saddle point'

>> summary(opt_type)
    maximum    minimum    saddle point
         4         2         2
```

The `summary` function prints a summary of the array.

Categorical arrays can be created from numbers as well as strings. Suppose the array

```
>> X = [5 1 4 5; 3 1 4 2; 1 4 2 3; 5 1 2 3];
```

represents five crops, identified by number, planted at certain locations in a field broken into a  $4 \times 4$  grid. We might record the actual crop names as follows:

```
>> X = categorical(X,1:5,{'barley','wheat','corn','maize','rapeseed'})
X =
    rapeseed    barley    maize    rapeseed
```

```

    corn      barley    maize    wheat
    barley    maize      wheat    corn
    rapeseed  barley    wheat    corn

```

Now we check where wheat is being grown:

```

>> X == 'wheat'
ans =
    4×4 logical array
    0  0  0  0
    0  0  0  1
    0  0  1  0
    0  0  1  0

```

The next example forms an ordinal categorical array. The first category is the smallest and the last one the largest:

```

>> periods = {'Georgian','Regency','Victorian','Edwardian'};
>> Y = categorical([1 3 4 3 1 2 4 1 4 3 4],1:4,periods,'Ordinal',true)
Y =
Columns 1 through 4
    Georgian    Victorian    Edwardian    Victorian
Columns 5 through 9
    Georgian    Regency    Edwardian    Georgian    Edwardian
Columns 10 through 11
    Victorian    Edwardian

```

Now we can compute with Y:

```

>> min(Y)
ans =
    Georgian

>> summary(Y)
    Georgian    Regency    Victorian    Edwardian
           3         1         3         4

>> find(Y < 'Edwardian')
ans =
    1     2     4     5     6     8    10

```

Notice that the function `min`, which we used with a numeric argument in Chapter 5, is being called here with a categorical array as argument. MATLAB calls a version of the function designed to work with categorical arrays, through a process known as overloading (see Chapter 19). The `histogram` function has also been overloaded to work with categorical arrays. Figure 18.2 shows the output from `histogram(Y)`.

Categorical arrays are often used within tables, as illustrated in the next section. For more details on categorical arrays see `doc categorical arrays`.

## 18.5. Datetime and Duration Arrays

Dates and times can be stored in datetime arrays, which support computation, sorting, comparison, plotting, and formatted display. These arrays are particularly useful in

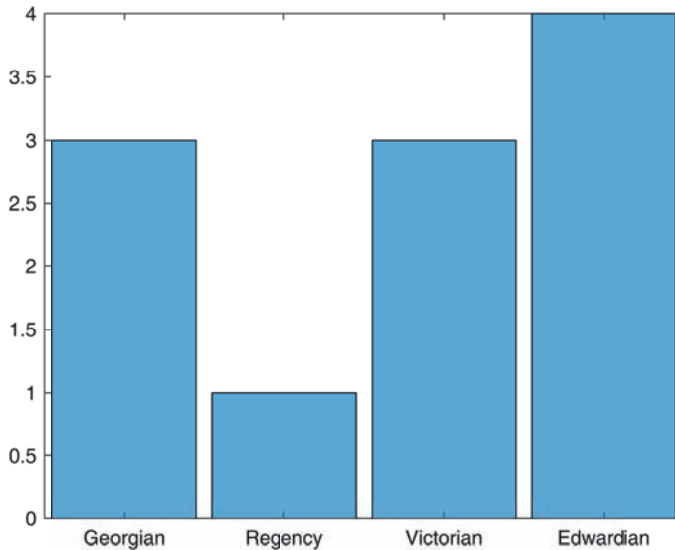


Figure 18.2. *Histogram of categorical array.*

tables, and play a key role in timetables (see Section 18.6). A datetime array can be set up with the `datetime` function code or by importing from a file using the `readtable` or `textscan` functions.

The `datetime` function sets up datetime arrays:

```
>> t = datetime('2017-09-30')
t =
    datetime
    30-Sep-2017

>> whos t
Name      Size      Bytes  Class      Attributes

t         1x1         17  datetime

>> details(t)

datetime with properties:

    SystemTimeZone: 'Europe/London'
    Methods, Superclasses
```

The `details` function prints properties of an array, and is particularly useful for objects, where it provides a hyperlink to the methods that can be applied to the object.

The input to `datetime` can be in one of several formats. If the format is not recognized you need to specify it, and the output format can also be specified:

```
>> datetime('30 Sep 2017', 'InputFormat', 'dd MMM yyyy')
```

Table 18.2. *Subset of identifiers supported in the datetime 'Format' and 'InputFormat' specifiers.*

Identifier	Description	Example
y	Year (with no leading zeros)	2017
yy	Year, using last two digits	17
M	Month, one or two digits	9
MM	Month, two digits	09
MMM	Month, abbreviated name	Sep
MMMM	Month, full name	September
d, dd	Day of month, one or two digits, two digits	3, 03
e, ee	Day of the week*, one or two digits, two digits	7, 07
eee	Day of the week, abbreviated name	Sat
eeee	Day of the week, full name	Saturday
h, hh	Hour in 12-hour clock, one or two digits, two digits	8, 08
H, HH	Hour in 24-hour clock, one or two digits, two digits	20, 20
m, s	Minute, second, one or two digits	32, 9

\* Sunday is the first day of the week.

```
ans =
  datetime
    30-Sep-2017

>> datetime(ans,'Format','eeee d MMMM, y')
ans =
  datetime
    Saturday 30 September, 2017
```

Format specifiers are used to specify both input and output formats and can draw on a large number of identifiers, a small subset of which is shown in Table 18.2. For the full list see `doc datetime properties`.

Datetime arrays can also be set up by giving numerical values for the year, month, day, hour, minute, and second components, and multiple dates and times can be specified in one call, as in the next examples:

```
>> t = datetime(2017,9,30,8,25,0)
t =
  datetime
    30-Sep-2017 08:25:00

>> t = datetime(2017,1:7,1,0,0,0)
t =
  1x7 datetime array
Columns 1 through 3
    01-Jan-2017 00:00:00    01-Feb-2017 00:00:00    01-Mar-2017 00:00:00
Columns 4 through 6
    01-Apr-2017 00:00:00    01-May-2017 00:00:00    01-Jun-2017 00:00:00
Column 7
```

```

01-Jul-2017 00:00:00

>> d = t(7)-t(1) % Time between first and last datetimes.
d =
    4344:00:00

>> d.Format = 'm'
d =
    duration
    2.6064e+05 min

>> whos d t
    Name      Size      Bytes  Class      Attributes

    d         1x1         10    duration
    t         1x7         113   datetime

```

The difference of two `datetime` values is a duration array, and such arrays can also be created with the `duration` function. Many functions are overloaded for date-time and duration arrays. The `caldiff` function is one of several functions that produces or works with `calendarDuration` arrays, which can also be created with the `calendarDuration` function:

```

>> d1 = diff(t) % Number of hours in each month
d1 =
    1x6 duration array
Columns 1 through 5
    744:00:00    672:00:00    744:00:00    720:00:00    744:00:00
Column 6
    720:00:00

>> d2 = caldiff(t,'days') % Number of days in each month.
    1x6 calendarDuration array
d2 =
    31d    28d    31d    30d    31d    30d

>> whos d1 d2
    Name      Size      Bytes  Class      Attributes

    d1         1x6         64    duration
    d2         1x6        598   calendarDuration

>> 2*d2
ans =
    1x6 calendarDuration array
    62d    56d    62d    60d    62d    60d

```

The next example shows the use of `interp1` to interpolate with a datetime array. It also uses the `caldays` function to produce a `calendarDuration` vector representing successive calendar days:

```

dates = datetime({'10 Jan 17','14 Feb 17','20 Mar 17','9 Apr 17'},...

```

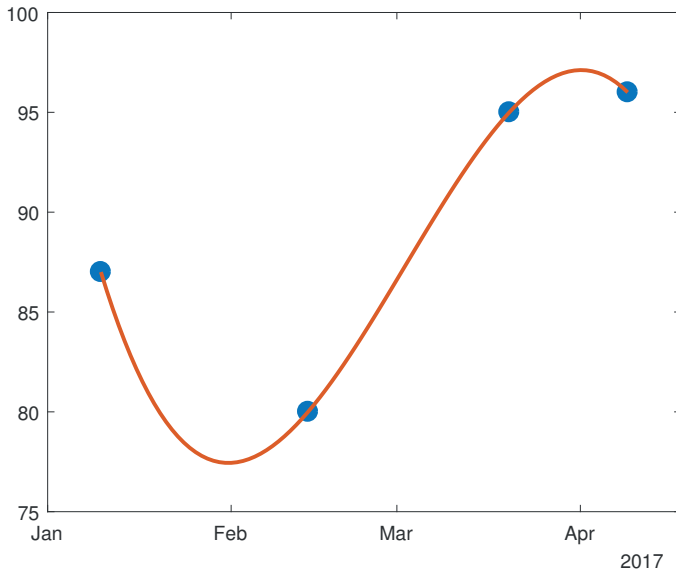


Figure 18.3. Plot of interpolated data with datetime vector on the x-axis.

```

                                'InputFormat','d MMM yy')
prices = [87 80 95 96];
daily = dates(1) + caldays(0:days(dates(end)-dates(1)));
prices_daily = interp1(dates,prices,daily,'spline');
plot(dates,prices,'.',daily,prices_daily,'-',...
     'MarkerSize',30,'LineWidth',2)
ax = xlim; ax(1) = datetime('1-Jan-2017'); xlim(ax)

```

The plot produced is shown in Figure 18.3.

MATLAB supports many other features for datetime arrays, including setting different time zones, converting dates to Julian date format, and displaying time zone offsets, and it has a NaT (Not-a-Time) value to represent an unknown or missing datetime value. The function `ismissing` detects missing values, including NaTs for datetime data and NaNs for double and single data.

## 18.6. Tables and Timetables

A table is a MATLAB data type designed to store tabular data, with columns representing variables of possibly different types. There are many ways to set up a table. One way is as follows. This example contains data about an annual conference held in different cities:

```

cities = {'Chicago';'London';'Atlanta';'Paris';'New_York';'Berlin'};
year = (2010:2015)';
month = {'July';'July';'August';'June';'May';'August'};
attendance = [2100; 2750; 1988; 2721; 2401; 2234];
reg_fee = [550; 575; 560; 600; 650; 610];
A = table(year,month,attendance,reg_fee,'RowNames',cities)

```

Running this code produces the output

```
A =
```

	year	month	attendance	reg_fee
Chicago	2010	'July'	2100	550
London	2011	'July'	2750	575
Atlanta	2012	'August'	1988	560
Paris	2013	'June'	2721	600
New_York	2014	'May'	2401	650
Berlin	2015	'August'	2234	610

Here, the columns are labeled by the variables from which they were constructed and the row labels were specified as character vectors. Notice that columns 1, 3, and 4 are numeric but the second column contains character vectors. The `summary` function prints a statistical summary of the table:

```
>> summary(A)
Variables:
  year: 6×1 double
    Values:
      min      2010
      median   2012.5
      max      2015
  month: 6×1 cell string
  attendance: 6×1 double
    Values:
      min      1988
      median   2317.5
      max      2750
  reg_fee: 6×1 double
    Values:
      min      550
      median   587.5
      max      650
```

We can remove a column by assigning the empty matrix to it:

```
>> A.reg_fee = []
A =
```

	year	month	attendance
Chicago	2010	'July'	2100
London	2011	'July'	2750
Atlanta	2012	'August'	1988
Paris	2013	'June'	2721
New_York	2014	'May'	2401
Berlin	2015	'August'	2234

Now we add a new column that specifies the number of presentations given at the conferences:



```
>> A.presentations = [1550; 1810; 1434; 2124; 2033; 1999]
A =
      year      month      attendance      presentations
      ----      -
Chicago 2010      'July'      2100      1550
London  2011      'July'      2750      1810
Atlanta 2012      'August'    1988      1434
Paris   2013      'June'      2721      2124
New_York 2014      'May'       2401      2033
Berlin  2015      'August'    2234      1999
```

Then we sort the table by the number of attendees and calculate the average number of attendees and presentations:

```
>> sortrows(A,3,'descend')
ans =
      year      month      attendance      presentations
      ----      -
London  2011      'July'      2750      1810
Paris   2013      'June'      2721      2124
New_York 2014      'May'       2401      2033
Berlin  2015      'August'    2234      1999
Chicago 2010      'July'      2100      1550
Atlanta 2012      'August'    1988      1434

>> format shortg
>> [mean(A.attendance), mean(A.presentations)]
ans =
      2365.7      1825
```

Next we convert the second column of the table to an ordinal categorical array:

```
>> A.month = categorical(A.month,{'January','February','March',...
      'April','May','June','July','August','September',...
      'October','November','December'},'Ordinal',true)
A =
      year      month      attendance      presentations
      ----      -
Chicago 2010      July      2100      1550
London  2011      July      2750      1810
Atlanta 2012      August    1988      1434
Paris   2013      June      2721      2124
New_York 2014      May       2401      2033
Berlin  2015      August    2234      1999
```

Note that the entries in the `month` column no longer have quotes around them, as they are not now character vectors. We could simply have written `A.month = categorical(A.month)`, which would have made a set of categories comprising just the months in the table. But the argument pair `'Ordinal',true`, which specifies that the categories are ordered, would then order alphabetically instead of by month. Now we can extract the conferences that took place in the second half of the year:



```

2016:04:07 09:11:57 'DSCF3913.RAF' 5 3200
2016:04:07 13:07:14 'DSCF3916.RAF' 4 1250

```

The field `DateTime` (which could have had any name) now represents labels for the rows, and the first column is now `Filename`:

```

>> E(1,1)
ans =
      DateTime      Filename
-----
2016:04:07 09:11:39 'DSCF3911.RAF'

```

The `timerange` function produces a subscript that can be used to select rows corresponding to a particular time interval:

```

>> E(timerange('18-Apr-2016','21-Apr-2016'),:) % April 18-20
ans =
      DateTime      Filename      Aperture      ISO
-----
2016:04:20 08:59:44 'IMG_4096.JPG' 2.2          25
2016:04:18 19:11:56 'IMG_4083.JPG' 2.2          125

```

By default the interval is open on the left and closed on the right, but a third argument allows the interval to be made open or closed at either end.

MATLAB has various other functions for working with timetables, which include `synchronize` for combining two tables and merging rows with a common time, and `retime`, which adjusts the table to a new set of times, with various options for how this is done.

MATLAB has a rich variety of functions for working with tables beyond those we have illustrated, including

- `writetable`, for writing tables to files in various formats, including CSV (“comma-separated value”) and `xls` (Excel spreadsheet) files;
- functions to convert to and from other MATLAB data types (for example, `array2table`, `table2array`); and
- functions to apply functions to variables (columns) and rows (`varfun`, `rowfun`).

For more details on tables see `doc table`.

## 18.7. Structures and Cell Arrays

Structures and cell arrays both provide a way to collect arrays of different types and sizes into a single array. They are used in many places within MATLAB. For example, structures are used by many of the numerical methods routines in Chapters 11 and 12 and in the graphics system, as described in the previous chapter. Structures also play an important role in object-oriented programming in MATLAB, which is described in the next chapter. Cell arrays are used by the `varargin` and `varargout` functions (Section 10.5), to specify text in graphics commands (p. 128), and in the `switch-case` construct (Section 6.2).

Suppose we want to build a collection of  $4 \times 4$  test matrices, recording for each matrix its name, the matrix elements, and the eigenvalues. We can build an array structure `testmat` having three fields, `name`, `mat`, and `eig`:

```

n = 4;
testmat(1).name = 'Hilbert';
testmat(1).mat = hilb(n);
testmat(1).eig = eig(hilb(n));
testmat(2).name = 'Pascal';
testmat(2).mat = pascal(n);
testmat(2).eig = eig(pascal(n));

```

Displaying the structure gives the field names but not the contents:

```

>> testmat
testmat =
1×2 struct array with fields:
    name
    mat
    eig

```

We can access individual fields using a period:

```

>> testmat(2).name
ans =
Pascal

>> testmat(1).mat
ans =
    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429

>> testmat(2).eig
ans =
    0.0380
    0.4538
    2.2034
   26.3047

```

For array fields, array subscripts can be appended to the field specifier:

```

>> testmat(1).mat(1:2,1:2)
ans =
    1.0000    0.5000
    0.5000    0.3333

```

Another way to set up the `testmat` structure is using the `struct` command:

```

testmat = struct('name',{ 'Hilbert','Pascal'},...
                'mat',{hilb(n),pascal(n)}, ...
                'eig',{eig(hilb(n)),eig(pascal(n))})

```

The arguments to the `struct` function are the field names, with each field name followed by the field contents listed within curly braces (that is, the field contents are cell arrays, which are described next). If the entire structure cannot be assigned with

one `struct` statement then it can be created with fields initialized to a particular value using `repmat`. For example, we can set up a test matrix structure for five matrices initialized with empty names and zero matrix entries and eigenvalues with

```
>> testmat = repmat(struct('name',{''}, 'mat',{zeros(n)}, ...
                          'eig',{zeros(n,1)}),5,1)

testmat =
5×1 struct array with fields:
    name
    mat
    eig

>> testmat(5) % Check last element of structure.
ans =
struct with fields:
    name: ''
    mat: [4×4 double]
    eig: [4×1 double]
```

For the benefits of such preallocation see Section 23.4.

When a structure passed as an argument to a function is used to set up options it is necessary for the routine to check which fields have been set. This can be done by using the `isfield` function to test for the existence of the fields, as here:

```
>> isfield(testmat(1),'eig')
ans =
logical
    1

>> isfield(testmat(1),'inverse')
ans =
logical
    0
```

Cell arrays differ from structures in that they are accessed using array indexing rather than named fields. One way to set up a cell array is by using curly braces as cell array constructors. In this example we set up a 2-by-2 cell array:

```
>> C = {1:3, pi; magic(2), 'A string'}
C =
2×2 cell array
[1×3 double]    [ 3.1416]
[2×2 double]    'A string'
```

Cell array contents are indexed using curly braces, and the colon notation can be used in the same way as for other arrays:

```
>> C{1,1}
ans =
    1    2    3

>> C{2,:}
```

```
ans =
     1     3
     4     2
ans =
A string
```

The test matrix example can be recast as a cell array as follows:

```
clear testmat
testmat{1,1} = 'Hilbert';
testmat{2,1} = hilb(n);
testmat{3,1} = eig(hilb(n));
testmat{1,2} = 'Pascal';
testmat{2,2} = pascal(n);
testmat{3,2} = eig(pascal(n));
```

The `clear` statement is necessary to remove the previous structure of the same name. Here, each collection of test matrix information occupies a column of the cell array, as can be seen from

```
>> testmat
testmat =
  3×2 cell array
    'Hilbert'    'Pascal'
    [4×4 double] [4×4 double]
    [4×1 double] [4×1 double]
```

The `celldisp` function can be used to display the contents of a cell array:

```
>> celldisp(testmat)
testmat{1,1} =
Hilbert
testmat{2,1} =
  1.0000    0.5000    0.3333    0.2500
  0.5000    0.3333    0.2500    0.2000
  0.3333    0.2500    0.2000    0.1667
  0.2500    0.2000    0.1667    0.1429
testmat{3,1} =
  0.0001
  0.0067
  0.1691
  1.5002
testmat{1,2} =
Pascal
testmat{2,2} =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
testmat{3,2} =
  0.0380
  0.4538
```

```

2.2034
26.3047

```

Another way to express the assignments to `testmat` above is by using standard array subscripting, as illustrated by

```
testmat(1,1) = {'Hilbert'};
```

Curly braces must appear on either the left or the right side of the assignment statement in order for the assignment to be valid.

When a component of a cell array is itself an array, its elements can be accessed using parentheses:

```

>> testmat{2,1}(4,4)
ans =
    0.1429

```

Although it was not necessary in our example, we could have preallocated the `testmat` cell array with the `cell` command:

```
testmat = cell(3,2);
```

After this assignment `testmat` is a 3-by-2 cell array of empty matrices.

Useful for visualizing the structure of a cell array is `cellplot`. Figure 18.4 was produced by `cellplot(testmat)`.

Cell arrays can replace comma-separated lists of variables. The `varargin` and `varargout` functions (see Section 10.5) provide good examples of this usage. To illustrate, consider

```

>> testmat{1,:}
ans =
Hilbert
ans =
Pascal

```

Two separate outputs are produced, and by feeding these into `char` we obtain a character array:

```

>> names = char(testmat{1,:})
names =
Hilbert
Pascal

```

```

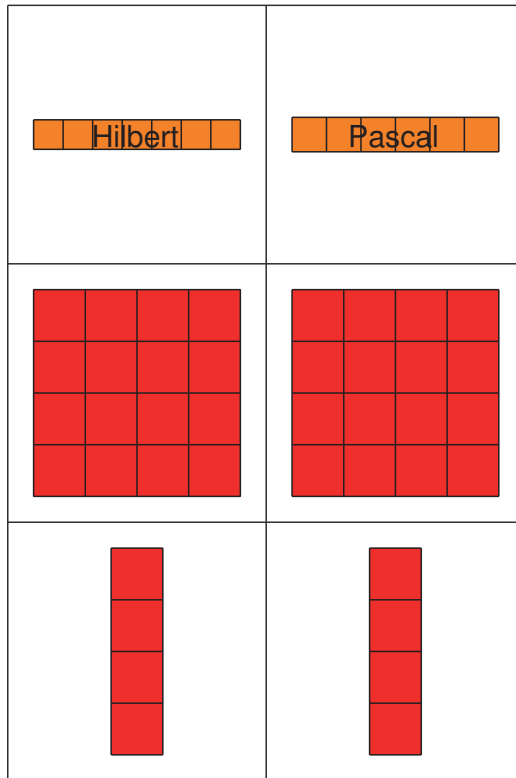
>> whos names
Name          Size          Bytes  Class

names        2x7              28   char array

```

Grand total is 14 elements using 28 bytes

The functions `cell2struct` and `struct2cell` convert between cell arrays and structures, while `num2cell` creates a cell array of the same size as the given numeric array. The `cat` function, discussed in Section 18.3, provides an elegant way to produce a numeric vector from a structure or cell array. In our test matrix example, if we want to produce a matrix having as its columns the vectors of eigenvalues, we can type

Figure 18.4. `cellplot(testmat)`.

```
cat(2,testmat.eig)
```

for the structure `testmat`, or

```
cat(2,testmat{3,:})
```

for the cell array `testmat`, in both cases obtaining the result

```
ans =
    0.0001    0.0380
    0.0067    0.4538
    0.1691    2.2034
    1.5002   26.3047
```

Here, the first argument of `cat` causes concatenation in the second dimension, that is, columnwise. If this argument is replaced by 1 then the concatenation is row-wise and a long vector is produced. An example of this use of `cat` is in Listing 17.1, where it extracts from a cell array a vector that can then be plotted.



*For many applications,  
the choice of the proper data structure is really  
the only major decision involved in the implementation;  
once the choice has been made,  
only very simple algorithms are needed.*

— ROBERT SEDGEWICK, *Algorithms* (1988)

# Chapter 19

## Object-Oriented Programming

Most of the code in this book fits into the paradigm of procedural programming, which is based on procedures (functions in MATLAB) that contain a sequence of computational steps. Another paradigm is functional programming, in which programs are entirely expressed in terms of mathematical functions. Function application is the only control structure and functions have no “side-effects”, that is, they do not do anything except return a value. Lisp is a classic example of a language designed for functional programming.

A third programming paradigm is object-oriented programming, which we have already seen is fundamental to the graphics system and which features in the next two chapters on the Symbolic Math Toolbox and graphs. Object-oriented programming is also widely used in developing GUIs. Among third-party software, Chebfun [38] is an excellent example of the power of object-oriented programming in MATLAB.

Object-oriented programming exploits the idea of data abstraction, in which the programmer defines classes and keeps separate the representation of the classes from implementation of operations on them. These operations are called methods. In general, a hierarchy of classes is built. Objects are specific instances of the classes with their own characteristics.

Object-oriented programming is a rich subject and we can only give a very brief introduction here, which we do via two examples. In each example we define a new class along with operations on objects in the class and we exploit overloading, in which MATLAB calls different versions of a function based on the class of its argument.

### 19.1. Max-Plus Algebra Class

Consider the algebra that is obtained using the two operations  $\oplus$  and  $\otimes$ , defined for  $a$  and  $b$  on the real line including  $-\infty$  by

$$\begin{aligned}a \oplus b &= \max(a, b), \\ a \otimes b &= a + b.\end{aligned}$$

In other words we replace plus by max and times by plus. From  $a \oplus -\infty = \max(a, -\infty) = a$  it follows that  $-\infty$  is the additive zero, and  $a \otimes 0 = a$  shows that 0 is the multiplicative identity.

Motivation for looking at this max-plus algebra comes from the observation that if two activities must be performed consecutively then the time required to complete both is the sum of the individual times, but if they may be performed concurrently then the time required is the maximum of the individual times. This suggests, correctly, that max-plus algebra is useful in scheduling [15], [63].

Our aim is to define a new MATLAB class (or data type) `maxplus` that enables us to carry out max-plus arithmetic. To do so we need to create a single file `maxplus.m` on the MATLAB path that contains all the necessary code in it. Alternatively, we could create a subdirectory `@maxplus` of a directory on the path and put within it multiple files that define the class.

The code file in Listing 19.1 is a bare-bones file that sets up max-plus arithmetic for scalars. The code comprises a `classdef` block within which there is a `methods` block that defines an object in the `maxplus` class and the operations that can be performed on it. The `maxplus` class is defined as a subclass (denoted by `<`) of the built-in `double` class. By making `maxplus` a subclass of `double`, we can take advantage of mathematical operators that already exist for `double`, though some of these will need redefining. The `obj` function defines an object of the `maxplus` class and the `obj@double(a)` syntax is MATLAB notation for calling the superclass `double` on `a`; this means that `maxplus('a')`, for example, is allowed, and returns a `maxplus` variable with value 97—the result of `double` converting the `char` to its ASCII value.

The `methods` block contains functions that define methods for the class, namely addition, multiplication, and powering. The function names `plus`, `times`, and `mpower` are the standard MATLAB names that are invoked when MATLAB encounters the symbols `+`, `*`, and `^`. We are therefore overloading these functions for `maxplus` arguments. We can now write

```
>> a = maxplus(1)
a =
  maxplus:
  double data:
      1
```

```
>> class(a)
ans =
  maxplus
```

We can carry out some simple max-plus computations:

```
>> a = maxplus(1); b = maxplus(-2); c = maxplus(4);
>> a*b + c + maxplus(2)
ans =
  maxplus:
  double data:
      4

>> a^5
ans =
  maxplus:
  double data:
      5
```

If we write the first of the previous expressions in a different way, the same answer is produced:

```
>> a*b + c + 2
ans =
  maxplus:
```

Listing 19.1. *Code file maxplus; version for scalars.*

```
classdef maxplus < double

    methods

        function obj = maxplus(a)
            % Maxplus class constructor.
            if ~isreal(a), error('Max-plus scalars must be real. '), end
            obj = obj@double(a);
        end

        function z = plus(x,y)
            z = maxplus(max(x,y));
        end

        function z = mtimes(x,y)
            z = maxplus(double(x) + double(y));
        end

        function z = mpower(x,k)
            % k a nonnegative integer.
            if (round(k) ~= k) || k < 0
                error('Only nonnegative integer powers are supported.')
            end
            if k == 0
                z = maxplus(0); return
            end
            z = x;
            for i = 2:k
                z = z*x;
            end
        end

    end

end
```

```
double data:
    4
```

Since the final 2 is a `double`, we might have expected MATLAB to carry out the second addition as a standard addition of doubles. However, MATLAB has a rule that user-defined classes have precedence over MATLAB fundamental classes, and this rule ensures that the second addition is carried out as a max-plus addition.

Our `maxplus` class works for scalars, but it will be much more useful if it is extended to matrices, so that we can carry out what is technically linear algebra over the tropical semiring. The `maxplus` and `plus` functions need no change, the latter automatically working componentwise when supplied with matrix arguments. However, `mtimes` does need modifying as we need it to carry out matrix multiplication, which is defined for  $m$ -by- $n$   $A$  and  $n$ -by- $p$   $B$  by

$$(A \otimes B)_{ij} = \sum_k^{\oplus} a_{ik} \otimes b_{kj} = \max_k(a_{ik} + b_{kj}).$$

Listing 19.2 shows an extended `maxplus` class in which `mtimes` handles the matrix case by explicitly computing the matrix product. For efficiency we vectorized the inner loop; the unvectorized code is shown in comments. We also overloaded the `disp` function in order to produce more concise output:

```
>> a = maxplus([1 2 3])
a =
    1     2     3
```

And we added the `times` method, which is called for componentwise multiplication of arrays. Note that `2*maxplus(rand(2))` gives an error, as we have not catered for the special case where one of the arguments is a scalar and the other a matrix.

This class also has a new feature: static methods, which are methods that do not require an object of the class as input. We have included matrix generation functions `eye` and `ones`, which generate the appropriate matrices for the max-plus algebra, namely matrices 0 and  $I$  such that  $A \oplus 0 = A$  and  $A \otimes I = A$ . These are called by specifying the class as an argument:

```
>> eye(2,4,'maxplus')
ans =
    0   -Inf   -Inf   -Inf
 -Inf    0   -Inf   -Inf

>> zeros(2,'maxplus')
ans =
 -Inf   -Inf
 -Inf   -Inf
```

Note that we have taken care that `eye` and `ones` accept arguments in the same way as the built-in functions.

Listing 19.2. *Code file maxplus; version for matrices.*

```

classdef maxplus < double

    methods

        function obj = maxplus(a)
            % Maxplus class constructor.
            if ~isreal(a), error('Max-plus matrices must be real. '), end
            obj = obj@double(a);
        end

        function z = plus(x,y)
            z = maxplus(max(x,y));
        end

        function z = times(x,y)
            z = maxplus(double(x) + double(y));
        end

        function z = mtimes(x,y)
            [m,n] = size(x);
            [n1,p] = size(y);
            if n ~= n1, error('Matrix dimensions not compatible. '), end
            if max([m n p]) == 1
                z = maxplus(double(x) + double(y)); % Scalar operands.
            else
                % Matrix multiplication.
                z = zeros(m,p,'maxplus'); % 'maxplus' argument essential here!
                for i = 1:m
                    for j = 1:p
                        z(i,j) = maxplus(max(double(x(i,:)).') + double(y(:,j))));
                        % Less efficient, but equivalent to previous line:
                        % for k = 1:n
                        z(i,j) = z(i,j) + x(i,k)*y(k,j);
                        % end
                    end
                end
            end
        end

        function z = mpower(x,k)
            % k a nonnegative integer.
            if (round(k) ~= k) || k < 0
                error('Only nonnegative integer powers are supported. ')
            end
            if k == 0
                z = eye(size(x),'maxplus'); return
            end
            z = x;
            for i = 2:k
                z = z*x;
            end
        end
    end
end

```

```
end

function disp(a)
    % Called by display function to display an object.
    disp(double(a))
end

end

methods (Static)
    % Methods that do not require an object of the class as input.
    % Support all three possible syntaxes of input to eye and zeros.

function A = eye(m,n)
    % Identity matrix in the max-plus algebra.
    if nargin == 1
        if length(m) == 1
            n = m;
        else
            n = m(2); m = m(1);
        end
    end
    A = repmat(-inf,[m,n]);
    A(1:m+1:m*min(m,n)) = 0; % Set diagonal to zero.
    A = maxplus(A);
end

function A = zeros(m,n)
    % Zero matrix in the max-plus algebra.
    if nargin == 1
        if length(m) == 1
            n = m;
        else
            n = m(2); m = m(1);
        end
    end
    A = maxplus(repmat(-inf,[m,n]));
end

end

end
```

We can obtain a summary of all the methods available for our class as follows:

```
>> methods('maxplus')
```

```
Methods for class maxplus:
```

```
abs                ctranspose         int64              permute
accumarray         cummax             int8               plot
acos              cummin            inv                plus
acosd             cumprod           isbanded          pochhammer
...
csch              int32             ordschur
```

```
Static methods:
```

```
eye                zeros
```

Here, most of the output has been omitted because there are so many methods inherited from `double` (many of which do not make sense in the max-plus algebra).

To illustrate the use of the `maxplus` class we consider the linear system  $x = A \otimes x \oplus b$  for an  $n$ -by- $n$   $A$ . In the usual algebra, the solution to the system  $x = Ax + b$  is  $x = (I - A)^{-1}b = (I + A + A^2 + \dots)b$  when  $\rho(A) < 1$ , where the spectral radius  $\rho(A)$  is the largest modulus of any eigenvalue of the matrix  $A$ . Somewhat analogously, under certain conditions that we will not describe, the max-plus system has the solution  $x = (I \oplus A \oplus \dots \oplus A^{n-1}) \otimes b$ , where the sum now terminates at the  $(n - 1)$ st power. We will check this formula for a system with  $n = 3$  for which the required conditions are known to hold:

```
>> A = maxplus([-1 0 0; -1 0 -3; -2 -4 0]);
>> b = maxplus([1 0 -1]');
>> K = eye(size(A), 'maxplus') + A + A^2; % Kleene star.

>> x = K*b
x =
     1
     0
    -1

>> res = norm(x - (A*x+b))
res =
     0
```

## 19.2. Circulant Matrix Class

A circulant matrix is a special kind of Toeplitz matrix of the form

$$C = C(c) = \begin{bmatrix} c_1 & c_n & \dots & c_2 \\ c_2 & c_1 & \dots & \vdots \\ \vdots & \ddots & \ddots & c_n \\ c_n & \dots & c_2 & c_1 \end{bmatrix}.$$



Note that the diagonals “wrap around”. Circulant matrices have a number of interesting properties. The sum and product of two circulant matrices is a circulant matrix and the inverse of a circulant matrix is a circulant matrix. Moreover, circulant matrices commute.

These properties all follow from the remarkable fact that a circulant matrix is diagonalized by the discrete Fourier transform matrix  $F_n$  [53, Thm. 4.8.2]:

$$F_n^{-1}CF_n = D = \text{diag}(d_i). \quad (19.1)$$

The matrix  $F_n$ , introduced in Section 11.4, is a complex matrix satisfying  $F_nF_n^* = nI$ ; it can be generated in MATLAB by either of the expressions `fft(eye(n))` or `sqrt(n)*gallery('orthog',n,3)'`. The vector  $d$  is given by  $d = F_n^*c = nF_n^{-1}c$ .

Multiplication by  $F_n$  is efficiently carried out by the FFT:  $y = F_n^{-1}x$  is equivalent to `y = ifft(x)`, and similarly  $x = F_ny$  is equivalent to `x = fft(y)`.

Using these properties we can carry out multiplication and inversion of circulant matrices, and solution of circulant linear systems, more efficiently than for general matrices. A good way to arrange these computations is with a circulant class that overloads addition, matrix multiplication, matrix inversion, and backslash. The class `circulant` in Figure 19.3 stores the defining vector `c` (the first column of the circulant matrix) and the vector of eigenvalues of the matrix, the latter to avoid recomputation of the eigenvalues.

The `circulant` class exploits the fact that the first column of  $F_n^{-1}$  is given by  $F_n^{-1}e_1 = n^{-1}e$ , where  $e$  is the vector of ones, so that, given  $D$  in (19.1), we can recover the first column of  $C$  from the equation  $Ce_1 = F_nDF_n^{-1}e_1 = n^{-1}F_nDe = n^{-1}F_nd$ .

This is a bare-bones class that would need extending for serious use. Nevertheless, for working with circulant matrices it can be very beneficial. In the following example we solve a circulant system  $Cx = b$  using the regular backslash and then with our circulant class. Note that `gallery('circul',c)` produces a circulant matrix whose first row is `c`, hence we transpose the matrix:

```
n = 10000; rng(1)
c = randn(n,1);
Cmat = gallery('circul',c)'; % n-by-n matrix representation.
C = circulant(c);          % circulant class representation.
normC = norm(C.eig,inf);   % Equivalent to norm(C).
b = randn(n,1);
tic, x1 = Cmat\b; toc      % Regular backslash.
rel_residual = norm(Cmat*x1 - b)/(normC*norm(x1) + norm(b))
tic, x2 = C\b; toc        % Overloaded backslash.
rel_residual = norm(Cmat*x2 - b)/(normC*norm(x2) + norm(b))
```

The output is

```
Elapsed time is 9.775281 seconds.
rel_residual =
    1.6632e-13
Elapsed time is 0.001755 seconds.
rel_residual =
    6.9252e-16
```

Listing 19.3. *Code file circulant.*

```

classdef circulant

    properties
        vector % Vector of coefficients in first row of matrix.
        eig    % Vector of eigenvalues.
    end

    methods

        function obj = circulant(c,e)
            obj.vector = c;
            if nargin < 2, e = length(c)*ifft(c); end
            obj.eig = e;
        end

        function x = plus(a,b)
            x = circulant(a.vector + b.vector,a.eig + b.eig);
        end

        function x = mldivide(a,b)
            for j = size(b,2):-1:1 % Reverse order preallocates x.
                x(:,j) = fft(a.eig .\ ifft(b(:,j)));
            end
        end

        function x = mtimes(a,b)
            n = length(a.vector);
            if n ~= length(b.vector)
                error('The matrices do not conform for multiplication');
            end
            e = a.eig.*b.eig; v = fft(e)/n;
            x = circulant(v,e);
        end

        function x = inv(a)
            if min(abs(a.eig)) <= eps*max(abs(a.eig))
                warning('Matrix is singular to working precision.')
            end
            n = length(a.vector);
            e = 1./a.eig; v = fft(e)/n;
            x = circulant(v,e);
        end

        function disp(a)
            disp(a.vector')
        end

    end

end
end

```

Not only has exploiting circulant structure reduced the solution time by three orders of magnitude (thanks to the efficiency of the FFT) but it has produced a smaller residual.

### 19.3. On Things Not Treated

This chapter has merely scratched the surface of object-oriented programming in MATLAB, presenting very simple, numerically oriented examples. We have barely touched on inheritance, public versus private properties, handle classes, or events and listeners. The best example of object-oriented programming in MATLAB is MATLAB itself, through its graphics system, unit testing framework, Symbolic Math Toolbox, and so on.

For further reading we recommend the MATLAB documentation and the following resources.

- An article by Davis on an “object-oriented backslash” for linear systems, based on a class called `factorization` [28].
- An article by Neidinger on an object-oriented implementation of automatic differentiation in MATLAB [134].
- The Chebfun system (<http://www.chebfun.org>) [38] for numerical computation with functions, which is built on piecewise polynomial interpolation at the extrema of Chebyshev polynomials.

*Objects allow anyone to add new data types to MATLAB.*

*By writing a handful of M-files,  
you can have your MATLAB do operations  
we never dreamed of at The MathWorks.*

— CLEVE B. MOLER, *Objectively Speaking. OOPS is Not an Apology* (1999)

# Chapter 20

## The Symbolic Math Toolbox

The Symbolic Math Toolbox is one of the many toolboxes that extend the functionality of MATLAB, and perhaps the one that does so in the most fundamental way. The toolbox is provided with the MATLAB and Simulink Student Suite but must be purchased as an extra with other versions of MATLAB. You can tell if your MATLAB installation contains the toolbox by issuing the `ver` command and seeing if the toolbox is listed.

The toolbox is based upon the MuPAD engine, which performs all the symbolic and variable-precision computations.

Not all functions in the toolbox can be mentioned here. To see a full list, type `doc symbolic` then select “Functions and Other Reference”.

All input and output in this chapter is shown as it appears in the Command Window. You may prefer to carry out symbolic computations in the Live Editor (see Section 16.7), since output is automatically typeset, making it easier to read.

### 20.1. Creating Symbolic Variables and Expressions

The Symbolic Math Toolbox defines a new data type: a symbolic object, denoted by `sym`. Symbolic objects can be created with the `sym` and `syms` commands. We can define symbolic variables `x` and `y` with

```
>> syms x y % Or, equivalently, x = sym('x'), y = sym('y')
```

```
>> whos
```

Name	Size	Bytes	Class	Attributes
x	1×1	112	sym	
y	1×1	112	sym	

Once the symbolic variables are defined, computations can be done with them:

```
>> expand((x+y)^4)
ans =
x^4 + 4*x^3*y + 6*x^2*y^2 + 4*x*y^3 + y^4

>> factor(ans)
ans =
[ x + y, x + y, x + y, x + y]
```

Care is needed when converting numeric expressions to symbolic form. It is important to ensure that expressions are not first converted to double precision. Consider this attempt to form a symbolic representation of  $e = \exp(1)$ :

```
>> e = sym(exp(1))
e =
3060513257434037/1125899906842624
```

Here, `exp(1)` has been evaluated in double precision and then `sym` has expressed that number in rational form, as the ratio of two large integers. The variable `e` only contains an approximation to  $e$ . The correct way to store a symbolic representation of  $e$  is as follows:

```
>> e = exp(sym(1))
e =
exp(1)
```

By applying `exp` to the symbolic constant `1` we ensure that all computations are symbolic.

The `sym` function has a feature that is both useful and potentially dangerous, namely that it tries to convert floating-point numbers to nearby symbolic expressions involving square roots and rational numbers with modest sized coefficients:

```
>> sym(0.1)
ans =
1/10

>> sym(sqrt(5))
ans =
5^(1/2)
```

In these two examples, the double-precision arguments `0.1` and  $\sqrt{5}$  are not exactly representable in double-precision arithmetic. Nevertheless, `sym` produces symbolic quantities as if there were no rounding errors in evaluating the arguments. The `sym` function has an optional second argument that specifies how to do the conversion to symbolic form. We highlight two of the options.

- `sym(expr, 'r')`. The documentation states that this rational mode “converts floating-point numbers obtained by evaluating expressions of the form  $p/q$ ,  $p\pi/q$ ,  $\sqrt{p}$ ,  $2^q$ , and  $10^q$  for modest sized integers  $p$  and  $q$  to the corresponding symbolic form”. If a simple rational approximation cannot be found then the procedure for the `'f'` option is followed. This is the default conversion, and explains why exact representations of `0.1` and  $\sqrt{5}$  were obtained above.
- `sym(expr, 'f')`. This floating-point mode returns an exact rational representation of the floating-point number given by the expression `expr`.

While the rational mode is convenient, the floating-point mode may be more appropriate for numerical computations, as we will explain in Section 20.6.

The `sym` function can also set up arrays of symbolic variables:

```
>> v = sym('v', [1 4])
v =
[ v1, v2, v3, v4]

>> A = sym('A', [2 2])
A =
```

```
[ A1_1, A1_2]
[ A2_1, A2_2]
```

```
>> whos A v
  Name      Size      Bytes  Class  Attributes
  A         2x2         112    sym
  v         1x4         112    sym
```

## 20.2. Equation Solving

Suppose we wish to solve the quadratic equation  $ax^2 + bx + c = 0$ . We first define symbolic variables:

```
>> syms a b c x
```

Now we can solve the quadratic using the powerful `solve` command:

```
>> y = solve(a*x^2+b*x+c == 0)
y =
-(b + (b^2 - 4*a*c)^(1/2))/(2*a)
-(b - (b^2 - 4*a*c)^(1/2))/(2*a)
```

MATLAB creates a 2-by-1 symbolic object `y` to hold the two solutions. Alternatively, we could have typed

```
y = solve(a*x^2+b*x+c)
```

In this case, since we did not specify an equals sign, MATLAB assumes the expression we provided is to be equated to zero. Less obvious is how MATLAB knows to solve for `x` and not one of the other symbolic variables. MATLAB applied its `symvar` function to the expression `a*x^2+b*x+c` to determine the variable closest alphabetically to `x`, and solved for that variable. The same procedure is used by other functions in the toolbox. In each case, this choice can be overridden by specifying the variable or variables as extra arguments. Thus we can solve the same equation for `a` as follows:

```
>> solve(a*x^2+b*x+c,a)
ans =
-(c + b*x)/x^2
```

Suppose we now wish to check that the components of `y` really do satisfy the quadratic equation. We evaluate the quadratic at `y`, using elementwise squaring since `y` is a vector:

```
>> a*y.^2+b*y+c
ans =
c + (b + (b^2 - 4*a*c)^(1/2))^2/(4*a)
- (b*(b + (b^2 - 4*a*c)^(1/2)))/(2*a)
c + (b - (b^2 - 4*a*c)^(1/2))^2/(4*a)
- (b*(b - (b^2 - 4*a*c)^(1/2)))/(2*a)
```

The result is not displayed as zero, but we can apply the `simplify` function to try to reduce it to zero:

```
>> simplify(ans)
ans =
  0
  0
```

It is characteristic of symbolic manipulation packages that postprocessing is often required to put the results in the most useful form.

Having computed a symbolic solution, a common requirement is to evaluate it for numerical values of the parameters. This can be done using the `subs` function, which replaces all occurrences of symbolic variables by specified expressions. To find the roots of the quadratic  $x^2 - x - 1$  (cf. p. 176) we can type

```
>> a = 1; b = -1; c = -1;
>> subs(y)
ans =
  1/2 - 5^(1/2)/2
  5^(1/2)/2 + 1/2
```

When given one symbolic argument the `subs` command returns that argument with all variables replaced by their assigned values (if they have any) from the workspace. Alternatively, `subs` can be called with three arguments in order to assign values to variables without changing those variables in the workspace:

```
>> subs(y, {a, b, c}, {1, -1, -1})
ans =
  1/2 - 5^(1/2)/2
  5^(1/2)/2 + 1/2
```

Note that the second and third arguments are cell arrays (see Section 18.7). In both cases we can convert the result to numerical form using `double`:

```
>> double(ans)
ans =
 -0.6180
  1.6180
```

Simultaneous equations can be specified one at a time to the `solve` function. In general, the number of solutions cannot be predicted. There are two ways to collect the output. As in the next example, if the same number of output arguments as unknowns is supplied then the results are assigned to the outputs (alphabetically):

```
>> [xs, ys] = solve(x^2+y^2 == 1, x^3-y^3 == 1)
xs =
          1
          0
 - (2^(1/2)*1i)/2 - 1
  (2^(1/2)*1i)/2 - 1
ys =
          0
         -1
  1 - (2^(1/2)*1i)/2
  (2^(1/2)*1i)/2 + 1
```

Alternatively, a single output argument can be provided, in which case a structure (see Section 18.7) containing the solutions is returned:

```
>> eqns = [y == 1/(1+x^2), y == 1.001 - 0.5*x];
>> S = solve(eqns)
S =
    struct with fields:

        x: [3×1 sym]
        y: [3×1 sym]

>> [S.x(1), S.y(1)]
ans =
[ 1001/500 - 2*root(z^3 - (1001*z^2)/500 + (1252001*z)/1000000 ...
```

The fields of the structure have the names of the variables, and in this example we looked at the first of the three solutions and truncated the output. Solutions have been returned in terms of the roots of a cubic polynomial. We can check that the equations are indeed satisfied using the `isAlways` function, which returns true or false for each equation it is given:

```
>> isAlways(subs(eqns,S))
ans =
    3×2 logical array
     1     1
     1     1
     1     1
```

The `sym` and `syms` commands have optional arguments `real`, `positive`, `integer`, and `rational` for specifying assumptions on a variable. For example:

```
>> syms x real, syms n integer, syms p positive

>> assumptions x
ans =
in(x, 'real')

>> assumptions % For all variables.
ans =
[ 0 < p, in(x, 'real'), in(n, 'integer')]
```

Here, we used the `assumptions` command to confirm what assumptions have been set. All assumptions on `x` can be cleared with

```
assume(x, 'clear')
```

The information that a variable satisfies certain constraints can be vital in symbolic computations. For example, consider

```
>> syms p x y
>> y = ((x^p)^(p+1))/x^(p-1);
>> simplify(y)
ans =
x*(x^p)^p
```



The Symbolic Math Toolbox assumes that the variables  $x$  and  $p$  are complex and is unable to simplify  $y$  further. With the additional information that  $x$  and  $p$  are positive, further simplification is obtained:

```
>> syms p x positive
>> simplify(y)
ans =
x^(p^2+1)
```

For another example, the identity  $\text{acos}(\cos z) = z$  is not true for all complex  $z$ , but it is true with appropriate restrictions on  $z$ . Here, we use the `assume` command, which allows quite general assumptions to be specified:

```
>> syms z
>> simplify(acos(cos(z)))
ans =
acos(cos(z))

>> assume(z > 0 & z < 1)

>> simplify(acos(cos(z)))
ans =
z
```

The function `complex` does not accept symbolic arguments, so to set up complex symbolic expressions you must use `sym(sqrt(-1))` or `sym(i)`. For example:

```
>> syms x y
>> z = x + sym(sqrt(-1))*y;
>> expand(z^2)
x^2 + x*y*2i - y^2
```

## 20.3. Calculus

The Symbolic Math Toolbox provides powerful symbolic integration and differentiation capabilities.

### 20.3.1. Integration

Integration is carried out with the `int` function. Here is a simple test of `int`:

```
>> syms x
>> int(sym(1)), int(x)
ans =
x
ans =
x^2/2
```

Note that the constant of integration is always omitted. A more complicated example is

```
>> int(sqrt(tan(x)))
ans =
(2^(1/2)*(log(2^(1/2)*tan(x)^(1/2) - tan(x) - 1) - log(tan(x) +
2^(1/2)*tan(x)^(1/2) + 1)))/4 + (2^(1/2)*(atan(2^(1/2)*tan(x)^(1/2)
- 1) + atan(2^(1/2)*tan(x)^(1/2) + 1)))/2
```

This answer is not easy to read. One possibility is to “prettyprint” it in the Command Window with the command `pretty(ans)`, but this is still not very readable in this case. Another possibility is to use `latex(ans)` to convert the output to  $\text{\LaTeX}$  and then typeset the output, which gives (after splitting the expression over two lines)

$$\frac{\sqrt{2} \left( \log \left( \sqrt{2} \sqrt{\tan(x)} - \tan(x) - 1 \right) - \log \left( \tan(x) + \sqrt{2} \sqrt{\tan(x)} + 1 \right) \right)}{4} + \frac{\sqrt{2} \left( \arctan \left( \sqrt{2} \sqrt{\tan(x)} - 1 \right) + \arctan \left( \sqrt{2} \sqrt{\tan(x)} + 1 \right) \right)}{2}.$$

An easier solution is to carry out the computations within the Live Editor (see Section 16.7). Here is how this example appears in the Live Editor (the output is cut off at the right margin in the PDF file from which this is taken, but within the Live Editor it can be scrolled).

```
syms x
int(sqrt(tan(x)))
```

```
ans =
```

$$\frac{\sqrt{2} \left( \log \left( \sqrt{2} \sqrt{\tan(x)} - \tan(x) - 1 \right) - \log \left( \tan(x) + \sqrt{2} \sqrt{\tan(x)} + 1 \right) \right)}{4} + \frac{\sqrt{2} \left( \operatorname{atan} \left( \sqrt{2} \sqrt{\tan(x)} - 1 \right) + \operatorname{atan} \left( \sqrt{2} \sqrt{\tan(x)} + 1 \right) \right)}{2}$$

Definite integrals  $\int_a^b f(x) dx$  can be evaluated by appending the limits of integration  $a$  and  $b$ . The following integral evaluates to Catalan’s constant, with is a function in the toolbox and so can be evaluated numerically:

```
>> int(log(x)/(1+x^2),0,1)
```

```
ans =
-catalan
```

```
>> double(ans)
```

```
ans =
-9.1597e-01
```

Here is an integral that has a singularity at the left endpoint, but which nevertheless has a finite value:

```
>> int(atan(x)/x^(3/2),0,1)
```

```
ans =
(pi*2^(1/2))/2 - pi/2 - 2^(1/2)*log(2 - 2^(1/2))*(1/4 - 1i/4)
+ 2^(1/2)*log(-1/(2^(1/2) - 2))*(1/4 + 1i/4)
+ (2^(1/2)*log(2^(1/2) + 2))/2
```

The answer is exact and is rather complicated. We can convert it to numeric form:

```
>> double(ans)
ans =
    1.8971 - 0.0000i
```

The result has a tiny imaginary component (in fact of order  $10^{-73}$ ), but obviously the result should be real.

It is important to realize that symbolic manipulation packages cannot “do” all integrals. This may be because the integral does not have a closed-form solution in terms of elementary functions, or because it has a closed-form solution that the package cannot find. Here is an example of the first kind:

```
>> int(sqrt(1+cos(x)^2))
ans =
    2^(1/2)*ellipticE(x, 1/2)
```

The integral is expressed in terms of an elliptic integral of the second kind, which itself is not expressible in terms of elementary functions. If we evaluate the same integral in definite form we obtain

```
>> int(sqrt(1+cos(x)^2),0,48)
ans =
    2^(1/2)*ellipticE(48, 1/2)
```

and MATLAB can evaluate the elliptic integral therein:

```
>> double(ans)
ans =
    58.4705
```

As a final example, consider the integral [176]

$$\int_0^1 \frac{x^4(1-x)^4}{1+x^2} dx, \quad (20.1)$$

which is clearly positive. Evaluation of the integral produces a surprising result:

```
>> int(x^4*(1-x)^4/(1+x^2),0,1 )
ans =
    22/7-pi
```

Hence we have an unexpected demonstration that the well-known approximation  $22/7$  to  $\pi$  is a strict overestimate.

### 20.3.2. Differentiation

Symbolic differentiation is carried out using `diff`. This function was mentioned on p. 68 as a means for forming differences of a numerical vector or matrix, and it is overloaded for a symbolic argument. You can tell whether a given function is overloaded from its help entry. Typing `help diff` produces (with ... denoting omitted output)

```
>> help diff
diff Difference and approximate derivative.
    diff(X), for a vector X, is [X(2)-X(1) X(3)-X(2) ... X(n)-X(n-1)].
```

`diff(X)`, for a matrix `X`, is the matrix of row differences,  
 $[X(2:n,:) - X(1:n-1,:)]$ .

...

See also `gradient`, `sum`, `prod`.

Reference page for `diff`

Other functions named `diff`

Clicking on the hyperlinked “Other functions named `diff`” reveals several other versions of `diff`, one of which, `sym/diff`, handles symbolic arguments. To obtain help directly for the symbolic version of `diff` type `doc sym/diff`.

Here are some examples of the use of `diff` for symbolic arguments:

```
>> syms a x n
>> diff(x^2)
ans =
2*x

>> diff(x^n,2)
ans =
n*x^(n - 2)*(n - 1)

>> diff(sin(x)*exp(-a*x^2))
ans =
exp(-a*x^2)*cos(x) - 2*a*x*exp(-a*x^2)*sin(x)

>> diff(x^4*exp(x),3)
ans =
36*x^2*exp(x) + 12*x^3*exp(x) + x^4*exp(x) + 24*x*exp(x)

>> simplify(ans)
ans =
x*exp(x)*(x^3 + 12*x^2 + 36*x + 24)
```

In the second and fourth calls to `diff` a second argument specifies the order of the required derivative; the default is the first derivative.

The `syms` command can also set up functions of symbolic variables. Here, we differentiate the product and composition of functions:

```
>> syms f(x) g(x)
>> diff(f*g) % Product rule
ans(x) =
f(x)*diff(g(x), x) + g(x)*diff(f(x), x)

>> diff(f(g)) % Chain rule
ans =
D(f)(g(x))*diff(g(x), x)
```

We can obtain mixed partial derivatives of functions of more than one variable by explicitly specifying the variable with respect to which each differentiation is done:

```
>> syms x y
```

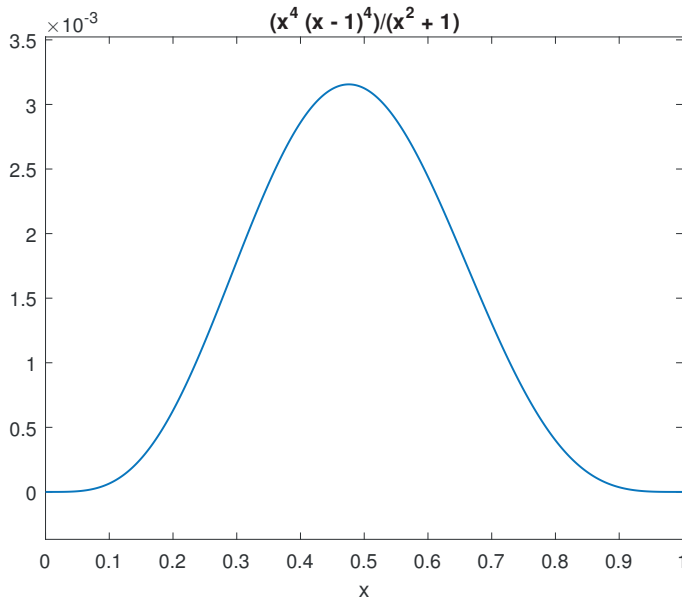


Figure 20.1. *The integrand in (20.1).*

```
>> f = x^2*exp(-y^2)-y/x
f =
x^2*exp(-y^2) - y/x

>> f_xy = diff(diff(f,x),y)
f_xy =
1/x^2 - 4*x*y*exp(-y^2)

>> f_yx = diff(diff(f,y),x)
f_yx =
1/x^2 - 4*x*y*exp(-y^2)
```

Finally, Figure 20.1 plots the integrand in (20.1). The plot appears symmetric and so the maximum might be thought to be at  $x = 0.5$ . To check, we can find the maximum symbolically:

```
>> g = diff( x^4*(1-x)^4/(1+x^2) ); s = double(solve(g))
s =
    0.0000 + 0.0000i
    1.0000 + 0.0000i
    0.4758 + 0.0000i
   -0.0712 - 1.1815i
   -0.0712 + 1.1815i
```

Three real stationary points have been found and the one that is the desired maximum is at 0.4758, not 0.5.

Other useful differentiation functions include `gradient`, `hessian`, and `jacobian`, for computing the multidimensional derivatives of the same names. For an example

of the use of `gradient` and `hessian` see Section 26.5.

### 20.3.3. Solving Differentiation Equations

Differential equations can be solved symbolically with `dsolve`. Initial conditions can optionally be specified after the equations, using the syntax  $y(\mathbf{a}) = \mathbf{b}$ ,  $Dy(\mathbf{a}) = \mathbf{c}$ , etc.; if none are specified then the solutions contain arbitrary constants of integration, denoted `C1`, `C2`, etc. For our first example we take the logistic differential equation

$$\frac{d}{dt}y(t) = cy - by^2,$$

solving it first with arbitrary  $c$  and  $b$ :

```
>> syms y(t) t b c
>> y = dsolve(diff(y) == c*y-b*y^2)
y =
                                0
                                c/b
(c*(tanh((c*(C2 + t))/2) + 1))/(2*b)
```

Three solutions have been returned, two of which are constant.

Now we solve the equation as an initial-value problem with particular values for  $b$  and  $c$  and then check that the solution satisfies the initial condition and the differential equation. Note that we need to initialize  $y(t)$  again in order to clear the previous solution:

```
>> syms y(t)
>> eqn = diff(y) - (10*y-y^2);
>> ys = dsolve(eqn,y(0) == 0.01)
ys =
10/(exp(log(999) - 10*t) + 1)

>> subs(ys,t,0)
ans =
1/100

>> res = simplify(subs(eqn,y,ys))
res(t) =
0
```

Next we try to find the general solution to the pendulum equation, which we solved numerically on p. 195. Here, we use a different syntax for expressing differential equations in which the equation is given as a string, with `D` denoting a first derivative, `D2` denoting a second derivative, and so on:

```
>> y = dsolve('D2theta + sin(theta) = 0')
y =
                                0
2*jacobiAM((2^(1/2)*(C15 - t)*(C13 - 1)^(1/2)*1i)/2, -2/(C13 - 1))
2*jacobiAM(-(2^(1/2)*(C15 - t)*(C13 - 1)^(1/2)*1i)/2, -2/(C13 - 1))
```

No explicit solution in terms of elementary functions could be found, but the solution is expressed in terms of the MuPAD Jacobi amplitude function `jacobiAM(u,m)`. If  $\theta$  is small we can approximate  $\sin \theta$  by  $\theta$ , and in this case `dsolve` is able to find both general and particular solutions:

```
>> y = dsolve('D2theta + theta = 0')
y =
C16*cos(t) + C17*sin(t)

>> y = dsolve('D2theta + theta = 0','theta(0)=1','Dtheta(0)=1')
y =
cos(t) + sin(t)
```

Note that the numbers associated with the `C` constants are unpredictable. Some constants are generated and used internally and not displayed. To force the constants to start being numbered at 1, you can use the command `reset(symengine)`, which restarts the symbolic engine (but does not clear the MATLAB workspace).

When using the string notation, if the independent variable is other than the default, `t`, it is important to specify the variable as the last input argument—otherwise, it will be treated as a constant:

```
>> y = dsolve('Dy-y*cos(x) = 0','y(0) = 1') % Incorrect if Dy=dy/dx.
y =
exp(t*cos(x))

>> y = dsolve('Dy-y*cos(x) = 0','y(0) = 1','x')
y =
exp(sin(x))
```

### 20.3.4. Taylor Expansions

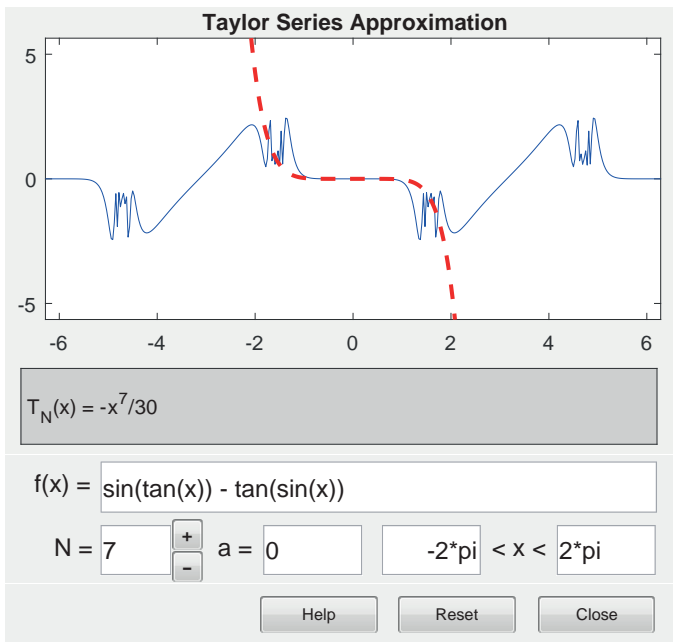
Taylor series can be computed using the function `taylor`:

```
>> syms x
>> taylor(log(1+x))
ans =
x^5/5 - x^4/4 + x^3/3 - x^2/2 + x
```

By default the Taylor series is expanded about 0 and terms up to order 5 are produced. Additional name–value arguments can specify different parameters. Terms up to degree less than the value specified for `'Order'` are included. Here, we use the `pretty` command to make the output more readable:

```
>> pretty(taylor(exp(-sin(x)),'Order',3,'ExpansionPoint',1))
      /
      | sin(1)  cos(1) |
exp(-sin(1)) + exp(-sin(1)) | ----- + ----- | (x - 1)
      \      2      2      /

- cos(1) exp(-sin(1)) (x - 1)
```

Figure 20.2. `taylor`tool window.

A function `taylor`tool provides a graphical interface to `taylor`, plotting both the function and the Taylor series. See Figure 20.2, which shows the interesting function  $\sin(\tan x) - \tan(\sin x)$ .

The input [176]

```
syms x, h = fplot(sin(x)+asin(x),[-0.8,0.8]); h.LineWidth = 1;
```

produces the plot in Figure 20.3. The curve looks straight, yet  $\sin$  and  $\arcsin$  have curvature. What is the explanation? Consider the following three Taylor series up to terms  $x^5$ :

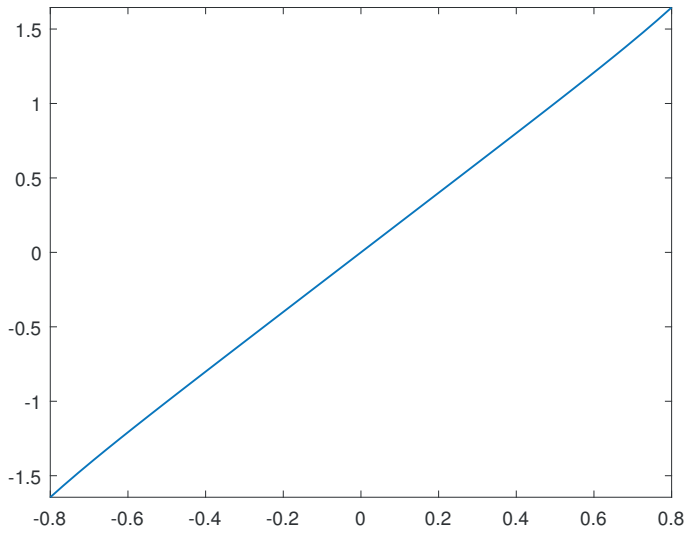
```
>> taylor(sin(x)), taylor(asin(x)), taylor(sin(x)+asin(x))
ans =
x^5/120 - x^3/6 + x
ans =
(3*x^5)/40 + x^3/6 + x
ans =
x^5/12 + 2*x
```

The  $x^3$  terms in the Taylor series for  $\sin$  and  $\arcsin$  cancel. Hence  $\sin x + \arcsin x$  agrees with  $2x$  up to terms of order  $x^5$ , and  $x^5$  is small on  $[-0.8, 0.8]$ .

## 20.4. Linear Algebra

Several of the MATLAB linear algebra functions have counterparts in the Symbolic Math Toolbox that take symbolic arguments. To illustrate we take the numeric and symbolic representations of the 5-by-5 Frank matrix:



Figure 20.3.  $\sin x + \arcsin x$ .

```
>> A_num = gallery('frank',5); A_sym = sym(A_num);
```

Since the Frank matrix has small integer entries the conversion is done exactly.

We can invert the double array `A_num` in the usual way:

```
inv(A_num)
ans =
    1.0000   -1.0000   -0.0000    0.0000     0
   -4.0000    5.0000   -1.0000   -0.0000     0
   12.0000  -15.0000    4.0000   -1.0000     0
  -24.0000   30.0000   -8.0000    3.0000   -1.0000
   24.0000  -30.0000    8.0000   -3.0000    2.0000
```

The trailing zeros show that the computed elements are not exactly integers. We can obtain the exact inverse by applying `inv` to `A_sym`, thereby invoking the Symbolic Math Toolbox's overloaded `inv` function:

```
inv(A_sym)
ans =
 [ 1, -1, 0, 0, 0]
 [ -4, 5, -1, 0, 0]
 [ 12, -15, 4, -1, 0]
 [ -24, 30, -8, 3, -1]
 [ 24, -30, 8, -3, 2]
```

Just as for numeric matrices, the backslash operator can be used to solve linear systems with a symbolic coefficient matrix. For example, we can compute the (5,1) element of the inverse of the Frank matrix with

```
>> [0 0 0 0 1]*(A_sym\[1 0 0 0 0]')
ans =
24
```

For a symbolic argument the `eig` function tries to compute the exact eigensystem. We know from Galois theory that this is not always possible in a finite number of operations for matrices of order 5 or more. For the 5-by-5 Frank matrix `eig` succeeds:

```
>> e = eig(A_sym)
e =
                                     1
7/2 - (55 - 14*10^(1/2))^(1/2)/2 - 10^(1/2)/2
(55 - 14*10^(1/2))^(1/2)/2 - 10^(1/2)/2 + 7/2
10^(1/2)/2 - (14*10^(1/2) + 55)^(1/2)/2 + 7/2
10^(1/2)/2 + (14*10^(1/2) + 55)^(1/2)/2 + 7/2

>> double(e)
ans =
    1.0000
    0.2812
    3.5566
    0.0994
   10.0629
```

As we noted in the example in Section 9.8.1, the eigenvalues come in reciprocal pairs. To check we can type

```
>> [e(2)*e(3); e(4)*e(5)]
ans =
-(55 - 14*10^(1/2))^(1/2)/2 - 10^(1/2)/2 + 7/2)*(10^(1/2)/2 + ...
(10^(1/2)/2 - (14*10^(1/2) + 55)^(1/2)/2 + 7/2)*(10^(1/2)/2 + ...

>> simplify(ans)
ans =
    1
    1
```

Note that we had to truncate the original output.

Finally, while we computed the characteristic polynomial numerically in Section 9.8.1, we can now obtain it exactly:

```
charpoly(A_sym,sym('x')) % Second argument for polynomial output.
ans =
x^5 - 15*x^4 + 55*x^3 - 55*x^2 + 15*x - 1
```

A partial list of linear algebra functions in the toolbox is given in Table 20.1; most of these overload corresponding functions defined for numeric arguments.

## 20.5. Polynomials and Rationals

Symbolic polynomial and rational expressions are easily formed using symbolic variables and the usual MATLAB notation. The function `coeffs` returns the (symbolic) coefficients and corresponding terms of a polynomial. The functions `sym2poly` and `poly2sym` convert between a symbolic polynomial and a numeric vector of coefficients of the polynomial. In all three functions the ordering of the coefficients and terms is in the standard MATLAB way from “highest power down to lowest power”. Examples:

Table 20.1. *Linear algebra functions in the Symbolic Math Toolbox.*

<code>diag</code>	Diagonal matrices and diagonals of matrix
<code>tril</code>	Extract lower triangular part
<code>triu</code>	Extract upper triangular part
<code>inv</code>	Matrix inverse
<code>det</code>	Determinant
<code>rank</code>	Rank
<code>rref</code>	Reduced row echelon form
<code>null</code>	Basis for null space (not orthonormal)
<code>orth</code>	Orthogonalization
<code>rank</code>	Rank
<code>chol</code>	Cholesky factorization
<code>lu</code>	LU factorization
<code>qr</code>	QR factorization
<code>eig</code>	Eigenvalues and eigenvectors
<code>svd</code>	Singular values and singular vectors
<code>poly</code>	Characteristic polynomial
<code>expm</code>	Matrix exponential
<code>logm</code>	Matrix logarithm
<code>funm</code>	General matrix function
<code>sqrtm</code>	Matrix square root
<code>colspace*</code>	Basis for column space
<code>jordan*</code>	Jordan canonical (normal) form
<code>smithForm*</code>	Smith normal form

\* Functions defined for symbolic arguments but not for doubles.

```

>> syms x
>> p = (2/3)*x^3-x^2-3*x+1
p =
(2*x^3)/3 - x^2 - 3*x + 1

>> [c,terms] = coeffs(p,x)
c =
[ 2/3, -1, -3, 1]
terms =
[ x^3, x^2, x, 1]

>> a = sym2poly(p)
a =
    0.6667    -1.0000    -3.0000     1.0000

>> q = poly2sym(a)
q =
(2*x^3)/3 - x^2 - 3*x + 1

```

As the coefficient of  $x^3$  in this example illustrates, `poly2sym` (which calls `sym`) attempts to convert floating-point numbers to nearby rationals. There is no function to return the degree of a polynomial, but it can be computed as `length(sym2poly(p)) - 1`.

Division of one polynomial by another is done by `quorem`, which returns the quotient and remainder (cf. `deconv` on p. 176):

```

>> [q,r] = quorem(p,x^2)
q =
(2*x)/3 - 1
r =
1 - 3*x

```

The function `numden` converts a rational into a normal form where the numerator and denominator are polynomials with integer coefficients and then returns the numerator and denominator. For a numeric argument, but not necessarily for symbolic ones, the returned numerator and denominator will be relatively prime:

```

>> [n,d] = numden(sym(16)/sym(84))
n =
4
d =
21

>> r = 1 + x^2/(3+x^2/5);
>> [p,q] = numden(r)
p =
6*x^2 + 15
q =
x^2 + 15

```

The `sort` function is overloaded for symbolic arguments. It sorts a polynomial in decreasing order of the powers:

Table 20.2. *Special polynomials.*

bernstein	Bernstein polynomials
chebyshevT	Chebyshev polynomials of the first kind
chebyshevU	Chebyshev polynomials of the second kind
gegenbauerC	Gegenbauer (ultraspherical) polynomials
hermiteH	Hermite polynomials
jacobiP	Jacobi polynomials
laguerreL	Generalized Laguerre function and Laguerre polynomials
legendreP	Legendre polynomials

```
>> p = x^2-3-3*x^3+x/2;
>> p = sort(p)
p =
- 3*x^3 + x^2 + x/2 - 3
```

Several special polynomials, mainly orthogonal polynomials, are provided: see Table 20.2. Figure 20.4 shows the output from the following code:

```
syms x, a = 1; b = 1;
hold on, grid on
for n = 1:5
    fplot(jacobiP(n,a,b,x),[-1 1],'LineWidth',1.5)
end
title(['Jacobi polynomials with a = ' num2str(a) ...
      ' and b = ' num2str(b)])
h = legend('1','2','3','4','5','Location','best');
legend('boxoff'), title(h,'Degree'), hold off
```

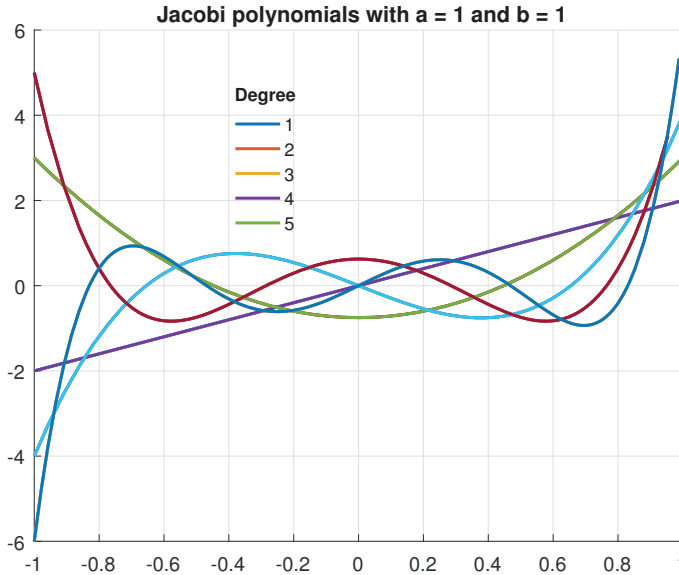
The `partfrac` function computes the partial fraction form of a rational function, optionally factoring the denominator in complex arithmetic:

```
>> partfrac((x^3+1)/(x^2+4))
ans =
x - (4*x - 1)/(x^2 + 4)

>> pretty(partfrac((x^3+1)/(x^2+4),'FactorMode','Complex'))
x + ----- + -----
      x - 2.0i      x + 2.0i
```

Finally, suppose we wish to evaluate a symbolic polynomial at a matrix argument. Take, for example, the polynomial  $p(x) = x^2 + x - 1$  and the matrix `diag(1,2,3)`. Here are two different attempts:

```
>> syms x; p = x^2 + x - 1;
>> A = sym(diag([1,2,3]));
>> P1 = polyvalm(sym2poly(p),A)
P1 =
[ 1, 0, 0]
```

Figure 20.4. *Jacobi polynomials.*

```
[ 0, 5, 0]
[ 0, 0, 11]
```

```
>> P2 = subs(p,x,A)
P2 =
[ 1, -1, -1]
[ -1, 5, -1]
[ -1, -1, 11]
```

The first result, P1, is the desired evaluation  $p(A) = A^2 + A - I$  in the matrix sense. The second evaluation gives a different answer because the `subs` function expands scalars into matrices: it converts the 1 into a matrix of 1s, whereas the first evaluation converts 1 into an identity matrix.

## 20.6. Variable-Precision Arithmetic

In addition to symbolic arithmetic, the Symbolic Math Toolbox supports variable-precision floating-point arithmetic. This is useful for problems where an accurate solution is required and an exact solution is impossible or too time-consuming to obtain. It can also be used to experiment with the effect of varying the precision of a computation.

The function `digits` returns the number of significant decimal digits to which variable-precision computations are carried out:

```
>> digits

Digits = 32
```

The default of 32 digits can be changed to `n` by the command `digits(n)`. Variable-precision computations are based on the `vpa` command. The simplest usage is to evaluate constants to variable accuracy:

```
>> pi_1 = vpa(pi)
pi_1 =
3.1415926535897932384626433832795
```

It is important to note the distinction between `pi_1`, a 32-digit approximation to  $\pi$ , and the exact representation

```
>> pi_2 = sym(pi)
pi_2 =
pi
```

The difference is apparent from

```
>> sin(pi_1)
ans =
-3.2101083013100396069547145883568e-40

>> sin(pi_2)
ans =
0
```

Note, however that both `pi_1` and `pi_2` are syms:

```
>> whos pi*
Name          Size          Bytes  Class    Attributes

pi_1          1x1            112   sym
pi_2          1x1            112   sym
```

The `vpa` function takes a second argument that overrides the current number of digits specified by `digits`:

```
>> vpa(pi,50)
ans =
3.1415926535897932384626433832795028841971693993751
```

In the next example we compute  $e$  to 40 digits and then check that taking the logarithm gives back 1 (to within 40 digits). By writing `sym(1)` we ensure that the exponential is evaluated symbolically before being converted to `vpa` form:

```
>> digits(40)
>> x = vpa(exp(sym(1)))
x =
2.718281828459045235360287471352662497757

>> vpa(log(x)) - 1
ans =
0.0
```

A minor modification of this example illustrates some pitfalls:

```

% Incorrect code.
>> digits(40)
>> y = vpa(exp(1))
y =
2.718281828459045534884808148490265011787

>> vpa(log(y))
ans =
1.000000000000000110188913283849495821818

```

We omitted to convert the 1 to a `sym`, so MATLAB evaluated `exp(1)` in double-precision floating-point arithmetic, converted that 16-digit result to 40 digits—thereby adding 24 meaningless digits—and then evaluated the exponential.

The function `vpasolve` is a counterpart to the numeric `fzero` and symbolic `solve` functions. Consider the following function based on that in Figure 20.2, which has many zeros near  $-\pi/2$ :

```

>> syms x, f = sin(tan(x)) - tan(sin(x)) - pi/2;
>> solve(f)
Warning: Cannot solve symbolically. Returning a numeric
proximation instead.
> In solve (line 303)
ans =
-234.88796307659113165754267656334

```

The `sym` function cannot find a symbolic solution and returns one numeric root; `vpasolve` produces the same root:

```

vpasolve(f)
ans =
-234.88796307659113165754267656334

```

However, `vpasolve` allows us to specify a starting value for the iterative method employed or an interval on which to search, and it can also choose a random starting value:

```

>> vpasolve(f, x, -pi/2)
ans =
-1.5707963267948966192313216916398

>> vpasolve(f, -pi/2 + [-0.1,0.1])
ans =
-1.4912337833414379465677549335376

>> vpasolve(f, -pi/2 + [-0.1,0.1], 'random', true)
ans =
-1.6500551377834450931692298548548

>> vpasolve(f, -pi/2 + [-0.1,0.1], 'random', true)
ans =
-1.5508990160785597633147812079701

```

The search interval can also be a rectangle in the complex plane:



```
>> vpasolve(f,2*[-1-i, 1+i],'random',true)
ans =
1.3891902153285393920800245007443 +
1.3046293160200280622294932555953i
```

Variable-precision linear algebra computations are performed by calling functions with variable-precision arguments. For example, we can compute the eigensystem of `pascal(4)` to 32 digits by

```
>> digits(32)
>> [V,E] = eig(vpa(pascal(4))); diag(E)
ans =
26.304703267097871286055226455526
2.2034461676473233016100756770366
0.45383455002566546509718436703794
0.038016015229139947237513500399509
```

The `vpa` function makes use of `sym` with the default rational mode conversion option. This can lead to unwanted conversion errors if a floating-point number happens to be close to being rational. A case in point is when one uses `vpa` arithmetic to compute the “exact” solution to a floating-point problem. Suppose we want to know the error in the eigenvalues computed by `eig` in the following example:

```
>> A = gallery('chebspec',4,1)
A =
-0.7071    -1.4142     0.7071    -0.2929
 1.4142     -0.0000    -1.4142     0.5000
-0.7071     1.4142     0.7071    -1.7071
 1.1716     -2.0000     6.8284    -5.5000

>> e = eig(A)    % Solution computed in double-precision arithmetic.
e =
-0.9588 + 2.3308i
-0.9588 - 2.3308i
-1.7912 + 0.7551i
-1.7912 - 0.7551i
```

We will compute the eigenvalues in 50-digit precision, convert the result back to double precision (hence obtaining the rounded version of the exact answer), and then compute the error. To do so, we need first to convert `A` to 50-digit `vpa` form. Simply evaluating `vpa(A)` will not give the desired result, however, as can be seen by looking at the (1,1) elements:

```
>> vpa(A(1,1))
ans =
-0.70710678118654752440084436210484903928483593768847

>> vpa(sym(A(1,1),'r')) % Same result as the previous vpa call.
ans =
-0.70710678118654752440084436210484903928483593768847

>> sym(A(1,1),'r')
```

```

ans =
-2^(1/2)/2

>> vpa(sym(A(1,1),'f')) % Force exact conversion.
ans =
-0.70710678118654768375961339188506826758384704589844

```

We can see that `vpa(A(1,1))` is converting `A(1,1)` to a nearby expression involving a square root and then approximating that to 50 digits, instead of exactly representing the floating-point value in 50-digit precision. We therefore need to use `sym` with the floating-point mode conversion option:

```

>> ex = eig(vpa(sym(A,'f'))); % Solution in 50-digit arithmetic.
>> format short e
>> err = e-double(ex)
err =
-1.8874e-15 + 1.3323e-15i
-1.8874e-15 - 1.3323e-15i
 1.9984e-15 - 9.9920e-16i
 1.9984e-15 + 9.9920e-16i

```

The output shows that the solution computed in double precision is accurate in essentially all of its 16 digits.

## 20.7. Other Features

The Symbolic Math Toolbox contains many other functions in areas we have not touched on, including the following.

- Fourier and Laplace transforms.
- The Dirac delta function and the Heaviside step function.
- The gamma function and related functions.
- The Riemann zeta function and the polylogarithm.
- Airy functions and Bessel functions.
- Error and exponential integral functions. (Among these functions are the Fresnel integrals, `fresnelc` and `fresnels`, which provide another way to evaluate the Fresnel spiral in Figure 12.1.)
- Hypergeometric and Whittaker functions.
- The Lambert W function and the Wright function.

Useful functions for postprocessing are `ccode`, `fortran`, `latex`, and `matlabFunction`, which produce C, Fortran,  $\text{\LaTeX}$ , and function handle representations, respectively, of a symbolic expression.

*[Babbage's Analytical Engine] can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraical notation, were provisions made accordingly.*

— AUGUSTA ADA BYRON, *Countess of Lovelace* (1843)

*I'm very good at integral and differential calculus,  
I know the scientific names of beings animalculous;  
In short, in matters vegetable, animal, and mineral,  
I am the very model of a modern Major-General.*

— WILLIAM SCHWENCK GILBERT, *The Pirates of Penzance*. Act 1 (1879)

*The particular form obtained by applying an analytical integration method may prove to be unsuitable for practical purposes.  
For instance, evaluating the formula may be numerically unstable (due to cancellation, for instance) or even impossible (due to division by zero).*

— ARNOLD R. KROMMER and CHRISTOPH W. UEBERHUBER, *Computational Integration* (1998)

*Maple has bugs. It has always had bugs...  
Every other computer algebra system also has bugs;  
often different ones,  
but remarkably many of these bugs are seen  
throughout all computer algebra systems,  
as a result of common design shortcomings.  
Probably the most useful advice I can give for dealing with this is  
be paranoid.  
Check your results at least two ways (the more the better).*

— ROB CORLESS, *Essential Maple 7* (2002)

# Chapter 21

## Graphs

A graph comprises a set of nodes (or vertices) and a set of edges (or links), with each edge connecting a pair of nodes. If the edges have a direction the graph is directed, otherwise it is undirected. MATLAB has classes `graph` and `digraph` for representing undirected and directed graphs, respectively. These classes allow node labels to be stored and  $x$ - $y$  locations to be specified (where appropriate), and they have a collection of functions for computing with graphs and visualizing them.

A graph can be characterized by an adjacency matrix, which is a matrix  $A$  of zeros and ones. For a directed graph,  $a_{ij} = 1$  if there is an edge from node  $i$  to node  $j$  and  $a_{ij} = 0$  otherwise. For an undirected graph,  $a_{ij} = 1$  if nodes  $i$  and  $j$  are connected by an edge and  $a_{ij} = 0$  otherwise. The adjacency matrix is symmetric for an undirected graph and nonsymmetric, in general, for a directed one.

### 21.1. Undirected Graphs

We begin with a simple example of how to construct a graph:

```
>> a = [1 1 1 2 2 3 3 4 4 5 7];
>> b = [2 3 4 3 7 4 6 5 7 6 8];
>> names = {'A','B','C','D','E','F','G','H'};
>> G = graph(a,b,[],names)
G =
    graph with properties:

    Edges: [11×1 table]
    Nodes: [8×1 table]

>> [numedges(G), numnodes(G)]
ans =
    11     8

>> degree(G)'
ans =
     3     3     4     4     2     2     3     1
```

Here, we have set up a graph with nodes named A to H and have specified the edges in the vectors `a` and `b`: there is an edge from the `a(i)`th node to the `b(i)`th node for  $i$  from 1 to `length(a)`. The third argument to `graph` is a vector that specifies weights to be applied to the edges; since we are constructing an unweighted graph (a graph for which every edge has unit weight) this argument is left empty. The `numedges` and `numnodes` functions return the number of edges and nodes in the graph,

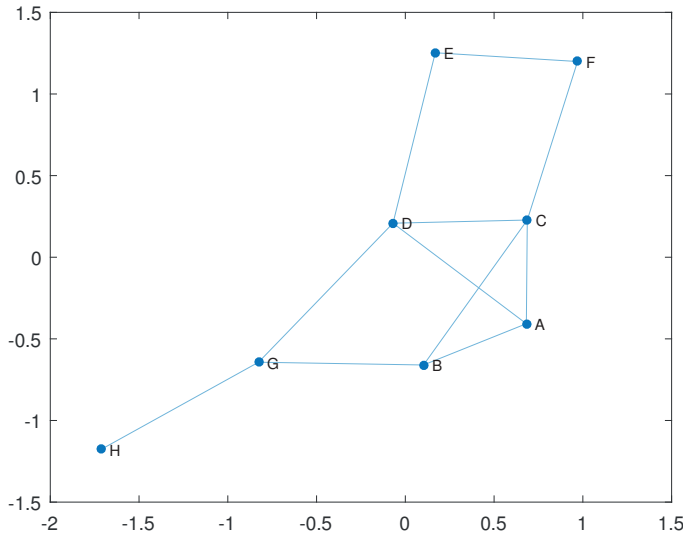


Figure 21.1. *Undirected graph.*

respectively. For undirected graphs, the `degree` function returns the degree of each node, which is the number of edges connected to the node. The function `plot` has been overloaded to visualize graphs, so the command `plot(G)` produces Figure 21.1. For small graphs such as this, `plot` shows the node labels. The `plot` function has several layout options, selected with the 'Layout' input argument, and by default chooses the layout automatically based on the size and structure of the graph.

Now we give each edge in the graph an associated weight, to produce a weighted graph, plot the graph with the weights shown, then compute and highlight the minimum spanning tree. A tree is an undirected graph in which any two nodes are connected by exactly one path. A spanning tree of a connected graph (one with a path between any two nodes) is a subgraph forming a tree that includes all the nodes (but usually not all the edges). A *minimum* spanning tree is a spanning tree for which the sum of the weights along the edges is minimal:

```
w = [5 3 1 2 6 3 1 9 4 4 7];
G = graph(a,b,w, names);
h = plot(G, 'EdgeLabel', G.Edges.Weight);
[T,p] = minspantree(G);
highlight(h,T)
axis off, box off
```

The plot is shown in Figure 21.2, with the edges of the minimal spanning tree thickened using the `highlight` function.

Next, we generate a random graph by sampling from a preferential attachment model, in which the graph is built node by node and a new node links to an existing node with a probability that is proportional to the current degree of that node. The graph is generated using the CONTEST [163] toolbox. The toolbox returns an adjacency matrix, so we use the ability of the `graph` function to accept an adjacency matrix as its first argument. The graph, shown in Figure 21.3, is undirected. We use the `shortestpathtree` function to find the shortest paths that connect node 1 with

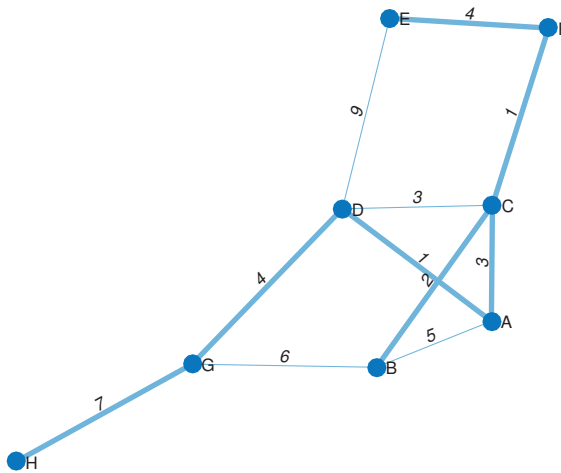


Figure 21.2. *Weighted undirected graph and a minimum spanning tree.*

the other nodes, highlight these paths on the plot, and then plot the tree, which is shown in Figure 21.4:

```
rng(25)
A = pref(100); % Function from CONTEST toolbox.
G = graph(A);
figure(1)
p = plot(G, 'NodeLabel', {});
axis tight, axis off
tree = shortestpathtree(G,1);
highlight(p,tree,'EdgeColor','r')
highlight(p,1,'NodeColor','r')
figure(2)
plot(tree)
axis tight, axis off
```

If the individual shortest paths are required they can be obtained with the command

```
tree = shortestpathtree(G,1,'OutputForm','cell');
```

which returns the paths in a cell array.

Table 21.1 lists some of the functions for working with graphs.

The computational functions have options to select between different algorithms, some of which may be applicable only to certain types of graph. For example, the `shortestpathtree` function includes Dijkstra's algorithm and the Bellman–Ford algorithm. The 'method' name–value argument is used to select between the algorithms.

## 21.2. Directed Graphs

Now we focus on directed graphs, in which the edges have a direction.

The next example creates and displays a directed graph whose nodes represent a selection of MATLAB topics. An edge from node  $i$  to node  $j$  indicates that topic  $j$  builds on topic  $i$ :

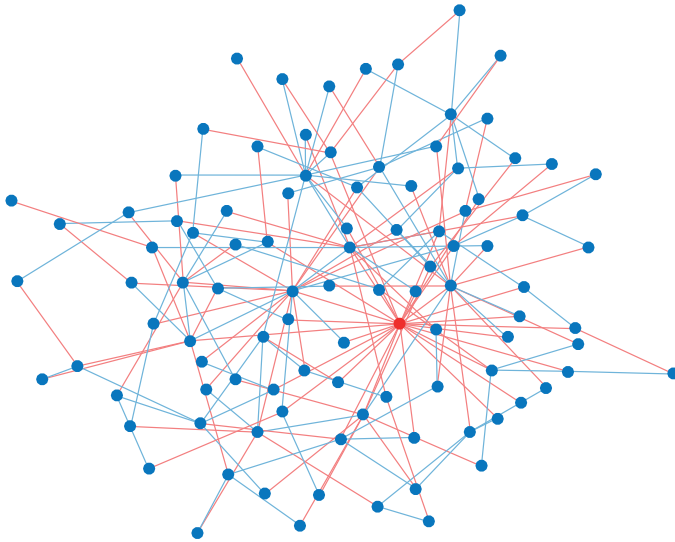


Figure 21.3. *Random graph from preferential attachment model. Red edges mark shortest paths from red node to the other nodes.*

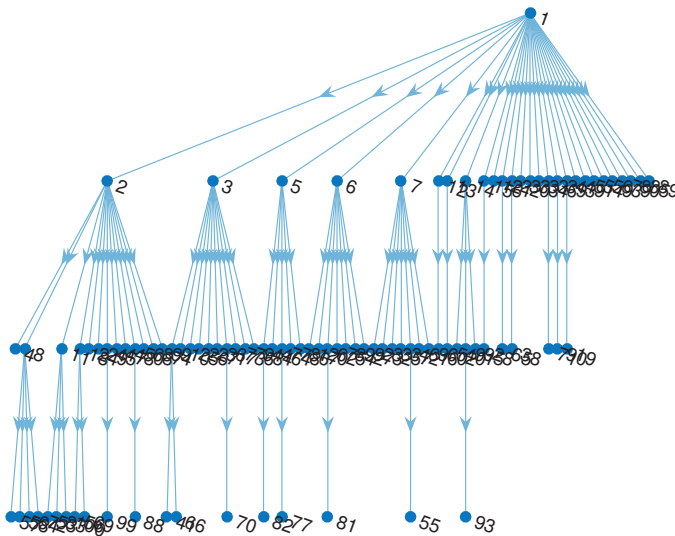


Figure 21.4. *Shortest path tree for graph in Figure 21.3.*

Table 21.1. *Selected graph functions.*

<code>graph</code>	Construct undirected graph
<code>digraph</code>	Construct directed graph
<code>plot</code>	Plot graph
<code>highlight</code>	Highlight nodes/edges in plotted graph
<code>degree</code>	Degree of graph nodes
<code>numedges</code>	Number of edges in graph
<code>numnodes</code>	Number of nodes in graph
<code>neighbors</code>	Neighbors of graph node
<code>nearest</code>	Nearest neighbors within radius
<code>findedge</code>	Locate edge in graph
<code>findnode</code>	Locate node in graph
<code>reordernodes</code>	Reorder graph nodes
<code>subgraph</code>	Extract subgraph
<code>bfsearch</code>	Breadth-first graph search
<code>dfsearch</code>	Depth-first graph search
<code>conncomp</code>	Connected graph components
<code>maxflow</code>	Maximum flow in graph
<code>minspantree</code>	Minimum spanning tree of graph
<code>centrality</code>	Quantify relative importance of nodes
<code>isomorphism</code>	Compute equivalence relation between two graphs
<code>isisomorphic</code>	Determine whether two graphs are isomorphic
<code>shortestpath</code>	Shortest path between two single nodes
<code>shortestpathtree</code>	Shortest path tree from node



```

A = [0 1 0 0 1 0 0 0 0 1 0 0;
      0 0 1 1 0 0 1 0 1 0 0 0;
      0 0 0 0 0 0 0 0 0 0 0 0;
      0 0 0 0 0 0 0 0 0 0 0 0;
      0 0 0 0 0 1 0 0 0 0 0 0;
      0 0 0 0 0 0 1 1 0 0 0 0;
      0 0 0 0 0 0 0 0 0 0 0 0;
      0 0 0 0 0 0 0 0 1 0 0 0;
      0 0 0 0 0 0 0 0 0 0 0 0;
      0 0 0 0 0 0 0 0 0 0 1 1;
      0 0 0 0 0 0 0 0 0 0 0 0;
      0 0 0 0 0 0 0 0 0 0 0 0];
names = {'Data Types','Matrices','Sparse Matrices', ...
         'Multi-Dim Arrays','Program Files','Functions','Graphs',...
         'ODEs','Stiff ODEs','Basic Plots','3D Graphics',...
         'Graphics Objects'};
G = digraph(A,names);
h = plot(G);
h.EdgeColor = 'm';
h.LineWidth = 1.5;
h.Marker = 'p';
h.MarkerSize = 10;
axis off

```

Figure 21.5 shows the plot. We specified the color and width of the edges and the style and size of the node markers. Because this directed graph has no cycles, we are able to construct a topological ordering of the nodes; that is, an ordering where every edge connects a node to one that is further down the list:

```

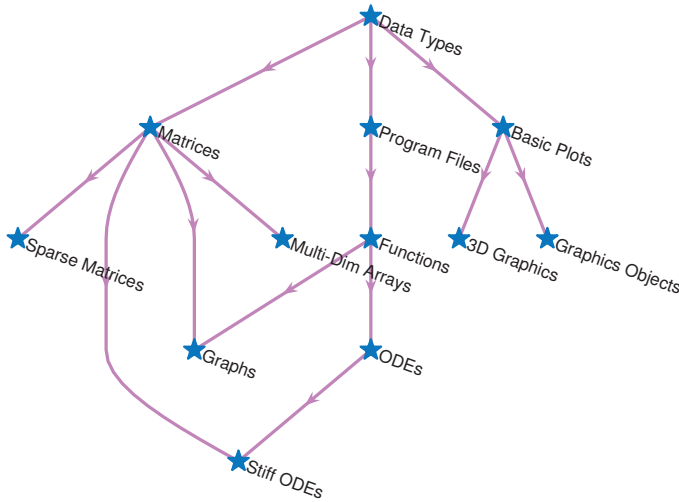
>> N = toposort(G)
N =
     1    10    12    11     5     6     8     2     9     7     4     3

>> names(N)
ans =
    12×1 cell array
    'Data Types'
    'Basic Plots'
    'Graphics Objects'
    '3D Graphics'
    'Program Files'
    'Functions'
    'ODEs'
    'Matrices'
    'Stiff ODEs'
    'Graphs'
    'Multi-Dim Arrays'
    'Sparse Matrices'

```

This provides a suitable order in which to digest the topics.

Figure 21.6 shows a larger directed network. Here, the 131 nodes represent

Figure 21.5. *Directed graph.*

frontal neurons in *C. elegans*, a transparent roundworm of length about 1 millimeter. The 764 directed edges correspond to experimentally observed anatomical connections and the nodes have well-defined locations in 2D physical space. Using the `celegans131.mat` file from Marcus Kaiser's web page at [http://www.dynamic-connectome.org/?page\\_id=25](http://www.dynamic-connectome.org/?page_id=25) we constructed the figure as follows:

```
load celegans131
% Gives celegans131labels, celegans131matrix, celegans131positions
G = digraph(celegans131matrix,celegans131labels);
h = plot(G,'XData',celegans131positions(:,1),...
        'YData',celegans131positions(:,2));
h.NodeColor = 'k'; set(h,'EdgeAlpha',1), set(h,'MarkerSize',4)
axis tight
```

Because of the size of the network, node labels were automatically suppressed. Note that the vertical scaling is an order of magnitude smaller than the horizontal scaling—the worm is long and thin.

To experiment further, we select the subgraph induced by the first 30 nodes, which is shown in Figure 21.7:

```
Gsub = subgraph(G,[1:30]);
X = celegans131positions(1:30,1);
Y = celegans131positions(1:30,2);
h = plot(Gsub,'XData',X,'YData',Y);
h.NodeColor = 'k'; set(h,'EdgeAlpha',1)
axis tight
```

Next we use the `centrality` function to quantify the relative importance of each node. This function supports several measures from the network science literature. We choose the measure attributed to Google's PageRank algorithm:

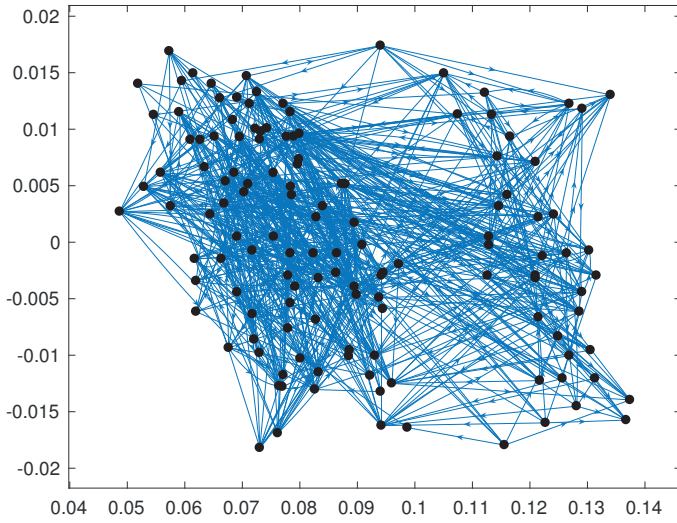


Figure 21.6. *Neuronal network of C. elegans.*

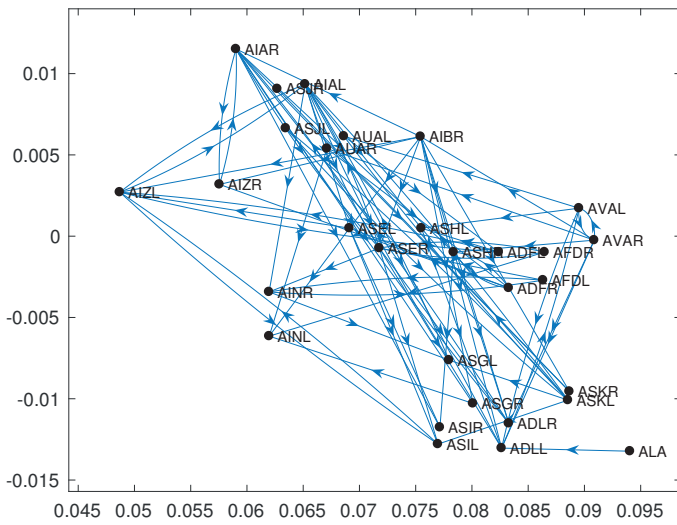


Figure 21.7. *Subnetwork from C. elegans data.*

```

>> pg_ranks = centrality(Gsub,'pagerank')
>> pg_ranks
pg_ranks =
Columns 1 through 6
    0.0533    0.0256    0.1011    0.0291    0.0180    0.0111
Columns 7 through 12
    0.0618    0.0425    0.0111    0.0334    0.0253    0.0794
Columns 13 through 18
    0.0133    0.0072    0.0256    0.0323    0.0285    0.0133
Columns 19 through 24
    0.0610    0.0281    0.0430    0.0258    0.0112    0.0185
Columns 25 through 30
    0.0233    0.0133    0.0278    0.0111    0.0971    0.0278

```

To visualize these centralities, we first reduce clutter by ignoring physical  $x$ - $y$  coordinates and using the default graph layout. Then we separate the nodes into three categories according to their centrality scores, and use corresponding marker sizes for the nodes. Here, we make use of the `discretize` function, which groups numeric values into discrete bins:

```

p = plot(Gsub);
axis off
Gsubedges = linspace(min(pg_ranks),max(pg_ranks),4);
Gsubbins = discretize(pg_ranks,Gsubedges);
p.MarkerSize = 5*Gsubbins;
p.NodeColor = 'r'; set(p,'EdgeAlpha',1)
p.EdgeColor = [0.61,0.30,0.08]; % saddlebrown

```

Figure 21.8 shows the result. Finally, we sort the centrality scores and display the top five nodes:

```

>> [pg,pgind] = sort(pg_ranks,'Descend');
>> top5 = celegans131labels(pgind(1:5))
top5 =
5×1 cell array
    'ADLL'
    'AVAL'
    'AIZL'
    'AIAL'
    'ASHL'

```

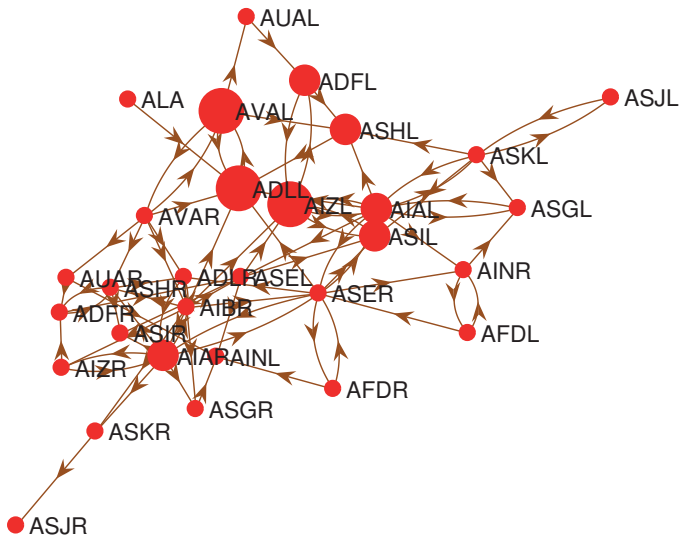


Figure 21.8. *Visualization of PageRank centrality in C. elegans subnetwork.*

#### CHEMICAL GRAPHS

Use of the word *graph* in the sense of this chapter first arose in the nineteenth century. Mathematicians realized that the objects they were studying had close connections with the *graphic notation* that chemists were using to describe molecular formulas. For example, methane,  $\text{CH}_4$ , could be represented as a central node, C, of degree four connected to four leaf nodes, H, of degree one. Sylvester (who coined the word “matrix” in 1850, and contributed much to the early development of matrix theory [72]) wrote [162]

“It may not be wholly without interest to some of the readers of *Nature* to be made acquainted with an analogy that has recently forcibly impressed me between branches of human knowledge apparently so dissimilar as modern chemistry and modern algebra.”

*The theory of graphs generally deals only with the number of elements in the network and their relationships with respect to each other in terms of the characteristics of the edge set. This very broad definition is capable of representing systems in bewildering detail.*

— DUNCAN J. WATTS, *Small Worlds. The Dynamics of Networks between Order and Randomness* (1999)

*Although graphs can be represented pictorially, most computations of graph properties are accomplished by way of either an adjacency matrix or adjacency list.*

— DUNCAN J. WATTS, *Small Worlds. The Dynamics of Networks between Order and Randomness* (1999)

*When ideas and tools move from one field to another, the movement is generally from the natural to the social sciences. In recent years, however, there has been a major movement in the opposite direction. The idea of centrality and the tools for its measurement were originally developed in the social science field of social network analysis. But currently the concept and tools of centrality are being used widely in physics and biology.*

— LINTON C. FREEMAN, *Going the Wrong Way Down a One-Way Street: Centrality in Physics and Biology* (2008)

# Chapter 22

## Large Data Sets

MATLAB has special features for working with data sets that are too large to fit into the computer's random access memory (RAM). In this chapter we briefly introduce some of these features.

### 22.1. Datastores

A datastore is a repository for collections of data that are arranged in one or more files having the same structure and formatting and are too large to fit in memory.

The best way to introduce datastores is with a simple example. We make use of a CSV file `twitter.csv`, which is a Twitter archive of one of the authors' Twitter accounts and consists of lines comprising fields (numbers or text) separated by commas. In our case the first line of the file is a header line whose fields contain column names.

We first create the datastore:

```
>> ds = datastore('twitter.csv')
ds =
    TabularTextDatastore with properties:

        Files: {
                'd:\twitter.csv'
               }
    FileEncoding: 'UTF-8'
    ReadVariableNames: true
    VariableNames: {'tweet_id', 'in_reply_to_status_id', ...
                   'in_reply_to_user_id' ... and 7 more}

Text Format Properties:
    NumHeaderLines: 0
    Delimiter: ','
    RowDelimiter: '\r\n'
    TreatAsMissing: ''
    MissingValue: NaN

Advanced Text Format Properties:
    TextscanFormats: {'%q', '%q', '%q' ... and 7 more}
    ExponentCharacters: 'eEdD'
    CommentStyle: ''
    Whitespace: ' \b\t'
```

```
MultipleDelimitersAsOne: false
```

```
Properties that control the table returned by preview, read, readall:
  SelectedVariableNames: {'tweet_id', 'in_reply_to_status_id', ...
                          'in_reply_to_user_id' ... and 7 more}
  SelectedFormats: {'%q', '%q', '%q' ... and 7 more}
  ReadSize: 20000 rows
```

We can preview the data (view a subset of it) using the `preview` function, but first we choose the columns to be shown, as a full row contains too much information to display:

```
>> ds.SelectedVariableNames = {'tweet_id','source'};
>> data = preview(ds)
data =
```

tweet_id	source
'685478255789473793'	'Twitter Web Client'
'685478112889565184'	'Twitter Web Client'
'685100412215685120'	'Tweetbot for iOS'
'684732471183818752'	'Hootsuite'
'684349803468316672'	'Hootsuite'
'684050694626766848'	'Twitter Web Client'
'684022799053250560'	'Hootsuite'
'683977163767439362'	'Twitter Web Client'

```
>> whos
```

Name	Size	Bytes	Class	Attributes
data	8×2	4266	table	
ds	1×1	8	matlab.io.datastore.TabularTextDatastore	

Notice that `data` is of class `table` (see Section 18.6).

At this point only the initial rows of `twitter.csv` shown in the preview have been read in. We can start to read the complete file using the `read` function:

```
>> tweetdata = read(ds); % Read in first chunk of data.

>> size(tweetdata)
ans =
    1319         2

>> hasdata(ds) % Is there any more data to read in?
ans =
    logical
     0
```

Data is read in a chunk at a time, the (default) size of a chunk being the `ReadSize`, which is 20,000 rows, as shown by the output from the `datastore` command. With only 1,319 rows this is a very small data set that can easily be held entirely in memory. But there are many Twitter data sets that are too voluminous to store (for example,



we might be looking at all the Twitter data for a period of time containing a certain hashtag).

For a sufficiently large data set it may not be possible to store even a subset of the columns, so we may need to process the data within each chunk as it is read in. To illustrate, suppose we wish to count the proportion of tweets that contain the string “SIAM”. We artificially set the chunk size to 100, in order to imitate the case of a large data set:

```

reset(ds) % Reset the datastore so reads are from the start.
ds.ReadSize = 100;
nSIAM = 0; nrows = 0;
ds.SelectedVariableNames = {'text'}; % The text of the tweet.

while hasdata(ds)
    tweet_text = read(ds);
    c = table2cell(tweet_text);
    len = length(c);
    nrows = nrows + len;
    for i = 1:len
        nSIAM = nSIAM + ~isempty(strfind(c{i}, 'SIAM'));
    end
end
fprintf('%g rows, %g occurrences of 'SIAM': ', nrows, nSIAM)
fprintf('percentage = %4.1f\n', (nSIAM/nrows)*100)

```

The output is

```
1319 rows, 300 occurrences of 'SIAM': percentage = 22.7
```

Now we carry out a separate computation that reads in the column giving the date and time of tweeting and plot histograms of the day of the week and the hour. The default chunk size reads the table in one go so we have no need for a `while hasdata(ds)` loop. We use the `datetime` function, which converts the date and time information into the MATLAB `datetime` data type, and extract from that the days of the week:

```

reset(ds)
ds.ReadSize = 2000; % Restore default.
ds.SelectedVariableNames = {'timestamp'};
tweet_times = read(ds);
t = datetime(table2cell(tweet_times),...
    'InputFormat', ['yyyy-MM-dd HH:mm:ss Z'], ...
    'TimeZone', 'local');

% Get day of the week and hour of tweets.
tweet_day = day(t, 'dayofweek');
valueset = 1:7;
catnames = {'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'};
tweet_day_names = categorical(tweet_day, valueset, catnames);
tweet_hour = hour(t);

subplot(2,1,1)

```

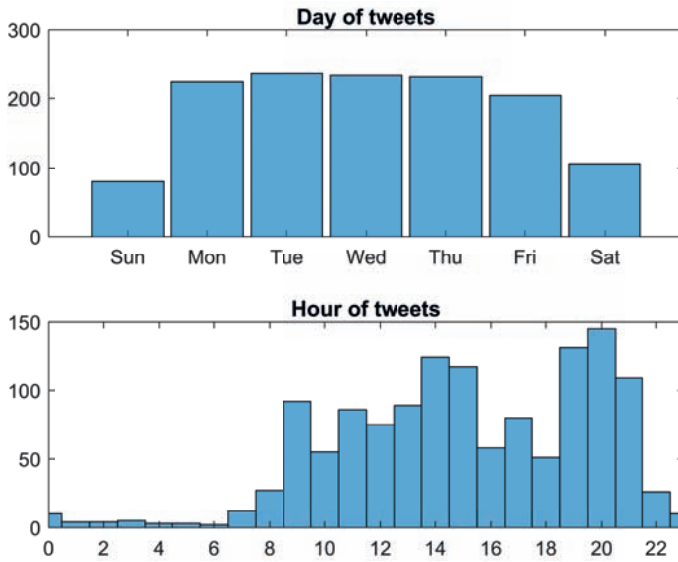


Figure 22.1. *Histograms of days and times of tweets.*

```

histogram(tweet_day_names), title('Day of tweets')
subplot(2,1,2)
histogram(tweet_hour), title('Hour of tweets')
xlim([0 23]), set(gca,'xtick',0:2:23)

```

Here, we formed a categorical array `C` (see Section 18.6) with category names that are abbreviations of the days of the week, and used the fact that `histogram` can plot such arrays. The histograms are shown in Figure 22.1.

## 22.2. MapReduce

An important methodology for working with large data sets is MapReduce, implemented in function `mapreduce`, which uses a datastore to process data in chunks that fit into memory. Each chunk undergoes a map phase, which preprocesses it, then the intermediate data chunks go through a reduce phase, which takes the outputs from the map function, does further computation, and outputs the results. There is great flexibility in how the outputs from the map function can be rearranged and combined before being passed to the reduce function.

The real power of MapReduce comes from the ability to parallelize the map operations, which are independent, and the Parallel Computing Toolbox comes into play here.

For details of `mapreduce` see `doc mapreduce`.

## 22.3. Tall Arrays

Tall arrays provide a way to work with data sets that do not fit into memory and are stored within a datastore. Calculations on tall arrays are delayed until a result is

explicitly requested, with MATLAB optimizing those calculations to try to minimize the number of passes through the data.

Tall arrays are created with the `tall` function, which take as argument an array (numeric, string, datetime, or one of several other data types) or a datastore.

We give an example that uses a text file in which each line contains a string representing the page locator for an index entry in the L<sup>A</sup>T<sub>E</sub>X source for a draft of this book. If a particular term is indexed  $m$  times on page  $n$  then there are  $m$  lines containing  $n$  in the file. The code

```
ds = datastore('index_entries.txt','Type','tabulartext');
ds_tall = tall(ds) % Tall table.
pages = ds_tall.Var1 % Tall numeric vector.
whos pages ds_tall
n = length(pages)
last_page = max(pages);
[n,last_page] = gather(n,last_page)
coverage = length(unique(pages))/last_page; % Prop. pages indexed.
coverage = gather(coverage)
h = histogram(pages,1:last_page);
h.EdgeColor = h.FaceColor; % Otherwise bars look black.
[~,most_indexed_page] = max(h.Values)
```

produces Figure 22.2 and the output

```
ds_tall =
  M×1 tall table
      Var1
  -----
  1.0000e+00
  1.0000e+00
  1.0000e+00
  1.0000e+00
  1.0000e+00
  1.0000e+00
  1.0000e+00
  1.0000e+00
  :
  :
pages =
  M×1 tall double column vector
  1
  1
  1
  1
  1
  1
  1
  1
  1
  :
  :
```

Name	Size	Bytes	Class	Attributes
ds_tall	1×1	310	tall	
pages	1×1	44	tall	

```

n =
    tall double
    ?
Preview deferred. Learn more.
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0 sec
Evaluation completed in 0 sec
n =
    2538
last_page =
    435
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0 sec
Evaluation completed in 0 sec
coverage =
    7.4023e-01
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0 sec
Evaluation completed in 0 sec
most_indexed_page =
    42

```

As the output shows, computations on tall data are deferred until the `gather` function is called with the relevant variables. In general, one should try to minimize the number of calls to `gather`. In this example the data is not very large and the computations could easily be done in memory, but the same code works with a data file too large to fit into memory.

The kinds of operations that should be carried out on tall arrays are ones whose results fit into memory, which can be thought of as reduction operations. Many MATLAB functions support tall arrays (including many in the Statistics and Machine Learning Toolbox); type `methods('tall')` for a list.

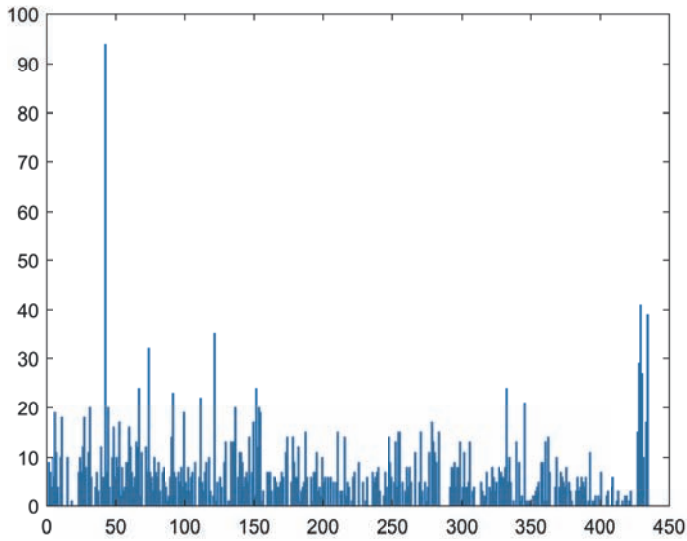


Figure 22.2. *Histogram of pages of index commands.*

*Kirk: "‘Insufficient data’ is not sufficient, Mr. Spock.  
You’re the Science Officer;  
you’re supposed to have sufficient data all the time."  
— Star Trek: The Immunity Syndrome (Stardate 4307.1)*

*A key word here is distillation; a lovely word.  
We must distil the outputs from the data.  
Many of us have sat through exhaustive presentations where  
diligent analysts have turned some handle and  
converted a kilogram of data into a kilogram of powerpoint,  
and they expect our gratitude.  
Decisions have to be both evidence-based and justified.  
Yet they should be ‘smart’ because they are data-driven.  
— PETER GRINDROD, *Mathematical Underpinnings  
of Analytics. Theory and Applications* (2015)*

*The thing which fascinates me is how we find patterns where previously  
others couldn’t find them.  
As the quantity and variety of data has increased,  
and our abilities to work with that raw material have got better,  
we start to see patterns which had previously remained hidden.  
— PETER LAFLIN, *Leeds’ role in the data revolution* (2013)*

*“Data! Data! Data!” he cried impatiently.  
“I can’t make bricks without clay.”  
— ARTHUR CONAN DOYLE, *The Adventure of the Copper Beeches* (1882)*

## Chapter 23

# Optimizing Codes

Most users of MATLAB find that computations are completed fast enough that execution time is not usually a cause for concern. Some computations, though, particularly when the problems are large, require a significant time and it is natural to ask whether anything can be done to speed them up. This chapter describes some techniques that produce better performance from MATLAB codes. They all exploit the fact that MATLAB lies somewhere between an interpreted language and a compiled language and has dynamic memory allocation.

MATLAB does automatic optimization of code via its Just-In-Time (JIT) accelerator, which compiles code at run time. These capabilities are under continuing development and improve with each release. Consequently, we will say very little about automatic optimization and will concentrate instead on useful programming techniques, some of which can probably never be replaced by automatic optimization of code.

The MATLAB profiler is a useful tool when you are optimizing codes, as it can help you decide which parts of the code to focus on. See Section 16.4 for details.

For more details of how to optimize the performance of codes see [2].

### 23.1. Timing Code

In order to understand how to optimize code we need a way of timing how long it takes to execute. MATLAB provides two main ways to do so. First, surrounding a piece of code by `tic` and `toc` causes the time it takes for that code to run to be printed when the `toc` statement is reached. Here, we apply `tic` and `toc` to the `fox_rabbit` function from Listing 12.4:

```
>> tic, fox_rabbit; toc
Elapsed time is 1.202347 seconds.
```

```
>> tic, fox_rabbit; toc
Elapsed time is 0.192736 seconds.
```

We ran the function twice and found that it executed around six times faster the second time than the first! The reason is that the first time a program is run MATLAB has to process and JIT-compile the code, but on subsequent executions it can reuse the information generated. Hence one should always run a code more than once in order to obtain reliable timings. Another complication is that if the code runs quickly (a small fraction of a second) it may be necessary to run it many times and take an average in order to obtain a time that is not unduly influenced by the natural variability of system overheads.

Note that the command `clear all` clears compiled code, among other things, so it is best avoided, especially within a program.

The function `timeit` deals with both the issues just mentioned. It takes a function as argument, runs the function once, then runs the function in a loop enough times that the loop takes about 1 millisecond to execute, and finally takes the median time. This strategy, together with several refinements, is designed to provide more reliable timings than the obvious use of `tic` and `toc`:

```
timeit(@fox_rabbit)
ans =
    7.6380e-02
```

To time a function with arguments, an anonymous function can be constructed:

```
>> A = gallery('ris',500); f = @()eig(A);

>> timeit(f)    % Call eig with one output argument.
ans =
    1.1784e-02

>> timeit(f,2) % Call eig with two output arguments.
ans =
    1.6634e-02
```

Here, we have used the optional second argument of `timeit`, which specifies how many output arguments to request when the first argument is called. Here, we are comparing the time to compute just the eigenvalues with the time to compute eigenvalues and eigenvectors (see Section 9.8.1).

Multiple `tic` and `toc` commands can be intertwined by using the syntax

```
timer1 = tic;
...
elapsed1 = toc(timer1)
```

in which the `toc` statement measures the elapsed time since `timer1` was set up. A second pair `timer2 = tic ... toc(timer2)` can overlap the first.

## 23.2. Vectorization

Since MATLAB is a matrix language, many of the matrix-level operations and functions are carried out internally using compiled C or assembly code and are therefore executed at near-optimum efficiency. This is true of the arithmetic operators `*`, `+`, `-`, `\`, `/` and of relational and logical operators. However, `for` loops may be executed relatively slowly: depending on what is inside the loop, MATLAB may or may not be able to optimize the loop. One of the most important tips for producing efficient code is to avoid `for` loops in favor of vectorized constructs, that is, to convert `for` loops into equivalent vector or matrix operations. Vectorization has important benefits beyond simply increasing speed of execution. It can lead to shorter and more readable MATLAB code. Furthermore, it expresses algorithms in terms of high-level constructs that are more appropriate for high-performance computing.

Consider the following example:

```

>> n = 5e7; x = randn(n,1);
>> tic, s = 0; for i=1:n, s = s + x(i)^2; end, toc
Elapsed time is 0.415446 seconds.

>> tic, s = sum(x.^2); toc
Elapsed time is 0.134303 seconds.

>> tic, s = norm(x)^2; toc
Elapsed time is 0.168398 seconds.

```

In this example we compute the sum of squares of the elements in a random vector in three ways: with a `for` loop, with an elementwise squaring followed by a call to `sum`, and with a single call to `norm`. The vectorized `sum` approach is three times faster than the loop. The `norm` call is slightly slower than `sum`, probably because the `norm` computation takes care to avoid underflow and overflow whenever possible. It is important to emphasize that these timings vary by a few percent from run to run.

The `for` loop in Listing 10.2 on p. 163 can be vectorized, assuming that `f` returns a vector output for a vector argument. The loop and the statement before it can be replaced by

```

x = linspace(0,1,n);
p = x*f(1) + (x-1)*f(0);
max_err = max(abs(f(x)-p));

```

For a slightly more complicated example of vectorization, consider the inner loop of Gaussian elimination applied to an  $n$ -by- $n$  matrix `A`, which can be written

```

for j = k+1:n
    for i = k+1:n;
        A(i,j) = A(i,j) - A(i,k)*A(k,j)/A(k,k);
    end
end

```

Both loops can be avoided, simply by deleting the two `for` and `end` statements:

```

j = k+1:n;
i = k+1:n;
A(i,j) = A(i,j) - A(i,k)*A(k,j)/A(k,k);

```

The approximately  $(n - k)^2$  scalar multiplications and additions have now been expressed as one matrix multiplication and one matrix addition. With  $n = 10,000$  and  $k = 1$  we timed the two-loop code at 2.49 seconds and the vectorized version at 0.77 seconds—again vectorization yields a substantial improvement.

The next example concerns premultiplication of a matrix by a Givens rotation in the  $(j, k)$ -plane, which replaces rows  $j$  and  $k$  by linear combinations of themselves. It might be coded as

```

temp = A(j,:);
A(j,:) = c*A(j,:) - s*A(k,:);
A(k,:) = s*temp + c*A(k,:);

```

By expressing the computation as a single matrix multiplication we can shorten the code and dispense with the temporary variable:



```
A([j k], :) = [c -s; s c] * A([j k], :);
```

The second version is approximately twice as fast for  $n = 10,000$ .

A good principle is to maximize the use of built-in MATLAB functions. Consider, for example, this code to assign to `row_norms` the  $\infty$ -norms of the rows of `A`:

```
for i=1:n
    row_norms(i) = norm(A(i,:), inf);
end
```

It can be replaced by the single statement

```
row_norms = max(abs(A), [], 2);
```

(see p. 68), which is shorter and runs much more quickly. Similarly, the factorial  $n!$  is more quickly computed by `prod(1:n)` than by

```
p = 1; for i = 1:n, p = p*i; end
```

(in fact, the MATLAB function `factorial` uses `prod` in this way).

As a final example, we start with the following code to generate and plot an approximate Brownian (standard Wiener) path [101], which produces Figure 23.1:

```
N = 1e6; dt = 1/N;
w(1) = 0;
for j = 2:N+1
    w(j) = w(j-1) + sqrt(dt)*randn;
end
plot([0:dt:1], w)
```

This computation can be speeded up by preallocating the array `w` (see the next section) and by computing `sqrt(dt)` outside the loop. However, we obtain a more dramatic improvement by vectorizing with the help of the cumulative sum function, `cumsum`:

```
N = 1e6; dt = 1/N;
w = sqrt(dt)*cumsum([0;randn(N,1)]);
plot([0:dt:1], w)
```

This produces Figure 23.1 roughly five times more quickly than the original version.

Vectorization plays an important role in the numerical methods codes of Chapter 12. These codes may require many function evaluations to solve their respective problems, and it can be much more efficient to carry out a certain number of evaluations on vectors than a larger number of scalar function evaluations, not least because of the reduced overheads. The function `integral` requires the integrand to be vectorized, while the stiff ODE solvers and `bvp4c` can take advantage of vectorized function evaluations (which the user specifies via `odeset` and `bvpset`).

### 23.3. Accessing Matrices by Column

MATLAB stores numeric arrays as a vector of elements, the order of the elements being that obtained when the array is accessed by looping over the subscripts from first to last. This is the order obtained with the `(:)` subscripting operation:

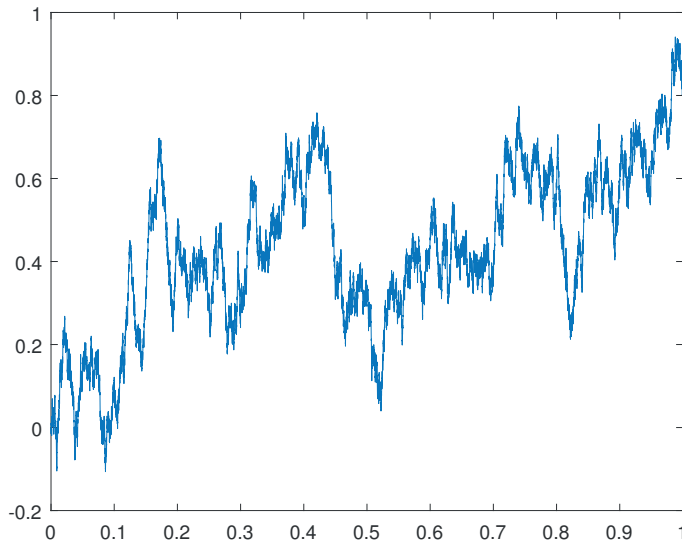


Figure 23.1. *Approximate Brownian path.*

```
>> A = 1:8; A = reshape(A,2,2,2)
A(:,:,1) =
     1     3
     2     4
A(:,:,2) =
     5     7
     6     8

>> A(:) '
ans =
     1     2     3     4     5     6     7     8
```

In particular, matrices are stored by column, that is, in column major order:

```
>> A = magic(2)
A =
     1     3
     4     2

>> A(:) '
ans =
     1     4     3     2
```

This is the reason why functions such as `max`, `sum`, and `sort` default to working on the columns, rather than the rows, when given a matrix argument.

For efficiency, it is important to access the elements of an array in the order in which they are stored, in order to minimize data movement between different levels of the computer's memory hierarchy. For a large matrix not all elements will fit into cache memory, and in an extreme case some elements may need to be paged to disk, so accessing noncontiguous elements may incur significant costs.

The Gaussian elimination example on p. 371 has two nested loops. It correctly has the outer  $j$  loop indexing the columns, so that the inner loop varies the row index and accesses the matrix down the columns. As that example shows, however, code can often be vectorized so that MATLAB itself makes the decision about the order in which to access matrix elements.

### 23.4. Preallocating Arrays

One of the attractions of MATLAB is that arrays need not be declared before first use: assignment to an array element beyond the upper bounds of the array causes MATLAB to extend the dimensions of the array as necessary. If overused, this flexibility can lead to inefficiencies, however. Consider the following implementation of a recurrence:

```
% x has not so far been assigned.
x(1:2) = 1;
for i=3:n, x(i) = 0.25*x(i-1)^2 - x(i-2); end
```

On each iteration of the loop, MATLAB must increase the length of the vector  $x$  by 1. In the next version  $x$  is preallocated as a vector of precisely the length needed, so no resizing operations are required during execution of the loop:

```
% x has not so far been assigned.
x = ones(n,1);
for i=3:n, x(i) = 0.25*x(i-1)^2 - x(i-2); end
```

With  $n = 1e7$ , the first piece of code took 0.81 seconds and the second 0.19 seconds, showing that the first version spends most of its time doing memory allocation rather than floating-point arithmetic.

Preallocation has the added advantage of reducing the fragmentation of memory resulting from dynamic memory allocation and deallocation.

If the order in which a loop body is evaluated does not matter then a trick to avoid preallocation is to run the loop backward, as in

```
for i = n:-1:1, x(i) = atanh(i/(i+1)); end
```

Here,  $x$  is allocated as an  $n$ -vector on the first iteration, just as if we had written  $x = \text{zeros}(1,n)$  before the loop.

You can preallocate an array structure with `repmat(struct(...))` and a cell array with the `cell` function; see Section 18.7.

### 23.5. Miscellaneous Optimizations

Suppose you wish to set up an  $n$ -by- $n$  matrix of twos. The obvious assignment is

```
A = 2*ones(n);
```

The  $n^2$  floating-point multiplications can be avoided by using

```
A = repmat(2,n);
```

The `repmat` approach is much faster for large  $n$ . This use of `repmat` is essentially the same as assigning

```
A = zeros(n); A(:) = 2;
```

in which scalar expansion is used to fill `A`.

There is one optimization that is automatically performed by MATLAB. Arguments that are passed to a function are not copied into the function's workspace *unless* they are altered within the function. Therefore there is no memory penalty for passing large variables to a function provided the function does not alter those variables.

## 23.6. Illustration: Bifurcation Diagram

For a practical example of optimizing MATLAB code we consider a problem from nonlinear dynamics. We wish to examine the long-term behavior of the iteration

$$y_k = F(y_{k-1}), \quad k \geq 2, \quad y_1 \text{ given,}$$

where the function  $F$  is defined by

$$F(y) = y + h \left( y + \frac{1}{2}hy(1-y) \right) \left( 1 - y - \frac{1}{2}hy(1-y) \right).$$

Here,  $h > 0$  is a parameter. (This map corresponds to the midpoint or modified Euler method [148] with stepsize  $h$  applied to the logistic ODE  $dy(t)/dt = y(t)(1-y(t))$  with initial value  $y_1$ .) For a range of  $h$ -values and for a few initial values,  $y_1$ , we would like to run the iteration for a “long time”, say as far as  $k = 500$ , and then plot the next 20 iterates  $\{y_i\}_{i=501}^{520}$ . For each  $h$  on the  $x$ -axis we will superimpose  $\{y_i\}_{i=501}^{520}$  onto the  $y$ -axis to produce a so-called bifurcation diagram.

Choosing values of  $h$  given by `1:0.005:4` and using initial values `0.2:0.5:2.7` we arrive at the script `bif1` in Listing 23.1. This is a straightforward implementation that uses three nested `for` loops and does not preallocate the array `y` before the first time around the inner loop. Figure 23.2 shows the result.

The script `bif2` in Listing 23.2 is an equivalent, but faster, implementation. Two of the loops have been removed and a single `plot` command is used. Here, we stack the iterates corresponding to all  $h$ - and  $y_1$ -values into one long vector, and use elementwise multiplication to perform the iteration simultaneously on the components of this vector. The array `Ydata`, which is used to store the data for the plot, is preallocated to the correct dimensions before use. The vectorized code produces Figure 23.2 about 18 times more quickly than the original version.

An example where a sequence of optimization steps is applied to a MATLAB code in mathematical finance may be found in [66].

## 23.7. External Codes

Instead of speeding up an existing MATLAB code you may prefer to call a routine coded in another language, perhaps from a library. Another reason to call an external routine would be if you cannot find a MATLAB function (built-in or from elsewhere) to carry out the computational task at hand.

Code and libraries written in other languages and frameworks, such as C, C++, Fortran, Java, Python, and .NET, can be called from MATLAB, in particular via the MEX facility that interfaces C, C++, and Fortran codes with MATLAB. For details, see `web([docroot '/matlab/calling-external-functions.html'])`.

Listing 23.1. *Script bif1.*

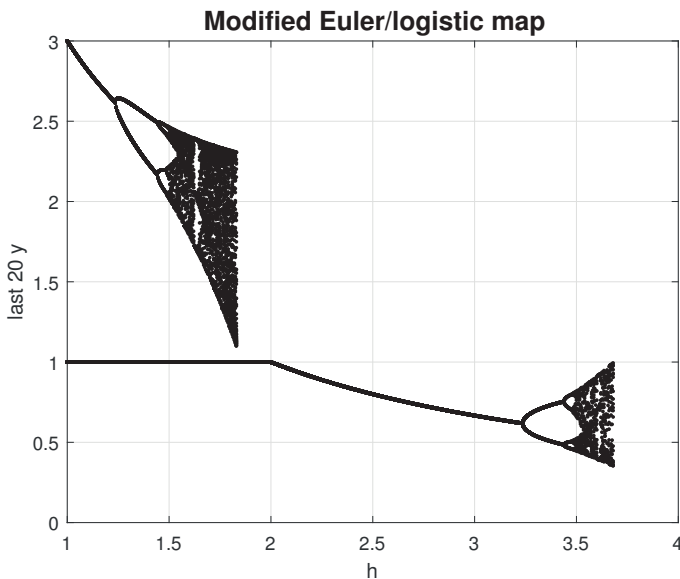
```

%BIF1 Bifurcation diagram for modified Euler/logistic map.
% Computes a numerical bifurcation diagram for a map of the form
%  $y_k = F(y_{k-1})$  arising from the modified Euler method
% applied to a logistic ODE.
%
% Slower version using multiple for loops.

for h = 1:0.005:4
    for iv = 0.2:0.5:2.7
        y(1) = iv;
        for k = 2:520
            y(k) = y(k-1) + h*(y(k-1)+0.5*h*y(k-1)*(1-y(k-1)))*...
                (1-y(k-1)-0.5*h*y(k-1)*(1-y(k-1)));
        end
        plot(h*ones(20,1),y(501:520),'.k'), hold on
    end
end

title('Modified Euler/logistic map','FontSize',14)
xlabel('h'), ylabel('last 20 y')
grid on, hold off

```

Figure 23.2. *Numerical bifurcation diagram.*

Listing 23.2. *Script bif2.*

```

%BIF2 Bifurcation diagram for modified Euler/logistic map.
%   Computes a numerical bifurcation diagram for a map of the form
%    $y_k = F(y_{k-1})$  arising from the modified Euler method
%   applied to a logistic ODE.
%
%   Fast, vectorized version.

h = (1:0.005:4)';
iv = 0.2:0.5:2.7;
hvals = repmat(h,length(iv),1);
Ydata = zeros((length(hvals)),20);
y = kron(iv',ones(size(h)));

for k=2:500
    y = y + hvals.*(y+0.5*hvals.*y.*(1-y)).*(1-y-0.5*hvals.*y.*(1-y));
end
for k=1:20
    y = y + hvals.*(y+0.5*hvals.*y.*(1-y)).*(1-y-0.5*hvals.*y.*(1-y));
    Ydata(:,k) = y;
end

plot(hvals,Ydata,'.k')
title('Modified Euler/Logistic Map','FontSize',14)
xlabel('h'), ylabel('last 20 y'), grid on

```

Third-party MATLAB toolboxes also provide new functionality and potentially faster execution. Of these we mention only the NAG Toolbox for MATLAB (<http://www.nag.co.uk>), which contains over 1,500 functions providing access to the NAG Library—a comprehensive library of codes for numerical computation that is older than MATLAB and, like MATLAB, is constantly evolving.

*Vectorization means using MATLAB language constructs to eliminate program loops, usually resulting in programs that run faster and are more readable.*

— STEVE EDDINS and LOREN SHURE, *MATLAB Digest* (September 2001)

*Entities should not be multiplied unnecessarily.*

— WILLIAM OF OCCAM (c. 1320)

*Life is too short to spend writing for loops.*

— The MathWorks, *Getting Started with MATLAB* (1998)

*In our six lines of MATLAB, not a single loop has appeared explicitly, though at least one loop is implicit in every line.*

— LLOYD N. TREFETHEN and DAVID BAU, III, *Numerical Linear Algebra* (1997)

*Make it right before you make it faster.*

— BRIAN W. KERNIGHAN and P. J. PLAUGER, *The Elements of Programming Style* (1978)

*A useful rule-of-thumb is that the execution time of a MATLAB function is proportional to the number of statements executed, no matter what those statements actually do.*

— CLEVE B. MOLER, *MATLAB News & Notes* (Spring 1996)

*The only recommendation that has withstood the test of time well for all MATLAB releases is to preallocate large data arrays.*

— YAIR ALTMAN, *Accelerating MATLAB Performance* (2015)

# Chapter 24

## Tricks and Tips

Our approach in this book has been to present material of interest to the majority of MATLAB users, omitting topics of more specialized interest. In this chapter we relax this philosophy and describe some tricks and tips that, while of limited use, can be invaluable when they are needed and are of general interest as examples of more advanced MATLAB matters.

### 24.1. Empty Arrays

In Section 5.4 we noted that MATLAB supports empty matrices—ones with one or both dimensions zero—and operations on them. More generally, multidimensional arrays can have zero dimensions:

```
>> double.empty(1,0,3)
ans =
    1×0×3 empty double array
```

We now give some examples of the utility of empty matrices and arrays.

The Schur complement of  $A_{11}$  in  $\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$  is the matrix  $A_{22} - A_{21}A_{11}^{-1}A_{12}$ . Suppose we compute the Schur complement in a context where the dimensions of the blocks can vary and that we encounter an edge case where  $A_{22}$  is the whole matrix. With empty matrices the Schur complement formula evaluates correctly to  $A_{22}$ :

```
>> m = 0; n = 2; rng(1)
>> A11 = rand(m,m); A12 = rand(m,n); A21 = rand(n,m); A22 = rand(n,n)
>> S = A22 - A21*(A11\A12)
A22 =
    4.1702e-01    1.1437e-04
    7.2032e-01    3.0233e-01
S =
    4.1702e-01    1.1437e-04
    7.2032e-01    3.0233e-01
```

Empty arrays can facilitate loop vectorization. Consider the nested loops

```
for i = j-1:-1:1
    s = 0;
    for k=i+1:j-1
        s = s + R(i,k)*R(k,j);
    end
end
```



The inner loop can be vectorized to give

```
for i = j-1:-1:1
    s = R(i,i+1:j-1)*R(i+1:j-1,j);
end
```

What happens when  $i = j-1$  and the index vector  $i+1:j-1$  is empty? Fortunately,  $R(i,i+1:j-1)$  evaluates to a 1-by-0 matrix and  $R(i+1:j-1,j)$  to a 0-by-1 matrix, and  $\mathbf{s}$  is assigned the desired value 0.

## 24.2. Exploiting Infinities

The infinities `inf` and `-inf` can be exploited to good effect.

Suppose you wish to find the maximum value of a function  $\mathbf{f}$  on a grid of points  $\mathbf{x}(1:n)$  and  $\mathbf{f}$  does not vectorize, so that you cannot write `max(f(x))`. Then you need to write a loop, with a variable `fmax` (say) initialized to some value at least as small as any value of  $\mathbf{f}$  that can be encountered. Simply assign `-inf`:

```
fmax = -inf;
for i=1:n
    fmax = max(fmax, f(x(i)));
end
```

Next, suppose that we are given  $p$  with  $1 \leq p \leq \infty$  and wish to evaluate the dual of the vector  $p$ -norm, that is, the  $q$ -norm, where  $p^{-1} + q^{-1} = 1$ . If we solve for  $q$  we obtain

$$q = \frac{1}{1 - 1/p}.$$

This formula clearly evaluates correctly for all  $1 < p < \infty$ . For  $p = \infty$  it yields the correct value 1, since  $1/\infty = 0$ , and for  $p = 1$  it yields  $q = 1/0 = \infty$ . So in MATLAB we can simply write `norm(x, 1/(1-1/p))` without treating the cases  $p = 1$  and  $p = \text{inf}$  specially.

## 24.3. Permutations

Permutations are important when using MATLAB for data processing and for matrix computations. A permutation can be represented as a vector or as a matrix. Consider first the vector form, which is produced by (for example) the `sort` function:

```
>> x = [10 -1 3 9 8 7]
x =
    10    -1     3     9     8     7

>> [s,ix] = sort(x)
s =
    -1     3     7     8     9    10
ix =
     2     3     6     5     4     1
```

The output of `sort` is a sorted vector  $\mathbf{s}$  and a permutation vector  $\mathbf{ix}$  such that  $\mathbf{x}(\mathbf{ix})$  equals  $\mathbf{s}$ . To regenerate  $\mathbf{x}$  from  $\mathbf{s}$  we need the inverse of the permutation  $\mathbf{ix}$ . This can be obtained as follows:

```

>> ix_inv(ix) = 1:length(ix)
ix_inv =
     6     1     2     5     4     3

>> s(ix_inv)
ans =
    10    -1     3     9     8     7

```

In matrix computations it is sometimes necessary to convert between the vector and matrix representations of a permutation. The following example illustrates how this is done, and shows how to permute the rows or columns of a matrix using either form:

```

>> p = [4 1 3 2]
p =
     4     1     3     2

>> I = eye(4);
>> P = I(p,:);
P =
     0     0     0     1
     1     0     0     0
     0     0     1     0
     0     1     0     0

>> A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> P*A
ans =
     4    14    15     1
    16     2     3    13
     9     7     6    12
     5    11    10     8

>> A(p,:)
ans =
     4    14    15     1
    16     2     3    13
     9     7     6    12
     5    11    10     8

>> A*P'
ans =
    13    16     3     2
     8     5    10    11
    12     9     6     7

```

```

    1     4    15    14

>> A(:,p)
ans =
    13    16     3     2
     8     5    10    11
    12     9     6     7
     1     4    15    14

>> p_from_P = (1:4)*P'
p_from_P =
     4     1     3     2

```

A random permutation vector can be generated with the function `randperm`:

```

>> randperm(8)
ans =
     2     4     1     5     8     6     3     7

```

Sometimes we want to swap two variables. More generally, we might wish to permute several variables in a given order. These tasks can be accomplished using the `deal` function:

```

>> a = 1; b = 2; [a,b] = deal(b,a)
a =
     2
b =
     1

>> a = 1; b = 2; c = 3; [a,b,c] = deal(b,c,a)
a =
     2
b =
     3
c =
     1

```

## 24.4. Rank-1 Matrices

A rank-1 matrix has the form  $A = xy^*$ , where  $x$  and  $y$  are both column vectors. Often we need to deal with special rank-1 matrices where  $x$  or  $y$  is the vector of ones. For  $y = \text{ones}(n,1)$  we can form  $A$  as an outer product as follows:

```

>> n = 4; x = (1:n)'; % Example choice of n and x.
>> A = x*ones(1,n)
A =
     1     1     1     1
     2     2     2     2
     3     3     3     3
     4     4     4     4

```

Recall that `x(:,1)` extracts the first column of `x`. Then `x(:, [1 1])` extracts the first column of `x` twice, giving an `n`-by-2 matrix. Extending this idea, we can form `A` using only indexing operations:

```
A = x(:,ones(n,1))
```

(This operation is known to MATLAB aficionados as “Tony’s trick”.) The revised code avoids the multiplication and is therefore faster.

Another way to construct the matrix is with `repmat`:

```
A = repmat(x,1,n);
```

See Section 23.5 for a discussion of how to form an even simpler rank-1 matrix.

For a practical example, consider the Cauchy matrix, which has  $(i, j)$  element  $1/(x_i + y_j)$ . It can be formed in two stages: by forming the matrix  $A$  with  $(i, j)$  element  $x_i + y_j$  and then inverting each element. We will consider just the formation of  $A$ , with `x` and `y` both `n`-by-1 vectors. The most obvious code is

```
A = x*ones(1,n) + ones(n,1)*y.');
```

As we have just seen this can alternatively be written as

```
A = x(:,ones(1,n)) + y(:,ones(1,n)).';
```

Even simpler, and used in `gallery('cauchy',x,y)`, is to exploit implicit expansion (described in Section 5.3.1):

```
A = x + y.');
```

For  $n = 10,000$  we found that the third code runs several times faster than the first two.

## 24.5. Set Operations

Suppose you need to find out whether any element of a vector `x` equals a scalar `a`. This can be done using `any` and an equality test, taking advantage of the way that MATLAB expands a scalar into a vector when necessary in an assignment or comparison:

```
>> x = 1:5; a = 3;
>> x == a
ans =
    1×5 logical array
     0     0     1     0     0

>> any(x == a)
ans =
    logical
     1
```

More generally, `a` might itself be a vector and you need to know how many of the elements of `a` occur within `x`. The test above will not work. One possibility is to loop over the elements of `a`, carrying out the comparison `any(x == a(i))`. Shorter and faster is to use the set function `ismember`:

```

>> x = 1:5; a = [-1 3 5];
>> ismember(a,x)
ans =
    1×3 logical array
     0     1     1

>> ismember(x,a)
ans =
    1×5 logical array
ans =
     0     0     1     0     1

```

As this example shows, `ismember(a,x)` returns a vector with  $i$ th element 1 if  $a(i)$  is in  $x$  and 0 otherwise. The number of elements of  $a$  that occur in  $x$  can be obtained with `sum(ismember(a,x))` or `nnz(ismember(a,x))`, the latter being faster as it involves no floating-point operations.

The `ismember` function works with various types of arrays, not just numeric arrays. To see the full list of MATLAB set functions type

```
web([docroot, '/matlab/set-operations.html'])
```

## 24.6. Subscripting Matrices as Vectors

MATLAB allows a two-dimensional array to be subscripted as though it were one dimensional, as we saw in the example of `find` applied to a matrix on p. 76. If  $A$  is  $m$ -by- $n$  and  $j$  is a scalar then  $A(j)$  means the same as  $a(j)$ , where  $a = A(:)$ ; in other words,  $A(j)$  is the  $j$ th element in the vector made up of the columns of  $A$  stacked one on top of the other.

To see how one-dimensional subscripting can be exploited suppose we wish to assign an  $n$ -vector  $v$  to the leading diagonal of an existing  $n$ -by- $n$  matrix  $A$ . This can be done by

```
A = A - diag(diag(A)) + diag(v);
```

but this code is neither elegant nor efficient. We can take advantage of the fact that the diagonal elements of  $A$  are equally spaced in the vector  $A(:)$  by writing

```
A(1:n+1:n^2) = v;
```

or

```
A(1:n+1:end) = v;
```

The main antidiagonal can be set in a similar way, by

```
A(n:n-1:n^2-n+1) = v;
```

For example,

```

>> A = spiral(5)
A =
    21    22    23    24    25
    20     7     8     9    10

```

```

    19     6     1     2    11
    18     5     4     3    12
    17    16    15    14    13

>> A(1:6:25) = -(1:5)
A =
   -1    22    23    24    25
   20    -2     8     9    10
   19     6    -3     2    11
   18     5     4    -4    12
   17    16    15    14    -5

>> A(5:4:21) = 0      % Using scalar expansion
A =
   -1    22    23    24     0
   20    -2     8     0    10
   19     6     0     2    11
   18     0     4    -4    12
     0    16    15    14    -5

```

One use of this trick is to shift a matrix by a multiple of the identity matrix:  $A \leftarrow A - \alpha I$ , a common operation in numerical analysis. This is accomplished with

```
A(1:n+1:end) = A(1:n+1:end) - alpha
```

It is not always easy to work out the appropriate one-dimensional subscripts with which to index a two-dimensional array. Suppose we wish to set to zero the (1,2), (1,4), (2,2), and (3,1) elements of a 4-by-4 array. Using two-dimensional subscripting this would require four separate assignment statements. Instead we can use the function `sub2ind` to convert the subscripts from two dimensions to one:

```

>> A = magic(4);
>> A(sub2ind(size(A), [1 1 2 3], [2 4 2 1])) = 0
A =
   16     0     3     0
     5     0    10     8
     0     7     6    12
     4    14    15     1

```

The input arguments of `sub2ind` are the array dimensions followed by the row subscripts then the column subscripts.

## 24.7. Avoiding If Statements

Statements involving `if` can sometimes be avoided by the careful use of relational operators. Suppose we wish to code the evaluation of the function

$$f(x) = \begin{cases} \sin x, & x < 0, \\ x, & 0 \leq x \leq 1, \\ 1, & 1 < x, \end{cases}$$

with  $x$  a double-precision matrix. Instead of the obvious `if-elseif-else` coding, we can write

```
y = sin(x).*(x < 0) + x.*(0 <= x & x <= 1) + (1 < x);
```

This evaluation exploits the fact that logical expressions evaluate componentwise to 1 (true) or 0 (false), and that MATLAB will happily perform arithmetic with logicals (here it automatically converts them to doubles first). A possible criticism of the evaluation is that it performs some unnecessary multiplications. The following version is more efficient, though a little less readable. It uses one-dimensional subscripting, as described in Section 24.6:

```
y = ones(size(x));
k = (x < 0);          y(k) = sin(x(k));
k = (0 <= x & x < 1); y(k) = x(k);
```

*A technique is a trick that works.*

— GIAN-CARLO ROTA

*A trick used three times becomes a standard technique.*

— GEORGE POLYA

*The MATLAB language ...  
is optimized for high-speed number crunching....  
Longtime practitioners of the language develop  
tricks and techniques that trade off speed of implementation,  
speed of execution, elegance, and compactness.*

— NED GULLEY, *In Praise of Tweaking: A Wiki-like Programming Contest* (2004)

## Chapter 25

# The Parallel Computing Toolbox

The Parallel Computing Toolbox is a toolbox that, like the Symbolic Math Toolbox, extends the capabilities of MATLAB. Type `ver` to see if it is available on your system.

MATLAB already exploits multicore processors and multiple processors through its built-in functions such as `mtimes` (matrix multiplication), `eig`, backslash, and `sort`, which execute on multiple computational threads in a single MATLAB session. The Parallel Computing Toolbox allows further, explicit exploitation of multicore processors, particularly for coarser-grained computations, and it supports the use of clusters (resources of connected computers) and graphics processing units (GPUs).

The purpose of parallel computing is to harness multiple computing units to obtain results as fast as possible and to solve larger problems than can be stored on a single unit. One measure of the success of a parallel computation is the speedup obtained compared with carrying out the same computation on a single processor.

Parallel computing is a complicated subject, for several reasons. First, computer architectures are continually evolving, so the techniques needed to exploit those architectures must also evolve. Second, parallelism introduces many issues, such as how the data is to be stored and how work can be shared across the computational resources (load balancing). Third, depending on the nature of the computation, it can be difficult to achieve satisfactory speed improvements by parallelization. Fourth, writing and debugging parallel programs is inherently difficult.

The Parallel Computing Toolbox makes it relatively simple to write parallel programs in MATLAB, using just minor extensions to the language. It is necessarily more limited than specialized parallel programming languages in the flexibility it provides.

The Parallel Computing Toolbox can be used to prototype parallel code by working on a local machine before finally running programs on a remote cluster. The cluster on which you run jobs must be running the MATLAB Distributed Computing Server.

In the terminology of the Parallel Computing Toolbox, the *client* is the MATLAB session running on the desktop. *Workers* are MATLAB computational engine processes running on a desktop computer or on a remote cluster, perhaps in the cloud. The workers do not have a MATLAB desktop and cannot display graphics, but they can communicate among themselves and with the client.

A set of workers forms a parallel pool. A parallel pool can be started with the command `parpool`, which by default uses the pool size set in Preferences-Parallel Computing Toolbox. There is a short delay when a parallel pool is set up, since MATLAB has to load on each worker:

```
>> parpool
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
ans =
```



Pool with properties:

```

        Connected: true
        NumWorkers: 4
        Cluster: local
AttachedFiles: {}
        IdleTimeout: 30 minute(s) (30 minutes remaining)
        SpmdEnabled: true

```

As well as the default `local` profile (your desktop machine) you may have a profile that provides access to a remote cluster. The defaults can be overridden: the command `parpool(poolsize)` uses the specified number of workers, and the two-argument form `parpool(profilename,poolsize)` also specifies the profile to be used. A parallel pool closes down after an idle time that can be set in Preferences-Parallel Computing Toolbox (it defaults to 30 minutes in this example). The pool can be explicitly closed as follows:

```

>> delete(gcf)
Parallel pool using the 'local' profile is shutting down.

```

With the default settings in Preferences-Parallel Computing Toolbox it is not essential to issue a `parpool` command, as the parallel language constructs start up a pool with the default pool size automatically if one does not already exist.

An icon in the bottom left corner of the MATLAB desktop has a tooltip that gives information about the current parallel pool and, when clicked, has options to start a parallel pool and to go to the preferences for the toolbox.

A number of MATLAB toolboxes provide support for the Parallel Computing Toolbox in that some of their functions use the toolbox if given certain input arguments. For example, in the Optimization Toolbox there is a setting `'UseParallel',true` in the `options` structure passed to the solvers.

## 25.1. The Parfor Loop

A `parfor` loop provides the simplest way to achieve parallelism with the Parallel Computing Toolbox, as it does not require any knowledge of parallel computing.

A `parfor` loop has the same form as a `for` loop but it splits the computation inside the loop among available workers, in an automatic way. Unlike in a `for` loop the iterations are carried out in an unspecified order, so a `parfor` loop should only be used when the loop iterations are completely independent.

The following code computes the values of the Lambert W function on a spiral in the complex plane, using the function `lambertw` from the Symbolic Math Toolbox. It computes the values twice: first with a `for` loop and again with a `parfor` loop:

```

n = 1000;
x = exp(1i*linspace(0,2*pi,n)) .* linspace(0,1e2,n);
y = ones(n,1); z = ones(n,1);

tic
for i = 1:n
    y(i) = lambertw(x(i));
end

```

```

toc

tic
parfor i = 1:n
    z(i) = lambertw(x(i));
end
toc
p = gcp; pool_size = p.NumWorkers
rel_diff = norm(y-z,1)/norm(y,1)

```

The output is

```

Elapsed time is 10.602350 seconds.
Elapsed time is 3.649462 seconds.
pool_size =
    4
rel_diff =
    0

```

With four workers we are getting a speedup of 2.9, and exactly the same vector of function values is computed. The speedup would be closer to 4 if the computations within the loop were more expensive, so that the relative overheads of the parallelization were smaller.

An ideal scenario for `parfor` is that we need to carry out a set of independent and almost identical computations, such as in a Monte Carlo simulation or a parameter sweep. The next example is of the latter type: it computes the error of the `integral` function applied to a parametrized function with known integral, for a range of values of the parameter:

```

n = 1024;
q = zeros(n,1);
exact = zeros(n,1);
lams = linspace(1,2,n);

parfor i=1:n
    lambda = lams(i);
    f = @(x) 0.1 ./ ( (x-lambda).^2 + 0.01);
    q(i) = integral(f,1,2);
    exact(i) = atan(10*(2-lambda)) - atan(10*(1-lambda));
end

semilogy(lams, abs(q-exact));
xlabel('\lambda','VerticalAlignment','top')
ylabel('Error','Rotation',0)
axis tight

```

The resulting plot is shown in Figure 25.1. The rough curve is typical of adaptive integrators. With four workers the speedup is 2.1 over the same code with `parfor` replaced by `for`.

A `parfor` loop must adhere to a number of restrictions for the loop to be valid.

1. The loop variable must increase in steps of 1. Examples:

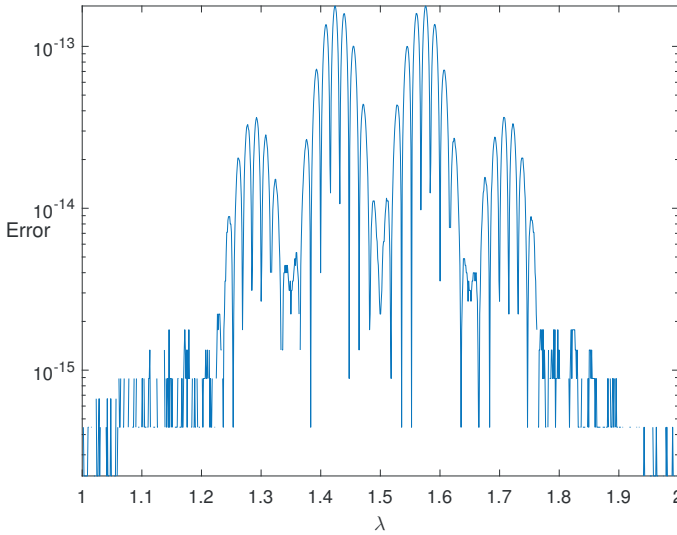


Figure 25.1. Error for integral function, for integral  $\int_1^2 0.1/((x - \lambda)^2 + 0.01) dx$  depending on  $\lambda$ .

```
parfor k = -2:10    % Allowed.
parfor k = 0:10:100 % Not allowed.
```

This limitation can be overcome by putting the desired values of the variable in a vector, `kvals` say, then writing `parfor i = 1:length(k), k = kvals(i) ...`

- There must be no data dependencies between different iterations. Examples:

```
n = 10; s = ones(n,1);
parfor i=2:10, s(i) = i^2; end           % Allowed.
parfor i=2:10, s(i) = s(i-1)^2 + i^2; end % Not Allowed.
```

In the third line the iterations cannot be carried out in an arbitrary order, as is necessary for `parfor` to be applicable.

- The body of a `parfor` loop cannot contain another `parfor` loop, but it can contain a call to a function that contains a `parfor` loop.

The second limitation is quite restricting. However, reduction assignments, which accumulate quantities across loop iterations, are allowed, as in the next example.

We now check the approximation  $\rho(A_n) \approx \sqrt{n}$  [51] to the spectral radius  $\rho(A)$  (defined on p. 321). We compute the average spectral radius over 25 random matrices for four different values of  $n$ , then print a vector of ratios that should be close to 1:

```
nvals = [125 250 500 1000];
nlen = length(nvals);
m = 25;

% Version 1. Serial.
```

```

tic
e = zeros(nlen,1);
for i = 1:nlen
    s = 0;
    for j = 1:m
        s = s + max(abs(eig(randn(nvals(i))))));
    end
    e(i) = s/m;
end
ratio = e'./sqrt(nvals)
toc

% Version 2. Parallel.
tic
e = zeros(nlen,1);
parfor i = 1:nlen
    s = 0;
    for j = 1:m
        s = s + max(abs(eig(randn(nvals(i))))));
    end
    e(i) = s/m;
end
ratio = e'./sqrt(nvals)
toc

```

The second version of the code uses `parfor` to parallelize the computations. The output is

```

ratio =
    1.0504e+00    1.0406e+00    1.0304e+00    1.0226e+00
Elapsed time is 21.192798 seconds.
ratio =
    1.0403e+00    1.0415e+00    1.0319e+00    1.0253e+00
Elapsed time is 20.694409 seconds.

```

The speedup, with four workers, is very modest. However, we can do better. The `parfor` loop is over the values of  $n$ , and so the computations will not be evenly split up amongst the workers: one will receive the  $n = 125$  computations and another the  $n = 1000$  computations. A natural improvement is to switch the order of the loops, so that the our loop is `parfor j = 1:m`. However, we can do even better by collapsing the two loops into one `parfor`, as is done in Listing 25.1. Within the `parfor` we reconstruct the old `i` loop index from the new `k` loop index (the `j` loop index can be reconstructed as well, but it is not needed in this example). The advantage of this transformation is that we now have a loop with a much greater number of iterations, which allows for a more even spread of computation among the workers. We have expressed this version as a function as we will use it again later in the chapter. Running the function with

```

nvals = [125 250 500 1000];
nlen = length(nvals);
m = 25;

```

Listing 25.1. *Function* `specrad_randn`.

```
function ratio = specrad_randn(m,nvals)
%SPECRAD_RANDN Spectral radius of randn matrices.
% ratio = specrad_randn(m,nvals) computes ratios of expected versus
% actual mean spectral radius of m randn matrices of dimensions
% given in nvals.

nlen = length(nvals);
s = zeros(nlen*m,1);
parfor k = 1:nlen*m
    i = ceil(k/m); % And, although not needed, j = k - m*(i-1);
    s(k) = max(abs(eig(randn(nvals(i))))));
end
t = reshape(s,m,nlen);
ratio = sum(t) ./ (m*sqrt(nvals(:)'));
```

```
tic
ratio = specrad_randn(m,nvals)
toc
```

gives

```
ratio =
    1.0407e+00    1.0403e+00    1.0297e+00    1.0240e+00
Elapsed time is 9.990371 seconds.
```

A reduction in time by about a factor 2 has been achieved. By comparison, simply reordering the loops gives a slightly greater run time of 10.76 seconds.

In this example we did not initialize the random number generator, in order to keep the code short. For parallel code it is not sufficient to use a single `rng` statement. Several lines of code are needed: search for `parfor random` in the MATLAB documentation.

Unlike the other parallel constructs, `parfor` is a command in core MATLAB. If the Parallel Computing Toolbox is not present (or automatic creation of a parallel pool is turned off) then `parfor` executes a standard `for` loop but in an unspecified order.

## 25.2. Asynchronous Computing with `Parfeval`

The `parfeval` function allows you to execute a function on one or more parallel pool workers without waiting for all the computations to complete. This is convenient if the computation has a target that may be reached early or if analysis or plotting of intermediate results is required. The `parfeval` function therefore contrasts with `parfor`, which runs until the loop is complete.

The syntax is

```
f = parfeval(fun,numout,in1,in2,...)
```

which requests asynchronous execution of the function `fun` on a worker contained in the current parallel pool, where `fun` returns `numout` output arguments and expects input arguments `in1`, `in2`, `...`. The returned `f` is an `FevalFuture` object, from which the results can be obtained when the worker has completed evaluating `fun`. A call to `fetchNext` of the form

```
[idx,B1,B2,...] = fetchNext(f)
```

waits for an unread `FevalFuture` in the array of futures `f` to finish and then returns the linear index `idx` of that future in `f` along with the future's results in `B1`, `B2`, `...`

Suppose we wish to implement a search for which the number of steps required is not known in advance. Perhaps the most obvious approach is to set up a loop with `N` calls to `parfeval` and then to execute another loop with `N` calls to `fetchNext` that waits for the results, repeating this process as necessary. A more effective way to employ parallelism is illustrated by the function `parfeval_specrad` in Listing 25.2. Here, an initial `N` calls to `parfeval` are issued, then repeatedly `N` further calls are issued and `N` results collected. When an iteration of the `while` loop finishes, the workers are still working while the client generates the next `N` tasks. Therefore there are always between 0 and `2N` tasks in progress at any one time, and usually at least `N`. The advantage of `parfeval_specrad` over the simple loop analogue can be clearly seen by considering what happens as the cost of `make_searches` tends to infinity.

Function `parfeval_specrad` searches for a `randn(n)` matrix whose spectral radius is at least  $1.75\sqrt{n}$ . As soon as such a matrix is found the tasks are terminated. Here is the (truncated) output from one run:

```
>> n = 100; c = 1.2; [A,specrad,index] = parfeval_specrad(n,c);
Iteration: 1, index = 3
Iteration: 1, index = 5
Iteration: 1, index = 6
Iteration: 1, index = 7
Iteration: 1, index = 2
Iteration: 1, index = 4
Iteration: 1, index = 8
Iteration: 1, index = 1
Iteration: 1, index = 9
Iteration: 1, index = 10
...
Iteration: 4, index = 91
Iteration: 4, index = 92
Iteration: 4, index = 93

>> specrad/(c*sqrt(n))
ans =
    1.0138e+00
```

The function `parfevalOnAll` is similar to `parfeval` but it requests the asynchronous execution of the specified function on all the workers in the parallel pool.

### 25.3. Batch Computations

When the `parfor` loops in the previous section are running, the command line is blocked until the computation is complete. The `batch` command allows a computation

Listing 25.2. *Function* `parfeval_specrad`.

```

function [A,specrad,index] = parfeval_specrad(n,c)
%PARFEVAL_SPECRAD Find random matrix with large spectral radius.
% [A, specrad] = PARFEVAL_SPECRAD(n,c) searches for a randn(n)
% matrix with spectral radius at least c*sqrt(n).
% Adapted from a code by Jos Martin.

N = 25; % Number of searches to launch at a time.
specrad_target = sqrt(n)*c;
% Launch initial N searches for the pool to work on.
f = make_searches; k = 1;
while 1
    % Add N more searches to the pool's list of tasks.
    f = [f make_searches];
    % Logical array to track the completed searches.
    complete = false(size(f));

    % Wait for any N of the 2*N searches to complete.
    for j = 1:N
        [index,found,specrad,A] = fetchNext(f);
        fprintf('Iteration: %d, index = %d\n',k,index);
        if found
            % Target met, so cancel the other searches and return.
            cancel(f), return
        end
        % Mark this particular search as complete, for removal below.
        complete(index) = true;
    end

    % Remove completed searches, to minimize length of f.
    f = f(~complete);
    k = k + 1;
end

function f = make_searches
%MAKE_SEARCHES Create block of N searches (nested function).
for i = N:-1:1 % Reverse loop, so no need to allocate f.
    f(i) = parfeval(@search,3,specrad_target,n);
end
end

function [found,specrad,A] = search(specrad_target,n)
%SEARCH Generate random matrix and check spectral radius.
A = randn(n);
specrad = max(abs(eig(A)));
found = (specrad >= specrad_target);
end

```

to be carried out in the background, so that further work can be done in the Command Window. Batch jobs are particularly attractive when the computations are done on a remote cluster. Indeed, in this case the client can even be shut down and the job will continue to run.

Consider the next script. If a parallel pool is open it should be closed before running this script:

```

clust = parcluster('local');
N = 4;
job = batch(clust,@specrad_randn, 1, {25, [125 250 500 1000]}, ...
           'Pool', N-1);    % Submit batch job.
wait(job,'finished')
ratios = fetchOutputs(job); % Retrieve results.
delete(job)

```

The `parcluster` command sets up a cluster, in this case using the local profile. The second input argument of `batch` is the function to run (`specrad_randn` in Listing 25.1), the third is the number of output arguments of that function, and the fourth is a cell array containing the input arguments to be passed to the function. We have to specify the number of workers, which must be at least one less than the total number of workers, since one worker is required to run the batch. When we invoke the script in the Command Window the prompt reappears and we can carry on working while the computations are performed. Our sample script uses the `wait` command to wait until the batch computations are finished and then it uses `fetchOutputs` to retrieve the results, displays them, and closes down the job. Alternatively, the state of a job with handle `job` can be checked periodically using `get(job,'State')`.

The Job Monitor, available from the menu selection Parallel-Monitor Jobs on the Home tab, displays the status of all open jobs for the selected cluster profile.

## 25.4. Single Program, Multiple Data

The single program, multiple data (SPMD) paradigm allows the same code to be run on multiple workers with each worker using different data, which might for example be different parts of the same array.

A general form of the `spmd` command is

```

spmd (n)
    statements
end

```

The code within the `spmd` construct is executed in parallel by `n` workers in the parallel pool, where `n` must not exceed the size of the pool. If `n` is omitted then all the workers are used.

Unlike with `parfor`, where a loop is split up among workers automatically, the `spmd` command allows the computations of individual workers to be explicitly specified, since each worker is identified by a unique `labindex` value.

Consider the script

```

f1 = @(x)cos(x.^2);
f2 = @(x)sin(x.^2);
f3 = @(x)tan(x.^2);

```



```

spmd
    switch labindex
        case 1, f = integral(f1,0,1);
        case 2, f = integral(f2,0,1);
        case 3, f = integral(f3,0,1);
    end
end
f{1:3} % Display results from composite variable.

```

The script evaluates three different integrals in parallel, assuming that at least three workers are available. As with the `parfor` command, if a parallel pool is not already running then `spmd` will set one up. We defined the functions outside the body of the `spmd` statement, as it is a limitation of the Parallel Computing Toolbox that anonymous functions cannot be defined within an `spmd` body. We retrieve the results back from the workers using the composite object `f`, which is accessed like a cell array, with one element per worker. The output is

```

ans =
    9.0452e-01
ans =
    3.1027e-01
ans =
    3.9841e-01

```

A composite object, with one element per worker, can be created with an assignment of the form

```
X = Composite()
```

When the elements of a composite object are set on the client they are immediately transferred to the appropriate workers. The next script computes the largest and smallest singular values of three matrices:

```

parpool(3);
X = Composite();
X{1} = hilb(100);
X{2} = pascal(100);
X{3} = magic(100);
spmd(3)
    s = svd(X);
    svals_extreme = [s(1) s(end)];
end
svals_extreme{1:3}

```

and the output is

```

ans =
    2.1827e+00    6.5355e-20
ans =
    3.0318e+58    5.2848e-02
ans =
    5.0005e+05    1.4304e-13

```

The differences between `parfor` and `spmd` are important to note.

- `parfor` divides a loop into smaller pieces automatically and requires no user input.
- `spmd` divides data into small pieces, but this must be explicitly programmed by the user. `spmd` allows dependencies between the tasks of different workers and communication between the workers (neither is illustrated in the simple examples above).

## 25.5. Distributed and Codistributed Arrays

A distributed array is created on the client but stored on the workers, partitioned among them. A codistributed array is created on the workers and partitioned among them. A distributed two-dimensional array is partitioned as evenly as possible by column. By contrast, the user has full control over how a codistributed array is partitioned. The advantage of a (co)distributed array is that different workers can work on different parts of the array in parallel.

In the following example we transform a random matrix by a matrix multiplication and some componentwise operations in three different ways, on a parallel pool with six workers. For this computation we started MATLAB with the command line option `-singleCompThread`, which limits MATLAB to a single computational thread and so gives a more dramatic illustration of the benefits of parallelism:

```
n = 5000;

% 1. Serial code.
tic
A = randn(n);
X = log( (A*A+eye(n)).^(1/2) );
toc

% 2. With distributed array.
tic
A = randn(n,'distributed');
X = log( (A*A+eye(n,'like',A)).^(1/2) );
toc

% 3. With codistributed array.
tic
spmd
    A = randn(n,'codistributed');
    X = log( (A*A+eye(n)).^(1/2) );
end
toc
```

The `'like',A` arguments in the `eye` statement ensure that the identity matrix is generated on the workers and not on the host; see the next section for more about `'like'`. The output is

```
Elapsed time is 25.328439 seconds.
Elapsed time is 6.505735 seconds.
Elapsed time is 5.840922 seconds.
```

In both the second and third computations the array `A` is created on the workers, and it is a little faster to generate it within the `spmd` statement.

If we delete the semicolon at the end of the first line inside the body of the `spmd` then we obtain insight into how the data is stored:

```
Lab 1:
  This worker stores A(:,1:834).
    LocalPart: [5000×834 double]
    Codistributor: [1×1 codistributor1d]
Lab 2:
  This worker stores A(:,835:1668).
    LocalPart: [5000×834 double]
    Codistributor: [1×1 codistributor1d]
Lab 3:
  This worker stores A(:,1669:2501).
    LocalPart: [5000×833 double]
    Codistributor: [1×1 codistributor1d]
Lab 4:
  This worker stores A(:,2502:3334).
    LocalPart: [5000×833 double]
    Codistributor: [1×1 codistributor1d]
Lab 5:
  This worker stores A(:,3335:4167).
    LocalPart: [5000×833 double]
    Codistributor: [1×1 codistributor1d]
Lab 6:
  This worker stores A(:,4168:5000).
    LocalPart: [5000×833 double]
    Codistributor: [1×1 codistributor1d]
```

We see that `A` has been split into blocks of about  $834 \approx 5000/6$  columns among the six workers.

## 25.6. GPU Computing

The Parallel Computing Toolbox supports GPU (graphics processing unit) computing on CUDA-enabled NVIDIA GPUs. Computations can potentially run faster on a GPU than on a CPU (central processing unit).

To determine if your machine has a suitable GPU, type `gpuDevice`. If a suitable GPU is found a handle is returned containing a list of properties of the GPU; otherwise an error message is produced:

```
>> gpuDevice

ans =
  CUDADevice with properties:
```

```

        Name: 'GeForce GTX 960M'
        Index: 1
    ComputeCapability: '5.0'
        SupportsDouble: 1
        DriverVersion: 7.5000
        ToolkitVersion: 7
    MaxThreadsPerBlock: 1024
        MaxShmemPerBlock: 49152
    MaxThreadBlockSize: [1024 1024 64]
        MaxGridSize: [2.1475e+09 65535 65535]
        SIMDWidth: 32
        TotalMemory: 2.1474e+09
        AvailableMemory: 1.2342e+09
    MultiprocessorCount: 5
        ClockRateKHz: 1097500
        ComputeMode: 'Default'
    GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
        CanMapHostMemory: 1
        DeviceSupported: 1
        DeviceSelected: 1

```

The function `gpuDeviceCount` returns the number of suitable GPUs present, and so can be used to test for the presence of a GPU without generating an error.

In order to carry out computations on the GPU, data must first be stored on it. This can be done in two ways: by transferring a numeric array from the CPU or, more efficiently, by generating the array directly on the GPU. An array is transferred to the GPU with the `gpuArray` function, which constructs an object of the `gpuArray` class. Functions such as `eye`, `ones`, `rand`, and `randn` can have an argument `'gpuArray'` appended in order to generate the array directly on the GPU. An array is transferred back to the CPU with the `gather` function:

```

>> A = gallery('randcorr',100);      % Generate on the CPU.
>> E = gpuArray(A);                 % Transfer to the GPU.
>> F = eye(100,'gpuArray');         % Generate on the GPU.
>> G = ones(100,'int64','gpuArray'); % Generate on the GPU.
>> B = gather(E); C = gather(F);
>> whos

```

Name	Size	Bytes	Class	Attributes
A	100×100	80000	double	
B	100×100	80000	double	
C	100×100	80000	double	
E	100×100	108	gpuArray	
F	100×100	108	gpuArray	
G	100×100	108	gpuArray	
ans	1×1	112	parallel.gpu.CUDADevice	

You can test whether an array `A` exists on the GPU with `existsOnGPU(A)`. This may be necessary because if the GPU is reset and its memory cleared, with `reset(h)`

where **h** is its handle, any variables on it will still show as existing in the MATLAB workspace until they are cleared.

A list of methods that operate on the `gpuArray` class can be obtained with the command `methods('gpuArray')`. It includes elementary functions (`exp`, `cos`, `abs`, `acosh`, ...), matrix functions (`lu`, `eig`, `svd`, `mldivide`, `gmres`, `expm`, ...), and numerical functions (`fft`, `interp1`, `polyfit`, ...). Help on the GPU versions of these functions is obtained with, for example, `help gpuArray/svd`.

Random numbers generated on the GPU are different from those generated on the CPU. For details see `doc control random number streams`.

Unlike on the CPU, complex arithmetic is not automatically used as necessary on the GPU. If complex arithmetic might be needed, the input must be explicitly set to be complex:

```
>> sqrt(gpuArray(-1))
Error using gpuArray/sqrt
SQRT: needs to return a complex result, but this is not supported
for real input X on the GPU. Use SQRT(COMPLEX(X)) instead.

>> acosh(gpuArray(-4))
Error using gpuArray/acosh
ACOSH: needs to return a complex result, but this is not supported
for real input X on the GPU. Use ACOSH(COMPLEX(X)) instead.

>> sqrt(gpuArray(complex(-1)))
ans =
    0.0000 + 1.0000i

>> acosh(gpuArray(complex(-4)))
ans =
    2.0634 + 3.1416i
```

Once data is on the GPU we can apply appropriate built-in MATLAB functions (from the list of methods above).

The functions `zeros`, `ones`, `eye`, `rand`, and `randn`, and a number of others, have a `'like'` option, illustrated by

```
X = randn(size(B), 'like', B);
```

This statement returns an array of normally distributed random numbers of the same size, type, and storage location as the existing array `B`. This usage is convenient when you are writing code that you may wish to run on the CPU or the GPU; after initializing an array on the CPU or GPU, subsequent calls to `randn`, etc., can reference its type so that arrays are created on the CPU or GPU as appropriate.

GPU computations can be timed with the `gputimeit` function, which is analogous to the `timeit` function discussed in Section 23.1.

Two important points need to be made concerning the efficiency of GPU computations. First, it is important to write vectorized code whenever possible. Second, GPUs are typically much more efficient at single-precision arithmetic than double-precision arithmetic, so single precision should be used as long as it is accurate enough for the task at hand.

## 25.7. On Things Not Treated

A function `pmode` starts up a Parallel Command Window that displays a `pmode` prompt (`P>>`). It allows interactive computations in SPMD form and displays output from each worker in a separate subwindow.

This chapter has only scratched the surface of the Parallel Computing Toolbox. It has given only trivial examples of its use, because serious examples would be much longer and harder to read. For one source of more realistic examples type

```
dir([matlabroot, '/toolbox/distcomp/examples/benchmark/hpcchallenge'])
```

You will see a number of functions that implement benchmarks from the HPC Challenge benchmark suite—in particular the HPL benchmark for solving a linear system  $Ax = b$ .

For further details of the Parallel Computing Toolbox consult the documentation as well as the video tutorials on the website of The MathWorks.

*Programmability for us will always trump performance.*

— GAURAV SHARMA and JOS MARTIN,  
*MATLAB<sup>®</sup>: A Language for Parallel Computing* (2009)

*Choosing which form of parallelism to use can be complicated.*

— CLEVE B. MOLER,  
*Parallel MATLAB: Multiple Processors and Multiple Cores* (2007)

# Chapter 26

## Case Studies

### 26.1. Introduction

To supplement the short bursts of MATLAB code that appear throughout the book, we now give some larger, more realistic examples. Their purpose is to demonstrate MATLAB in use on nontrivial problems and to illustrate good programming practice. We focus on problems that are

1. easy to explain in words,
2. easy to set up mathematically,
3. suited to graphical display, and
4. solvable with around one page or less of MATLAB code.

For each case study, after summarizing the problem we list the relevant code and give a walk-through that points out notable techniques and MATLAB functions. The walk-throughs are not intended to explain the codes line by line. For further details on MATLAB functions used in this chapter consult the index and the MATLAB documentation.

### 26.2. Brachistochrone

Suppose a particle slides down a frictionless wire. If we fix the endpoints, what shape of wire minimizes the travel time? This problem dates back to the times of Johann Bernoulli (1667–1748) and its solution involves a curve known as the *brachistochrone*. The name comes from the Greek “brachistos” (shortest) and “chronos” (time).

For this problem it is traditional to use “upside-down” coordinates so that the  $x$ -axis points in a horizontal direction but the  $y$ -axis points vertically downward. Suppose the wire starts at the origin,  $(0, 0)$ , and ends at  $(b_x, b_y)$ . If we let  $y(x)$  denote the curve followed by the wire, then the particle’s sliding time takes the form

$$T = \int_0^{b_x} \sqrt{\frac{1 + (dy/dx)^2}{2gy(x)}} dx,$$

where  $g$  is the constant of acceleration due to gravity. Minimizing  $T$  over an appropriate class of functions  $y(x)$  produces the brachistochrone, which may be defined in terms of the parameter  $\theta$  and a constant  $R$  by

$$x(\theta) = R(\theta - \sin \theta), \quad y(\theta) = R(1 - \cos \theta).$$

The curve must finish at  $(b_x, b_y)$  so we require  $0 \leq \theta \leq \theta^*$ , where  $b_x = R(\theta^* - \sin \theta^*)$  and  $b_y = R(1 - \cos \theta^*)$ . Eliminating  $R$ , we find  $\theta^*$  by solving

$$b_y \theta^* - b_y \sin \theta^* + b_x \cos \theta^* - b_x = 0, \quad (26.1)$$

and then we set  $R = b_y / (1 - \cos \theta^*)$ .

Now we consider a wire formed from straight-line segments. More precisely, divide the  $x$  interval  $[0, b_x]$  into  $N$  equally spaced subintervals  $[x_{k-1}, x_k]$  with  $x_k = k\Delta x$  and  $\Delta x = b_x/N$ , and let  $y_k$  denote  $y(x_k)$ . Imagine that the wire is produced by joining the  $y_k$  heights with straight lines. Since  $y_0 = 0$  and  $y_N = b_y$ , our join-the-dots curve is completely determined by specifying the heights  $\{y_k\}_{k=1}^{N-1}$  at the internal points. For any such curve it may be shown that the particle's slide time is given by

$$T = \sum_{k=1}^N \frac{2\sqrt{\Delta x^2 + (y_k - y_{k-1})^2}}{\sqrt{2gy_k} + \sqrt{2gy_{k-1}}}. \quad (26.2)$$

For fixed  $b_x$ ,  $b_y$ , and  $N$ , our task is to find the straight-line segments that minimize the slide time. This is an optimization problem; we wish to minimize  $T = T(y_1, y_2, \dots, y_{N-1})$  in (26.2), with  $y_0 = 0$  and  $y_N = b_y$ , over all possible  $\{y_k\}_{k=1}^{N-1}$ . It is then of interest to see how well the resulting curve approximates the brachistochrone. More details about the brachistochrone problem may be found, for example, at <http://mathworld.wolfram.com/BrachistochroneProblem.html>. The idea of optimizing over piecewise linear wires is taken from [116].

### Code and Walk-through

The function `brach` in Listing 26.1 compares a number of exact and join-the-dots brachistochrones, as shown in Figure 26.1. The nested function `Btime` computes the piecewise linear slide time  $T$  in (26.2), and the anonymous function `tzero` returns the left-hand side of (26.1). We set  $b_x = 1$  and use ten  $b_y$ -values equally spaced between 0.2 and 2 and two  $N$ -values, 4 and 8. In each case the initial guess `yinit` that we pass to `fminsearch` represents the straight line from  $(0, 0)$  to  $(b_x, b_y)$ . After using `fminsearch` to find the optimal curve, we plot `-[0 y by]`, rather than `[0 y by]`, to account for the upside-down coordinate system. The true brachistochrone is computed via `fzero`. The  $N = 4$  and  $N = 8$  cases are plotted in the `subplot(1,2,1)` and `subplot(1,2,2)` regions, respectively.

## 26.3. Small-World Networks

As discussed in Chapter 21, an undirected network, or graph, is defined by a list of nodes and a list of edges connecting pairs of nodes. A network of  $N$  nodes may be stored in a symmetric  $N$ -by- $N$  adjacency matrix  $A$ , with  $a_{ij} = 1$  if nodes  $i$  and  $j$  have an edge between them, and  $a_{ij} = 0$  otherwise. In the case where  $A$  represents a social network, for example, the nodes are people and the edges represent acquaintanceships:  $a_{ij} = a_{ji} = 1$  if persons  $i$  and  $j$  know each other. The pathlength between nodes  $i$  and  $j$  is the minimum number of edges that must be crossed in order to get from  $i$  to  $j$ .

A sparse network may be said to have small-world characteristics if

- (a) it is highly clustered—if  $i$  knows  $j$  and  $j$  knows  $k$ , then, with high frequency,  $i$  knows  $k$ —and



Listing 26.1. *Function brach.*

```

function brach
%BRACH   Brachistochrone illustration.
%   Computes and plots approximate brachistochrone by optimization,
%   using fminsearch, and exact brachistochrone, using fzero.

bx = 1; g = 9.81;
byvals = linspace(0.2,2,10);
Nvals = [4 8];
for i = 1:2
    N = Nvals(i);
    subplot(1,2,i)
    for k = 1:length(byvals)

        % Approximate brachistochrone.
        by = byvals(k);
        dy = by/N; dx = bx/N;
        yinit = [dy:dy:by-dy];
        y = fminsearch(@Btime,yinit);

        plot([0:dx:bx],[-[0 y by],'ro-')
        hold on

        % True brachistochrone.
        tzero = @(theta)(by*theta - by*sin(theta) + bx*cos(theta) - bx);
        tstar = fzero(tzero,pi);
        R = by/(1-cos(tstar));
        thetaval = linspace(0,tstar,100);
        xcoord = R*(thetaval-sin(thetaval));
        ycoord = R*(1-cos(thetaval));
        plot(xcoord,-ycoord,'b--','Linewidth',2)

    end
    title(sprintf('N = %1.0f',N),'FontWeight','normal','FontSize',12)
    xlim([0,bx]), axis off
end
hold off

function T = Btime(y)
%BTIME   Travel time for a particle.
%   Piecewise linear path with equispaced y between (0,0) and (bx,by).

yvals = [0 y by];           % End points do not vary.
N = length(y)+1; d = bx/N;
T = sum(2*sqrt( d^2 + (diff(yvals)).^2 )./( sqrt(2*g*yvals(2:end)) + ...
        sqrt(2*g*yvals(1:end-1) )));
end

end

```

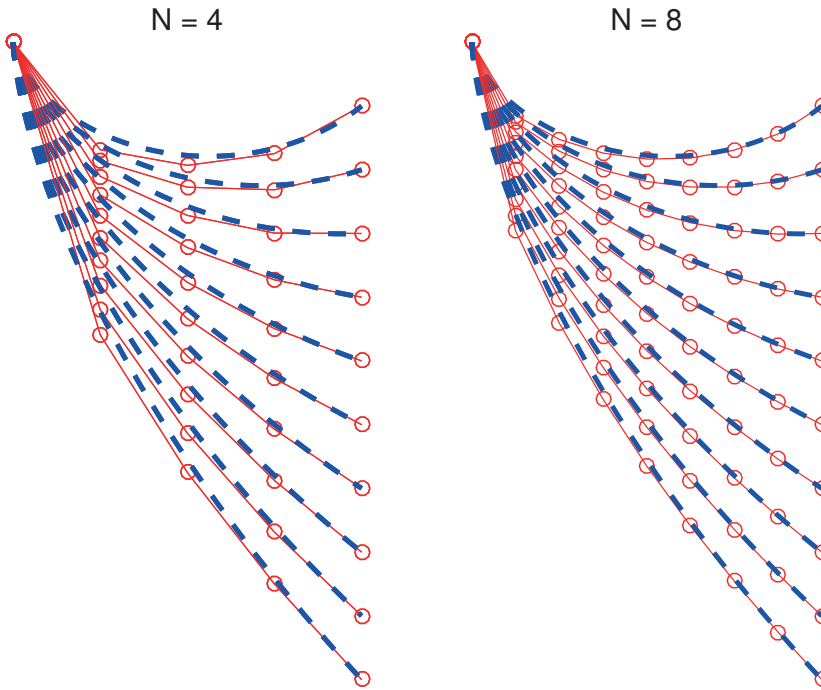


Figure 26.1. *Output from brach.*

(b) it has a small average pathlength.

Watts and Strogatz [180] coined the phrase small-world network and found several real-life examples. They also showed that randomly rewiring a regular lattice is a mechanism for creating a small world. Here, we will focus on property (b), the average pathlength, for a variation of the Watts–Strogatz model in the spirit of [135] that uses shortcuts rather than rewiring.

We begin with a  $k$ -nearest-neighbor ring network. Arranging the  $N$  nodes like the hours on a clock, we set  $a_{ij} = 1$  if  $j$  can be reached by moving at most  $k$  steps away from  $i$ , either clockwise or counterclockwise. In the case  $N = 7$ ,  $k = 2$ , the adjacency matrix is

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}.$$

In general,  $A$  could be defined by the MATLAB commands

```
r = zeros(1,N); r(2:k+1) = 1; r(N-k+1:N) = 1;
A = toeplitz(r);
```

Now we superimpose random shortcuts on the network; that is, we add nonzeros to the matrix at random locations, according to the following process. We look at  $N$

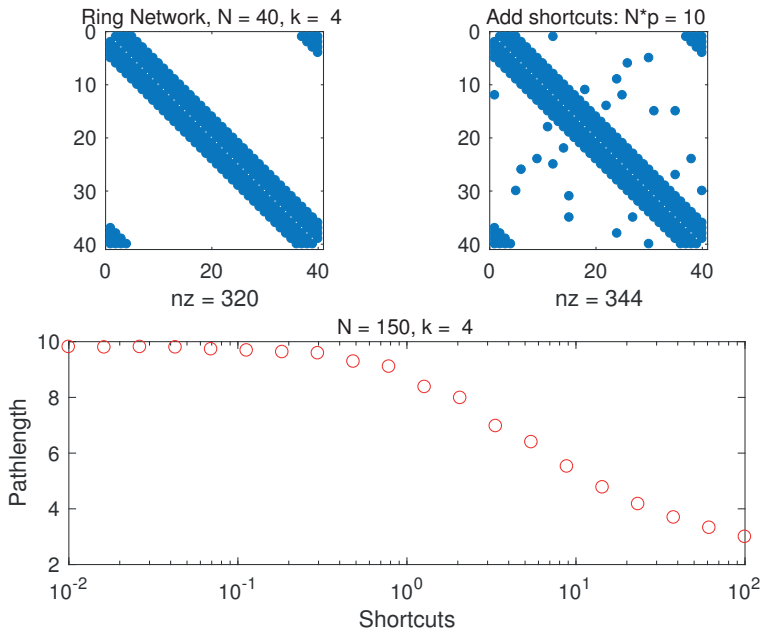


Figure 26.2. Output from the small-world simulations of `small_world`. Upper: adjacency matrices. Lower: pathlength decay.

flips of a biased coin that lands heads with probability  $p$ . If the  $i$ th flip shows heads then we choose a column  $1 \leq j \leq N$  uniformly and set  $a_{ij} = a_{ji} = 1$ . (In other words, we add a new link—a shortcut—from the  $i$ th node to a randomly chosen  $j$ th node.) On average, the overall number of shortcuts that we create is  $Np$ .

Generating shortcuts in this way makes it easier to get around the network, and hence decreases the average pathlength. Our aim is to investigate, for fixed  $N$  and  $k$ , how sharply the average pathlength decays as the average number of shortcuts is increased.

For computational purposes, we may use the characterization that, for  $r > 1$ , the pathlength between nodes  $i$  and  $j$  is  $r$  if and only if  $(A^r)_{ij} > 0$  and  $(A^k)_{ij} = 0$  for all  $0 < k \leq r - 1$ . Hence, we can find all pathlengths by raising the adjacency matrix to increasingly higher powers, until no zeros remain. For each pair of nodes,  $i$  and  $j$ , we must record the power at which the  $(i, j)$  element first becomes nonzero.

### Code and Walk-through

The script `small_world` in Listing 26.2 produces Figure 26.2. The upper left `spy` plot shows the adjacency matrix for a four-nearest-neighbor ring of 40 nodes. In the upper right picture, we see an instance of the same network with shortcuts added, using  $Np = 10$ . The lower picture gives the results of a large-scale computation. Here, we show the average pathlength of a 150-node, four-nearest-neighbor ring as a function of the average number of shortcuts,  $Np$ .

The first part of the code sets up an adjacency matrix for the ring using `toeplitz`. We then generate shortcuts with the `sparse` facility. The line `v = find(rand(N,1)<p);` simulates the coin flips. Then

Listing 26.2. *Script small\_world.*

```

%SMALL_WORLD    Small-world network example.
%   Display ring and small-world adjacency matrices.
%   Then compute average pathlengths.

rng(100), options = {'FontSize',10,'FontWeight','normal'};

N = 40; k = 4; short_ave = 10; p = short_ave/N;
r = zeros(1,N); r(2:k+1) = 1; r(N-k+1:N) = 1;
A = toeplitz(r);
subplot(2,2,1), spy(A)
title(sprintf('Ring Network, N = %2.0f, k = %2.0f',N, k),options{:})

subplot(2,2,2)
v = find(rand(N,1)<p);
Ashort = sparse(v,ceil(N*rand(size(v))),ones(size(v)),N,N);
spy(A+Ashort+Ashort')
title(sprintf('Add shortcuts: N*p = %2.0f',N*p),options{:})
h = waitbar(0,'Computing average pathlengths');

%%%%%% Average pathlength as a function of N*p %%%%%
N = 150; k = 4; M = 20; Smax = 150; Np = logspace(-2,2,M);
r = zeros(1,N); r(2:k+1) = 1; r(N-k+1:N) = 1;
B = toeplitz(r);
lmean = zeros(M,1);
for i = 1:M
    waitbar(i/M)
    p = Np(i)/N;
    smean = zeros(Smax,1);
    for s = 1:Smax
        v = find(rand(N,1)<p);
        Bshort = sparse(v,ceil(N*rand(size(v))),ones(size(v)),N,N);
        Bnetwork = B + Bshort + Bshort' + eye(N); % Full array.
        L = sign(Bnetwork); % Convert to matrix of 0s and 1s.
        power = 1;
        Bnew = Bnetwork;
        while any(any(Bnew==0))
            power = power + 1;
            Bold = Bnew;
            Bnew = Bnew*Bnetwork;
            L = L + ( (L == 0) & (Bnew > 0) )*power;
        end
        smean(s) = mean(mean(L-diag(diag(L))))*N/(N-1);
    end
    lmean(i) = mean(smean);
end
close(h)

subplot(2,2,3:4), semilogx(Np,lmean,'ro')
xlabel('Shortcuts'), ylabel('Pathlength')
title(sprintf('N = %2.0f, k = %2.0f',N, k),options{:})

```

```
Ashort = sparse(v,ceil(N*rand(size(v))),ones(size(v)),N,N);
```

finds a column index for each successful row index and inserts the appropriate edges. Because an edge from  $i$  to  $j$  automatically implies an edge from  $j$  to  $i$ , we apply `spy` to `A+Ashort+Ashort'`. The shortcut matrix in this case is

```
>> Ashort
Ashort =
    (12,1)      1
    (13,9)      1
    (18,11)     1
    (25,12)     1
    (35,15)     1
    (14,22)     1
    (9,24)      1
    (38,24)     1
    (6,26)      1
    (5,30)      1
    (15,31)     1
    (27,35)     1
    (33,36)     1
    (30,40)     1
```

We see that 14 shortcuts have been added. Two of these, at (13,9) and (33,36), will have no effect on the pathlengths, as they repeat existing edges.

The second part of the script performs the big simulation. The outer loop, `for i = 1:M`, runs over the  $Np$ -values, and the inner loop, `for s = 1:Smax`, drives a Monte Carlo simulation—for each  $Np$ -value, we approximate the average pathlength over all networks by the computed average over `Smax` networks. Although, for convenience, the shortcuts are created with the `sparse` function, we compute with a full matrix `Bnew` because we know that it will eventually fill in completely.

We include the identity matrix in the assignment

```
Bnetwork = B + Bshort + Bshort' + eye(N); % Full array.
```

to make the diagonal nonzero. This allows `any(any(Bnew==0))` to be used as the termination criterion for the `while` loop. The line `L = sign(Bnetwork);` ensures that any multiply assigned edges are only counted once. The `while` loop powers up the adjacency matrix until it is full of nonzeros. If the  $(i,j)$  element first becomes nonzero at level `power`, then we enter this value in `L(i,j)`. On leaving the `while` loop, we compute the average over the off-diagonal entries of `L` using `mean(mean(L-diag(diag(L))))*N/(N-1)`. The resulting plot shows that the pathlength starts to drop significantly when an average of  $O(1)$  shortcuts are added to the  $O(N)$  network. Upping the network size to `N = 1000`, which of course increases the runtime, produces results that agree qualitatively with the related computations in [180].

## 26.4. Performance Profiles

A common task in scientific computing is to compare several competing methods on a set of test problems. Assuming a scalar measure of performance has been chosen

(typically speed or accuracy), how best to present the results from the tests is a nontrivial question. Some natural approaches have drawbacks. Plotting the average performance of the methods tends to make difficult problems dominate the results, and it is unclear how to handle problems that a method failed to solve. Ranking the solvers, by plotting the number of times a solver came in  $k$ th place, for  $k$  from 1 to the number of solvers, provides no information on the size of the improvement between one place and the next.

A way of presenting results called a *performance profile* overcomes these disadvantages. This technique, introduced by Dolan and Moré [35], is not to be confused with an older technique of the same name that has been applied mainly in the context of numerical integration [117].

Suppose we have a set  $P$  of  $m$  test problems and a set  $S$  of  $n$  solvers (we use the term “solver” instead of “method” to emphasize that we are considering a particular implementation in software of a method). Let  $t_s(p)$  measure the performance of solver  $s \in S$  on problem  $p \in P$ , where the smaller the value of  $t_s(p)$  the better the performance. Typically,  $t_s(p)$  is the runtime, the flop count, the reciprocal of the flop rate, or a measure of accuracy or stability. Define the performance ratio

$$r_{p,s} := \frac{t_s(p)}{\min\{t_\sigma(p) : \sigma \in S\}} \geq 1,$$

which is the performance of solver  $s$  on problem  $p$  divided by the best performance of all the solvers on this problem. The performance profile of solver  $s$  is the function

$$\phi_s(\theta) = \frac{1}{m} \times \text{number of } p \in P \text{ such that } r_{p,s} \leq \theta,$$

which is monotonically increasing. In words,  $\phi_s(\theta)$  is the probability that the performance of solver  $s$  is within a factor  $\theta$  of the best performance over all solvers on the given set of test problems. Technically,  $\phi_s(\theta)$  is the (cumulative) distribution function for the performance ratio of solver  $s$ .

The formulas above reduce to simple array arithmetic. Let the performance data be an  $m$ -by- $n$  array  $A$ , where  $a_{ij}$  is the performance of solver  $j$  on problem  $i$ . Then

$$\phi_j(\theta) = \frac{1}{m} \times \text{number of } i \text{ among } 1:m \text{ such that } a_{ij} \leq \theta \min\{a_{ik} : k = 1:n\}. \quad (26.3)$$

To view the performance profiles we simply plot  $\phi_j(\theta)$  against  $\theta$  for all solvers  $j$ .

## Code and Walk-through

Function `perfprof` in Listing 26.3 computes and plots performance profiles. This function could be written in several ways. The shortest approach would be to make use of the MATLAB function `stairs`, but it is more instructive to code the necessary computations directly, as we have done here.

Note first that  $\phi_j(\theta)$  is a piecewise constant function whose possible values are 0,  $1/m$ ,  $2/m$ ,  $\dots$ , 1, and whose value changes when  $\theta = a_{ij} / \min\{a_{ik} : k = 1:n\}$  for some  $i$ . We will exploit the latter property but not the former.

To understand the code, consider the  $j$ th solver and a given scalar  $\theta_k$ . We need to compute  $\phi_j(\theta_k)$ , which is  $m^{-1}$  times the number of  $i$  for which `coli`  $\leq \theta_k$ , where

Listing 26.3. *Function* perfprof.

```

function [th_max,h] = perfprof(A,th_max)
%PERFPROF Performance profile.
% [th_max, h] = PERFPROF(A,th_max) produces a
% performance profile for the data in the M-by-N matrix A,
% where A(i,j) > 0 measures the performance of the j'th solver
% on the i'th problem, with smaller values of A(i,j) denoting
% "better". For each solver theta is plotted against the
% probability that the solver is within a factor theta of
% the best solver over all problems, for theta on the interval
% [1, th_max].
% Set A(i,j) = NaN if solver j failed to solve problem i.
% TH_MAX defaults to the smallest value of theta for which
% all probabilities are 1 (modulo any NaN entries of A).
% h is a vector of handles to the lines with h(j)
% corresponding to the j'th solver.

minA = min(A,[],2);
if nargin < 2, th_max = max( max(A,[],2)./minA ); end
tol = sqrt(eps); % Tolerance.

[m,n] = size(A); % m problems, n solvers.

for j = 1:n % Loop over solvers.

    col = A(:,j)./minA; % Performance ratios.
    col = col(~isnan(col)); % Remove NaNs.
    if isempty(col), continue; end
    theta = unique(col)'; % Unique elements, in increasing order.
    r = length(theta);
    prob = sum( col(:,ones(r,1)) <= theta(ones(length(col),1),:) ) / m;
    % Assemble data points for staircase plot.
    k = [1:r; 1:r]; k = k(:)';
    x = theta(k(2:end)); y = prob(k(1:end-1));

    % Ensure endpoints plotted correctly.
    if x(1) >= 1 + tol, x = [1 x(1) x]; y = [0 0 y]; end
    if x(end) < th_max - tol, x = [x th_max]; y = [y y(end)]; end
    h(j) = plot(x,y); hold on

end
hold off
xlim([1 th_max])

```

$\text{col}_i = a_{ij} / \min\{a_{ik} : k = 1:n\}$ . In MATLAB notation, exploiting scalar expansion,  $\phi_j(\theta_k)$  is<sup>8</sup>

```
sum( col <= theta(k) )/m
```

We need to carry out this computation for each element in the 1-by- $r$  row vector `theta`, which can be done with the loop

```
for k = 1:r
    prob(k) = sum( col <= theta(k) )/m
end
```

Using the indexing trick from Section 24.4, this loop can be vectorized to

```
prob = sum( col(:,ones(r,1)) <= theta(ones(length(col),1),:) ) / m
```

The reason for writing `length(col)` rather than `m` is that `perfprof` first needs to remove NaNs from `col`, and hence when `prob` is formed `col` may have fewer than `m` elements. The elements of `theta`, which are the distinct and sorted elements of `col`, are obtained with the MATLAB function `unique`.

Some further work is needed to produce a plot that properly displays the piecewise linear nature of the curves  $\phi_j(\theta)$  (i.e., a staircase plot). The last few lines of the loop construct the data pairs to be passed to `plot`.

To make the plots as readable as possible it is necessary to set line styles, marker types, a legend, and so on. Achieving this within `perfprof` via input arguments would be clumsy. Instead it is left for the user to set the relevant properties of the graphics objects after calling `perfprof`.

Function `ode_pp` in Listing 26.4 illustrates the use of `perfprof`. It times the three MATLAB nonstiff ODE solvers (see Table 12.2) on six test problems. (Function `fox1` is the function in Listing 12.3.) The function illustrates various advanced MATLAB programming techniques that have been discussed in the book. In particular, it includes both nested functions and local functions (with a nested function inside a local function). Note that for the timings from `ode_pp` to be meaningful we need to run the function twice, as the first time it is run there is the unwanted overhead of MATLAB compiling `ode_pp` and the solvers into its internal format.

The performance profile plot from `ode_pp` is shown in Figure 26.3. We now explain how to interpret the figure. But, first, we emphasize that *this example is purely illustrative and the results should not be taken at face value*. Indeed, some of the test problems are difficult and this experiment does not check the correctness of the solutions computed. The experiment was designed simply to give an interesting performance profile. The numbers on which the figure is based are shown in Table 26.1 (this is the transpose of the array `T` returned by `ode_pp`).

We now explain how to interpret Figure 26.3. Note first that since there are only  $m = 6$  test problems we have explicitly set the  $y$ -axis tick marks to the possible values of  $\phi_j(\theta)$  ( $0, 1/m, 2/m, \dots, 1$ ) and then assigned appropriately short tick labels; for larger  $m$ , automatic tick marks will probably give better readability.

- Left-hand side of plot,  $\phi_s(1)$ : `ode23` is the fastest solver on 50% of the problems, with `ode45` and `ode113` being fastest on 33% and 17% of the problems, respectively.

<sup>8</sup>Since the argument to `sum` is a vector of zeros and ones, it would be more efficient to replace `sum` by `nnz` here, but `nnz` does not produce the desired result in the vectorized expression used in `perfprof`.



Table 26.1. *Data in transpose of array T from ode\_pp.*

Problem	1	2	3	4	5	6
ode23	1.26e-2	2.41e-1	3.74e-2	3.37e0	1.44e-1	5.06e-1
ode45	6.20e-3	1.53e-1	5.00e-2	6.45e0	1.56e-1	1.07e0
ode113	1.56e-2	1.97e-1	6.68e-2	7.86e0	3.76e-2	1.50e0

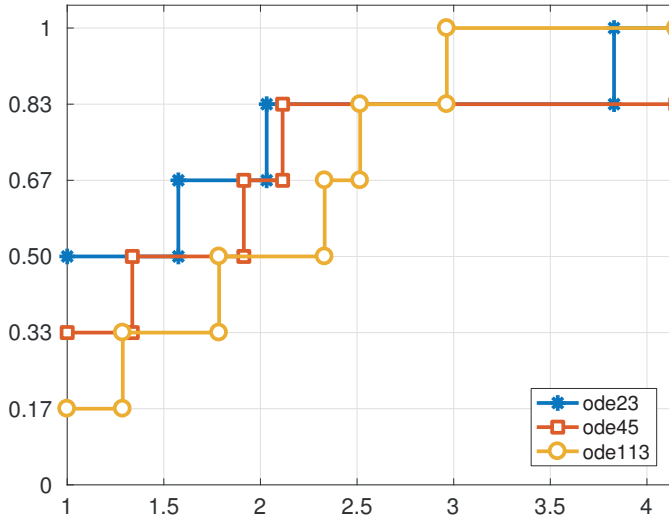


Figure 26.3. *Performance profile produced by ode\_pp.*

- Middle of plot, where the curves all cross: If our criterion for choosing a solver is that it has an 83% chance of being within a factor 2.5 of the fastest solver then all three solvers are equally good.
- Middle to right-hand side of plot, looking where the curves first hit probability 1: `ode113` is within a factor  $\theta$  of being the fastest solver on every problem for  $\theta \approx 3$ . For the same to be true for `ode23` and `ode45` we need to increase  $\theta$  to 3.8 and 4.1, respectively.

The performance profile therefore answers several different aspects of the question, “Which is the best solver?” It shows that `ode23` is most often the fastest, `ode113` is the most reliable in the sense of being the least likely to be much slower than the fastest, and `ode45` treads a middle ground between the two (each statement applying only to this very small and unrepresentative set of test problems).

Listing 26.4. *Function ode\_pp.*

```

function T = ode_pp
%ODE_PP    Performance profile of three ODE solvers.

solvers = {@ode23, @ode45, @ode113};  nsolvers = length(solvers);
nproblems = 6;
nruns = 5;  % Number of times to run solver to get more reliable timing.

for j = 1:nsolvers
    code = solvers{j}
    for i = 1:nproblems

        options = [];
        switch i
            case 1
                fun = @fox1; tspan = [0 10]; yzero = [3;0];
            case 2
                fun = @crossler; tspan = [0 100]; yzero = [1;1;1];
                options = odeset('AbsTol',1e-7,'RelTol',1e-4);
            case 3
                fun = @fvdpol; tspan = [0 20]; yzero = [2;1]; mu = 10;
            case 4
                fun = @fvdpol; tspan = [0 20]; yzero = [2;1]; mu = 1000;
            case 5
                fun = @drug_transport; tspan = [0 6]; yzero = [0;0];
            case 6
                fun = @knee; tspan = [0 2]; yzero = 1;
        end

        t0 = clock;
        for k = 1:nruns
            [t,y] = code(fun,tspan,yzero,options);
        end
        T(i,j) = etime(clock,t0)/nruns;

    end
end

[~,h] = perfprof(T);
ylim([0 1.05]), grid

yvals = 0:1/nproblems:1;
ax = gca;
ax.YTick = yvals;
ax.YAxis.TickLabelFormat = '%4.2f '
ax.YTickLabel{1} = '0 '; ax.YTickLabel{end} = '1 ';
ax.FontSize = 12;

legend('ode23','ode45','ode113','Location','SE')
set(h,{'Marker'},{'*','s','o'})      % Vectorized set.
set(h,'MarkerSize',8)
set(h,'MarkerFaceColor','auto') % Make marker interiors non-transparent.

```

```

set(h,{'LineStyle'},{'-','-','-'}) % Vectorized set.
set(h,'LineWidth',2)

function yprime = fvdpol(x,y)
%FVDPOL Van der Pol equation written as first order system.
% Parameter MU.
yprime = [y(2); mu*y(2)*(1-y(1)^2)-y(1)];
end

end

function yprime = rossler(t,y)
%ROSSLER Rossler system, parameterized.
a = 0.2; b = 0.2; c = 2.5;
yprime = [-y(2)-y(3); y(1)+a*y(2); b+y(3)*(y(1)-c)];
end

function yprime = drug_transport(t,y)
%DRUG_TRANSPORT Two-compartment pharmacokinetics example.
% Reference: Shampine (1994, p. 105).
yprime = [-5.6*y(1) + 48*pulse(t,1/48,0.5); 5.6*y(1) - 0.7*y(2)];

function pls = pulse(t,w,p)
%PULSE Pulse of height 1, width W, period P.
pls = (rem(t,p) <= w);
end

end

function yprime = knee(t,y)
%KNEE Knee problem.
% Reference: Shampine (1994, p. 115).
epsilon = 1e-4;
yprime = (1/epsilon)*((1-t)*y - y^2);
end

```

For a well-chosen set of test problems, the inequality  $\phi_i(\theta) \leq \phi_j(\theta)$  holding for all  $\theta$  is certainly strong evidence that solver  $j$  is superior to solver  $i$ , but this inequality does not imply that solver  $j$  performs better than solver  $i$  on every test problem. This is illustrated by `ode45` and `ode23` in Table 26.1 and Figure 26.3.

In some applications a solver may fail to solve a problem. For example, an optimization code may fail to converge or it may converge to a nonoptimal point. The failure of solver  $j$  to solve problem  $i$  can be accounted for by setting  $A(i, j) = \text{NaN}$ . To illustrate, consider this script, based on entirely fictitious data:

```

A = [1    2    3    4
     2    4    6    8
     NaN  3    NaN  2
     1    2    5    2
     NaN 10    NaN 20
     1    1    4    6
     3    NaN  4    5

```

```

      4      2      3      1
NaN      2      2      2
NaN      3      5      5
NaN     NaN     NaN     5
NaN      2      1      3];

th_max = 7;
[~,h] = perfprof(A,th_max*1.1);
xlim([1 th_max]), ylim([0 1.05])
legend('Column 1','Column 2','Column 3','Column 4','Location','SE')
set(h,{'LineStyle'},{'-','-','--',':'}) % Vectorized set.
set(h,{'Color'},{'r','b','g','k'}) % Vectorized set.
set(h,{'Marker'},{'*','o','s','+'}) % Vectorized set.
set(h,'MarkerFaceColor','auto') % Marker interiors non-transparent.
set(h,'LineWidth',2), set(h,'MarkerSize',8)
set(gca,'YTick',0:0.1:1)
set(gca,'FontSize',12)

```

Figure 26.4 is produced. The intersection of the curves with the right-hand axis shows the proportion of problems that could be solved, ranging from 0.5 to 1. The solver corresponding to the red line and the “\*” marker (and represented by the first column of  $A$ ) was most often the best but solved the fewest problems. Such information would be hard to discern from a large array of data, but it is immediately apparent from the figure. This example illustrates the use of the second input argument `th_max` of `perfprof`. By default, the  $x$ -axis would cover the range  $[0, 6]$ . We have called `perfprof` with a larger value of `th_max` in order to better display the “flatlining” effect. The reason we pass a second argument to `perfprof` that is slightly larger than the intended  $x$ -axis upper limit is to avoid markers being plotted where the lines meet the right-hand edge of the plot, since these intersections are not data points. With a more complicated data set it may be necessary to call `perfprof` twice: once to compute the default `th_max` and a second time with an increased value (as in this example).

We note two further refinements. First, for data sets that produce a large `th_max`, setting a logarithmic scale on the  $x$ -axis can help make the performance profile more readable. This can be done after calling `perfprof` by issuing the command `set(gca,'XScale','log')` (see Section 17.1). Second, when the data are relative errors of the order `eps`, the performance profile can be skewed by the presence of a few abnormally small relative errors much smaller than `eps`. In this case, the simple transformation of the data suggested in [34] can produce a more meaningful performance profile.

## 26.5. Multidimensional Calculus

The calculus features of the Symbolic Math Toolbox are very useful for solving problems that are too tedious to treat by hand yet small and simple enough that symbolic manipulation is feasible. We illustrate by finding and classifying the stationary points of the function

$$F(x, y) = 4x^2 - 3x^4 + x^6/3 + xy - 4y^2 + 4y^4. \quad (26.4)$$

This is a slight variation of the function pictured in Figure 8.17.

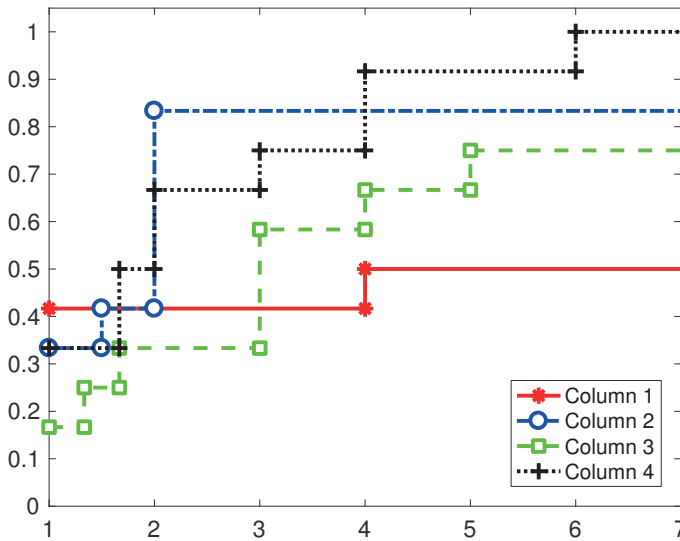


Figure 26.4. Performance profile for fictitious data in 12-by-4 array A.

The stationary points are the points where the gradient vector

$$\nabla F(x, y) = \begin{bmatrix} \frac{\partial F}{\partial x} \\ \frac{\partial F}{\partial y} \end{bmatrix}$$

is zero. The nature of a stationary point—minimum, maximum, or saddle point—can be determined from the signs of the eigenvalues of the Hessian matrix,

$$\nabla^2 F(x, y) = \begin{bmatrix} \frac{\partial^2 F}{\partial x^2} & \frac{\partial F}{\partial x \partial y} \\ \frac{\partial F}{\partial y \partial x} & \frac{\partial^2 F}{\partial y^2} \end{bmatrix},$$

provided the matrix is nonsingular.

### Code and Walk-through

Script `camel_solve` in Listing 26.5 symbolically computes the gradient and then uses `solve` to find the points where the gradient is zero. A general principle is that results from a symbolic manipulation package should always be tested and should not automatically be trusted. For each putative stationary point, the code checks numerically that the gradient is small enough to be regarded as zero, and it discards complex solutions, which are not of interest.

The gradient and Hessian are computed symbolically using the toolbox commands `gradient` and `hessian`, for conciseness of code. The eigenvalues of the numerically evaluated Hessians are then used to classify the stationary points. Finally, the stationary points are printed, arranged by type, and the contour plot shown in Figure 26.5 is produced. The output from the script is as follows:

Listing 26.5. *Script camel\_solve.*

```

%CAMEL_SOLVE Find stationary points of the camel function.
%           This script requires the Symbolic Math Toolbox.

format short e
syms x y
f = 4*x^2 - 3*x^4 + x^6/3 + x*y - 4*y^2 + 4*y^4;

g = gradient(f,[x y])
disp('Original solutions:')
s = solve(g)

H = hessian(f,[x,y])
n = length(s.x); j = 1; minx = []; maxx = []; saddlex = [];

for i = 1:n % Loop over stationary points.
    fprintf('Point %2.0f: ',i)
    xi = s.x(i); yi = s.y(i); pointi = double([xi yi]);
    gi = double(subs(g,{x,y},{xi,yi}));
    % Filter out nonreal points and points where gradient not zero.
    if norm(gi) > eps
        fprintf('gradient is nonzero!\n')
    elseif ~isreal(pointi)
        fprintf('is nonreal!\n')
    else
        fprintf('(%10.2e,%10.2e) ', pointi)
        Hi = double(subs(H,{x,y},{xi,yi}));
        eig_Hi = eig(Hi);
        if all(eig_Hi > 0)
            minx = [minx; pointi]; fprintf('minimum\n')
        elseif all(eig_Hi < 0)
            maxx = [maxx; pointi]; fprintf('maximum\n')
        elseif prod(eig_Hi) < 0
            saddlex = [saddlex; pointi]; fprintf('saddle point\n')
        else
            fprintf('nature of stationary point unclear\n')
        end
    end
end
end
minx, maxx, saddlex

plot(minx(:,1),minx(:,2),'*k', maxx(:,1),maxx(:,2),'ok',...
      saddlex(:,1),saddlex(:,2),'xk','MarkerSize',8)
hold on, a = axis;
[x,y] = meshgrid(linspace(a(1),a(2),200),linspace(a(3),a(4),200));
z = subs(f); % Replaces symbolic x, y with numeric values from workspace.
contour(x,y,z,30)
map = hot; colormap(map(1:40,:)); % Darker part of this color map.
xlim([-2.5 2.5]) % Fine tuning.
legend('Min', 'Max', 'Saddle')
g = findall(gca,'type','axes'); set(g,'FontSize',12)
hold off

```

```

g =
  2*x^5 - 12*x^3 + 8*x + y
      16*y^3 - 8*y + x
Original solutions:
s =
  x: [15×1 sym]
  y: [15×1 sym]
H =
 [ 10*x^4 - 36*x^2 + 8,      1]
 [                1, 48*y^2 - 8]
Point 1: ( 0.00e+00, 0.00e+00) saddle point
Point 2: ( 8.82e-01, 1.13e-01) maximum
Point 3: ( 9.18e-01, 6.41e-01) saddle point
Point 4: (-9.02e-02, 7.13e-01) minimum
Point 5: (-8.14e-01, 7.53e-01) saddle point
Point 6: (-2.30e+00, 8.21e-01) minimum
Point 7: is nonreal!
Point 8: is nonreal!
Point 9: (-8.82e-01, -1.13e-01) maximum
Point 10: (-9.18e-01, -6.41e-01) saddle point
Point 11: ( 9.02e-02, -7.13e-01) minimum
Point 12: ( 8.14e-01, -7.53e-01) saddle point
Point 13: ( 2.30e+00, -8.21e-01) minimum
Point 14: is nonreal!
Point 15: is nonreal!
minx =
 -9.0183e-02  7.1268e-01
 -2.2969e+00  8.2144e-01
  9.0183e-02 -7.1268e-01
  2.2969e+00 -8.2144e-01
maxx =
  8.8223e-01  1.1318e-01
 -8.8223e-01 -1.1318e-01
saddlex =
      0      0
  9.1833e-01  6.4063e-01
 -8.1410e-01  7.5335e-01
 -9.1833e-01 -6.4063e-01
  8.1410e-01 -7.5335e-01

```

Fifteen stationary points are found symbolically, 11 of which are verified to be real and have a zero gradient.

There is no guarantee that the `solve` function yields all the solutions of the system it is asked to solve, so further analysis is needed to determine whether `camel_solve` has found all the stationary points. More sophisticated methods for solving this type of problem are explained in Chapter 4, titled “Think Globally, Act Locally”, of [13], which contains some MATLAB code.

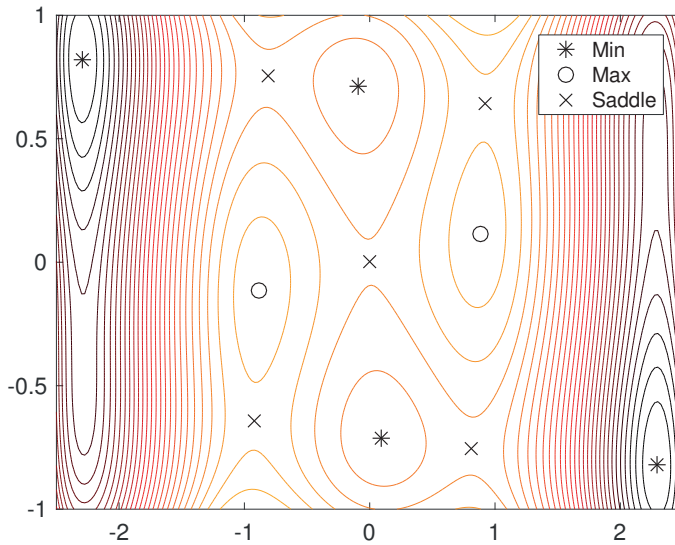


Figure 26.5. *Contours and stationary points of camel function (26.4).*

## 26.6. L-Systems and Turtle Graphics

The L-system formulation provides a simple means to draw plant-like objects. We will consider the case where such objects are represented by strings from an alphabet of five characters: F, [, ], +, -. Here, the [ and ] characters must appear in matching pairs. We may view a string formally using the *turtle graphics* idea. Imagine a turtle equipped with a pen. The turtle reads the characters in the string sequentially, from left to right, interpreting them as instructions, and thereby draws a picture. At any given stage, the turtle has a *current position*,  $(x, y)$ , and a *current move vector*,  $(dx, dy)$ . The characters have the following precise meanings.

F means perform the current move; that is, draw a line from  $(x, y)$  to  $(x + dx, y + dy)$ .

Update the current position to  $(x + dx, y + dy)$ . Keep the current move vector as  $(dx, dy)$ .

+ means turn clockwise through a prespecified angle  $\theta^+$ ; that is, change the current move vector from  $(dx, dy)$  to  $(\cos(\theta^+)dx + \sin(\theta^+)dy, -\sin(\theta^+)dx + \cos(\theta^+)dy)$ .

- means turn counterclockwise through a prespecified angle  $\theta^-$ ; that is, change the current move vector from  $(dx, dy)$  to  $(\cos(\theta^-)dx - \sin(\theta^-)dy, \sin(\theta^-)dx + \cos(\theta^-)dy)$ .

[ means record the current values of  $(x, y)$  and  $(dx, dy)$ ; that is, push them onto a stack. Then scale  $(dx, dy)$  by a prespecified factor. The turtle does not move. When the matching ] marker is reached, that position  $(x, y)$  and move vector  $(dx, dy)$  are popped off the stack; the turtle returns to  $(x, y)$  (without drawing) and resets its current move vector to  $(dx, dy)$ .

In order to create our strings, we must define an *initial state* and a *production rule*. We will always take the initial state to be F. Then, in general, to get from one



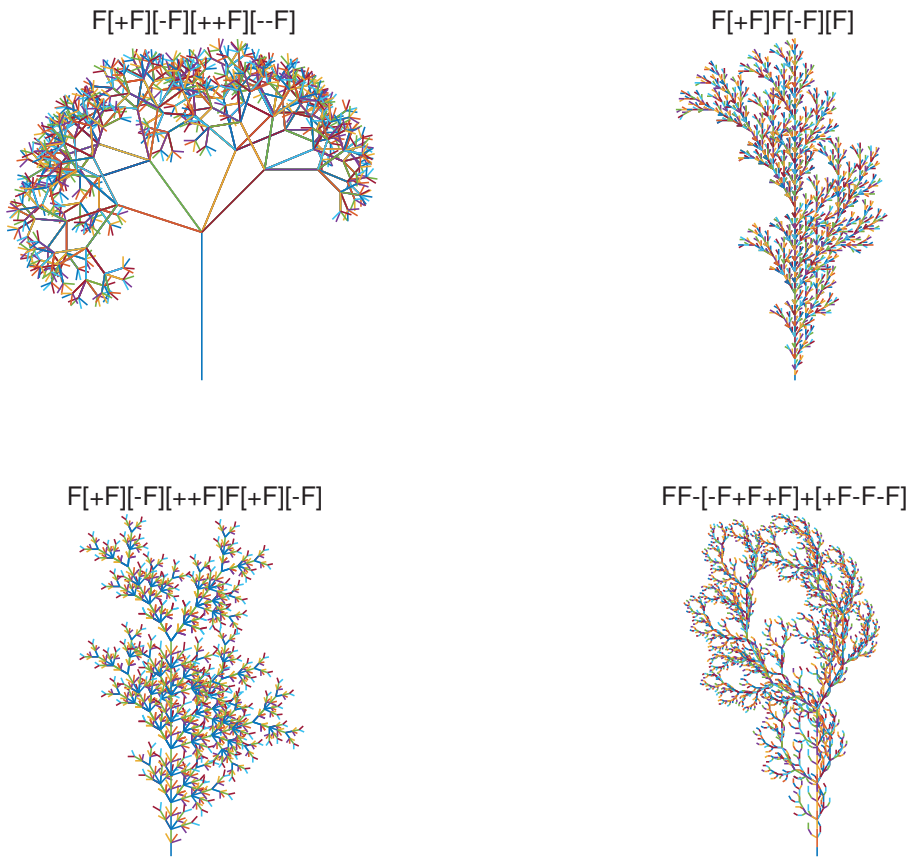


Figure 26.6. *Members of the genus Matlabius Floribundum produced by 1sys.*

generation to the next we replace every occurrence of  $F$  by the production rule. For example, with the production rule  $F[+F]F[-F]F$  we have

**Initial state**  $F$

**1st generation**  $F[+F]F[-F]F$

**2nd generation**  $F[+F]F[-F]F[+F][+F]F[-F]F[+F]F[-F]F[-F][+F]F[-F]F$   
 $F[+F]F[-F]F$

The process is akin to using the “search and replace all” facility available in a typical text editor, with  $F$  being searched for and replaced by the production rule. Our aim is now to draw the picture that arises when the rule, generation level, turning angles, and scale factor are specified.

The book [143] gives a very readable discussion of the ideas behind L-systems, which are named after the Swedish biologist Aristid Lindenmayer (1925–89).

### Code and Walk-through

The recursive function `1sys` in Listing 26.6 combines the string production and string interpretation phases. The input variable `rule` is the required production rule. `1sys`

uses the `switch` construct to parse the rule, taking the appropriate action for each character. In particular, `F` results in a recursive call to `lsys` with the generation decremented by one. Note that the arrays `cstack` and `dstack` are not preallocated. Since the stack is usually just a few levels deep this is not a major inefficiency. Called with `gen` equal to 1, `lsys` draws the first generation plant.

The script `lsys_run` in Listing 26.7 calls `lsys` with four different sets of parameters, producing the pictures in Figure 26.6.

## 26.7. Black–Scholes Delta Surface

A European call option is a financial product that gives its holder the right to purchase from its writer an asset at a specified price, known as the *exercise price*, at some specified time in the future, known as the *expiry date*. A seminal paper by Black and Scholes shows how the writer of an option can eliminate risk by dynamically hedging with a portfolio of asset and cash. The amount of asset that the writer must hold is known as the *delta* of the option. Black and Scholes's formula for the delta is  $N(d_1)$ , where

$$d_1 = \frac{\log(S(t)/E) + (r + \frac{1}{2}\sigma^2)(T - t)}{\sigma\sqrt{T - t}}. \quad (26.5)$$

Here,

- $t$  denotes time, with  $t = 0$  and  $t = T$  specifying the start and expiry dates,
- $S(t)$  is the asset price at time  $t$ ,
- $E$  is the exercise price,
- $r$  is the interest rate,
- $\sigma$  is the asset volatility,
- $N(\cdot)$  is the distribution function for a standard normal random variable, defined as

$$N(x) := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-s^2/2} ds.$$

MATLAB has a function `erf` that evaluates the *error function*

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

from which the normal distribution function may be obtained as

$$N(x) = \frac{1 + \text{erf}(x/\sqrt{2})}{2}.$$

The Black–Scholes theory is derived under the assumption that the asset price  $S(t_i)$  at time  $t_i$  evolves into  $S(t_{i+1})$  at time  $t_{i+1} > t_i$  according to

$$S(t_{i+1}) = S(t_i) \exp\left(\left(\mu - \frac{1}{2}\sigma^2\right)(t_{i+1} - t_i) + \sigma\sqrt{t_{i+1} - t_i}Z_i\right), \quad (26.6)$$

where  $Z_i$  is a standard normal random variable. Here,  $\mu$  is a constant that governs the expected increase in the asset.

Listing 26.6. *Function* lsys.

```

function [coord,mov] = lsys(rule,coord,mov,angle,scale,gen)
%LSYS    Recursively generated L-system.
%    LSYS(rule,coord,mov,angle,scale,gen) generates the L-system
%    produced by gen generations of the production rule given
%    in the string rule.
%    coord and mov are the initial (x,y) and (dx,dy) values.
%    angle is a 2-vector, with angle(1) specifying the clockwise
%    rotations and angle(2) the counterclockwise rotations.
%    scale is the scale factor for branch length.

%    During recursion, gen, coord, and mov record the current state.

if gen == 0
    % Draw line, then update location.
    plot([coord(1),coord(1)+mov(1)],[coord(2),coord(2)+mov(2)])
    coord = coord + mov;
    hold on
else
    stack = 0;
    for k=1:length(rule)
        switch rule(k)
            case 'F'
                [coord,mov] = lsys(rule,coord,mov,angle,scale,gen-1);
            case '+'
                mov = [cos(angle(1)) sin(angle(1));
                    -sin(angle(1)) cos(angle(1))]*mov;
            case '-'
                mov = [cos(angle(2)) -sin(angle(2));
                    sin(angle(2)) cos(angle(2))]*mov;
            case '['
                stack = stack + 1;
                cstack(1:2,stack) = coord;
                dstack(1:2,stack) = mov;
                mov = scale*mov;
            case ']'
                coord = cstack(1:2,stack);
                mov = dstack(1:2,stack);
                stack = stack - 1;
        end
    end
end
end

```

Listing 26.7. *Script lsys\_run.*

```

%LSYS_RUN   Runs lsys function to draw L-systems.

subplot(2,2,1)
rule = 'F[+F][-F][++F][--F]';
[c,d] = lsys(rule,[0;0],[0;1],[pi/8;pi/5],0.6,5);
options = {'FontSize',8,'FontWeight','normal'};
title(rule,options{:}), axis equal, axis off

subplot(2,2,2)
rule = 'F[+F]F[-F][F]';
[c,d] = lsys(rule,[0;0],[0;1],[pi/6;pi/6],1,5);
title(rule,options{:}), axis equal, axis off

subplot(2,2,3)
rule = 'F[+F][-F][++F]F[+F][-F]';
[c,d] = lsys(rule,[0;0],[0;1],[pi/5;pi/6],0.8,4);
title(rule,options{:}), axis equal, axis off

subplot(2,2,4)
rule = 'FF[-F+F+F][+F-F-F]';
[c,d] = lsys(rule,[0;0],[0;1],[pi/6;pi/6],0.7,4);
title(rule,options{:}), axis equal, axis off

```

Our aim is to plot the delta-value  $N(d_1)$  as a function of asset price  $S$  and time  $t$ , with all the other parameters,  $T$ ,  $E$ ,  $r$ ,  $\sigma$ , and  $\mu$ , fixed. This will give a surface above the  $(S, t)$ -plane. We will then generate three asset paths over  $[0, T]$ , using (26.6) to update the price between finely spaced time points, with the  $Z_i$  generated by calls to the normal pseudorandom number generator, `randn`. We will sit these paths on the delta surface, that is, plot the path of  $N(d_1)$  when  $d_1$  in (26.5) takes the values given by  $(S_i, t_i)$ . This illustrates the amount of asset that the option writer must maintain as the asset price evolves, in each of the three cases.

For further details on financial option valuation, see, for example, [67] and [183].

### Code and Walk-through

The script `bsdelta` in Listing 26.8 produces the picture in Figure 26.7. A key issue to address here is the division-by-zero arising when  $t = T$  in (26.5). This difficulty is avoided by defining  $N(d_1)$  at  $t = T$  in terms of its limit from below:

$$\lim_{t \rightarrow T^-} N(d_1) = \begin{cases} 1 & \text{if } S(T) > E, \\ \frac{1}{2} & \text{if } S(T) = E, \\ 0 & \text{if } S(T) < E. \end{cases} \quad (26.7)$$

After initializing parameters, we use `meshgrid` to set up the arrays `Sgrid` and `tgrid` that are needed by `surf`, with the final time level omitted. We compute  $d_1$  and  $N(d_1)$  at these points, and then fill in the  $t = T$  values separately, using (26.7). We call `surf` to display the surface, and then mark and label the axes.

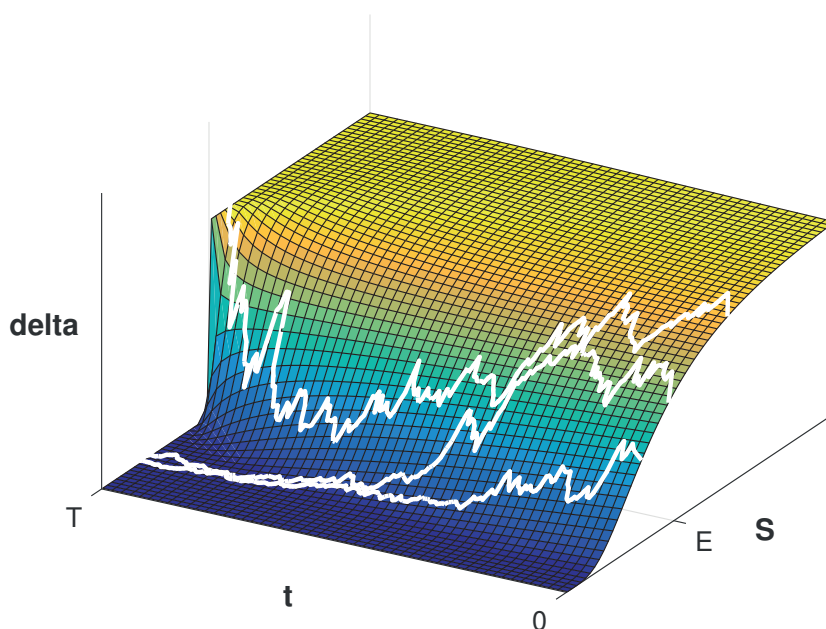


Figure 26.7. *Black-Scholes delta picture from bsdelta.*

The second part of the code generates the three asset paths. We take starting values of 1.5, 0.95, and 0.7. The cumulative product function, `cumprod`, is used to apply (26.6) over all time points. We then use the formula (26.5) for  $t < T$  and (26.7) for  $t = T$  to give a path of surface heights, `Npath`. These are superimposed by the 3D line plotter, `plot3`, with an increment of `deltaN = 0.1` added to the heights so that the paths are visible above the surface. Finally, we alter the default view to give a clearer picture.

## 26.8. Chutes and Ladders

In the game of Chutes and Ladders (also known as Snakes and Ladders) a player moves between squares on a board according to the roll of a die. On each turn, the number rolled, 1, 2, 3, 4, 5, or 6, determines how many squares to advance, with the constraints that

- if the new location is the foot of a ladder, the player automatically jumps up to the square at the top of that ladder,
- if the new location is the top of a chute (head of a snake), the player automatically slides down to the square at the end of that chute (tail of that snake),
- if the player would progress beyond the final square, that turn is discarded, and the player's location is unchanged.

The game typically involves two or more players, with the first to reach the final square being deemed the winner. However, as there is no interaction between players,

Listing 26.8. *Script bsdelta.*

```

%BSDELTA Black--Scholes delta surface with three asset paths superimposed.

rng(51)
E = 1; r = 0.05; sigma = 0.6; mu = 0.05; T = 1;
N1 = 50; Dt = T/N1; N2 = 60;

tvals = [0:Dt:T-Dt]; % Avoid division by zero.
Svals = linspace(.01,2.5,N2);
[Sgrid,tgrid] = meshgrid(Svals,tvals);

d1grid = (log(Sgrid/E) + ...
          (r+0.5*sigma^2)*(T-tgrid))./(sigma*sqrt(T-tgrid));
Ngrid = 0.5*(1+erf(d1grid/sqrt(2)));

tvals = [0:Dt:T]; % Add expiry date.
[Sgrid,tgrid] = meshgrid(Svals,tvals); % Extend the grid.
Ngrid(end+1,:) = 0.5*(sign(Svals - E) + 1); % Append final time values.

surf(Sgrid,tgrid,Ngrid)
hx = xlabel('S','FontWeight','bold','FontSize',16);
hx.Position = hx.Position + [0.2 -0.1 0.2];
hy = ylabel('t','FontWeight','bold','FontSize',16);
hy.Position = hy.Position + [0 0.15 0];
zlabel('delta','FontWeight','bold','FontSize',16,...
       'Rotation',0,'HorizontalAlignment','right')
ylim([0 T]), xlim([0 2.5])
set(gca,'ZTick',[])
set(gca,'YTick',[0,T]), set(gca,'YTickLabel',{'0','T'},'FontSize',12)
set(gca,'XTick',E), set(gca,'XTickLabel','E','FontSize',12)

% Superimpose asset paths.
hold on
L = 200; Dt = T/L;
tpath = [0:Dt:T-Dt]';
Szero = [1.5;0.95;0.7];
for k = 1:3
    factors = exp((mu-0.5*sigma^2)*Dt+sigma*sqrt(Dt)*randn(L,1));
    Spath = [Szero(k);Szero(k)*cumprod(factors)];
    dpath = (log(Spath(1:end-1)/E) + ...
            (r+0.5*sigma^2)*(T-tpath))./(sigma*sqrt(T-tpath));
    Npath = 0.5*(1+erf(dpath/sqrt(2)));
    Npath = [Npath;0.5*(sign(Spath(end)-E)+1)];
    deltaN = 0.1;
    Npath(2:end) = Npath(2:end)+deltaN;
    plot3(Spath,[tpath;T],Npath,'w-','Linewidth',2)
end
hold off, view(-60,35)

```



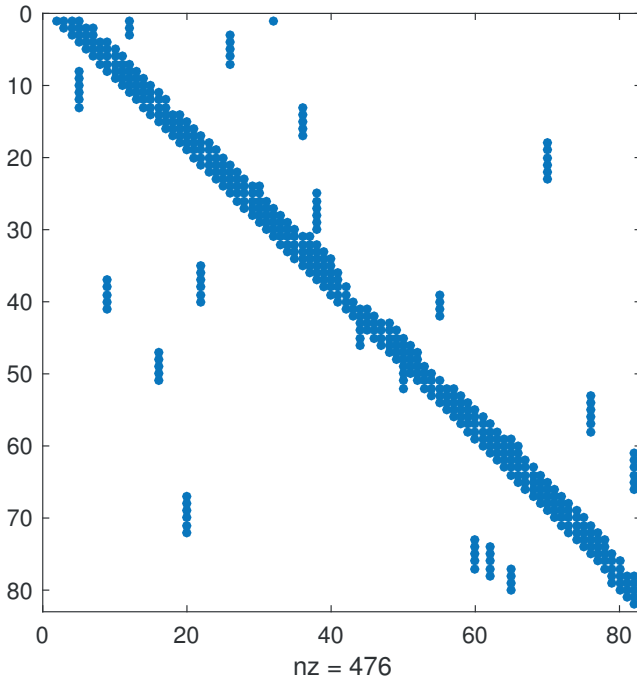


Figure 26.8. *spy plot of transition matrix from chute.*

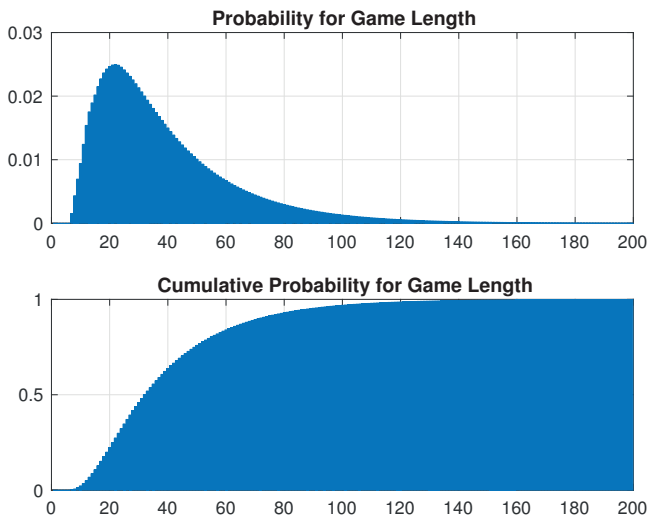


Figure 26.9. *Probability of finishing chutes and ladders game in exactly  $n$  rolls (upper) and at most  $n$  rolls (lower).*



Listing 26.9. *Script chute.*

```

%CHUTE    Chutes and ladders analysis.
%    Probability of finishing in exactly n moves and in at least n moves.

N = 100;    % Start at square zero, finish at square N.

% "+1" translates square to state.
top = [ 1  4  9 16 21 28 36 47 49 51 56 62 64 71 80  87 93 95 98] + 1;
bot = [38 14 31  6 42 84 44 26 11 67 53 19 60 91 100 24 73 75 78] + 1;

P = toeplitz(zeros(1,N+1),[0 ones(1,6) zeros(1,N-6)]);
for k = N-4:N+1, P(k,k) = k-N+5; end
P = P/6;

for k = 1:length(top)
    r = top(k); s = bot(k);    % Chute or ladder from r to s.
    P(:,s) = P(:,s) + P(:,r); % Add column r to column s.
end
P(top,:) = []; P(:,top) = []; % Remove starts of chutes and ladders.

figure(1)
spy(P)

M = 200;
cumprob = zeros(M,1);
cumprob(1) = P(1,end);
v = P(1,:);
for n = 2:M
    v = v*P;
    cumprob(n) = v(end);
end

figure(2)
colormap lines
subplot(2,1,1)
bar(diff([0;cumprob]))
title('Probability for Game Length')
grid on
xlim([0 M])
subplot(2,1,2)
bar(cumprob)
title('Cumulative Probability for Game Length')
grid on
xlim([0 M])

```

Having constructed the final transition matrix, which has dimension  $N + 1 - 19 = 82$ , we use `spy` to reveal the nonzero structure, as shown in Figure 26.8. We then compute the array `cumprob`, whose  $n$ th entry stores the probability of reaching the final state in no more than  $n$  rolls, for  $1 \leq n \leq 200$ . Applying `diff` to `[0;cumprob]` gives an array whose  $n$ th entry stores the probability of reaching the final state in exactly  $n$  rolls. The appropriate histograms are shown in Figure 26.9.

## 26.9. Pythagorean Sum

The Pythagorean sum  $\sqrt{a^2 + b^2}$  of two scalars is a commonly occurring quantity and can be regarded as an “atomic operation” on a par with the four elementary operations  $+$ ,  $-$ ,  $*$ ,  $/$  and the square root of a single scalar. Fast, reliable ways of computing Pythagorean sums are therefore needed. It is desirable to avoid explicitly computing a square root, since the square root is a relatively expensive operation, and also to avoid squaring  $a$  or  $b$ , since the squares could overflow or underflow despite  $\sqrt{a^2 + b^2}$  being within the range of the arithmetic. The MATLAB function `hypot` computes the Pythagorean sum of numeric arrays. Here, we will look at an approach particularly well-suited to high-precision computation.

Given  $x_0 \geq 0$  and  $y_0 \geq 0$  the following iteration computes  $p = \sqrt{x_0^2 + y_0^2}$ :

$$x_{n+1} = x_n \left( 1 + 2 \frac{y_n^2}{4x_n^2 + y_n^2} \right), \quad (26.9a)$$

$$y_{n+1} = \frac{y_n^3}{4x_n^2 + y_n^2}. \quad (26.9b)$$

To be precise, it can be shown that as  $n \rightarrow \infty$ ,  $x_n$  converges monotonically to  $p$  from below, and  $y_n$  decreases monotonically to 0 from above, with  $\sqrt{x_n^2 + y_n^2} = p$ . The rate of convergence of the iteration is cubic, which means that ultimately the error in  $x_n$  and  $y_n$  is bounded by a multiple of the cube of the error in  $x_{n-1}$  and  $y_{n-1}$ , respectively.

The iteration was originally suggested by Moler and Morrison, who develop an elegant floating-point arithmetic implementation that avoids overflow [129]. Our interest is in using the iteration to compute the Pythagorean sum to arbitrary precision, and for simplicity we will not scale to avoid overflow. It is when working to high precision that iterations with cubic or higher orders of convergence are particularly attractive.

### Code and Walk-through

Function `pythag` in Listing 26.10 implements iteration (26.9) in the `vpa` arithmetic of the Symbolic Math Toolbox. It has a third input argument, `d`, that specifies the required number of significant digits. The computations are done with `d + 10` digits, since rounding errors can be expected to make the last few digits incorrect. The first two input arguments can be symbolic expressions, so they are converted to `vpa` form before beginning the iteration; absolute values are also taken, to allow the routine to work for negative arguments. We use the function `narginchk` discussed in Section 10.6 to check that the requisite number of input arguments has been provided.

For efficiency of the iteration it is important to order the starting values so that  $y_0 \leq x_0$ , since otherwise the first few iterations are spent making  $x_n$  (which tends to a positive value) greater than  $y_n$  (which tends to zero). Note the use of the variables `yn2` and `temp` to reduce the amount of computation.

Listing 26.10. *Function* pythag.

```

function [xn,k] = pythag(x,y,d,noprnt)
%PYTHAG    Pythagorean sum in variable precision arithmetic.
%    p = PYTHAG(x,y,d) computes the Pythagorean sum
%    sqrt(x^2+y^2) of the real numbers x and y correct to about d
%    significant digits, using an iteration that avoids computing
%    square roots.  d defaults to 50.
%    By default, the progress of the iteration is printed;
%    the call PYTHAG(x,y,d,1) suppresses this.
%    [x,k] = PYTHAG(x,y,d) returns also the number of
%    iterations, k.

narginchk(2,4)    % Check number of input arguments.
if nargin < 4, noprint = 0; end
if nargin < 3, d = 50; end

d_old = digits;
% Work with slightly more accuracy than requested for final result.
digits(d+10)
x = abs(vpa(x)); y = abs(vpa(y));

xn = max(x,y); % Take max since xn increases to Pyth. sum.
yn = min(x,y);

k = 0;
x_change = 0;

while abs(x_change) < d
    k = k + 1;
    yn2 = yn^2;
    temp = yn2/(4*xn^2+yn2);
    xnp1 = xn*(1 + 2*temp);
    ynp1 = yn*temp;
    x_change = double( log10(abs((xnp1-xn)/xnp1)) );
    y_exp = double( log10(ynp1) );
    if ~noprint
        fprintf('log(rel_change_x_n): %6.0f, log(y_n): %6.0f\n', ...
                x_change, y_exp)
    end
    xn = xnp1; yn = ynp1;
end
xn = vpa(xn,d); % Return requested number of digits.
digits(d_old)    % Restore original value.

```

The convergence test checks whether the *d*th significant digit has changed since the previous iteration. The absolute value of the relative change in two successive iterates can be smaller than the smallest positive double-precision number (`realmin`), in which case it underflows to zero if converted to double precision. Hence in implementing the convergence test we compute the base 10 logarithm of the relative change, which *is* representable in double precision. The function prints to the screen information that shows the convergence of the iteration.

To illustrate, we compute  $\sqrt{(1/10)^2 + e^2}$  to 2,500 significant digits and check the result against the answer computed directly in `vpa` arithmetic. The code

```
d = 2500; digits(d)
x = sym('0.1'); y = sym(exp(sym(1)));
z = pythag(x,y,d);
z = char(z); [z(1:60) '...']

% Check:
p = vpa(sqrt(x^2+y^2), d+10);
p = char(p);
test_equal = strcmp(z(1:d),p(1:d))
```

produces the output

```
log(rel_change_x_n):    -3, log(y_n):    -4
log(rel_change_x_n):   -10, log(y_n):   -15
log(rel_change_x_n):   -31, log(y_n):   -46
log(rel_change_x_n):   -93, log(y_n):  -140
log(rel_change_x_n): -281, log(y_n): -421
log(rel_change_x_n): -843, log(y_n): -1264
log(rel_change_x_n): -Inf, log(y_n): -3795
ans =
2.7201206037473136693255563235433096109025216407472208575162...
test_equal =
1
```

The `-Inf` is a result of `xn` and `xnp1` being exactly equal. The cubic convergence is evident in the increase in size of the logarithms by a factor approximately three from one line to the next.

Figure 26.10 plots the execution time of `pythag` versus the number of requested digits for `x = vpa('1/3')`, `y = vpa('1/7')`. We see approximately linear growth of time with the number of digits. The number of iterations increases very slowly because of the cubic convergence, varying from 5 to 10.

## 26.10. Fisher's Equation

Fisher's equation, a PDE of the form

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + u(1 - u),$$

is used as a model for various biological phenomena. The right-hand side of the equation combines a diffusion term with a logistic growth term. It is common to

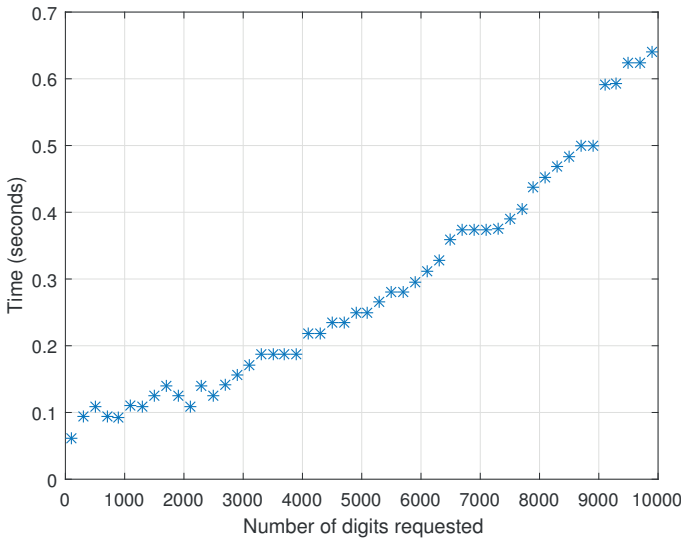


Figure 26.10. *Execution time of `pythag` versus requested accuracy.*

pose the equation over the whole  $x$ -axis,  $-\infty < x < \infty$ , and to specify boundary conditions  $u(x, t) \rightarrow 1$  as  $x \rightarrow -\infty$  and  $u(x, t) \rightarrow 0$  as  $x \rightarrow \infty$ . In this context, traveling-wave solutions of the form  $u(x, t) = f(x - ct)$  have been widely studied. For such a solution, the function  $f$  defines a fixed profile that is transported along the  $x$ -axis as time evolves. If we let  $z = x - ct$ , then the solution  $u(z, t)$  becomes stationary (independent of time) in the moving coordinate system  $(z, t)$ . In the hope of catching a traveling wave, we will take a large space interval,  $-50 \leq x \leq 50$ , and specify Neumann boundary conditions  $\partial u / \partial x = 0$  at  $x = \pm 50$ . We will solve the PDE for  $0 \leq t \leq 20$  with two different initial conditions: the step function

$$u(x, 0) = \begin{cases} 0.99, & x \leq -20, \\ 0, & x > -20; \end{cases} \quad (26.10)$$

and the small hump

$$u(x, 0) = \begin{cases} \frac{1}{4} \cos^2\left(\frac{\pi x}{10}\right), & |x| \leq 5, \\ 0, & |x| > 5. \end{cases} \quad (26.11)$$

### Code and Walk-through

The function `fisher` in Listing 26.11 uses `pdepe` to solve Fisher's equation, producing the pictures in Figures 26.11 and 26.12. The local function `fica` implements the first initial condition, (26.10), and a mesh plot of the resulting solution is displayed in the upper left region of the first figure window. We set `view(30,30)` in order to get a more revealing perspective. It appears that the solution is indeed evolving into a fixed profile that progresses linearly in time. To investigate further, the upper right picture shows `contour` applied to the solution, specifying contour levels of 0.2, 0.4, 0.6, and 0.8. The contours appear to settle into equally spaced straight lines. Although Fisher's equation admits traveling waves of any speed  $c \geq 2$ , it may be argued (see,

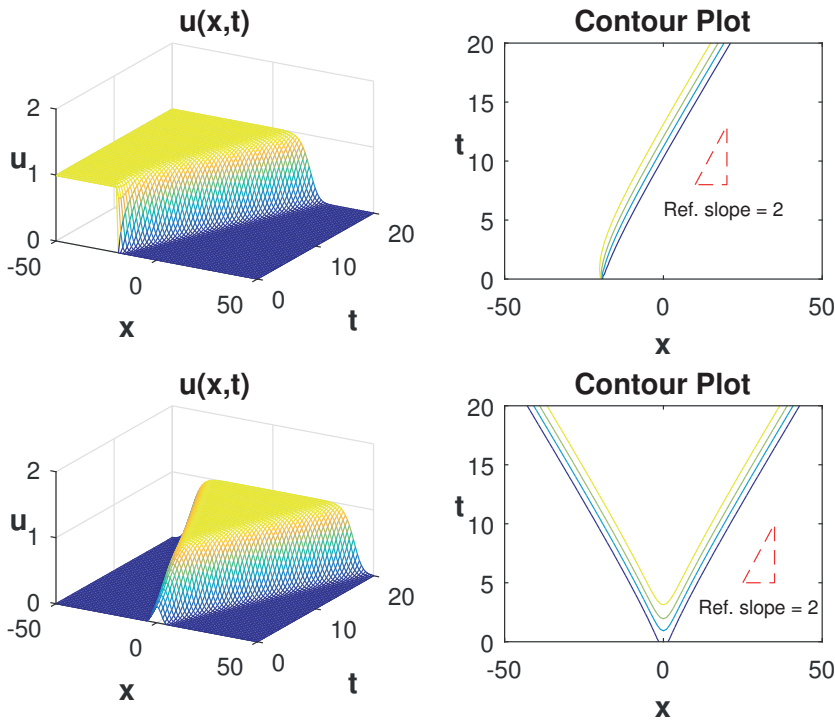


Figure 26.11. *Traveling-wave solutions for Fisher's equation, from fisher.*

for example, [131]) that a wave of speed  $c = 2$  is the most likely to be observed. A reference triangle of this slope has been added with a basic `plot` command to give a visual check.

The second initial condition, (26.11), implemented in `ficb`, gives rise to the pictures in the lower half of Figure 26.11. In this case, two wavefronts are generated, emanating from each side of the initial hump, and the contour plot is again consistent with wave speed  $c = 2$ .

In the second figure window, as shown in Figure 26.12, we `waterfall` the solution for (26.10) in the moving coordinate system  $(x - 2t, t)$  to give further visual confirmation that a traveling wave of speed  $c = 2$  has emerged.

Listing 26.11. *Script fisher.*

```

function fisher
%FISHER    Displays solutions to Fisher PDE.

m = 0; a = -50; b = 50; t0 = 0; tf = 20;
xvals = linspace(a,b,101); tvals = linspace(t0,tf,51);
[xmesh, tmesh] = meshgrid(xvals,tvals);

figure(1), subplot(2,2,1), sol = pdepe(m,@fpde,@fica,@fbc,xvals,tvals);
ua = sol(:,:,1); mesh(xmesh,tmesh,ua)
xlabel('x'), ylabel('t'), zlabel('u','Rotation',0), title('u(x,t)')
text_set, view(30,30)

subplot(2,2,2), contour(xmesh,tmesh,ua,[0.2:0.2:0.8])
xlabel('x'), ylabel('t','Rotation',0), title('Contour Plot')
text_set, hold on, txt = {'Ref. slope = 2','FontSize',8};
plot([10,20,20,10],[8,13,8,8],'r--'), text(0,6,txt{:}), hold off

subplot(2,2,3), sol = pdepe(m,@fpde,@ficb,@fbc,xvals,tvals);
ub = sol(:,:,1); mesh(xmesh,tmesh,ub)
xlabel('x'), ylabel('t'), zlabel('u','Rotation',0), title('u(x,t)')
text_set, view(30,30)

subplot(2,2,4), contour(xmesh,tmesh,ub,[0.2:0.2:0.8])
xlabel('x'), ylabel('t','Rotation',0), title('Contour Plot')
text_set, hold on
plot([25,35,35,25],[5,10,5,5],'r--'), text(11,3,txt{:}), hold off

figure(2), zmesh = xmesh - 2*diag(tvals)*ones(size(xmesh));
waterfall(zmesh,tmesh,ua)
xlabel('x-2t'), ylabel('t'), zlabel('u','Rotation',0), title('u(x-2t,t)')
zlim([0 1]), text_set, view(15,30)

%----- Local functions -----%
function [c,f,s] = fpde(x,t,u,DuDx)
%FDE Fisher PDE.
c = 1; f = DuDx; s = u*(1-u);

function u0 = fica(x)
%FIC Fisher initial condition: 1st case.
u0 = 0.99*(x<=-20);

function [pa,qa,pb,qb] = fbc(xa,ua,xb,ub,t)
%FBC Fisher boundary conditions.
pa = 0; qa = 1; pb = 0; qb = 1;

function u0 = ficb(x)
%FIC2 Fisher initial condition: 2nd case.
u0 = 0.25*(cos(0.1*pi*x).^2).*(abs(x)<=5);

function text_set
h = findall(gca,'type','text'); set(h,'FontSize',12,'FontWeight','bold')

```

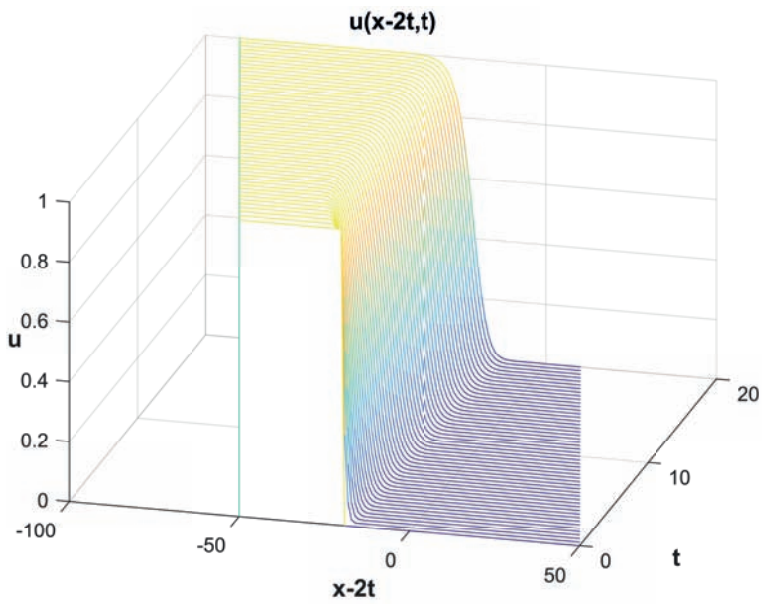


Figure 26.12. *Solution of Fisher's equation for initial conditions (26.10) in moving coordinate system, from fisher.*



*Example is always more efficacious than precept.*

— SAMUEL JOHNSON (1759)

*The computation of  $\text{sqrt}(a^2 + b^2)$  is required in many matrix algorithms,  
particularly those involving complex arithmetic.*

*A new approach to carrying out this operation is described by*

*Moler and Morrison ....*

*In MATLAB, the algorithm is used for complex modulus,  
Euclidean vector norm, plane rotations,  
and the shift calculation in the eigenvalue and singular value iterations.*

— CLEVE B. MOLER, *MATLAB Users' Guide* (1982)

*Performance profiles can be used to compare the performance of two solvers,  
but performance profiles are most useful in comparing several solvers.*

*Because large amounts of data are generated in these situations,  
trends in performance are often difficult to see.*

— ELIZABETH D. DOLAN and JORGE J. MORÉ,

*Benchmarking Optimization Software with Performance Profiles* (2002)

*One of the reasons MATLAB is so good at signal processing is that  
it was not designed for signal processing.*

*It was designed for mathematics.*

— JAMES MCCLELLAN

*And none of this would have been any fun without MATLAB.*

— NOËL M. NACHTIGAL, SATISH C. REDDY, and LLOYD N. TREFETHEN,  
*How Fast Are Nonsymmetric Matrix Iterations?* (1992)

# Appendix A

## The Top 111 MATLAB Functions

This appendix lists the 111 MATLAB functions that we believe are the most useful for the typical MATLAB user. Information about these functions can be found by looking in the index of this book or by using the online MATLAB documentation.

Table A.1. *Elementary and specialized vectors and matrices.*

<code>zeros</code>	Zeros array
<code>ones</code>	Ones array
<code>eye</code>	Identity matrix
<code>rand</code>	Uniformly distributed random numbers
<code>randn</code>	Normally distributed random numbers
<code>gallery</code>	Test matrices
<code>linspace</code>	Linearly spaced vector

Table A.2. *Special variables and functions.*

<code>ans</code>	Most recent answer
<code>eps</code>	Floating-point relative accuracy
<code>i, j</code>	Imaginary unit ( $\sqrt{-1}$ )
<code>inf</code>	$\infty$
<code>NaN</code>	Not a Number
<code>pi</code>	$\pi$

Table A.3. *Array information and manipulation.*

<code>size</code>	Array dimensions
<code>length</code>	Length of array (size of longest dimension)
<code>reshape</code>	Change size of array
<code>:</code>	Regularly spaced vector and index into matrix
<code>end</code>	Last index in an indexing expression
<code>diag</code>	Diagonal matrices and diagonals of matrix
<code>tril</code>	Extract lower triangular part
<code>triu</code>	Extract upper triangular part
<code>repmat</code>	Replicate and tile array

Table A.4. *Logical operators.*

<code>all</code>	Test for all nonzeros
<code>any</code>	Test for any nonzeros
<code>find</code>	Find indices of nonzero elements
<code>isempty</code>	Test for empty array
<code>isequal</code>	Test if arrays are equal

Table A.5. *Flow control.*

<code>error</code>	Display error message and abort function
<code>for</code>	Repeat statements a specific number of times
<code>if</code>	Conditionally execute statements
<code>switch, case</code>	Choose among several cases
<code>while</code>	Repeat statements indefinitely

Table A.6. *Basic data analysis.*

<code>max</code>	Largest component
<code>min</code>	Smallest component
<code>mean</code>	Average or mean value
<code>std</code>	Standard deviation
<code>sum</code>	Sum of elements
<code>prod</code>	Product of elements
<code>sort</code>	Sort elements

Table A.7. *Graphics.*

<code>plot</code>	<i>x-y</i> plot
<code>fplot</code>	Function plotter
<code>semilogy</code>	Plot with logarithmically scaled <i>y</i> -axis
<code>bar</code>	Bar graph
<code>histogram</code>	Histogram
<code>axis</code>	Axis control
<code>xlim</code>	Set <i>x</i> -axis limits
<code>ylim</code>	Set <i>y</i> -axis limits
<code>grid</code>	Grid lines
<code>xlabel</code>	Label <i>x</i> -axis
<code>ylabel</code>	Label <i>y</i> -axis
<code>title</code>	Title graph
<code>legend</code>	Display legend
<code>text</code>	Text annotation
<code>subplot</code>	Create axes in grid pattern
<code>hold</code>	Hold current graph
<code>contour</code>	Contour plot
<code>mesh</code>	Wireframe surface
<code>surf</code>	Solid surface
<code>colormap</code>	Set color map
<code>spy</code>	Visualize sparsity pattern
<code>print</code>	Print figure
<code>clf</code>	Clear current figure
<code>shg</code>	Make current figure visible
<code>close</code>	Close figure

Table A.8. *Linear algebra.*

<code>norm</code>	Norm of vector or matrix
<code>cond</code>	Condition number of matrix (with respect to inversion)
<code>\</code>	Solve linear system of equations
<code>eig</code>	Eigenvalues and eigenvectors
<code>lu</code>	LU factorization
<code>qr</code>	QR factorization
<code>svd</code>	Singular value decomposition

Table A.9. *Functions connected with program files.*

<code>edit</code>	Invoke MATLAB editor
<code>lookfor</code>	Search H1 line (first comment line) of all program files for keyword
<code>nargin</code>	Number of function input arguments
<code>nargout</code>	Number of function output arguments
<code>type</code>	List file in Command Window
<code>which</code>	Display full pathname of MATLAB file

Table A.10. *Miscellaneous.*

<code>clc</code>	Clear Command Window
<code>demo</code>	Demonstrations
<code>diary</code>	Save Command Window text to file
<code>dir</code>	Display directory listing
<code>doc</code>	Display HTML documentation in Help browser
<code>help</code>	Display help in Command Window
<code>tic, toc</code>	Start/stop stopwatch timer
<code>what</code>	List MATLAB files in current directory grouped by type

Table A.11. *Data types and conversions.*

<code>double</code>	Convert to double precision
<code>char</code>	Create or convert to character array (string)
<code>cell</code>	Create cell array
<code>num2str</code>	Convert number to string
<code>sparse</code>	Create sparse matrix
<code>struct</code>	Create or convert to structure array

Table A.12. *Managing the workspace.*

<code>clear</code>	Clear items from workspace
<code>who, whos</code>	List variables in workspace
<code>load</code>	Load workspace variables from disk
<code>save</code>	Save workspace variables to disk
<code>exit, quit</code>	Terminate MATLAB session

Table A.13. *Input and output.*

<code>disp</code>	Display text or array
<code>format</code>	Set output format
<code>fprintf</code>	Write formatted data to screen or file
<code>sprintf</code>	Write formatted data to string
<code>input</code>	Prompt for user input

Table A.14. *Numerical methods.*

<code>bvp4c</code>	Solve two-point boundary-value problem
<code>fft</code>	Discrete Fourier transform
<code>fminbnd</code>	Minimize function of one variable on fixed interval
<code>fzero</code>	Find zero of function of one variable
<code>integral</code>	Numerical integration
<code>interp1</code>	One-dimensional interpolation (several methods)
<code>ode45</code>	Explicit Runge–Kutta pair for nonstiff differential equations
<code>polyfit</code>	Least-squares polynomial fit
<code>roots</code>	Roots of polynomial
<code>spline</code>	Cubic spline interpolation

# Glossary

**Array Editor.** A tool allowing array contents to be viewed and edited in tabular format.

**class.** A class describes a set of objects with a common set of properties and the operations that can be performed on the data within the properties.

**Command History.** A tool that lists MATLAB commands previously typed in the current and past sessions and allows them to be copied or executed.

**Command Window.** The window in which the MATLAB prompt `>>` appears and in which commands are typed. It is part of the MATLAB desktop.

**Current Folder browser.** A browser for viewing program files and other files and performing operations on them.

**Editor/Debugger.** A tool for creating, editing, and debugging program files.

**FIG-file.** A file with a `.fig` extension that contains a representation of a figure that can be reloaded into MATLAB.

**figure.** A MATLAB window for displaying graphics.

**function.** A program file with a `.m` extension that can accept input arguments and return output arguments and whose variables are local to the function.

**Help browser.** A browser that allows you to view and search the documentation for MATLAB and other MathWorks products.

**IEEE arithmetic.** A standard for floating-point arithmetic [87], [88] to which the arithmetic in MATLAB conforms.

**LAPACK.** A Fortran 77 library of programs for linear equation, least-squares, eigenvalue, and singular value computations [3]. Many of the MATLAB linear algebra functions are based on LAPACK.

**Live Editor.** An interactive environment for editing and running MATLAB code that allows you to see your results together with the code that produced them.

**live script.** A program file with a `.mlx` extension that contains MATLAB commands and the output they produce. It is created and edited in the Live Editor.

**MAT-file.** A file with a `.mat` extension that contains MATLAB variables. Created and accessed with the `save` and `load` commands.

**MATLAB desktop.** A user interface for managing files, tools, and applications associated with MATLAB.

**MATLAB Toolstrip.** The strip or ribbon of tools, organized in several tabs, that appears at the top of the desktop. The tabs can be opened up or collapsed by clicking the icon containing a triangle located at the top right-hand corner of the desktop.

**MATLAB Web browser.** A web browser that is part of the MATLAB system. Used for displaying `profile` reports, for example.

**MEX-file.** A subroutine produced from C, C++, or Fortran code whose name has a platform-specific extension. It behaves like a program file or built-in function.

**program file.** A file that contains a sequence of MATLAB commands. It is of one of four types: a function, a script, a live script, or a file defining a class.

**script.** A program file with a `.m` extension that takes no input or output arguments and operates on data in the workspace.

**toolbox.** A collection of program files that extends the capabilities of MATLAB, usually in a particular application area.

**Workspace browser.** A browser that allows the contents of the workspace to be viewed and managed.



# Bibliography

- [1] Forman S. Acton. *Numerical Methods That Work*. Harper and Row, New York, 1970. xviii+541 pp. Reprinted by Mathematical Association of America, Washington, D.C., with new preface and additional problems, 1990. ISBN 0-88385-450-3. (Cited on pp. 158, 187.)
- [2] Yair Altman. *Accelerating MATLAB Performance. 1101 Tips to Speed up MATLAB Programs*. CRC Press, Boca Raton, FL, USA, 2015. xxv+743 pp. ISBN 978-1-4822-1129-0. (Cited on p. 369.)
- [3] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, and D. C. Sorensen. *LA-PACK Users' Guide*. Third edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. xxvi+407 pp. ISBN 0-89871-447-8. (Cited on pp. 135, 445.)
- [4] Mary Aprahamian and Nicholas J. Higham. Matrix inverse trigonometric and inverse hyperbolic functions: Theory and algorithms. *SIAM J. Matrix Anal. Appl.*, 37(4): 1453–1477, 2016. (Cited on p. 269.)
- [5] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. xvii+314 pp. ISBN 0-89871-412-5. (Cited on p. 210.)
- [6] Russell Ash. *The Top 10 of Everything*. Dorland Kindersley, London, 1994. 288 pp. ISBN 0-7513-0137-X. (Cited on p. 289.)
- [7] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. Second edition, Wiley, New York, 1989. xvi+693 pp. ISBN 0-471-50023-2. (Cited on pp. 175, 191, 192.)
- [8] Zhaojun Bai, James W. Demmel, Jack J. Dongarra, Axel Ruhe, and Henk A. Van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. xxix+410 pp. ISBN 0-89871-471-0. (Cited on p. 135.)
- [9] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1994. xiii+112 pp. ISBN 0-89871-328-5. (Cited on p. 154.)
- [10] Jon L. Bentley. *Programming Pearls*. Addison-Wesley, Reading, MA, USA, 1986. viii+195 pp. ISBN 0-201-10331-1. (Cited on p. 243.)
- [11] Jon L. Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, Reading, MA, USA, 1988. viii+207 pp. ISBN 0-201-11889-0. (Cited on p. 133.)
- [12] David Borland and Russell M. Taylor II. Rainbow color map (still) considered harmful. *IEEE Computer Graphics and Applications*, 27(2):14–17, 2007. (Cited on p. 119.)
- [13] Folkmar Bornemann, Dirk Laurie, Stan Wagon, and Jörg Waldvogel. *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004. xi+306 pp. ISBN 0-89871-561-X. (Cited on p. 419.)

- [14] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996. x+256 pp. Corrected republication of work first published in 1989 by North-Holland, New York. ISBN 0-89871-353-6. (Cited on p. 210.)
- [15] David S. Broomhead. Applications of max-plus algebra. In [83], pages 795–800. (Cited on p. 315.)
- [16] A. Buchheim. On the theory of matrices. *Proc. London Math. Soc.*, 16:63–82, 1884. (Cited on p. 140.)
- [17] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009. xii+277 pp. ISBN 978-0-898716-68-9. (Cited on p. 185.)
- [18] James W. Cooley. How the FFT gained acceptance. In *A History of Scientific Computing*, Stephen G. Nash, editor, Addison-Wesley, Reading, MA, USA, 1990, pages 133–140. (Cited on p. 186.)
- [19] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19(90):297–301, 1965. (Cited on p. 186.)
- [20] Robert M. Corless. *Essential Maple 7: An Introduction for Scientific Programmers*. Springer-Verlag, New York, 2002. xv+282 pp. ISBN 0-387-95352-3. (Cited on p. 348.)
- [21] Robert M. Corless and Nicolas Fillion. *A Graduate Introduction to Numerical Methods From the Viewpoint of Backward Error Analysis*. Springer-Verlag, London, 2013. xxxix+868 pp. ISBN 978-1-4614-8452-3. (Cited on p. 175.)
- [22] Tony Crilly. The appearance of set operators in Cayley’s group theory. *Notices of the South African Mathematical Society*, 31:9–22, 2000. (Cited on p. 140.)
- [23] Germund Dahlquist and Åke Björck. *Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1974. xviii+573 pp. Translated by Ned Anderson. ISBN 0-13-627315-7. (Cited on pp. 175, 191.)
- [24] Harold T. Davis. *Introduction to Nonlinear Differential and Integral Equations*. Dover, New York, 1962. xv+566 pp. ISBN 0-486-60971-5. (Cited on p. 201.)
- [25] Timothy A. Davis. SuiteSparse: A suite of sparse matrix software. <http://faculty.cse.tamu.edu/davis/suitesparse.html>. (Cited on p. 252.)
- [26] Timothy A. Davis. Algorithm 832: UMFPACK V4.3—An unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software*, 30(2):196–199, 2004. (Cited on p. 252.)
- [27] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006. xii+217 pp. ISBN 0-89871-613-6. (Cited on p. 249.)
- [28] Timothy A. Davis. Algorithm 930: FACTORIZE: An object-oriented linear system solver for MATLAB. *ACM Trans. Math. Software*, 39(4):28:1–28:18, 2013. (Cited on pp. 158, 324.)
- [29] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016. (Cited on p. 249.)
- [30] Edvin Deadman and Nicholas J. Higham. Testing matrix function algorithms using identities. *ACM Trans. Math. Software*, 42(1):4:1–4:15, 2016. (Cited on p. 269.)
- [31] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. xi+419 pp. ISBN 0-89871-389-7. (Cited on p. 135.)

- [32] James W. Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Richard Vuduc, R. Clint Whaley, and Katherine Yellick. Self-adapting linear algebra algorithms and software. *Proc. IEEE*, 93(2):293–312, 2005. (Cited on p. 186.)
- [33] Edsger W. Dijkstra. On the cruelty of really teaching computing science. <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1036.html>, December 1998. (Cited on p. 272.)
- [34] Nicholas J. Dingle and Nicholas J. Higham. Reducing the influence of tiny normwise relative errors on performance profiles. *ACM Trans. Math. Software*, 39(4):24:1–24:11, 2013. (Cited on p. 416.)
- [35] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Programming*, 91:201–213, 2002. (Cited on pp. 410, 437.)
- [36] Jack J. Dongarra and Francis Sullivan. Introduction to the top 10 algorithms. *Computing in Science and Engineering*, 2(1):22–23, 2000. (Cited on p. 187.)
- [37] David L. Donoho and Victoria Stodden. Reproducible research in the mathematical sciences. In [83], pages 916–925. (Cited on p. 49.)
- [38] Tobin A. Driscoll, Nicholas Hale, and Lloyd N. Trefethen. *Chebfun Guide*. Pafnuty Publications, Oxford, 2014. (Cited on pp. 315, 324.)
- [39] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986. xiii+341 pp. ISBN 0-19-853408-6. (Cited on p. 255.)
- [40] Iain S. Duff. MA57—A code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Software*, 30(2):118–144, 2004. (Cited on p. 252.)
- [41] Steven L. Eddins. Automated software testing for MATLAB. *Computing in Science and Engineering*, 11(6):48–54, 2009. (Cited on p. 272.)
- [42] Steven L. Eddins. Rainbow color map critiques: An overview and annotated bibliography. <http://mathworks.com/company/newsletters/articles/rainbow-color-map-critiques-an-overview-and-annotated-bibliography.html>, 2014. (Cited on p. 119.)
- [43] Alan Edelman. Eigenvalue roulette and random test matrices. In *Linear Algebra for Large Scale and Real-Time Applications*, Marc S. Moonen, Gene H. Golub, and Bart L. De Moor, editors, volume 232 of *NATO ASI Series E*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993, pages 365–368. (Cited on pp. 83, 84.)
- [44] Alan Edelman, Eric Kostlan, and Michael Shub. How many eigenvalues of a random matrix are real? *J. Amer. Math. Soc.*, 7(1):247–267, 1994. (Cited on p. 84.)
- [45] Mark Embree and Lloyd N. Trefethen. Growth and decay of random Fibonacci sequences. *Proc. Roy. Soc. London Ser. A*, 455:2471–2485, 1999. (Cited on p. 8.)
- [46] Bengt Fornberg. *A Practical Guide to Pseudospectral Methods*. Cambridge University Press, Cambridge, UK, 1995. x+231 pp. ISBN 0-521-49582-2. (Cited on p. 282.)
- [47] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1977. xi+259 pp. ISBN 0-13-165332-6. (Cited on pp. 175, 184.)
- [48] Linton C. Freeman. Going the wrong way down a one-way street: Centrality in physics and biology. *J. Social Structure*, 9, 2008. (Cited on p. 359.)
- [49] Matteo Frigo and Steven G. Johnson. FFTW. <http://www.fftw.org/>. (Cited on p. 186.)
- [50] C. W. Gear and R. D. Skeel. The development of ODE methods: A symbiosis between hardware and numerical analysis. In *A History of Scientific Computing*, Stephen G. Nash, editor, Addison-Wesley, Reading, MA, USA, 1990, pages 88–105. (Cited on p. 213.)

- [51] Stuart Geman. The spectral radius of large random matrices. *Ann. Probab.*, 14(4): 1318–1328, 1986. (Cited on p. 390.)
- [52] John R. Gilbert, Cleve B. Moler, and Robert S. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992. (Cited on p. 255.)
- [53] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Fourth edition, Johns Hopkins University Press, Baltimore, MD, USA, 2013. xxi+756 pp. ISBN 978-1-4214-0794-4. (Cited on pp. 57, 135, 146, 322.)
- [54] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. xiii+220 pp. ISBN 0-89871-396-X. (Cited on p. 154.)
- [55] David F. Griffiths, John W. Dold, and David J. Silvester. *Essential Partial Differential Equations: Analytical and Computational Aspects*. Springer-Verlag, London, 2015. xi+368 pp. ISBN 978-3-319-22568-5. (Cited on p. 175.)
- [56] David F. Griffiths and Desmond J. Higham. *Numerical Methods for Ordinary Differential Equations*. Springer-Verlag, London, 2010. x+271 pp. ISBN 978-0-85729-147-9. (Cited on pp. 175, 195.)
- [57] David F. Griffiths and Desmond J. Higham. *Learning L<sup>A</sup>T<sub>E</sub>X*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2016. x+103 pp. ISBN 978-1-611974-41-6. (Cited on pp. 106, 109, 130.)
- [58] Peter Grindrod. *Mathematical Underpinnings of Analytics. Theory and Applications*. Oxford University Press, New York, 2015. xiii+261 pp. ISBN 978-0-19-872509-1. (Cited on p. 367.)
- [59] Ned Gulley. In praise of tweaking: A wiki-like programming contest. *Interactions*, 11(3):18–23, 2004. (Cited on p. 386.)
- [60] E. Hairer and G. Wanner. *Analysis by Its History*. Springer-Verlag, New York, 1996. x+374 pp. ISBN 0-387-94551-2. (Cited on p. 191.)
- [61] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Second edition, Springer-Verlag, Berlin, 1996. xv+614 pp. ISBN 3-540-60452-9. (Cited on pp. 205, 231.)
- [62] Leonard Montague Harrod, editor. *Indexers on Indexing: A Selection of Articles Published in The Indexer*. R. K. Bowker, London, 1978. x+430 pp. ISBN 0-8352-1099-5. (Cited on p. 452.)
- [63] Bernd Heidergott, Geert Jan Olsder, and Jacob van der Woude. *Max Plus at Work. Modeling and Analysis of Synchronized Systems: A Course on Max-Plus Algebra and Its Applications*. Princeton University Press, Princeton, NJ, USA, 2006. xi+213 pp. ISBN 978-0-691-11763-8. (Cited on p. 315.)
- [64] Piet Hein. *Grooks*. Number 85 in *Borgens Pocketbooks*. Second edition, Borgens Forlag, Copenhagen, Denmark, 1992. 53 pp. First published in 1966. ISBN 87-418-1079-1. (Cited on p. 248.)
- [65] Kurt Hensel. Über den Zusammenhang zwischen den Systemen und ihren Determinanten. *J. Reine Angew. Math.*, 159(4):246–254, 1928. (Cited on p. 140.)
- [66] Desmond J. Higham. Nine ways to implement the binomial method for option valuation in MATLAB. *SIAM Rev.*, 44(4):661–677, 2002. (Cited on p. 375.)
- [67] Desmond J. Higham. *An Introduction to Financial Option Valuation: Mathematics, Stochastics and Computation*. Cambridge University Press, Cambridge, UK, 2004. xxi+273 pp. ISBN 0-521-83884-3. (Cited on pp. 133, 424.)
- [68] Nicholas J. Higham. Algorithm 694: A collection of test matrices in MATLAB. *ACM Trans. Math. Software*, 17(3):289–305, 1991. (Cited on p. 51.)

- [69] Nicholas J. Higham. The Test Matrix Toolbox for MATLAB (version 3.0). Numerical Analysis Report No. 276, Manchester Centre for Computational Mathematics, Manchester, England, September 1995. 70 pp. (Cited on p. 51.)
- [70] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. xxx+680 pp. ISBN 0-89871-521-0. (Cited on pp. 39, 42, 55, 144, 282.)
- [71] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008. xx+425 pp. ISBN 978-0-898716-46-7. (Cited on p. 158.)
- [72] Nicholas J. Higham. Sylvester's influence on applied mathematics. *Mathematics Today*, 50(4):202–206, 2014. (Cited on p. 358.)
- [73] Nicholas J. Higham. Color spaces and digital imaging. In [83], pages 808–813. (Cited on p. 99.)
- [74] Nicholas J. Higham. Functions of matrices. In [83], pages 97–99. (Cited on p. 158.)
- [75] Nicholas J. Higham. Numerical linear algebra and matrix analysis. In [83], pages 263–281. (Cited on p. 135.)
- [76] Nicholas J. Higham. Programming languages: An applied mathematics view. In [83], pages 828–839. (Cited on p. 259.)
- [77] Nicholas J. Higham. The singular value decomposition. In [83], pages 126–127. (Cited on p. 146.)
- [78] Nicholas J. Higham. Iterating MATLAB commands. <https://nickhigham.wordpress.com/2016/05/13/iterating-matlab-commands>, May 2016. (Cited on p. 94.)
- [79] Nicholas J. Higham. The one-line maze program in MATLAB. <https://nickhigham.wordpress.com/2016/06/29/the-one-line-maze-program-in-matlab>, June 2016. (Cited on p. 295.)
- [80] Nicholas J. Higham. The top 10 algorithms in applied mathematics. <https://nickhigham.wordpress.com/2016/03/29/the-top-10-algorithms-in-applied-mathematics>, March 2016. (Cited on p. 187.)
- [81] Nicholas J. Higham and Awad H. Al-Mohy. Computing matrix functions. *Acta Numerica*, 19:159–208, 2010. (Cited on p. 158.)
- [82] Nicholas J. Higham and Edwin Deadman. A catalogue of software for matrix functions. Version 2.0. MIMS EPrint 2016.3, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, January 2016. 22 pp. Updated March 2016. (Cited on p. 158.)
- [83] Nicholas J. Higham, Mark R. Dennis, Paul Glendinning, Paul A. Martin, Fadil Santosa, and Jared Tanner, editors. *The Princeton Companion to Applied Mathematics*. Princeton University Press, Princeton, NJ, USA, 2015. xvii + 994 + 16 color plates pp. ISBN 978-0-691-15039-0. (Cited on pp. 187, 448, 449, 451.)
- [84] Nicholas J. Higham and Françoise Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. *SIAM J. Matrix Anal. Appl.*, 21(4):1185–1201, 2000. (Cited on p. 137.)
- [85] Francis B. Hildebrand. *Advanced Calculus for Applications*. Second edition, Prentice-Hall, Englewood Cliffs, NJ, USA, 1976. xiii+733 pp. ISBN 0-13-011189-9. (Cited on p. 217.)
- [86] Doug Hoyte. *Let Over Lambda. 50 Years of Lisp*. <http://letoverlambda.com>, 2008. iv+376 pp. ISBN 978-1-4357-1275-1. (Cited on p. 174.)

- [87] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987. (Cited on pp. 39, 445.)
- [88] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985)*. IEEE Computer Society, New York, 2008. 58 pp. ISBN 978-0-7381-5752-8. (Cited on pp. 39, 445.)
- [89] D. S. Jones and B. D. Sleeman. *Differential Equations and Mathematical Biology*. CRC Press, Boca Raton, FL, USA, 2003. 408 pp. ISBN 1-58488-296-4. (Cited on pp. 227, 229.)
- [90] William M. Kahan. Handheld calculator evaluates integrals. *Hewlett-Packard Journal*, 31(8):23–32, 1980. (Cited on p. 231.)
- [91] David K. Kahaner, Cleve B. Moler, and Stephen G. Nash. *Numerical Methods and Software*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1989. xii+495 pp. ISBN 0-13-627258-4. (Cited on p. 175.)
- [92] Irving Kaplansky. Reminiscences. In *Paul Halmos: Celebrating 50 Years of Mathematics*, John H. Ewing and F. W. Gehring, editors, Springer-Verlag, Berlin, 1991, pages 87–89. (Cited on p. 158.)
- [93] Roger Emanuel Kaufman. *A FORTRAN Coloring Book*. The MIT Press, Cambridge, MA, USA, 1978. ISBN 0-262-61026-4. (Cited on pp. 82, 174, 240.)
- [94] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995. xiii+165 pp. ISBN 0-89871-352-8. (Cited on p. 154.)
- [95] C. T. Kelley. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. xv+180 pp. ISBN 0-89871-433-8. (Cited on p. 185.)
- [96] C. T. Kelley. *Solving Nonlinear Equations with Newton's Method*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003. xiii+104 pp. ISBN 0-89871-546-6. (Cited on p. 181.)
- [97] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. Second edition, McGraw-Hill, New York, 1978. xii+168 pp. ISBN 0-07-034207-5. (Cited on pp. 95, 174, 240, 248, 272, 378.)
- [98] Jacek Kierzenka. Tutorial on solving BVPs with BVP4C, 2016. <https://mathworks.com/matlabcentral/fileexchange/3819-tutorial-on-solving-bvps-with-bvp4c>. (Cited on pp. 220, 455.)
- [99] Jacek Kierzenka. Tutorial on solving DDEs with DDE23, 2016. <http://mathworks.com/matlabcentral/fileexchange/3899-tutorial-on-solving-ddes-with-dde23>. (Cited on p. 223.)
- [100] Jacek A. Kierzenka and Lawrence F. Shampine. A BVP solver based on residual control and the MATLAB PSE. *ACM Trans. Math. Software*, 27(3):229–316, 2001. (Cited on p. 220.)
- [101] Peter E. Kloeden and Eckhard Platen. *Numerical Solution of Stochastic Differential Equations*. Springer-Verlag, Berlin, 1992. xxxv+632 pp. ISBN 3-540-54062-8. (Cited on p. 372.)
- [102] G. Norman Knight. Book indexing in Great Britain: A brief history. *The Indexer*, 6(1):14–18, 1968. Reprinted in [62, pp. 9–13]. (Cited on p. 459.)
- [103] Donald E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, 1974. Reprinted in [105]. (Cited on p. 272.)
- [104] Donald E. Knuth. *The T<sub>E</sub>Xbook*. Addison-Wesley, Reading, MA, USA, 1986. ix+483 pp. ISBN 0-201-13448-9. (Cited on p. 106.)

- [105] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Stanford University, Stanford, CA, USA, 1992. xv+368 pp. ISBN 0-9370-7380-6. (Cited on p. 452.)
- [106] Donald E. Knuth. *Digital Typography*. CSLI Lecture Notes Number 78. Center for the Study of Language and Information, Stanford University, Stanford, CA, USA, 1999. xv+685 pp. ISBN 0-57586-010-4. (Cited on p. 134.)
- [107] Helmut Kopka and Patrick W. Daly. *Guide to L<sup>A</sup>T<sub>E</sub>X*. Fourth edition, Addison-Wesley, Boston, MA, USA, 2004. xii+597 pp. ISBN 0-321-17385-6. (Cited on pp. 106, 109, 130.)
- [108] Arnold R. Krommer and Christoph W. Ueberhuber. *Computational Integration*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. xix+445 pp. ISBN 0-89871-374-9. (Cited on p. 348.)
- [109] Peter Kunkel and Volker Mehrmann. *Differential-Algebraic Equations: Analysis and Numerical Solution*. European Mathematical Society, Zurich, Switzerland, 2006. viii+377 pp. ISBN 3-03719-017-5. (Cited on p. 210.)
- [110] Peter Laffin. Leeds' role in the data revolution. <http://www.leedsdatathing.co.uk/data-in-a-day/data-in-a-day-peter-laflin-on-leeds-role-in-the-data-revolution>, April 2013. (Cited on p. 367.)
- [111] Jeffrey C. Lagarias. The  $3x+1$  problem and its generalizations. *Amer. Math. Monthly*, 92(1):3–23, 1985. (Cited on p. 10.)
- [112] Leslie Lamport. *L<sup>A</sup>T<sub>E</sub>X: A Document Preparation System. User's Guide and Reference Manual*. Second edition, Addison-Wesley, Reading, MA, USA, 1994. xvi+272 pp. ISBN 0-201-52983-1. (Cited on pp. 106, 109, 130.)
- [113] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. xv+142 pp. ISBN 0-89871-407-9. (Cited on p. 155.)
- [114] F. M. Leslie. Liquid crystal devices. Technical report, Institute Wiskundige Dienstverlening, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1992. (Cited on p. 216.)
- [115] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007. xv+341 pp. ISBN 978-0-898716-29-0. (Cited on p. 175.)
- [116] Shangzhi Li, Falai Chen, Yaohua Wu, and Yunhua Zhang. *Mathematics Experiments*. World Scientific, New Jersey, USA, 2003. ix+217 pp. ISBN 978-981-238-049-4. (Cited on p. 404.)
- [117] J. N. Lyness and J. J. Kaganove. Comments on the nature of automatic quadrature routines. *ACM Trans. Math. Software*, 2(1):65–81, 1976. (Cited on p. 410.)
- [118] Tom Marchioro. Putting math to work: An interview with Cleve Moler. *Computing in Science and Engineering*, 1(4):10–13, 1999. (Cited on p. 37.)
- [119] Annik Martin and Shigui Ruan. Predator-prey models with delay and prey harvesting. *J. Math. Biol.*, 43:247–267, 2001. (Cited on p. 221.)
- [120] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Engrg.*, SE-2(4):308–320, 1976. (Cited on p. 259.)
- [121] Cleve B. Moler. Demonstration of a matrix laboratory. In *Numerical Analysis, Mexico 1981*, J. P. Hennart, editor, volume 909 of *Lecture Notes in Mathematics*, Springer-Verlag, Berlin, 1982, pages 84–98. (Cited on p. 37.)

- [122] Cleve B. Moler. MATLAB users' guide. Technical Report CS81-1 (revised), Department of Computer Science, University of New Mexico, Albuquerque, New Mexico, August 1982. 60 pp. (Cited on pp. 33, 437.)
- [123] Cleve B. Moler. Yet another look at the FFT. *The MathWorks Newsletter*, 1992. (Cited on p. 102.)
- [124] Cleve B. Moler. MATLAB's magical mystery tour. *The MathWorks Newsletter*, 7(1): 8–9, 1993. (Cited on p. 51.)
- [125] Cleve B. Moler. Objectively speaking. OOPS is not an apology. *MATLAB News and Notes*, pages 6–7, 1999. (Cited on p. 324.)
- [126] Cleve B. Moler. Parallel MATLAB: Multiple processors and multiple cores. *The MathWorks News and Notes*, pages 26–28, 2007. (Cited on p. 401.)
- [127] Cleve B. Moler. Backslash. <http://blogs.mathworks.com/cleve/2013/08/19/backslash>, August 2013. (Cited on p. 140.)
- [128] Cleve B. Moler. Modernization of numerical integration, from quad to integral. <http://blogs.mathworks.com/cleve/2016/05/23/modernization-of-numerical-integration-from-quad-to-integral>, May 2016. (Cited on p. 191.)
- [129] Cleve B. Moler and Donald Morrison. Replacing square roots by Pythagorean sums. *IBM J. Res. Develop.*, 27(6):577–581, 1983. (Cited on p. 430.)
- [130] Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter. `10 PRINT CHR$(205.5+RND(1)); : GOTO 10`. The MIT Press, Cambridge, MA, USA, 2013. xi+309 pp. ISBN 978-0-262-01846-3. (Cited on p. 295.)
- [131] J. D. Murray. *Mathematical Biology I. An Introduction*. Springer-Verlag, Berlin, 2002. xxiii+551 pp. ISBN 0-387-95223-3. (Cited on p. 434.)
- [132] Noël M. Nachtigal, Satish C. Reddy, and Lloyd N. Trefethen. How fast are nonsymmetric matrix iterations? *SIAM J. Matrix Anal. Appl.*, 13(3):778–795, 1992. (Cited on p. 437.)
- [133] Salih N. Neftci. *An Introduction to the Mathematics of Financial Derivatives*. Second edition, Academic Press, San Diego, CA, USA, 2000. xxvii+527 pp. ISBN 0-12-515392-9. (Cited on p. 226.)
- [134] Richard D. Neidinger. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Rev.*, 52(3):545–563, 2010. (Cited on p. 324.)
- [135] M. E. J. Newman, C. Moore, and D. J. Watts. Mean-field solution of the small-world network model. *Physical Review Letters*, 84:3201–3204, 2000. (Cited on p. 406.)
- [136] J. R. Norris. *Markov Chains*. Cambridge University Press, Cambridge, UK, 1997. ISBN 0-521-48181-3. (Cited on p. 427.)
- [137] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic: Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001. xiv+104 pp. ISBN 0-89871-482-6. (Cited on p. 39.)
- [138] Karen Hunger Parshall. *James Joseph Sylvester. Life and Work in Letters*. Oxford University Press, 1998. xv+321 pp. ISBN 0-19-850391-1. (Cited on p. 140.)
- [139] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Fractals for the Classroom. Part One: Introduction to Fractals and Chaos*. Springer-Verlag, New York, 1992. xiv+450 pp. ISBN 0-387-97041-X. (Cited on pp. 12, 18, 119, 170.)
- [140] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Fractals for the Classroom. Part Two: Complex Systems and Mandelbrot Set*. Springer-Verlag, New York, 1992. xii+500 pp. ISBN 0-387-97722-8. (Cited on p. 12.)



- [141] E. Pitts. The stability of pendent liquid drops. Part 1. Drops formed in a narrow gap. *J. Fluid Mech.*, 59(4):753–767, 1973. (Cited on p. 214.)
- [142] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Second edition, Cambridge University Press, Cambridge, UK, 1992. xxvi+963 pp. ISBN 0-521-43064-X. (Cited on p. 185.)
- [143] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, 1990. ISBN 0387-97297-8. (Cited on p. 421.)
- [144] *A Million Random Digits with 100,000 Normal Deviates*. RAND, Santa Monica, CA, USA, 2001. Reprint of work originally published in 1955 by The Free Press, Glencoe, Illinois. ISBN 978-0833030474. (Cited on p. 7.)
- [145] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003. xviii+528 pp. ISBN 0-89871-534-2. (Cited on p. 154.)
- [146] Robert Sedgewick. *Algorithms*. Second edition, Addison-Wesley, Reading, MA, USA, 1988. xii+657 pp. ISBN 0-201-06673-4. (Cited on p. 314.)
- [147] Peter Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, Oxford, 2009. xiii+617 pp. ISBN 978-1-4302-1948-4. (Cited on p. 272.)
- [148] Lawrence F. Shampine. *Numerical Solution of Ordinary Differential Equations*. Chapman and Hall, New York, 1994. x+484 pp. ISBN 0-412-05151-6. (Cited on pp. 195, 197, 205, 375.)
- [149] Lawrence F. Shampine. Solving  $0 = F(t, y(t), y'(t))$  in MATLAB. *Journal of Numerical Mathematics*, 10(4):291–310, 2002. (Cited on p. 213.)
- [150] Lawrence F. Shampine. Singular boundary value problems for ODEs. *Appl. Math. Comput.*, 138(1):99–112, 2003. (Cited on p. 220.)
- [151] Lawrence F. Shampine, Richard C. Allen, Jr., and Steven Pruess. *Fundamentals of Numerical Computing*. Wiley, New York, 1997. x+268 pp. ISBN 0-471-16363-5. (Cited on pp. 175, 191, 192.)
- [152] Lawrence F. Shampine, Ian Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge University Press, Cambridge, UK, 2003. viii+263 pp. ISBN 0-521-53094-6. (Cited on p. 223.)
- [153] Lawrence F. Shampine, Jacek A. Kierzenka, and Mark W. Reichelt. Solving boundary value problems for ordinary differential equations in MATLAB with `bvp4c`, 2000. In [98]. 27 pp. (Cited on p. 231.)
- [154] Lawrence F. Shampine and Mark W. Reichelt. The MATLAB ODE suite. *SIAM J. Sci. Comput.*, 18(1):1–22, 1997. (Cited on pp. 207, 231.)
- [155] Lawrence F. Shampine and S. Thompson. Solving DDEs in MATLAB. *Appl. Numer. Math.*, 37:441–458, 2001. (Cited on p. 223.)
- [156] Gaurav Sharma and Jos Martin. MATLAB<sup>®</sup>: A language for parallel computing. *Int. J. Parallel Prog.*, 37(1):3–36, 2009. (Cited on p. 401.)
- [157] G. W. Stewart. *Matrix Algorithms. Volume I: Basic Decompositions*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. xx+458 pp. ISBN 0-89871-414-1. (Cited on p. 135.)
- [158] G. W. Stewart. *Matrix Algorithms. Volume II: Eigensystems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001. xix+469 pp. ISBN 0-89871-503-2. (Cited on p. 135.)
- [159] Josef Stoer and Christoph Witzgall. *Convexity and Optimization in Finite Dimensions I*. Springer-Verlag, Berlin, 1970. ix+293 pp. (Cited on p. 70.)

- [160] Gilbert Strang. *Introduction to Linear Algebra*. Third edition, Wellesley-Cambridge Press, Wellesley, MA, USA, 2003. viii+568 pp. ISBN 0-9614088-9-8. (Cited on p. 135.)
- [161] Steven H. Strogatz. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering*. Addison-Wesley, Reading, MA, USA, 1994. xi+498 pp. ISBN 0-201-54344-3. (Cited on pp. 12, 195, 197, 199.)
- [162] J. J. Sylvester. Chemistry and algebra. *Nature*, 17:284, 1878. (Cited on p. 358.)
- [163] Alan Taylor and Desmond J. Higham. CONTEST: A controllable test matrix toolbox for MATLAB. *ACM Trans. Math. Software*, 35(4):26:1–26:17, 2009. (Cited on p. 350.)
- [164] Test set for IVP solvers, release 2.4. <http://pitagora.dm.uniba.it/~testset>. (Cited on pp. 210, 211.)
- [165] Lloyd N. Trefethen. *Spectral Methods in MATLAB*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. xvi+165 pp. ISBN 0-89871-465-6. (Cited on pp. xxvi, 133, 289.)
- [166] Lloyd N. Trefethen. *Approximation Theory and Approximation Practice*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2013. viii+305 pp. ISBN 978-1-611972-39-9. (Cited on pp. 105, 133, 265.)
- [167] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. xii+361 pp. ISBN 0-89871-361-7. (Cited on p. 135.)
- [168] Lloyd N. Trefethen and J. A. C. Weideman. The exponentially convergent trapezoidal rule. *SIAM Rev.*, 56(3):385–458, 2014. (Cited on p. 192.)
- [169] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA, 1983. 197 pp. (Cited on pp. 133, 134, 289.)
- [170] Edward R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, USA, 1990. 126 pp. (Cited on p. 133.)
- [171] Edward R. Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, CT, USA, 1997. 158 pp. ISBN 0-9613921-2-6. (Cited on p. 133.)
- [172] Charles F. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992. xiii+273 pp. ISBN 0-89871-285-8. (Cited on p. 187.)
- [173] Charles F. Van Loan. Using examples to build computational intuition. *SIAM News*, 28:1, 7, 1995. (Cited on p. xxvi.)
- [174] Charles F. Van Loan. *Introduction to Scientific Computing: A Matrix-Vector Approach Using MATLAB*. Prentice-Hall, Englewood Cliffs, NJ, USA, 2000. xi+367 pp. ISBN 0-13-949157-0. (Cited on p. 175.)
- [175] D. Viswanath. Random Fibonacci sequences and the number 1.3198824... *Math. Comp.*, 69(231):1131–1155, 2000. (Cited on p. 8.)
- [176] Stan Wagon. *Mathematica in Action*. Second edition, TELOS division of Springer-Verlag, New York, NY, USA, 2000. xvi+592 pp. ISBN 0-387-98684-7. (Cited on pp. 332, 337.)
- [177] A. J. Wathen. Preconditioning. *Acta Numerica*, 24:329–376, 2015. (Cited on p. 154.)
- [178] David S. Watkins. *Fundamentals of Matrix Computations*. Third edition, Wiley, New York, 2010. xvi+644 pp. ISBN 978-0-470-52833-4. (Cited on p. 135.)
- [179] Duncan J. Watts. *Small Worlds. The Dynamics of Networks between Order and Randomness*. Princeton University Press, Princeton, NJ, USA, 1999. xv+262 pp. ISBN 978-0-691-11704-7. (Cited on p. 359.)

- [180] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393:440–442, 1998. (Cited on pp. 406, 409.)
- [181] Junjie Wei and Shigui Ruan. Stability and bifurcation in a neural network model with two delays. *Physica D*, 130:255–272, 1999. (Cited on p. 223.)
- [182] Maurice V. Wilkes. *Memoirs of a Computer Pioneer*. The MIT Press, Cambridge, MA, USA, 1985. viii+240 pp. ISBN 0-262-23122-0. (Cited on p. 248.)
- [183] Paul Wilmott, Sam Howison, and Jeff Dewynne. *The Mathematics of Financial Derivatives: A Student Introduction*. Cambridge University Press, Cambridge, UK, 1995. xiii+317 pp. ISBN 0-521-49699-3. (Cited on pp. 226, 424.)

# Index

*The 18th century saw the advent of the professional indexer.  
He was usually of inferior status—a Grub Street hack—  
although well-read and occasionally a university graduate.*

— G. NORMAN KNIGHT, *Book Indexing in Great Britain: A Brief History* (1968)

*I find that a great part of the information I have was acquired by  
looking up something and finding something else on the way.*

— FRANKLIN P. ADAMS

A suffix “t” after a page number denotes a table, “f” a figure, “g” the glossary, “ℓ” a listing, and “n” a footnote. Entries in typewriter font are MATLAB functions, unless marked as objects or properties,

- ! (system command), 27
- (:), 56
- , (comma), 4, 6, 24, 50
- .\* (array multiplication), 58, 86
- ... (continuation), 26
- ./ (array right division), 58
- .\ (array left division), 58
- .^ (array exponentiation), 59
- .' (transpose), 59
- / (right division), 58
- : (colon), 48 t, 54–57, 440 t
- ; (semicolon), 1, 2, 4, 6, 23, 50
- <, 71
- <=, 71
- == (logical equal), 71, 293, 295
- >, 71
- >=, 71
- @ (function handle), 159
- [. .] (matrix building), 6, 50
- [] (empty matrix), 63, 68
- % (comment), 10, 24, 84
- %% (cell in program file), 265
- %{ (block comment), 91
- & (logical and), 74
- && (logical and, for scalars, with short-circuiting), 74
- \ (left division), *see* backslash (\)
- ^, 59
- | (logical or), 74
- || (logical or, for scalars, with short-circuiting), 74
- >> (prompt), 1, 23, 24
- ' (conjugate transpose), 59
- ' (string delimiter), 5
- '' , 79, 235
- ~ (discard output arguments), 90
- ~ (logical not), 74
- ~= (logical not equal), 71
- 2D plotting functions, 114 t, 441 t
- 3D plotting functions, 123 t
- 3x + 1 problem, 10
- aborting a computation, 26
- abs, 42 t
- AbsTol option, 189
- acos, 42 t, 269
- acosd, 42 t
- acosh, 42 t
- acot, 42 t
- acotd, 42 t
- acoth, 42 t
- acsc, 42 t
- acscd, 42 t
- acsch, 42 t

- addpath, 92
- addpoints, 281
- adjacency matrix, 349, 404
- airy, 42 t
- algebraic equations, *see* linear equations; nonlinear equations
- Algol 68, 57
- algorithms, top ten, 187 t, 187
- all, 74, 75, 440 t
- amd, 252
- and, 74
- angle, 42 t
- AnimatedLine object, 281
- animation, 279–281
- annotation, 109
- anonymous function, 160–161, 183, 185, 404
  - origin in Lisp, 161
- ans, 2, 4, 23, 439 t
- any, 74, 75, 383, 440 t
- App, 268
- area, 114 t, 128
- area graph, 128
- arguments
  - default values for, 163–165
  - variable number of, 165–166
- ARPACK, 155
- array
  - categorical, 299–300, 306–307
  - character, 292–295
  - codistributed, 397–398
  - distributed, 397–398
  - empty, 379–380
    - testing for, 73 t
  - generation, 4–7, 47–54
  - logical, 77–78
  - multidimensional, 297–298
  - order of storage, 372–374
  - preallocating, 374
  - subscripts start at 1, 54, 56, 247
  - tall, 364–366
- Array Editor, 31, 32 f, 445 g
- array operations
  - elementary, 59 t
  - elementwise, 3, 58, 86
- array2table, 308
- ASCII file
  - loading from, 31
  - saving to, 31
- asec, 42 t
- asecd, 42 t
- asech, 42 t
- aside
  - anonymous function and lambda expression, 161
  - backslash notation, 140
  - before pseudorandom number generators, 7
  - char maze, 295
  - chemical graphs, 358
  - classic MATLAB, 33
  - code complexity, 260
  - colon notation, 57
  - color maps, 119
  - color spaces, 99
  - fast Fourier transform, 186
  - iterating MATLAB commands, 94
  - reproducible research, 49
  - stiffness, 213
  - strings, 293
  - top ten algorithms, 187
  - zero-based versus one-based indexing, 56
- asin, 42 t
- asind, 42 t
- asinh, 42 t
- assert, 242–243, 269
- assertions, 242–243
- assume, 329, 330
- assumptions, 329
- atan, 42 t
- atan2, 42 t
- atand, 42 t
- atanh, 42 t
- attribute, **sparse**, 249
- Axes object, 273, 278
- axes, superimposing, 282
- axis, 16, 102–104, 441 t
  - options, 103 t
- backslash (\), 4, 58, 138–142, 144, 252, 338, 441 t, *see also* linsolve
- balance, 150
- balancing, 150
- bar, 114 t, 125, 441 t
- bar graph, 12, 125–126
- bar3, 123 t, 126
- bar3h, 123 t, 126
- barh, 114 t, 125

- batch, 393, 395
- batch computations, 393–395
- bench, 27 t
- bernstein, 342 t
- bessel, 42 t
- beta, 42 t
- Bezier curve, 109
- bfsearch, 353 t
- bicg, 154, 155 t
- bicgstab, 155 t
- bicgstabl, 155 t
- bifurcation diagram, 375
- binomial coefficient, *see* `nchoosek`
- Black–Scholes
  - delta surface, 422–425
  - PDE, 226–227
- blkdiag, 50
- block comment, 91
- boundary value problem (BVP) solver, 213–220
  - dealing with an unknown parameter, 217–219
  - example files, 220
  - input and output arguments, 215–216
- boundary-value problem (BVP), two-point, 213
- Box property, 276
- box, 101, 114, 276
- brachistochrone, 403
- break, 80
- breakpoint, 245
- Brownian path, 372
- bsxfun, 63
- bvp4c, 213–220, 372, 443 t
- bvp5c, 220
- bvpget, 220
- bvpinit, 216, 220
- bvpset, 216, 220, 372
- bvpval, 220
- C. elegans*, 355
- caldays, 303
- caldiff, 303
- calendarDuration, 303
- calendarDuration array, 303–304
- cardioid, 134
- cart2pol, 42 t
- cart2sph, 42 t
- case, 80, 440 t
- case sensitivity, 1, 24, 31
- cat, 298, 298 t, 312–313
- categorical, 299–300
- categorical array, 299–300, 306–307
- Cayley–Hamilton theorem, 176
- ccode, 347
- cd, 27
- ceil, 42 t
- cell, 308, 312, 374, 442 t
- cell array, 101, 128, 165, 308–313
  - converting to and from, 312
  - displaying contents, 311–312
  - indexing, 310
  - preallocating, 312, 374
  - replacing comma-separated list, 312
- cell2struct, 312
- celldisp, 311
- cellplot, 312
- centrality, 353 t, 355
- cgs, 155 t
- char, 292–295, 442 t
- characteristic polynomial, 148, 176, 339
- charpoly, 339
- Chebfun, 324
- chebyshevT, 342 t
- chebyshevU, 342 t
- checkcode, 259, 260
- Children property, 278
- chol, 145, 252, 340 t
- Cholesky factorization, 138, 145, 252
- cholupdate, 145
- circle, drawing with `rectangle`, 282
- circshift, 298 t
- circulant matrix, 321–324
- clabel, 115–116
- class, 83, 315–324, 445 g, *see also* data type
- class, 44, 291
- classdef, 316
- clc, 26, 442 t
- clear, 9, 31, 92, 246, 311, 442 t
- clearing
  - Command Window, *see* `clc`
  - figure window, *see* `clf`
  - workspace, *see* `clear`
- clearvars, 31
- clf, 102, 246, 441 t
- client, 387
- close, 8, 102, 246, 441 t

- CMYK color space, 99
- code file, *see* program file
- codistributed array, 397–398
- coeffs, 339
- colamd, 253
- Collatz iteration, 10
- colon notation, 5–6, 54–57
- Color property, 98, 99 t
- color maps, 119
- color spaces, 99
- colorbar, 115
- colormap, 118, 119, 441 t
- colormapeditor, 119
- colors, default order for plotting, 98 f
- colperm, 253
- colspace, 340 t
- comet, 114 t, 280–281
- comet3, 123 t, 280
- comma
  - to separate matrix elements, 6, 50
  - to separate statements, 4, 24
- Command History, 26, 445 g
- command line, editing, 26, 27 t
- Command Window, 1, 23, 445 g
  - clearing, 26
- command/function duality, 93–94, 130, 194
- comment block, 91
- comment line, 10, 24, 84
- companion, 52 t
- complex, 12, 30, 72, 330
- complex arithmetic, 30, 37
- complex numbers
  - entering, 30, 72
  - functions for manipulating, 42 t
- complex variable, visualizing function
  - of  $a$ , 122
- complexity, of a code, 260
- Composite, 396
- composite object, 396
- computer, 27 t
- cond, 137, 143, 441 t
- condeig, 150
- condest, 137–138, 143, 252
- condition number (of eigenvalue), 150
- condition number (of matrix), 137
  - estimation, 137
- conj, 30, 42 t
- conjugate transpose, 59
- conncomp, 353 t
- contains, 296
- contentsrpt, 268
- continuation line, 26
- continuation method, 216
- continue, 80
- continued fraction, 7
- contour, 114, 115, 123 t, 441 t
- contour3, 123 t
- contourf, 12, 123 t
- conv, 176–177
- copyfile, 27
- cos, 42 t
- cosd, 42 t
- cosh, 42 t
- cot, 42 t
- cotd, 42 t
- coth, 42 t
- cov, 174
- cplxroot, 122
- cross, 60
- cross product, 60
- csc, 42 t
- cscd, 42 t
- csch, 42 t
- CSV file, 308, 361
- ctranspose, 60
- Ctrl-C (stop execution), 26
- cummax, 67 t
- cummin, 67 t
- cumprod, 67 t
- cumsum, 67 t, 191, 372
- Current Folder browser, 264, 268, 445 g
- Curvature property, 282
- Cuthill–McKee ordering, 253
- cyclomatic complexity, 259, 260
- data analysis
  - basic functions, 67 t, 440 t
  - dealing with large data sets, 361–368
- data fitting
  - interpolation, *see* interpolation
  - least-squares polynomial, 177
- data type
  - calendarDuration array, 303–304
  - categorical array, 299–300, 306–307
  - cell, 308
  - char array, 292–295
  - datetime array, 300–304

- determining, 291
- double, 39, 42–44
- duration array, 300–304
- function handle, 159–160
- fundamental types, 291
- int\*, 44–45
- logical, 71
- multidimensional array, 297–298
- single, 42–44
- string array, 295–297
- struct, 308
- table, 304–308
- timetable, 307–308
- uint\*, 44–45
- DataAspectRatio property, 282
- datastore, 361–364
- datetime, 301, 363
- datetime array, 300–304
- dbclear, 245
- dbcont, 245
- dbdown, 245
- dbquit, 245
- dbstop, 245
- dbtype, 245
- dbup, 245
- dde23, 221–225
- ddensd, 225
- ddesd, 223
- ddeset, 223
- deal, 382
- debugger, 245–246
- debugging, 245–247
- decic, 213
- deconv, 176–177, 341
- degree, 350, 353 t
- delay-differential equations (DDEs), 221–225
- delete, 27, 276, 388
- demo, 27 t, 442 t
- deprpt, 268
- det, 142–143, 340 t
- details, 301
- determinant, 142
- deval, 199, 216
- dfsearch, 353 t
- diag, 64 t, 65–66, 340 t, 440 t
- diagonal matrix, 65–66
- diagonals
  - of matrix, 65–66
  - assigning to, 384–385
  - of sparse matrix, 250–251
- diary, 32, 233, 442 t
- diff, 67 t, 68, 332–334
- differential equations
  - numerical solution, 12, 193–229
  - symbolic solution, 335–336
- differential-algebraic equations (DAEs), 210–213
- digits, 343–347
- digraph, 353 t, 351–357
- dimension of array/matrix, 36, 47, 297
- dir, 9, 27, 442 t
- directed graph, 349
- direction field, *see* vector field
- directory
  - changing, making, listing, printing, 27
  - private, 169
- discretize, 357
- disp, 32, 234, 442 t
- distributed array, 397–398
- division, 58–59
  - left, 57
  - right, 57
- doc, 28, 442 t
- docsearch, 28
- dot, 2, 60
- dot product, 2, 60
- double (function), 332, 442
- double data type, 39, 42–44
- double integral, 192
- drawnow, 281
- dsolve, 335–336
- duration, 303
- duration array, 300–304
- echo, 234
- edit, 9, 90, 441 t
- Editor/Debugger, 9, 90, 91 f, 174, 245, 246, 257, 445 g
- eig, 5, 148–152, 255, 339, 340 t, 441 t
- eigenvalue, 148
  - generalized, 151
- eigenvalue problem
  - generalized, 151
  - Hermitian definite, 152
  - numerical solution
    - direct, 148–153
    - iterative, 155–156



- polynomial, 153
  - quadratic, 153
  - standard, 148
  - symbolic solution, 339
- eigenvector, 148
  - generalized, 151
- eigs, 155–156, 253, 255
- elementary functions, 42 t
- elementary matrix functions, 48 t
- ellipj, 42 t
- ellipke, 42 t
- elliptic integral, 332
- empty array, 379–380, *see also* empty matrix
  - testing for, 73 t
- empty matrix, 63–64, 68, *see also* empty array
  - as subscript, 379–380
- end (flow control), 78–80
- end (subscript), 55, 440
- epicycloid, 104–105
- eps, 39, 43, 439 t
- equations
  - algebraic, *see* linear equations; non-linear equations
  - differential, *see* differential equations
  - linear, *see* linear equations
  - nonlinear, *see* nonlinear equations
- erf, 42 t, 422
- error, 201, 242, 440 t
- error messages, understanding, 241–242
- errorbar, 114 t
- errors, 241–242
- eval, 294
- event location, 201–203
- exist, 82, 92
- existsOnGPU, 399
- exit, 1, 25, 442 t
- exp, 42 t
- expint, 42 t
- expm, 156, 174, 340 t
- expm1, 42, 42 t
- exponential, of matrix, 156
- exponentiation, 59
- export\_fig, 131
- external codes, calling, 375–378
- eye, 6, 43, 47, 48 t, 439 t
- factor, 42 t
- factorial, 42 t, 372
- false, 71
- fast Fourier transform, 185–187
- fcontour, 114
- fetchNext, 393
- fetchOutputs, 395
- FevalFuture object, 393
- fft, 185, 322, 443 t
- fft2, 186
- fftn, 186
- fftw, 186
- Fibonacci sequence, 242 *l*
  - random, 8
- FIG-file, 131, 445 g
- figure, 445 g
- figure, 102
- Figure object, 273
- figure window, 97
  - clearing, 102
  - closing, 102
  - creating new, 102
- figure, saving and printing, 129–131
- File Exchange, *see* MATLAB Central File Exchange
- fill, 109, 114 t, 215
- fill3, 123 t
- find, 75–78, 250, 440 t
- findall, 278, 282
- findedge, 353 t
- findnode, 353 t
- findobj, 278, 282
- findstr, 294
- Fisher’s equation (PDE), 432–434
- fix, 42 t
- flip, 64 t
- fliplr, 64 t
- flipud, 64 t
- float, 291
- floating-point arithmetic, 39
  - IEEE standard, 39
  - range, 39
  - subnormal number, 40
  - unit roundoff, 39
- floor, 42 t
- flow control, 78–82
- fmesh, 118, 123 t
- fminbnd, 174, 184, 443 t
- fminsearch, 184, 185, 404
- folder, *see* directory

- FontAngle property, 100, 276
  - default, 101 t
- FontSize property, 100, 106, 281, 282
  - default, 101 t
- FontWeight property, 109, 276
- fopen, 236
- for, 7, 79, 440 t
- for loops, avoiding by vectorizing, 370–372
- format, 3, 25, 26 t, 41, 442 t
- fortran, 347
- forward slash operator, /, 138
- Fourier transform, discrete, 185–187, 322
- fplot, 109–112, 114 t, 441 t
- fplot3, 123 t
- fprintf, 234–236, 442 t
- fractal landscape, 119
- Frank matrix, 150, 337–339
- fread, 237
- Fresnel integrals, 191, 347
- fresnelc, 347
- fresnels, 347
- fscanf, 237
- fsurf, 118, 123 t
- full, 249
- function, 83–94, 159–174, 445 g
  - anonymous, 160–161, 183, 185, 404
    - origin in Lisp, 161
  - arguments, copying of, 375
  - arguments, defaults, 163–165
  - arguments, multiple input and output, 35–37, 86–90
  - definition line, 84
  - determining program files it calls, 268–269
  - documenting, 85
  - evaluating with feval, 160
  - existence of, testing, 92
  - H1 line, 85, 92
  - handle, 159–160, 162
  - local, 161–163
  - of a matrix, 61, 156–158
  - nested, 168–169, 198, 216, 404, 412
  - passing as argument, 159–160, 162
  - private, 169–170
  - recursive, 12–16, 119, 170–171, 420–422
  - subfunction, *see* function, local
- function/command duality, 93–94
- funm, 157–158, 340 t
- fwrite, 237
- fzero, 181, 182, 184, 219, 404, 443 t
- gallery, 51, 52 t, 53 t, 170, 253, 337, 439 t
- gamma, 42 t
- gather, 365, 366, 399
- Gaussian elimination, *see* LU factorization
- gca, 276
- gcd, 42 t
- gcf, 276
- gco, 276, 278
- gegenbauerC, 342 t
- generalized eigenvalue problem, 151
- generalized real Schur decomposition, 153
- generalized Schur decomposition, 153
  - reordering, 153
- generalized singular value decomposition, 148
- get, 26, 278–279, 395
- getframe, 279
- ginput, 233
- Git, 264
- GitHub, 269
- Givens rotation, 371
- global, 83, 173
- global variables, 173
- glossary, 445–446
- gmres, 155, 155 t
- GPU Computing, 398–400
- gpuArray, 399–400
- gpuDevice, 398
- gpuDeviceCount, 399
- gputimeit, 400
- grabcode, 265
- gradient, 334, 335, 417
- graph
  - directed, 349
  - undirected, 349, 404
- graph, 349–351, 353 t
- Graphical User Interface (GUI) tools, 278
- graphics, 97–133, 273–282
  - 2D, 97–113
    - summary of functions, 114 t, 441 t
  - 3D, 113–122

- summary of functions, 123 t
- animation, 279–281
- hierarchical object structure, 274 f, 278
- labelling, 101, 114
- legends, 105–106
- NaN in function arguments, 122
- optimizing for presentations, 279
- optimizing for printed output, 130–131, 279
- Plot Editor, 97
- property values, factory-defined, 278–279
- saving and printing, 129–131
- specialized, 125–129
- grid, 12, 104, 441 t
- Gridalpha property, 104, 275
- GridColor property, 104, 275
- griddata, 179–180
- griddata3, 180
- griddatan, 180
- griddedInterpolant, 180
- GridLineStyle property, 104, 275
- groot, 278
- gsvd, 148, 174
- gtext, 106
- GUI tools, *see* Graphical User Interface (GUI) tools
- H1 line, 85, 92
- hadamard, 52 t, 174
- handle
  - to function, *see* function handle
  - to graphics object, 273
- HandleVisibility property, 282
- hankel, 51, 52 t
- hasdata, 363
- help, 1, 28, 85, 442 t
  - for local functions, 162
- Help browser, 28, 30 f, 445 g
- hermiteH, 342 t
- Hermitian matrix, 135
- hess, 151–153
- Hessenberg factorization, 151, 152
- Hessenberg matrix, 138
- hessian, 334, 335, 417
- hidden, 116
- highlight, 350, 353 t
- hilb, 51, 52 t
- Hilbert matrix, 51
- histogram, 8, 126–128
- histogram, 8, 97, 114 t, 126–128, 300, 364, 441 t
- hold, 101, 441 t
- HorizontalAlignment property, 109, 282
- Horner’s method, 175
- hypot, 42 t, 430
- $i$  ( $\sqrt{-1}$ ), 30, 246, 439 t
- ichol, 154
- identity matrix, 47
  - sparse, 250
- IEEE arithmetic, 39–41, 445 g
- if, 10, 78–79, 440 t
- ifft, 185, 322
- ifft2, 186
- ifftn, 186
- ilu, 154
- imag, 30, 42 t
- image, 133
- imaginary unit ( $i$  or  $j$ ), 5, 30
- implicit expansion, 61–63
- Import Wizard, 32
- imread, 81
- indexing, *see* subscripting
- inf, 39, 439 t
  - exploiting, 380
- inmem, 268
- inner product, 2, 60
- input
  - from file, 236–237
  - from the keyboard, 233
  - via mouse clicks, 233
- input, 10, 233, 442 t
- inputParser, 168
- int, 330–332
- int\* data type, 44–45
- int16, 44
- int2str, 130, 235, 293, 294
- int32, 44
- int64, 44
- int8, 44
- integer, 292
- integral, 189–192, 372, 389, 443 t
  - default error tolerances, 189, 189 t
- integral2, 192–193
- integral3, 193
- integration
  - double, 192–193

- numerical, *see* numerical integration
- symbolic, 330–332
- interp1, 178–179, 303, 443t
- interp2, 179
- interp3, 180
- interp<sub>n</sub>, 180
- interpolation
  - 1D (interp1), 178
  - 2D (interp2, griddata), 179
  - multidimensional, 180
  - polynomial, 177
  - spline, 177
- Interpreter property, 109
- intmax, 45
- intmin, 45
- inv, 25, 142, 338, 340t
- inverse matrix, 142
  - symbolic computation, 338
- invhilib, 51, 52t
- ipermute, 298t
- isa, 291
- isAlways, 329
- isbanded, 136t
- ischar, 73t, 294
- iscolumn, 73t
- isdiag, 136t
- isempty, 73t, 163, 440t
- isequal, 72, 73t, 440t
- isequaln, 73t
- isfield, 310
- isfinite, 73t, 75
- isfloat, 73t, 292
- isgraphics, 274
- ishermitian, 136t
- isinf, 72, 73t
- isinteger, 73t, 292
- isisomorphic, 353t
- iskeyword, 92
- islogical, 73t
- ismac, 93
- ismatrix, 73t
- ismember, 383, 384
- ismissing, 304
- isnan, 69, 72, 73t
- isnumeric, 73t, 292
- isomorphism, 353t
- ispc, 93
- isprime, 42t
- isreal, 72, 73t
- isrow, 73t
- isscalar, 73t
- issorted, 73t
- issparse, 73t, 252
- isstring, 73t
- issymmetric, 136t
- istril, 136t
- istriu, 136t
- isunix, 93
- isvector, 73t
- iterative eigenvalue problem solvers, 155–156
- iterative linear equation solvers, 140t, 155t, 153–155
- $j(\sqrt{-1})$ , 30, 246, 439t
- jacobian, 334
- jacobiP, 342t
- jet color map, 120t
- Job Monitor, 395
- join, 296
- jordan, 340t
- Just-In-Time (JIT) accelerator, 369
- keyboard, 245
- keypresses for command line editing, 27t
- keywords, 92
- Koch curves, 170–171
- Koch snowflake, 170–171
- kron, 60
- Kronecker product, 60
- L-systems, 420–422
- LabelFontSizeMultiplier property, 276
- labindex, 395
- laguerreL, 342t
- lambda expression, 161
- lambertw, 388
- LAPACK, 135, 137, 445g
- lasterr, 242
- lastwarn, 243
- L<sup>A</sup>T<sub>E</sub>X, 109, 129, 331, 347
  - formatting output for, 238
  - TikZ and PGFPlots packages, 131
- latex, 331, 347
- lcm, 42t
- ldl, 144, 252

- least-squares data fitting, by polynomial, 177
- least-squares solution to overdetermined system, 141
- legend, 105–106, 441 t
- legendre, 42 t
- legendreP, 342 t
- length, 47, 440 t
- license, 27 t
- 'like' argument, 397, 400
- line, 282
- linear algebra functions, symbolic, 340 t
- linear equations, 138–142, *see also* overdetermined system; underdetermined system
  - numerical solution
    - direct, 138–142
    - iterative, 153–155
    - symbolic solution, 338
- LineWidth property, 99
  - default, 101 t
- linsolve, 140 t, 138–140
- linspace, 12, 48 t, 57, 439 t
- Lisp, 161, 315
- Live Editor, 264–265, 331, 445 g
- live script, 83, 264–265, 445 g
- load, 31, 52, 442 t
- local function, 161–163
- log, 42 t
- log10, 42 t
- log1p, 42, 42 t
- log2, 42 t
- logarithm, of matrix, 156
- logical, 77–78
- logical array, 77–78
- logical data type, 71
- logical operators, 73–78
- logistic differential equation, 335
- loglog, 100, 114 t
- logm, 156, 340 t
- logspace, 48 t
- lookfor, 92, 441 t
- loop structures, *see* for, while
- lorenz, 12, 281
- Lorenz equations, 12
- ls, 9, 27
- lsqnonneg, 141
- lsqr, 155 t
- lu, 90 t, 143, 252, 340 t, 441 t
- LU factorization
  - partial pivoting, 138, 143–144, 252
  - threshold pivoting, 252
- M-file, *see* program file
- magic, 51, 52 t
- Mandelbrot set, 12
- MapReduce, 364
- mapreduce, 364
- MarkerEdgeColor property, 100
  - default, 101 t
- MarkerFaceColor property, 100
  - default, 101 t
- MarkerSize property, 99
  - default, 101 t
- MAT-file, 31, 133, 445 g
- MATLAB -singleCompThread, 397
- MATLAB Central File Exchange, 83, 269
- MATLAB desktop, 2 f, 23, 445 g
- MATLAB releases, xxiii t
- MATLAB Toolstrip, 23, 446 g
- MATLAB Web browser, 446 g
- MATLAB, classic, 33
- matlab.apputil.package, 269
- matlab.codetools.requiredFilesAndProducts, 268
- matlab.lang.makeValidName, 258
- matlab2tikz, 131
- matlabFunction, 347
- matlabrc, 92
- matrix
  - adjacency, 349, 404
  - block diagonal, 50
  - block form, 50
  - circulant, 321–324
  - condition number, 137
  - conjugate transpose, 59
  - deleting a row or column, 64
  - diagonal, 65–66
  - empty, 63–64, *see* empty matrix
  - exponential, 156
  - Frank, 150, 337–339
  - function of, 61, 156–158
  - generation, 4, 6–7, 47–52
  - Hermitian, 135
  - Hermitian positive definite, 138
  - Hessenberg, 138
  - Hilbert, 51
  - identity, 47

- shifting by multiple of, 385
    - sparse, 250
  - inverse, 142
  - logarithm, 156
  - manipulation functions, 64 t, 440 t
  - norm, 136–137
  - orthogonal, 135
  - rank-1, forming, 382–383
  - reshaping, 64
  - skew-Hermitian, 135
  - skew-symmetric, 135
  - square root, 156
  - storage in column major order, 373–374
  - submatrix, 6, 54
  - subscripting as vector, 384–385
  - symmetric, 135
  - transpose, 59
  - triangular, 66, 138
  - unitary, 135
  - Wathen, 154
- matrix operations
- elementary, 59 t
  - elementwise, 3, 58, 86
- max, 36, 62, 67 t, 67–68, 372, 440 t
- max-plus algebra, 315–321
- maxflow, 353 t
- McCabe complexity, 259
- mean, 67 t, 440 t
- median, 67 t
- membrane, 116, 260
- mesh, 116, 118, 123 t, 441 t
- meshc, 116, 123 t
- meshgrid, 12, 48 t, 115, 179, 180, 424
- meshz, 121, 123 t
- method of lines, 208
- methods, 316, 400
- MEX interface, 375, 446
- mfilename, 167
- min, 62, 67 t, 67–68, 440 t
- minimization of nonlinear function, 184–185
- minimum-degree ordering, 253
- minres, 153, 155 t
- minspantree, 353 t
- mkdir, 27
- mkpp, 178
- mldivide, 138
- mlintrpt, 260
- mod, 42 t
- mode, 67 t
- more, 92
- movie, 279
- movies, 279–280
- movmax, 67 t
- movmean, 67 t
- movmedian, 67 t
- movmin, 67 t
- movstd, 67 t
- movsum, 67 t
- mrdivide, 138
- multidimensional array, 297–298
- functions for manipulating, 298 t
- multiplication, 58–59
- namelengthmax, 31
- NaN, 439 t
- NaN (Not a Number), 40, 67, 439 t
- in graphics function arguments, 122
  - removing, 69
  - for representing missing data, 69
  - testing for, 72
- nargin, 88, 163, 441 t
- narginchk, 167, 430
- nargout, 88, 166, 441 t
- nargoutchk, 167
- NaN, 304
- nchoosek, 42 t, 174
- ndgrid, 298 t
- ndims, 297, 298 t
- nearest, 353 t
- neighbors, 353 t
- Nelder–Mead simplex algorithm, 185
- nested function, 168–169, 198, 216, 404, 412
- nextpow2, 42 t
- nnz, 250, 384, 412 n
- nonlinear equations
- numerical solution, 180–184
  - symbolic solution, 327–329
- nonlinear minimization, 184–185
- nonzeros, 250
- norm
- evaluating dual, 380
  - matrix, 136–137
  - vector, 35–36, 136
- norm, 35–36, 136–137, 380, 441 t
- normally distributed random numbers, 6, 48

- normest, 137
- normest1, 137–138
- not, 74
- nthroot, 42 t
- null, 147, 340 t
- num2cell, 312
- num2str, 235, 293, 442 t
- number theoretic functions, 42 t
- numden, 341
- numedges, 349, 353 t
- numeric, 292
- numerical integration, 189–193
  - adaptive, 191
  - double integral, 192–193
- numnodes, 349, 353 t
- nzmax, 250
  
- object-oriented programming, 315–324
- ode113, 208 t
- ode15i, 208 t, 211
- ode15s, 205, 208, 208 t, 210, 211, 226
- ode23, 208 t
- ode23s, 208 t
- ode23t, 208 t, 210
- ode23tb, 208 t
- ode45, 12, 174, 193–199, 208 t, 443 t
  - behavior on stiff problems, 205–207
- odeexamples, 213
- odeget, 213
- odephas2, 213
- odeset, 197, 201, 205, 207, 211, 213, 226, 372
- onCleanup, 258, 259
- ones, 6, 43, 47, 48 t, 297, 439 t
- open, 131
- openvar, 31
- operator precedence, 75, 76 t
  - arithmetic, 41, 41 t
- operators
  - logical, 73–78
  - relational, 71–78
- Optimization Toolbox, 181 n, 185
- optimizing codes, 369–375
- optimset, 182, 184
- or, 74
- ordeig, 151
- ordinary differential equation (ODE)
  - solvers, 208 t
  - default error tolerances, 189 t, 197
  - error control, 197
  - evaluating solution at intermediate values, 199
  - event location, 201–203
  - example files, 213
  - input and output arguments, 194, 197
  - Jacobian, specifying, 207–210
  - mass matrix, specifying, 210–213
  - obtaining solutions at specific times, 194
  - option setting with `odeset`, 197
  - output as structure, `sol`, 198
  - symbolic, 335–336
  - tolerance proportionality, 197
- ordinary differential equations (ODEs), 12, 193–220
  - boundary-value problem (BVP), 213
  - higher order, 195
  - initial-value problem, 193
  - pursuit problem, 201–203
  - Robertson problem, 205
  - Rössler system, 197–199
  - simple pendulum equation, 195
  - stiff, 205–213
- ordqz, 153
- ordschur, 151
- orth, 147, 340 t
- otherwise, 80
- outer product, 3
- output
  - to file, 236–237
  - to screen, 234–236
- output arguments
  - discarding with `~`, 90
  - function behavior depending on number of, 36
- overdetermined system, 140–141
  - basic solution, 141
  - minimal 2-norm solution, 141
- overloading, 300, 303, 315, 316, 318, 322, 332, 338, 339, 341, 350
  
- P-code, 261–264
- Parallel Computing Toolbox, 387–401
- parcluster, 395
- Parent property, 278
- parfeval, 392–393
- parfevalOnAll, 393
- parfor, 388–392, 397

- parpool, 387, 388
- partfrac, 342
- Partial Differential Equation Toolbox, 229
- partial differential equations (PDEs), 208, 225–229
- parula color map, 119, 120 t
- pascal, 52 t
- path, 92
- path (MATLAB search path), 91–92
- pathtool, 92
- pause, 233
- Pause button, 91
- pcg, 155 t, 153–155, 174
- pchip, 178
- pcode, 261–264
- PDE solver, 225–229
- pdepe, 225–229
- pdeval, 226
- peaks, 122
- performance profile, 409–416
- performance testing, 272
- perms, 42 t
- permutations, 380–382
- permute, 298 t
- persistent, 173
- persistent variables, 173
- phase plane plot, 196
- phase, of complex number, *see* angle
- pi, 4, 30, 439 t
- pie, 114 t, 128
- pie charts, 128
- pie3, 123 t, 128
- pinv, 141, 143
- pitfalls, 246–247
- pivoting, *see* partial pivoting; rook pivoting
- plot, 8, 97–100, 114 t, 350, 353 t, 441 t
  - options, 99 t
- Plot Editor, 97
- plot3, 113, 123 t
- plottedit, 97, 278
- plotting functions
  - 2D, 114 t, 441 t
  - 3D, 123 t
- pmode, 401
- point, 100
- pol2cart, 42 t
- polarplot, 114 t
- poly, 148, 176, 340 t
- poly2sym, 339, 341
- polyder, 176
- polyeig, 153
- polyfit, 177, 443 t
- polynomial
  - division, 176
  - eigenvalue problem, 153
  - evaluation, 175–176
    - of derivative, 176
  - multiplication, 176
  - representation, 175
  - root finding, 176
- polynomials, symbolic, 342 t
- polyval, 175, 176
- polyvalm, 175, 176
- Position property, 276, 282
- positive definite matrix, 138
  - testing for, 145
- pow2, 42 t
- power method, 137
- ppval, 178
- preallocating arrays, 374
- precedence
  - of arithmetic operators, 41, 41 t
  - of operators, 75, 76 t
- precision, 25, 39
- preconditioning, 154
- pretty, 331, 336
- preview, 362
- primes, 42 t, 57
- print, 129–131, 441 t
- printing
  - a figure, 129–131
  - to file, 236–237
  - to screen, 234–236
- private function, 169–170
- prod, 67 t, 372, 440 t
- profile, 260–261
- profiling, 260–261
- program file, 83–94, 159–174, 257–261, 446 g
  - commenting out a block, 91
  - determining codes it calls, 268–269
  - editing, 90–91
  - function, 83–94, 159–174, 445 g, *see also* function
  - H1 line, 85, 92
  - live script, 83



- names, 90, 92
- optimizing, 369–375
- script, 9, 24, 83–84, 446 g
- search path, 91
- style, 257–258
- vectorizing, 370–372
- pseudo-inverse matrix, 143
- psi, 42 t
- publish, 265
- pursuit problem, 201–203
- pwd, 9, 27
- Pythagorean sum, 430–432
  
- qmr, 154, 155 t
- qr, 145–146, 253, 340 t, 441 t
- QR algorithm, 151
- QR factorization, 145–146
  - column pivoting, 141, 146
  - of sparse matrix, 253
- qrdelete, 146
- qrinsert, 146
- qrupdate, 146
- quad, 191
- quadgk, 168, 191
- quadl, 191
- quadratic eigenproblem, 153
- quadrature, *see* numerical integration
- quit, 1, 25, 442 t
- quiver, 114 t, 195
- quorem, 341
- quote mark, representing within string, 79, 235
- qz, 153
- QZ algorithm, 152, 153
  
- rainbow color map, 119
- rand, 6, 25, 48, 48 t, 297, 439 t
- randi, 48 t
- randn, 6, 48, 48 t, 297, 424, 439 t
- random number generators
  - period, 49
  - RAND corporation book, 7
  - seed, 6
  - state, 49
- randperm, 382
- rank, 147, 340 t
- rat, 42 t
- rats, 42 t
- rcond, 137, 138, 142, 143
- reaction–diffusion equations, 227–229
  
- read, 362
- ReadSize property, 362
- readtable, 236, 301, 307
- real, 30, 42 t, 247
- real Schur decomposition, 151
  - generalized, 153
- realmax, 39, 43
- realmin, 40, 43
- rectangle, 282
- recursive function, 12–16, 119, 170–171, 420–422
- Refine option, 207
- relational operators, 71–78
- RelTol option, 189
- rem, 42 t
- reordernodes, 353 t
- repelem, 60
- replace, 296
- repmat, 48 t, 51, 310, 374, 383, 440 t
- reproducible research, 49
- reset, 279, 336, 399
- reshape, 64, 64 t, 440 t
- retime, 308
- return, 88, 245
- reverse, 296
- RGB color space, 99
- Riemann surface, 122, 124 f
- rng, 6, 49
- Robertson ODE problem, 205
- rook pivoting, 144
- Root object, 273
  - assignable properties, 278
- roots, 176, 443 t
- rosser, 52 t
- Rössler ODE system, 197–199
- rot90, 64 t
- Rotation property, 106
- round, 42 t
- rounding error, 25
- rowfun, 308
- rref, 148, 340 t
- run, 92
- Runge–Kutta method, 193
- runperf, 272
- runttests, 269
  
- save, 31, 442 t
- saveas, 131
- scalar expansion, 57, 60, 375, 385, 412
- scatter, 114 t

- scatter3, 123t
- scatteredInterpolant, 180
- schur, 151
- Schur decomposition, 151
  - generalized, 153
  - generalized real, 153
  - real, 151
  - reordering, 151
- script, 9, 24, 83–84
  - live, 83, 445g
  - one-line, 295
- search path, 91
- sec, 42t
- secd, 42t
- sech, 42t
- semicolon
  - to denote end of row, 2, 4, 6, 50
  - to suppress output, 1, 4, 23
- semilogx, 101, 114t
- semilogy, 9, 101, 114t, 441t
- set, 26, 275, 278–279, 282
- set operations, 383–384
- shading, 118
- shg, 102, 441t
- shiftdim, 298t
- short-circuiting of logical expressions, 75
- shortestpath, 353t
- shortestpathtree, 350, 351, 353t
- Sierpinski gasket, 12–18
- sign, 42t
- simplify, 327
- Simpson’s rule, 191
- sin, 42t
- sind, 42t
- single (function), 43
- single data type, 42–44
- single program, multiple data (SPMD), 395–397
- singular value decomposition (SVD), 146–148
  - generalized, 148
- sinh, 42t
- size, 36, 47, 297, 440t
- skew-Hermitian matrix, 135
- skew-symmetric matrix, 135
- small-world networks, 404–409
- smithForm, 340t
- solve, 327–329, 417
- sort, 67t, 66–68, 296, 341, 380, 440t
- source control, 264
- sparse (function), 249–250, 407, 442
- sparse attribute, 249
- sparse matrix, 153–156, 249–255
  - reordering, 253
  - storage required, 250
  - visualizing, 252
- spdiags, 208, 250–251
- special functions, 42t
- special matrix functions, 52t
- spectral radius, 321
- speye, 250
- sph2cart, 42t
- spiral, 52t, 384
- spline, 177, 443t
- spline interpolation, 177
- split, 296
- spmd, 395–397
- spones, 250
- spparms, 255
- sprand, 251
- sprandn, 251
- sprintf, 235, 293, 442t
- spy, 252, 253, 441t
- sqrt, 42t, 61
- sqrtm, 61, 156, 340t
- sqrtm\_tri, 174
- square root of a matrix, 156
- squeeze, 298t
- stairs, 114t, 410
- standard deviation, 67
- startup, 92
- std, 67t, 440t
- stem3, 123t
- stiff ordinary differential equation, 205–213
- storage allocation, automatic, 35
- str2func, 160
- strcat, 293
- strcmp, 293
- strcmpi, 293
- strfind, 294
- string
  - comparison, 293
  - concatenation, 293
  - conversion, 292
  - representation, 292
  - TeX notation in, 106, 107t

- string, 293, 295–297
- string array, 295–297
- struct, 308, 309, 374, 442 t
- struct2cell, 312
- structure, 308–313, 329
  - accessing fields, 309
  - bvpset, for BVP solver options, 216
  - odeset, for ODE solver options, 197
  - optimset, for nonlinear equation solver options, 182
  - preallocating, 309–310, 374
- sub2ind, 385
- subclass, 316
- subfunction, *see* local function
- subgraph, 353 t
- submatrix, 6, 54
- subnormal number, 40
- subplot, 109–112, 441 t
  - irregular grid, 112
- subs, 328, 343
- subscripting, 54–57
  - end, 55
  - single subscript for matrix, 76, 384–385
  - subscripts start at 1, 54, 56, 247
  - zero-based versus one-based, 56, 247
- Subversion, 264
- sum, 67, 67 t, 67–68, 440 t
- summary, 299, 305
- superclass, 316
- surf, 116, 118, 123 t, 441 t
- surfc, 116, 122, 123 t
- svd, 147, 340 t, 441 t
- svds, 156, 253
- switch, 80–81, 174, 422, 440 t
- sylveste, 140 t
- sym, 325–327, 329–330
- sym2poly, 339
- symamd, 253
- Symbolic Math Toolbox, 325–347
- symbolic object, 325
- symmetric matrix, 135
- symmlq, 153, 155 t
- symrcm, 253
- syms, 325, 329–330
- symvar, 327
- synchronize, 308
- system command (!), 27
- tab completion, 1, 26
- table, 304–308
- table2array, 308
- table2timetable, 307
- tall, 365
- tall array, 364–366
- tan, 42 t
- tand, 42 t
- tanh, 42 t
- taylor, 336–337
- Taylor series, 336–337
- taylortool, 337
- test matrix
  - functions, 52 t, 53 t
  - properties, 55 t
- TeX commands, in text strings, 106, 107 t
- texlabel, 106
- text, 106, 441 t
- textscan, 237, 301
- tfqmr, 155 t
- tic, 156, 369–370, 442 t
- tick marks, 282, 412
- TickLength property, 282
- timeit, 370, 400
- timerange, 308
- timetable, 307–308
- timing a computation, 156, 369–370
- title, 101, 441 t
- TitleFontSizeMultiplier property, 276
- toc, 156, 369–370, 442 t
- toeplitz, 51, 52 t, 406, 407, 427
- tolerance, mixed absolute/relative, 189
- Tony’s trick, 383, 412
- toolbox, 446 g
  - creating, 265–268
- Toolboxes
  - Optimization Toolbox, 181 n, 185
  - Parallel Computing, 387–401
  - Symbolic Math Toolbox, 325–347
- Toolstrip, 90, 264
- transpose, 60
- trapezium rule, 192
- trapz, 192
- triangular matrix, 138
- triangular parts, 66
- trigonometric functions, 42 t

- tril, 64t, 66, 340t, 440t
- triu, 64t, 66, 340t, 440t
- true, 71
- turtle graphics, 420–422
- type, 92, 441t
  
- uiimport, 32
- uint\* data type, 44–45
- uint16, 44
- uint32, 44
- uint64, 44
- uint8, 44
- UMFPACK, 252
- underdetermined system, 141–142
  - basic solution, 141
  - minimal 2-norm solution, 141
- undirected graph, 349, 404
- uniformly distributed random numbers,
  - 6, 48
- unique, 365, 412
- unit roundoff, 39
- unit tests, 269–272
- unitary matrix, 135
- Units property, 276
- unmkpp, 178
- unwrap, 42t
  
- validateattributes, 167
- validatestring, 168
- vander, 52t
- var, 67t, 174
- varargin, 165–166, 174, 312
- varargout, 165–166, 174, 312
- varfun, 308
- variable names, 31
  - camel case, 258
  - choosing, 92, 173, 258
  - pothole case, 258
  - snake case, 258
- variable-precision arithmetic, 343–347
- variables
  - global, 173
  - listing, 7, 31
  - loading, 31
  - persistent, 173
  - removing, 9, 31
  - saving, 31
  - testing for existence of, 92
- variance, 67
- vector
  - field, 195, 196
  - generation, 5–6
    - linearly spaced, *see* linspace
    - logarithmically spaced, *see* log-space
  - logical, 77–78
  - norm, 136
  - product, 60
- vectorizing
  - codes, 370–372
  - empty subscript produced, 379–380
- ver, 27t, 268, 325
- version, 27t
- version control, 264
- VerticalAlignment property, 282
- VideoWriter, 280
- view, 118, 119, 433
- visdiff, 260
- vpa, 344–347
- vpasolve, 345
  
- wait, 395
- waitbar, 12, 278
- warning, 138, 243–245
- waterfall, 118, 123t, 434
- Wathen matrix, 154, 253
- Waypoints, 190
- web, xxiv
- what, 92, 442t
- which, 92, 246, 441t
- while, 10, 80, 440t
- who, 7, 31, 442t
- whos, 7, 31, 44, 250, 442t
- why, 95, 174
- wilkinson, 52t
- worker, 387
- workspace, *see* variables
- workspace, 31
- Workspace browser, 31, 32f, 446g
- writetable, 236, 308
  
- XData property, 279
- xlabel, 12, 101, 441t
- xlim, 103t, 104, 441t
- xlsread, 236
- xlswrite, 236
- xor, 74
- XScale property, 416
- XTick property, 282
- xtickangle, 113

`xtickformat`, 113

`xticklabels`, 113

`xticks`, 113

`YData` property, 279

`YDir` property, 282

`ylabel`, 12, 101, 441 t

`ylim`, 103 t, 104, 441 t

`ytickangle`, 113

`ytickformat`, 113

`YTickLabel` property, 282

`yticklabels`, 113

`yticks`, 113

`yyaxis`, 114 t, 282

`ZData` property, 279

`zeros`, 6, 43, 47, 48 t, 297, 439 t

`zlabel`, 114

`zoom`, 102

MATLAB is an interactive system for numerical computation that is widely used for teaching and research in industry and academia. It provides a modern programming language and problem solving environment, with powerful data structures, customizable graphics, and easy-to-use editing and debugging tools.

This third edition of *MATLAB Guide* completely revises and updates the best-selling second edition and is 25 percent longer. The book remains a lively, concise introduction to the most popular and important features of MATLAB and the Symbolic Math Toolbox.

Key features are

- a tutorial in Chapter 1 that gives a hands-on overview of MATLAB,
- a thorough treatment of MATLAB mathematics, including the linear algebra and numerical analysis functions and the differential equation solvers, and
- a web page at <http://www.siam.org/books/ot150> that provides example program files, updates, and links to MATLAB resources.

The new edition

- contains color figures throughout,
- includes pithy discussions of related topics in new “Asides” boxes that augment the text,
- has new chapters on the Parallel Computing Toolbox, object-oriented programming, graphs, and large data sets,
- covers important new MATLAB data types such as categorical arrays, string arrays, tall arrays, tables, and timetables,
- contains more on MATLAB workflow, including the Live Editor and unit tests, and
- fully reflects major updates to the MATLAB graphics system.

This book is suitable for both beginners and more experienced users, including students, researchers, and practitioners.



**Desmond J. Higham** is 1966 Chair of Numerical Analysis at the University of Strathclyde, UK. His research focuses on stochastic computation, network science, and city analytics. He is a SIAM Dahlquist Prize winner, a SIAM Fellow, and a Fellow of the Royal Society of Edinburgh.



**Nicholas J. Higham** is Richardson Professor of Applied Mathematics at the University of Manchester, UK. His research focuses on numerical linear algebra, and he has contributed numerous functions to MATLAB. He is a Fellow of the Royal Society, a SIAM Fellow, and a Member of Academia Europaea.

For more information about SIAM books, journals, conferences, memberships, or activities, contact:

**siam**

Society for Industrial and Applied Mathematics  
3600 Market Street, 6th Floor  
Philadelphia, PA 19104-2688 USA  
+1-215-382-9800 • Fax +1-215-386-7999  
[siam@siam.org](mailto:siam@siam.org) • [www.siam.org](http://www.siam.org)

ISBN 978-1-611974-65-2



9781611974652

OT150