

GNU Octave

A high-level interactive language for numerical computations
Edition 3 for Octave version 2.9.13
July 2007

John W. Eaton

Copyright © 1996, 1997, 2007 John W. Eaton.

This is the third edition of the Octave documentation, and is consistent with version 2.9.13 of Octave.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the same conditions as for modified versions.

Portions of this document have been adapted from the **gawk**, **readline**, **gcc**, and **C** library manuals, published by the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301-1307, USA.

Table of Contents

Preface	1
Acknowledgements	1
How You Can Contribute to Octave	3
Distribution	4
1 A Brief Introduction to Octave	5
1.1 Running Octave	5
1.2 Simple Examples	5
1.2.1 Creating a Matrix	5
1.2.2 Matrix Arithmetic	6
1.2.3 Solving Linear Equations	6
1.2.4 Integrating Differential Equations	6
1.2.5 Producing Graphical Output	7
1.2.6 Editing What You Have Typed	7
1.2.7 Help and Documentation	7
1.3 Conventions	8
1.3.1 Fonts	8
1.3.2 Evaluation Notation	8
1.3.3 Printing Notation	9
1.3.4 Error Messages	9
1.3.5 Format of Descriptions	9
1.3.5.1 A Sample Function Description	9
1.3.5.2 A Sample Command Description	10
1.3.5.3 A Sample Variable Description	10
2 Getting Started	13
2.1 Invoking Octave from the Command Line	13
2.1.1 Command Line Options	13
2.1.2 Startup Files	16
2.2 Quitting Octave	16
2.3 Commands for Getting Help	17
2.4 Command Line Editing	19
2.4.1 Cursor Motion	19
2.4.2 Killing and Yanking	20
2.4.3 Commands For Changing Text	20
2.4.4 Letting Readline Type For You	21
2.4.5 Commands For Manipulating The History	21
2.4.6 Customizing <code>readline</code>	24
2.4.7 Customizing the Prompt	24
2.4.8 Diary and Echo Commands	25
2.5 How Octave Reports Errors	26
2.6 Executable Octave Programs	27
2.7 Comments in Octave Programs	28

3	Data Types	31
3.1	Built-in Data Types	31
3.1.1	Numeric Objects	31
3.1.2	Missing Data	32
3.1.3	String Objects	32
3.1.4	Data Structure Objects	32
3.1.5	Cell Array Objects	32
3.2	User-defined Data Types	32
3.3	Object Sizes	32
4	Numeric Data Types	35
4.1	Matrices	36
4.1.1	Empty Matrices	38
4.2	Ranges	39
4.3	Integer Data Types	40
4.3.1	Integer Arithmetic	41
4.4	Bit Manipulations	42
4.5	Logical Values	44
4.6	Predicates for Numeric Objects	45
5	Strings	47
5.1	Creating Strings	48
5.2	Comparing Strings	50
5.3	Manipulating Strings	51
5.4	String Conversions	56
5.5	Character Class Functions	60
6	Data Structures	63
7	Cell Arrays	69
7.1	Creating Cell Array	69
7.2	Indexing Cell Arrays	71
7.3	Cell Arrays of Strings	72
7.4	Processing Data in Cell Arrays	73
8	Variables	77
8.1	Global Variables	77
8.2	Persistent Variables	78
8.3	Status of Variables	79
8.4	Summary of Built-in Variables	82
8.5	Defaults from the Environment	84

9	Expressions	87
9.1	Index Expressions	87
9.2	Calling Functions	89
9.2.1	Call by Value	90
9.2.2	Recursion	91
9.3	Arithmetic Operators	91
9.4	Comparison Operators	93
9.5	Boolean Expressions	94
9.5.1	Element-by-element Boolean Operators	94
9.5.2	Short-circuit Boolean Operators	95
9.6	Assignment Expressions	96
9.7	Increment Operators	98
9.8	Operator Precedence	99
10	Evaluation	101
10.1	Calling a Function by its Name	101
10.2	Evaluation in a Different Context	102
11	Statements	105
11.1	The <code>if</code> Statement	105
11.2	The <code>switch</code> Statement	107
11.2.1	Notes for the C programmer	108
11.3	The <code>while</code> Statement	109
11.4	The <code>do-until</code> Statement	110
11.5	The <code>for</code> Statement	110
11.5.1	Looping Over Structure Elements	111
11.6	The <code>break</code> Statement	112
11.7	The <code>continue</code> Statement	113
11.8	The <code>unwind_protect</code> Statement	114
11.9	The <code>try</code> Statement	114
11.10	Continuation Lines	115
12	Functions and Script Files	117
12.1	Defining Functions	117
12.2	Multiple Return Values	119
12.3	Variable-length Argument Lists	121
12.4	Variable-length Return Lists	122
12.5	Returning From a Function	123
12.6	Default Arguments	124
12.7	Function Files	124
12.7.1	Manipulating the load path	125
12.7.2	Subfunctions	127
12.7.3	Overloading and Autoloading	127
12.7.4	Function Locking	129
12.8	Script Files	129
12.9	Function Handles, Inline Functions, and Anonymous Functions	131

12.9.1	Function Handles	131
12.9.2	Anonymous Functions	131
12.9.3	Inline Functions	132
12.10	Commands	133
12.11	Organization of Functions Distributed with Octave	134
13	Errors and Warnings	137
13.1	Handling Errors	137
13.1.1	Raising Errors	137
13.1.2	Catching Errors	139
13.2	Handling Warnings	141
13.2.1	Issuing Warnings	141
13.2.2	Enabling and Disabling Warnings	142
14	Debugging	147
14.1	Entering Debug Mode	147
14.2	Breakpoints	147
14.3	Debug Mode	149
15	Input and Output	151
15.1	Basic Input and Output	151
15.1.1	Terminal Output	151
15.1.1.1	Paging Screen Output	154
15.1.2	Terminal Input	155
15.1.3	Simple File I/O	156
15.1.3.1	Saving Data on Unexpected Exits	160
15.1.4	Rational Approximations	161
15.2	C-Style I/O Functions	162
15.2.1	Opening and Closing Files	162
15.2.2	Simple Output	164
15.2.3	Line-Oriented Input	164
15.2.4	Formatted Output	165
15.2.5	Output Conversion for Matrices	166
15.2.6	Output Conversion Syntax	167
15.2.7	Table of Output Conversions	167
15.2.8	Integer Conversions	168
15.2.9	Floating-Point Conversions	169
15.2.10	Other Output Conversions	170
15.2.11	Formatted Input	170
15.2.12	Input Conversion Syntax	171
15.2.13	Table of Input Conversions	172
15.2.14	Numeric Input Conversions	172
15.2.15	String Input Conversions	173
15.2.16	Binary I/O	173
15.2.17	Temporary Files	176
15.2.18	End of File and Errors	177
15.2.19	File Positioning	177

16	Plotting	179
16.1	Two-Dimensional Plotting	179
16.2	Specialized Two-Dimensional Plots	182
16.3	Three-Dimensional Plotting	187
16.4	Manipulating Existing Plots	188
16.5	Plot Annotations	190
16.6	Multiple Plots on One Page	191
16.7	Multiple Plot Windows	192
16.8	Printing Plots	192
16.9	Test Plotting Functions	193
16.10	Interaction with <code>gnuplot</code>	194
17	Matrix Manipulation	195
17.1	Finding Elements and Checking Conditions	195
17.2	Rearranging Matrices	197
17.3	Applying a Function to an Array	203
17.4	Special Utility Matrices	204
17.5	Famous Matrices	209
18	Arithmetic	213
18.1	Utility Functions	213
18.2	Complex Arithmetic	218
18.3	Trigonometry	219
18.4	Sums and Products	222
18.5	Special Functions	223
18.6	Coordinate Transformations	227
18.7	Mathematical Constants	227
19	Linear Algebra	231
19.1	Techniques used for Linear Algebra	231
19.2	Basic Matrix Functions	231
19.3	Matrix Factorizations	235
19.4	Functions of a Matrix	240
20	Nonlinear Equations	243

21	Sparse Matrices	245
21.1	The Creation and Manipulation of Sparse Matrices	245
21.1.1	Storage of Sparse Matrices	245
21.1.2	Creating Sparse Matrices	246
21.1.3	Finding out Information about Sparse Matrices	253
21.1.4	Basic Operators and Functions on Sparse Matrices	257
21.1.4.1	Sparse Functions	257
21.1.4.2	The Return Types of Operators and Functions	258
21.1.4.3	Mathematical Considerations	258
21.2	Linear Algebra on Sparse Matrices	265
21.3	Iterative Techniques applied to sparse matrices	271
21.4	Real Life Example of the use of Sparse Matrices	275
22	Numerical Integration	279
22.1	Functions of One Variable	279
22.2	Orthogonal Collocation	281
22.3	Functions of Multiple Variables	281
23	Differential Equations	283
23.1	Ordinary Differential Equations	283
23.2	Differential-Algebraic Equations	285
24	Optimization	293
24.1	Linear Programming	293
24.2	Quadratic Programming	299
24.3	Nonlinear Programming	300
24.4	Linear Least Squares	301
25	Statistics	303
25.1	Descriptive Statistics	303
25.2	Basic Statistical Functions	306
25.3	Statistical Plots	309
25.4	Tests	310
25.5	Models	317
25.6	Distributions	318
25.7	Random Number Generation	325
26	Financial Functions	331
27	Sets	333
27.1	Set Operations	333
28	Polynomial Manipulations	335

29	Interpolation	341
29.1	One-dimensional Interpolation	341
29.2	Multi-dimensional Interpolation	345
30	Geometry	349
31	Control Theory	351
31.1	System Data Structure	351
31.1.1	Variables common to all OCST system formats	352
31.1.2	tf format variables	352
31.1.3	zp format variables	352
31.1.4	ss format variables	353
31.2	System Construction and Interface Functions	353
31.2.1	Finite impulse response system interface functions	353
31.2.2	State space system interface functions	354
31.2.3	Transfer function system interface functions	359
31.2.4	Zero-pole system interface functions	361
31.2.5	Data structure access functions	362
31.3	System display functions	367
31.4	Block Diagram Manipulations	368
31.5	Numerical Functions	376
31.6	System Analysis-Properties	380
31.7	System Analysis-Time Domain	385
31.8	System Analysis-Frequency Domain	389
31.9	Controller Design	393
31.10	Miscellaneous Functions (Not yet properly filed/documentated)	401
32	Signal Processing	405
33	Image Processing	415
33.1	Loading and Saving Images	415
33.2	Displaying Images	416
33.3	Representing Images	417
33.4	Plotting on top of Images	421
33.5	Color Conversion	421
34	Audio Processing	423
35	Quaternions	425

36	System Utilities	427
36.1	Timing Utilities	427
36.2	Filesystem Utilities	435
36.3	File Archiving Utilities	441
36.4	Networking Utilities	442
36.5	Controlling Subprocesses	443
36.6	Process, Group, and User IDs	448
36.7	Environment Variables	448
36.8	Current Working Directory	448
36.9	Password Database Functions	449
36.10	Group Database Functions	450
36.11	System Information	451
36.12	Hashing Functions	453
37	Packages	455
37.1	Installing and Removing Packages	455
37.2	Using Packages	456
37.3	Administrating Packages	456
37.4	Creating Packages	456
37.4.1	The DESCRIPTION File	458
37.4.2	The INDEX file	459
37.4.3	PKG_ADD and PKG_DEL directives	460
Appendix A Dynamically Linked Functions		
	461
A.1	Oct-Files	461
A.1.1	Getting Started with Oct-Files	461
A.1.2	Matrices and Arrays in Oct-Files	464
A.1.3	Character Strings in Oct-Files	467
A.1.4	Cell Arrays in Oct-Files	468
A.1.5	Structures in Oct-Files	469
A.1.6	Sparse Matrices in Oct-Files	471
A.1.6.1	The Differences between the Array and Sparse Classes	471
A.1.6.2	Creating Sparse Matrices in Oct-Files	472
A.1.6.3	Using Sparse Matrices in Oct-Files	475
A.1.7	Accessing Global Variables in Oct-Files	476
A.1.8	Calling Octave Functions from Oct-Files	477
A.1.9	Calling External Code from Oct-Files	479
A.1.10	Allocating Local Memory in Oct-Files	481
A.1.11	Input Parameter Checking in Oct-Files	481
A.1.12	Exception and Error Handling in Oct-Files	483
A.1.13	Documentation and Test of Oct-Files	484
A.2	Mex-Files	485
A.2.1	Getting Started with Mex-Files	485
A.2.2	Working with Matrices and Arrays in Mex-Files	487
A.2.3	Character Strings in Mex-Files	490

A.2.4	Cell Arrays with Mex-Files	490
A.2.5	Structures with Mex-Files	491
A.2.6	Sparse Matrices with Mex-Files	494
A.2.7	Calling Other Functions in Mex-Files	497
A.3	Standalone Programs	498
Appendix B	Test and Demo Functions	501
B.1	Test Functions	501
B.2	Demonstration Functions	505
Appendix C	Tips and Standards	509
C.1	Writing Clean Octave Programs	509
C.2	Tips for Making Code Run Faster	509
C.3	Tips on Writing Comments	510
C.4	Conventional Headers for Octave Functions	510
C.5	Tips for Documentation Strings	512
Appendix D	Known Causes of Trouble	519
D.1	Actual Bugs We Haven't Fixed Yet	519
D.2	Reporting Bugs	519
D.3	Have You Found a Bug?	520
D.4	Where to Report Bugs	520
D.5	How to Report Bugs	520
D.6	Sending Patches for Octave	522
D.7	How To Get Help with Octave	523
Appendix E	Installing Octave	525
E.1	Installation Problems	527
Appendix F	Emacs Octave Support	531
F.1	Installing EOS	531
F.2	Using Octave Mode	531
F.3	Running Octave From Within Emacs	535
F.4	Using the Emacs Info Reader for Octave	536
Appendix G	GNU GENERAL PUBLIC LICENSE	539
Preamble		539
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION		540
Appendix: How to Apply These Terms to Your New Programs		544
Concept Index		545
Variable Index		549

Function Index 551

Operator Index 563

Preface

Octave was originally intended to be companion software for an undergraduate-level textbook on chemical reactor design being written by James B. Rawlings of the University of Wisconsin-Madison and John G. Ekerdt of the University of Texas.

Clearly, Octave is now much more than just another ‘courseware’ package with limited utility beyond the classroom. Although our initial goals were somewhat vague, we knew that we wanted to create something that would enable students to solve realistic problems, and that they could use for many things other than chemical reactor design problems.

There are those who would say that we should be teaching the students Fortran instead, because that is the computer language of engineering, but every time we have tried that, the students have spent far too much time trying to figure out why their Fortran code crashes and not enough time learning about chemical engineering. With Octave, most students pick up the basics quickly, and are using it confidently in just a few hours.

Although it was originally intended to be used to teach reactor design, it has been used in several other undergraduate and graduate courses in the Chemical Engineering Department at the University of Texas, and the math department at the University of Texas has been using it for teaching differential equations and linear algebra as well. If you find it useful, please let us know. We are always interested to find out how Octave is being used in other places.

Virtually everyone thinks that the name Octave has something to do with music, but it is actually the name of a former professor of mine who wrote a famous textbook on chemical reaction engineering, and who was also well known for his ability to do quick ‘back of the envelope’ calculations. We hope that this software will make it possible for many people to do more ambitious computations just as easily.

Everyone is encouraged to share this software with others under the terms of the GNU General Public License (see [Appendix G \[Copying\]](#), page 539) as described at the beginning of this manual. You are also encouraged to help make Octave more useful by writing and contributing additional functions for it, and by reporting any problems you may have.

Acknowledgements

Many people have already contributed to Octave’s development. The following people have helped write parts of Octave or helped out in various other ways (listed alphabetically).

Andy Adler	Joel Andersson	Muthiah Annamalai
Shai Ayal	Roger Banks	Ben Barrowes
Alexander Barth	David Bateman	Heinz Bauschke
Karl Berry	David Billingham	Don Bindner
Jakub Bogusz	Marcus Brinkmann	Remy Bruno
Marco Caliri	Daniel Calvelo	John C. Campbell
Jean-Francois Cardoso	Joao Cardoso	Larrie Carr
David Castelow	Vincent Cautaearts	Clinton Chee
Albert Chin-A-Young	J. D. Cole	Martin Costabel
Michael Creel	Jeff Cunningham	Martin Dalecki
Jorge Barros de Abreu	Philippe Defert	Bill Denney
David M. Doolin	Pascal A. Dupuis	John W. Eaton

Dirk Eddelbuettel	Paul Eggert	Stephen Eglén
Peter Ekberg	Rolf Fabian	Stephen Fegan
Ramon Garcia Fernandez	Torsten Finke	Jose Daniel Munoz Frias
Castor Fu	Eduardo Galleste	Walter Gautschi
Klaus Gebhardt	Driss Ghaddab	Nicolo Giorgetti
Michael Goffioul	Glenn Golden	Tomislav Goles
Keith Goodman	Etienne Grossmann	Peter Gustafson
Kai Habel	William P. Y. Hadisoese	Benjamin Hall
Kim Hansen	Soren Hauberg	Daniel Heiserer
Yozo Hida	Roman Hodek	A. Scottedward Hodel
Richard Allan Holcombe	Tom Holroyd	David Hoover
Kurt Hornik	Christopher Hulbert	Cyril Humbert
Teemu Ikonen	Alan W. Irwin	Geoff Jacobsen
Mats Jansson	Cai Jianming	Steven G. Johnson
Heikki Junes	Atsushi Kajita	Mohamed Kamoun
Lute Kamstra	Joel Keay	Mumit Khan
Paul Kienzie	Aaron A. King	Arno J. Klaassen
Geoffrey Knauth	Heine Kolltveit	Ken Kouno
Oyvind Kristiansen	Piotr Krzyzanowski	Volker Kuhlmann
Mirosław Kwasniak	Rafael Laboissiere	Kai Labusch
Claude Lacoursiere	Walter Landry	Bill Lash
Dirk Laurie	Maurice LeBrun	Friedrich Leisch
Benjamin Lindner	Ross Lippert	David Livings
Erik de Castro Lopo	Massimo Lorenzin	Hoxide Ma
James Macnicol	Jens-Uwe Mager	Ricardo Marranita
Orestes Mas	Makoto Matsumoto	Laurent Mazet
G. D. McBain	Stefan Monnier	Antoine Moreau
Kai P. Mueller	Victor Munoz	Carmen Navarrete
Todd Neal	Al Niessner	Rick Niles
Takuji Nishimura	Eric Norum	Michael O'Brien
Thorsten Ohl	Arno Onken	Luis F. Ortiz
Scott Pakin	Gabriele Pannocchia	Sylvain Pelissier
Per Persson	Jim Peterson	Danilo Piazzalunga
Nicholas Piper	Hans Ekkehard Plesser	Tom Poage
Orion Poplawski	Ondrej Popp	Jef Poskanzer
Francesco Potorti	James B. Rawlings	Eric S. Raymond
Balint Reczey	Michael Reifenger	Petter Risholm
Matthew W. Roberts	Andrew Ross	Mark van Rossum
Kevin Ruland	Olli Saarela	Toni Saarela
Juhani Saastamoinen	Ben Sapp	Alois Schloegl
Michel D. Schmid	Nicol N. Schraudolph	Sebastian Schubert
Ludwig Schwardt	Daniel J. Sebald	Dmitri A. Sergatskov
Baylis Shanks	Joseph P. Skudlarek	John Smith
Julius Smith	Shan G. Smith	Joerg Specht
Quentin H. Spencer	Christoph Spiel	Richard Stallman
Russell Standish	Doug Stewart	Thomas Stuart
Ariel Tankus	Georg Thimm	Duncan Temple Lang

Olaf Till	Thomas Treichl	Utkarsh Upadhyay
Stefan van der Walt	Peter Van Wieren	James R. Van Zandt
Gregory Vanuxem	Ivana Varekova	Thomas Walter
Olaf Weber	Thomas Weber	Bob Weigel
Andreas Weingessel	Michael Weitzel	Fook Fah Yap
Michael Zeising	Federico Zenith	Alex Zvoleff

Special thanks to the following people and organizations for supporting the development of Octave:

- The National Science Foundation, through grant numbers CTS-0105360, CTS-9708497, CTS-9311420, and CTS-8957123.
- The industrial members of the Texas-Wisconsin Modeling and Control Consortium ([TWMCC](#)).
- The Paul A. Elfers Endowed Chair in Chemical Engineering at the University of Wisconsin-Madison.
- Digital Equipment Corporation, for an equipment grant as part of their External Research Program.
- Sun Microsystems, Inc., for an Academic Equipment grant.
- International Business Machines, Inc., for providing equipment as part of a grant to the University of Texas College of Engineering.
- Texaco Chemical Company, for providing funding to continue the development of this software.
- The University of Texas College of Engineering, for providing a Challenge for Excellence Research Supplement, and for providing an Academic Development Funds grant.
- The State of Texas, for providing funding through the Texas Advanced Technology Program under Grant No. 003658-078.
- Noel Bell, Senior Engineer, Texaco Chemical Company, Austin Texas.
- John A. Turner, Group Leader, Continuum Dynamics (CCS-2), Los Alamos National Laboratory, for registering the [octave.org](#) domain name.
- James B. Rawlings, Professor, University of Wisconsin-Madison, Department of Chemical and Biological Engineering.
- Richard Stallman, for writing GNU.

This project would not have been possible without the GNU software used in and used to produce Octave.

How You Can Contribute to Octave

There are a number of ways that you can contribute to help make Octave a better system. Perhaps the most important way to contribute is to write high-quality code for solving new problems, and to make your code freely available for others to use.

If you find Octave useful, consider providing additional funding to continue its development. Even a modest amount of additional funding could make a significant difference in the amount of time that is available for development and support.

If you cannot provide funding or contribute code, you can still help make Octave better and more reliable by reporting any bugs you find and by offering suggestions for ways to

improve Octave. See [Appendix D \[Trouble\]](#), page 519, for tips on how to write useful bug reports.

Distribution

Octave is *free* software. This means that everyone is free to use it and free to redistribute it on certain conditions. Octave is not in the public domain. It is copyrighted and there are restrictions on its distribution, but the restrictions are designed to ensure that others will have the same freedom to use and redistribute Octave that you have. The precise conditions can be found in the GNU General Public License that comes with Octave and that also appears in [Appendix G \[Copying\]](#), page 539.

Octave is available on CD-ROM with various collections of other free software, and from the Free Software Foundation. Ordering a copy of Octave from the Free Software Foundation helps to fund the development of more free software. For more information, write to

Free Software Foundation
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301-1307
USA

Octave is also available on the Internet from <ftp://ftp.octave.org/pub/octave>, and additional information is available from <http://www.octave.org>.

1 A Brief Introduction to Octave

This manual documents how to install, run, and use GNU Octave, and how to report bugs.

GNU Octave is a high-level language, primarily intended for numerical computations. It provides a convenient command line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments. It may also be used as a batch-oriented language.

GNU Octave is also freely redistributable software. You may redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation. The GPL is included in this manual in [Appendix G \[Copying\]](#), page 539.

This document corresponds to Octave version 2.9.13.

1.1 Running Octave

On most systems, the way to invoke Octave is with the shell command ‘octave’. Octave displays an initial message and then a prompt indicating it is ready to accept input. You can begin typing Octave commands immediately afterward.

If you get into trouble, you can usually interrupt Octave by typing *Control-C* (usually written *C-c* for short). *C-c* gets its name from the fact that you type it by holding down **CTRL** and then pressing **C**. Doing this will normally return you to Octave’s prompt.

To exit Octave, type *quit*, or *exit* at the Octave prompt.

On systems that support job control, you can suspend Octave by sending it a SIGTSTP signal, usually by typing *C-z*.

1.2 Simple Examples

The following chapters describe all of Octave’s features in detail, but before doing that, it might be helpful to give a sampling of some of its capabilities.

If you are new to Octave, I recommend that you try these examples to begin learning Octave by using it. Lines marked with ‘octave:13>’ are lines you type, ending each with a carriage return. Octave will respond with an answer, or by displaying a graph.

1.2.1 Creating a Matrix

To create a new matrix and store it in a variable so that it you can refer to it later, type the command

```
octave:1> A = [ 1, 1, 2; 3, 5, 8; 13, 21, 34 ]
```

Octave will respond by printing the matrix in neatly aligned columns. Ending a command with a semicolon tells Octave to not print the result of a command. For example

```
octave:2> B = rand (3, 2);
```

will create a 3 row, 2 column matrix with each element set to a random value between zero and one.

To display the value of any variable, simply type the name of the variable. For example, to display the value stored in the matrix B, type the command

```
octave:3> B
```

1.2.2 Matrix Arithmetic

Octave has a convenient operator notation for performing matrix arithmetic. For example, to multiply the matrix `a` by a scalar value, type the command

```
octave:4> 2 * A
```

To multiply the two matrices `a` and `b`, type the command

```
octave:5> A * B
```

and to form the matrix product $A^T A$, type the command

```
octave:6> A' * A
```

1.2.3 Solving Linear Equations

To solve the set of linear equations $Ax = b$, use the left division operator, '`\`':

```
octave:7> A \ b
```

This is conceptually equivalent to $A^{-1}b$, but avoids computing the inverse of a matrix directly.

If the coefficient matrix is singular, Octave will print a warning message and compute a minimum norm solution.

1.2.4 Integrating Differential Equations

Octave has built-in functions for solving nonlinear differential equations of the form

$$\frac{dx}{dt} = f(x, t), \quad x(t = t_0) = x_0$$

For Octave to integrate equations of this form, you must first provide a definition of the function $f(x, t)$. This is straightforward, and may be accomplished by entering the function body directly on the command line. For example, the following commands define the right hand side function for an interesting pair of nonlinear differential equations. Note that while you are entering a function, Octave responds with a different prompt, to indicate that it is waiting for you to complete your input.

```
octave:8> function xdot = f (x, t)
>
>   r = 0.25;
>   k = 1.4;
>   a = 1.5;
>   b = 0.16;
>   c = 0.9;
>   d = 0.8;
>
>   xdot(1) = r*x(1)*(1 - x(1)/k) - a*x(1)*x(2)/(1 + b*x(1));
>   xdot(2) = c*a*x(1)*x(2)/(1 + b*x(1)) - d*x(2);
>
> endfunction
```

Given the initial condition

```
x0 = [1; 2];
```

and the set of output times as a column vector (note that the first output time corresponds to the initial condition given above)

```
t = linspace (0, 50, 200)';
```

it is easy to integrate the set of differential equations:

```
x = lsode ("f", x0, t);
```

The function `lsode` uses the Livermore Solver for Ordinary Differential Equations, described in A. C. Hindmarsh, *ODEPACK, a Systematized Collection of ODE Solvers*, in: Scientific Computing, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, 1983, pages 55–64.

1.2.5 Producing Graphical Output

To display the solution of the previous example graphically, use the command

```
plot (t, x)
```

If you are using a graphical user interface, Octave will automatically create a separate window to display the plot.

To save a plot once it has been displayed on the screen, use the `print` command. For example,

```
print -deps foo.eps
```

will create a file called ‘foo.eps’ that contains a rendering of the current plot. The command

```
help print
```

explains more options for the `print` command and provides a list of additional output file formats.

1.2.6 Editing What You Have Typed

At the Octave prompt, you can recall, edit, and reissue previous commands using Emacs- or vi-style editing commands. The default keybindings use Emacs-style commands. For example, to recall the previous command, press **Control-p** (usually written **C-p** for short). Doing this will normally bring back the previous line of input. **C-n** will bring up the next line of input, **C-b** will move the cursor backward on the line, **C-f** will move the cursor forward on the line, etc.

A complete description of the command line editing capability is given in this manual in [Section 2.4 \[Command Line Editing\]](#), page 19.

1.2.7 Help and Documentation

Octave has an extensive help facility. The same documentation that is available in printed form is also available from the Octave prompt, because both forms of the documentation are created from the same input file.

In order to get good help you first need to know the name of the command that you want to use. This name of the function may not always be obvious, but a good place to start is to just type `help`. This will show you all the operators, reserved words, functions, built-in variables, and function files. An alternative is to search the documentation using the `lookfor` function. This function is described in [Section 2.3 \[Getting Help\]](#), page 17.

Once you know the name of the function you wish to use, you can get more help on the function by simply including the name as an argument to `help`. For example,

```
help plot
```

will display the help text for the `plot` function.

Octave sends output that is too long to fit on one screen through a pager like `less` or `more`. Type a `(RET)` to advance one line, a `(SPC)` to advance one page, and `(q)` to exit the pager.

The part of Octave’s help facility that allows you to read the complete text of the printed manual from within Octave normally uses a separate program called `Info`. When you invoke `Info` you will be put into a menu driven program that contains the entire Octave manual. Help for using `Info` is provided in this manual in [Section 2.3 \[Getting Help\]](#), page 17.

1.3 Conventions

This section explains the notational conventions that are used in this manual. You may want to skip this section and refer back to it later.

1.3.1 Fonts

Examples of Octave code appear in this font or form: `svd(a)`. Names that represent arguments or metasyntactic variables appear in this font or form: *first-number*. Commands that you type at the shell prompt sometimes appear in this font or form: ‘`octave --no-init-file`’. Commands that you type at the Octave prompt sometimes appear in this font or form: *foo --bar --baz*. Specific keys on your keyboard appear in this font or form: `(ANY)`.

1.3.2 Evaluation Notation

In the examples in this manual, results from expressions that you evaluate are indicated with ‘ \Rightarrow ’. For example,

```
sqrt (2)
 $\Rightarrow$  1.4142
```

You can read this as “`sqrt (2)` evaluates to 1.4142”.

In some cases, matrix values that are returned by expressions are displayed like this

```
[1, 2; 3, 4] == [1, 3; 2, 4]
 $\Rightarrow$  [ 1, 0; 0, 1 ]
```

and in other cases, they are displayed like this

```
eye (3)
 $\Rightarrow$   1  0  0
      0  1  0
      0  0  1
```

in order to clearly show the structure of the result.

Sometimes to help describe one expression, another expression is shown that produces identical results. The exact equivalence of expressions is indicated with ‘ \equiv ’. For example,

```

rot90 ([1, 2; 3, 4], -1)
≡
rot90 ([1, 2; 3, 4], 3)
≡
rot90 ([1, 2; 3, 4], 7)

```

1.3.3 Printing Notation

Many of the examples in this manual print text when they are evaluated. Examples in this manual indicate printed text with ‘`+`’. The value that is returned by evaluating the expression (here 1) is displayed with ‘`⇒`’ and follows on a separate line.

```

printf ("foo %s\n", "bar")
+ foo bar
⇒ 1

```

1.3.4 Error Messages

Some examples signal errors. This normally displays an error message on your terminal. Error messages are shown on a line starting with `error:`.

```

struct_elements ([1, 2; 3, 4])
error: struct_elements: wrong type argument 'matrix'

```

1.3.5 Format of Descriptions

Functions, commands, and variables are described in this manual in a uniform format. The first line of a description contains the name of the item followed by its arguments, if any. The category—function, variable, or whatever—is printed next to the right margin. The description follows on succeeding lines, sometimes with examples.

1.3.5.1 A Sample Function Description

In a function description, the name of the function being described appears first. It is followed on the same line by a list of parameters. The names used for the parameters are also used in the body of the description.

Here is a description of an imaginary function `foo`:

```

foo (x, y, ...) [Function]
The function foo subtracts x from y, then adds the remaining arguments to the result.
If y is not supplied, then the number 19 is used by default.
foo (1, [3, 5], 3, 9)
⇒ [ 14, 16 ]
foo (5)
⇒ 14

```

More generally,

```

foo (w, x, y, ...)
≡
x - w + y + ...

```

Any parameter whose name contains the name of a type (e.g., *integer*, *integer1* or *matrix*) is expected to be of that type. Parameters named *object* may be of any type. Parameters

with other sorts of names (e.g., *new_file*) are discussed specifically in the description of the function. In some sections, features common to parameters of several functions are described at the beginning.

Functions in Octave may be defined in several different ways. The category name for functions may include another name that indicates the way that the function is defined. These additional tags include

Function File

The function described is defined using Octave commands stored in a text file. See [Section 12.7 \[Function Files\]](#), page 124.

Built-in Function

The function described is written in a language like C++, C, or Fortran, and is part of the compiled Octave binary.

Loadable Function

The function described is written in a language like C++, C, or Fortran. On systems that support dynamic linking of user-supplied functions, it may be automatically linked while Octave is running, but only if it is needed. See [Appendix A \[Dynamically Linked Functions\]](#), page 461.

Mapping Function

The function described works element-by-element for matrix and vector arguments.

1.3.5.2 A Sample Command Description

Command descriptions have a format similar to function descriptions, except that the word ‘Function’ is replaced by ‘Command’. Commands are functions that may be called without surrounding their arguments in parentheses. For example, here is the description for Octave’s `cd` command:

```
cd dir [Command]
chdir dir [Command]
```

Change the current working directory to *dir*. For example, `cd ~/octave` changes the current working directory to ‘~/octave’. If the directory does not exist, an error message is printed and the working directory is not changed.

1.3.5.3 A Sample Variable Description

A *variable* is a name that can hold a value. Although any variable can be set by the user, *built-in variables* typically exist specifically so that users can change them to alter the way Octave behaves (built-in variables are also sometimes called *user options*). Ordinary variables and built-in variables are described using a format like that for functions except that there are no arguments.

Here is a description of the imaginary variable `do_what_i_mean_not_what_i_say`.

```
do_what_i_mean_not_what_i_say [Built-in Variable]
```

If the value of this variable is nonzero, Octave will do what you actually wanted, even if you have typed a completely different and meaningless list of commands.

Other variable descriptions have the same format, but ‘Built-in Variable’ is replaced by ‘Variable’, for ordinary variables, or ‘Constant’ for symbolic constants whose values cannot be changed.

2 Getting Started

This chapter explains some of Octave's basic features, including how to start an Octave session, get help at the command prompt, edit the command line, and write Octave programs that can be executed as commands from your shell.

2.1 Invoking Octave from the Command Line

Normally, Octave is used interactively by running the program `'octave'` without any arguments. Once started, Octave reads commands from the terminal until you tell it to exit.

You can also specify the name of a file on the command line, and Octave will read and execute the commands from the named file and then exit when it is finished.

You can further control how Octave starts by using the command-line options described in the next section, and Octave itself can remind you of the options available. Type `'octave --help'` to display all available options and briefly describe their use (`'octave -h'` is a shorter equivalent).

2.1.1 Command Line Options

Here is a complete list of all the command line options that Octave accepts.

- `--debug`
- `-d` Enter parser debugging mode. Using this option will cause Octave's parser to print a lot of information about the commands it reads, and is probably only useful if you are actually trying to debug the parser.
- `--echo-commands`
- `-x` Echo commands as they are executed.
- `--eval code`
 Evaluate *code* and exit when done unless `--persist` is also specified.
- `--exec-path path`
 Specify the path to search for programs to run. The value of *path* specified on the command line will override any value of `OCTAVE_EXEC_PATH` found in the environment, but not any commands in the system or user startup files that set the built-in variable `EXEC_PATH`.
- `--help`
- `-h`
- `-?` Print short help message and exit.
- `--image-path path`
 Specify the path to search for images. The value of *path* specified on the command line will set the value of `IMAGE_PATH` found in the environment.
- `--info-file filename`
 Specify the name of the info file to use. The value of *filename* specified on the command line will override any value of `OCTAVE_INFO_FILE` found in the environment, but not any commands in the system or user startup files that use the `info_file` function.

`--info-program program`
Specify the name of the info program to use. The value of *program* specified on the command line will override any value of `OCTAVE_INFO_PROGRAM` found in the environment, but not any commands in the system or user startup files that use the `info_program` function.

`--interactive`
`-i` Force interactive behavior. This can be useful for running Octave via a remote shell command or inside an Emacs shell buffer. For another way to run Octave within Emacs, see [Appendix F \[Emacs\]](#), page 531.

`--no-history`
`-H` Disable command-line history.

`--no-init-file`
Don't read the `'~/.octaverc'` or `'~/.octaverc'` files.

`--no-line-editing`
Disable command-line editing.

`--no-site-file`
Don't read the site-wide `'octaverc'` file.

`--norc`
`-f` Don't read any of the system or user initialization files at startup. This is equivalent to using both of the options `--no-init-file` and `--no-site-file`.

`--path path`
`-p path` Specify the path to search for function files. The value of *path* specified on the command line will override any value of `OCTAVE_PATH` found in the environment, but not any commands in the system or user startup files that set the internal load path through one of the path functions.

`--persist`
Go to interactive mode after `--eval` or reading from a file named on the command line.

`--silent`
`--quiet`
`-q` Don't print the usual greeting and version message at startup.

`--traditional`
`--braindead`
For compatibility with MATLAB, set initial values for user-preferences to the following values

```

PS1                        = ">> "
PS2                        = ""
beep_on_error              = true
crash_dumps_octave_core    = false
default_save_options       = "-mat-binary"
fixed_point_format         = true
history_timestamp_format_string = "%!-- %D %I:%M %p --%"
page_screen_output         = false
print_empty_dimensions     = false

```

and disable the following warnings

```

Octave:fopen-file-in-path
Octave:function-name-clash
Octave:load-file-in-path

```

--verbose
-V Turn on verbose output.

--version
-v Print the program version number and exit.

file Execute commands from *file*. Exit when done unless **--persist** is also specified.

Octave also includes several built-in variables that contain information about the command line, including the number of arguments and all of the options.

argv () [Built-in Function]
 Return the command line arguments passed to Octave. For example, if you invoked Octave using the command

```
octave --no-line-editing --silent
```

argv would return a cell array of strings with the elements **--no-line-editing** and **--silent**.

If you write an executable Octave script, **argv** will return the list of arguments passed to the script. See [Section 2.6 \[Executable Octave Programs\], page 27](#), for an example of how to create an executable Octave script.

program_name () [Built-in Function]
 Return the last component of the value returned by **program_invocation_name**.
See also: **program_invocation_name**.

program_invocation_name () [Built-in Function]
 Return the name that was typed at the shell prompt to run Octave.
 If executing a script from the command line (e.g., **octave foo.m**) or using an executable Octave script, the program name is set to the name of the script. See [Section 2.6 \[Executable Octave Programs\], page 27](#), for an example of how to create an executable Octave script.

See also: **program_name**.

Here is an example of using these functions to reproduce Octave's command line.

```
printf ("%s", program_name ());
arg_list = argv ();
for i = 1:nargin
    printf (" %s", arg_list{i});
endfor
printf ("\n");
```

See [Section 9.1 \[Index Expressions\]](#), page 87, for an explanation of how to properly index arrays of strings and substrings in Octave, and See [Section 12.1 \[Defining Functions\]](#), page 117, for information about the variable `nargin`.

2.1.2 Startup Files

When Octave starts, it looks for commands to execute from the files in the following list. These files may contain any valid Octave commands, including function definitions.

octave-home/share/octave/site/m/startup/octaverc

Where *octave-home* is the directory in which all of Octave is installed (the default is `‘/usr’`). This file is provided so that changes to the default Octave environment can be made globally for all users at your site for all versions of Octave you have installed. Some care should be taken when making changes to this file, since all users of Octave at your site will be affected.

octave-home/share/octave/version/m/startup/octaverc

Where *octave-home* is the directory in which all of Octave is installed (the default is `‘/usr’`), and *version* is the version number of Octave. This file is provided so that changes to the default Octave environment can be made globally for all users for a particular version of Octave. Some care should be taken when making changes to this file, since all users of Octave at your site will be affected.

`~/.octaverc`

This file is normally used to make personal changes to the default Octave environment.

`.octaverc`

This file can be used to make changes to the default Octave environment for a particular project. Octave searches for this file in the current directory after it reads `‘~/.octaverc’`. Any use of the `cd` command in the `‘~/.octaverc’` file will affect the directory that Octave searches for the file `‘.octaverc’`.

If you start Octave in your home directory, commands from from the file `‘~/.octaverc’` will only be executed once.

A message will be displayed as each of the startup files is read if you invoke Octave with the `--verbose` option but without the `--silent` option.

2.2 Quitting Octave

`exit (status)`

[Built-in Function]

quit (*status*) [Built-in Function]

Exit the current Octave session. If the optional integer value *status* is supplied, pass that value to the operating system as the Octave's exit status. The default value is zero.

atexit (*fcn*) [Built-in Function]

Register a function to be called when Octave exits. For example,

```
function bye_bye ()
    disp ("Bye bye");
endfunction
atexit ("bye_bye");
```

will print the message "Bye bye" when Octave exits.

atexit (*fcn*, *flag*) [Built-in Function]

Register or unregister a function to be called when Octave exits, depending on *flag*. If *flag* is true, the function is registered, if *flag* is false, it is unregistered. For example, after registering the function `bye_bye` as above,

```
atexit ("bye_bye", false);
```

will remove the function from the list and Octave will not call the function `bye_bye` when it exits.

Note that **atexit** only removes the first occurrence of a function from the list, so if a function was placed in the list multiple times with **atexit**, it must also be removed from the list multiple times.

2.3 Commands for Getting Help

The entire text of this manual is available from the Octave prompt via the command **doc**. In addition, the documentation for individual user-written functions and variables is also available via the **help** command. This section describes the commands used for reading the manual and the documentation strings for user-supplied functions and variables. See [Section 12.7 \[Function Files\]](#), [page 124](#), for more information about how to document the functions you write.

help *name* [Command]

Display the help text for *name*. If invoked without any arguments, **help** prints a list of all the available operators and functions.

For example, the command **help help** prints a short message describing the **help** command.

The **help** command can give you information about operators, but not the comma and semicolons that are used as command separators. To get help for those, you must type **help comma** or **help semicolon**.

See also: **doc**, **which**, **lookfor**.

doc *function_name* [Command]

Display documentation for the function *function_name* directly from an on-line version of the printed manual, using the GNU Info browser. If invoked without any arguments, the manual is shown from the beginning.

For example, the command `doc rand` starts the GNU Info browser at this node in the on-line version of the manual.

Once the GNU Info browser is running, help for using it is available using the command `C-h`.

See also: `help`.

```
lookfor str [Command]
lookfor -all str [Command]
[fun, helpstring] = lookfor (str) [Function]
[fun, helpstring] = lookfor ('-all', str) [Function]
```

Search for the string *str* in all of the functions found in the function search path. By default `lookfor` searches for *str* in the first sentence of the help string of each function found. The entire help string of each function found in the path can be search if the `'-all'` argument is supplied. All searches are case insensitive.

Called with no output arguments, `lookfor` prints the list of matching functions to the terminal. Otherwise the output arguments *fun* and *helpstring* define the matching functions and the first sentence of each of their help strings.

Note that the ability of `lookfor` to correctly identify the first sentence of the help of the functions is dependent on the format of the functions help. All of the functions in Octave itself will correctly find the first sentence, but the same can not be guaranteed for other functions. Therefore the use of the `'-all'` argument might be necessary to find related functions that are not part of Octave.

See also: `help`, `which`.

The following function can be used to change which programs are used for displaying the documentation, and where the documentation can be found.

```
val = info_file () [Built-in Function]
old_val = info_file (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the name of the Octave info file. The default value is `"octave-home/info/octave.info"`, in which *octave-home* is the directory where all of Octave is installed.

See also: `info_program`, `doc`, `help`, `makeinfo_program`.

```
val = info_program () [Built-in Function]
old_val = info_program (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the name of the info program to run. The default initial value is `"octave-home/libexec/octave/version/exec/arch/info"■` in which *octave-home* is the directory where all of Octave is installed, *version* is the Octave version number, and *arch* is the system type (for example, `i686-pc-linux-gnu`). The default initial value may be overridden by the environment variable `OCTAVE_INFO_PROGRAM`, or the command line argument `--info-program NAME`.

See also: `info_file`, `doc`, `help`, `makeinfo_program`.

```
val = makeinfo_program () [Built-in Function]
old_val = makeinfo_program (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the name of the makeinfo program that Octave runs to format help text containing Texinfo markup commands. The default initial value is "makeinfo".

See also: info_file, info_program, doc, help.

```
val = suppress_verbose_help_message () [Built-in Function]
old_val = suppress_verbose_help_message (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave will add additional help information to the end of the output from the `help` command and usage messages for built-in commands.

2.4 Command Line Editing

Octave uses the GNU readline library to provide an extensive set of command-line editing and history features. Only the most common features are described in this manual. Please see The GNU Readline Library manual for more information.

To insert printing characters (letters, digits, symbols, etc.), simply type the character. Octave will insert the character at the cursor and advance the cursor forward.

Many of the command-line editing functions operate using control characters. For example, the character **Control-a** moves the cursor to the beginning of the line. To type **C-a**, hold down `(CTRL)` and then press `(a)`. In the following sections, control characters such as **Control-a** are written as **C-a**.

Another set of command-line editing functions use Meta characters. On some terminals, you type **M-u** by holding down `(META)` and pressing `(u)`. If your terminal does not have a `(META)` key, you can still type Meta characters using two-character sequences starting with **ESC**. Thus, to enter **M-u**, you could type `(ESC)(u)`. The **ESC** character sequences are also allowed on terminals with real Meta keys. In the following sections, Meta characters such as **Meta-u** are written as **M-u**.

2.4.1 Cursor Motion

The following commands allow you to position the cursor.

C-b	Move back one character.
C-f	Move forward one character.
<code>(DEL)</code>	Delete the character to the left of the cursor.
C-d	Delete the character underneath the cursor.
M-f	Move forward a word.
M-b	Move backward a word.
C-a	Move to the start of the line.
C-e	Move to the end of the line.
C-l	Clear the screen, reprinting the current line at the top.

C-<u>_</u>	
C-/	Undo the last thing that you did. You can undo all the way back to an empty line.
M-r	Undo all changes made to this line. This is like typing the ‘undo’ command enough times to get back to the beginning.

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. On most terminals, you can also use the arrow keys in place of **C-f** and **C-b** to move forward and backward.

Notice how **C-f** moves forward a character, while **M-f** moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

The function `clc` will allow you to clear the screen from within Octave programs.

2.4.2 Killing and Yanking

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* it back into the line. If the description for a command says that it ‘kills’ text, then you can be sure that you can get the text back in a different (or the same) place later.

Here is the list of commands for killing text.

C-k	Kill the text from the current cursor position to the end of the line.
M-d	Kill from the cursor to the end of the current word, or if between words, to the end of the next word.
M-<u>DEL</u>	Kill from the cursor to the start of the previous word, or if between words, to the start of the previous word.
C-w	Kill from the cursor to the previous whitespace. This is different than M-<u>DEL</u> because the word boundaries differ.

And, here is how to *yank* the text back into the line. Yanking means to copy the most-recently-killed text from the kill buffer.

C-y	Yank the most recently killed text back into the buffer at the cursor.
M-y	Rotate the kill-ring, and yank the new top. You can only do this if the prior command is C-y or M-y .

When you use a kill command, the text is saved in a *kill-ring*. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it in one clean sweep. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

2.4.3 Commands For Changing Text

The following commands can be used for entering characters that would otherwise have a special meaning (e.g., *TAB*, **C-q**, etc.), or for quickly correcting typing mistakes.

C-q	
C-v	Add the next character that you type to the line verbatim. This is how to insert things like C-q for example.

<code>M-<u>TAB</u></code>	Insert a tab character.
<code>C-t</code>	Drag the character before the cursor forward over the character at the cursor, also moving the cursor forward. If the cursor is at the end of the line, then transpose the two characters before it.
<code>M-t</code>	Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.
<code>M-u</code>	Uppercase the characters following the cursor to the end of the current (or following) word, moving the cursor to the end of the word.
<code>M-l</code>	Lowercase the characters following the cursor to the end of the current (or following) word, moving the cursor to the end of the word.
<code>M-c</code>	Uppercase the character following the cursor (or the beginning of the next word if the cursor is between words), moving the cursor to the end of the word.

2.4.4 Letting Readline Type For You

The following commands allow Octave to complete command and variable names for you.

`TAB` Attempt to do completion on the text before the cursor. Octave can complete the names of commands and variables.

`M-?` List the possible completions of the text before the cursor.

`val = completion_append_char ()` [Built-in Function]

`old_val = completion_append_char (new_val)` [Built-in Function]

Query or set the internal character variable that is appended to successful command-line completion attempts. The default value is " " (a single space).

`completion_matches (hint)` [Built-in Function]

Generate possible completions given *hint*.

This function is provided for the benefit of programs like Emacs which might be controlling Octave and handling user input. The current command number is not incremented when this function is called. This is a feature, not a bug.

2.4.5 Commands For Manipulating The History

Octave normally keeps track of the commands you type so that you can recall previous commands to edit or execute them again. When you exit Octave, the most recent commands you have typed, up to the number specified by the variable `history_size`, are saved in a file. When Octave starts, it loads an initial list of commands from the file named by the variable `history_file`.

Here are the commands for simple browsing and searching the history list.

`LFD`

`RET` Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`C-p` Move ‘up’ through the history list.

C-n	Move ‘down’ through the history list.
M-<	Move to the first line in the history.
M->	Move to the end of the input history, i.e., the line you are entering!
C-r	Search backward starting at the current line and moving ‘up’ through the history as necessary. This is an incremental search.
C-s	Search forward starting at the current line and moving ‘down’ through the history as necessary.

On most terminals, you can also use the arrow keys in place of **C-p** and **C-n** to move through the history list.

In addition to the keyboard commands for moving through the history list, Octave provides three functions for viewing, editing, and re-running chunks of commands from the history list.

history options [Command]

If invoked with no arguments, **history** displays a list of commands that you have executed. Valid options are:

- w file** Write the current history to the file *file*. If the name is omitted, use the default history file (normally ‘`~/.octave_hist`’).
- r file** Read the file *file*, replacing the current history list with its contents. If the name is omitted, use the default history file (normally ‘`~/.octave_hist`’).
- n** Only display the most recent *n* lines of history.
- q** Don’t number the displayed lines of history. This is useful for cutting and pasting commands if you are using the X Window System.

For example, to display the five most recent commands that you have typed without displaying line numbers, use the command **history -q 5**.

edit_history options [Command]

If invoked with no arguments, **edit_history** allows you to edit the history list using the editor named by the variable `EDITOR`. The commands to be edited are first copied to a temporary file. When you exit the editor, Octave executes the commands that remain in the file. It is often more convenient to use **edit_history** to define functions rather than attempting to enter them directly on the command line. By default, the block of commands is executed as soon as you exit the editor. To avoid executing any commands, simply delete all the lines from the buffer before exiting the editor.

The **edit_history** command takes two optional arguments specifying the history numbers of first and last commands to edit. For example, the command

```
edit_history 13
```

extracts all the commands from the 13th through the last in the history list. The command

```
edit_history 13 169
```

only extracts commands 13 through 169. Specifying a larger number for the first command than the last command reverses the list of commands before placing them

in the buffer to be edited. If both arguments are omitted, the previous command in the history list is used.

`run_history` [*first*] [*last*] [Command]
 Similar to `edit_history`, except that the editor is not invoked, and the commands are simply executed as they appear in the history list.

Octave also allows you customize the details of how and where the history is saved.

`val = history_file ()` [Built-in Function]
`old_val = history_file (new_val)` [Built-in Function]
 Query or set the internal variable that specifies the name of the file used to store command history. The default value is `"~/octave_hist"`, but may be overridden by the environment variable `OCTAVE_HISTFILE`.

See also: `history_size`, `saving_history`, `history_timestamp_format_string`.

`val = history_size ()` [Built-in Function]
`old_val = history_size (new_val)` [Built-in Function]
 Query or set the internal variable that specifies how many entries to store in the history file. The default value is 1024, but may be overridden by the environment variable `OCTAVE_HISTSIZE`.

See also: `history_file`, `history_timestamp_format`, `saving_history`.

`val = saving_history ()` [Built-in Function]
`old_val = saving_history (new_val)` [Built-in Function]
 Query or set the internal variable that controls whether commands entered on the command line are saved in the history file.

See also: `history_file`, `history_size`, `history_timestamp_format`.

`val = history_timestamp_format_string ()` [Built-in Function]
`old_val = history_timestamp_format_string (new_val)` [Built-in Function]
 Query or set the internal variable that specifies the format string for the comment line that is written to the history file when Octave exits. The format string is passed to `strftime`. The default value is

`"# Octave VERSION, %a %b %d %H:%M:%S %Y %Z <USER@HOST>"`

See also: `strftime`, `history_file`, `history_size`, `saving_history`.

`val = EDITOR ()` [Built-in Function]
`old_val = EDITOR (new_val)` [Built-in Function]
 Query or set the internal variable that specifies the editor to use with the `edit_history` command. If the environment variable `EDITOR` is set when Octave starts, its value is used as the default. Otherwise, `EDITOR` is set to `"emacs"`.

See also: `edit_history`.

2.4.6 Customizing readline

As mentioned earlier Octave uses the GNU readline library for command-line editing and history features. It is possible to customize how readline works through a configuration file.

`read_readline_init_file (file)` [Built-in Function]

Read the readline library initialization file *file*. If *file* is omitted, read the default initialization file (normally `~/inputrc`).

See [section “Readline Init File”](#) in *GNU Readline Library*, for details.

2.4.7 Customizing the Prompt

The following variables are available for customizing the appearance of the command-line prompts. Octave allows the prompt to be customized by inserting a number of backslash-escaped special characters that are decoded as follows:

<code>'\t'</code>	The time.
<code>'\d'</code>	The date.
<code>'\n'</code>	Begins a new line by printing the equivalent of a carriage return followed by a line feed.
<code>'\s'</code>	The name of the program (usually just <code>'octave'</code>).
<code>'\w'</code>	The current working directory.
<code>'\W'</code>	The basename of the current working directory.
<code>'\u'</code>	The username of the current user.
<code>'\h'</code>	The hostname, up to the first <code>'.'</code> .
<code>'\H'</code>	The hostname.
<code>'\#'</code>	The command number of this command, counting from when Octave starts.
<code>'\!'</code>	The history number of this command. This differs from <code>'\#'</code> by the number of commands in the history list when Octave starts.
<code>'\\$'</code>	If the effective UID is 0, a <code>'#'</code> , otherwise a <code>'\$'</code> .
<code>'\nnn'</code>	The character whose character code in octal is <i>nnn</i> .
<code>'\'</code>	A backslash.

`val = PS1 ()` [Built-in Function]

`old_val = PS1 (new_val)` [Built-in Function]

Query or set the primary prompt string. When executing interactively, Octave displays the primary prompt when it is ready to read a command.

The default value of the primary prompt string is `"\s:\#> "`. To change it, use a command like

```
octave:13> PS1 = "\\u@\\H> "
```

which will result in the prompt `'boris@kremvax> '` for the user `'boris'` logged in on the host `'kremvax.kgb.su'`. Note that two backslashes are required to enter a backslash into a double-quoted character string. See [Chapter 5 \[Strings\]](#), page 47.

See also: PS2, PS4.

`val = PS2 ()` [Built-in Function]
`old_val = PS2 (new_val)` [Built-in Function]

Query or set the secondary prompt string. The secondary prompt is printed when Octave is expecting additional input to complete a command. For example, if you are typing a for loop that spans several lines, Octave will print the secondary prompt at the beginning of each line after the first. The default value of the secondary prompt string is "> ".

See also: PS1, PS4.

`val = PS4 ()` [Built-in Function]
`old_val = PS4 (new_val)` [Built-in Function]

Query or set the character string used to prefix output produced when echoing commands when `echo_executing_commands` is enabled. The default value is "+ ". See [Section 2.1 \[Invoking Octave from the Command Line\], page 13](#), for a description of `--echo-commands`.

See also: `echo_executing_commands`, PS1, PS2.

2.4.8 Diary and Echo Commands

Octave's diary feature allows you to keep a log of all or part of an interactive session by recording the input you type and the output that Octave produces in a separate file.

diary options [Command]

Create a list of all commands *and* the output they produce, mixed together just as you see them on your terminal. Valid options are:

- on** Start recording your session in a file called 'diary' in your current working directory.
- off** Stop recording your session in the diary file.
- file** Record your session in the file named *file*.

Without any arguments, **diary** toggles the current diary state.

Sometimes it is useful to see the commands in a function or script as they are being evaluated. This can be especially helpful for debugging some kinds of problems.

echo options [Command]

Control whether commands are displayed as they are executed. Valid options are:

- on** Enable echoing of commands as they are executed in script files.
- off** Disable echoing of commands as they are executed in script files.
- on all** Enable echoing of commands as they are executed in script files and functions.
- off all** Disable echoing of commands as they are executed in script files and functions.

If invoked without any arguments, **echo** toggles the current echo state.

```
val = echo_executing_commands () [Built-in Function]
old_val = echo_executing_commands (new_val) [Built-in Function]
```

Query or set the internal variable that controls the echo state. It may be the sum of the following values:

- 1 Echo commands read from script files.
- 2 Echo commands from functions.
- 4 Echo commands read from command line.

More than one state can be active at once. For example, a value of 3 is equivalent to the command `echo on all`.

The value of `echo_executing_commands` is set by the `echo` command and the command line option `--echo-input`.

2.5 How Octave Reports Errors

Octave reports two kinds of errors for invalid programs.

A *parse error* occurs if Octave cannot understand something you have typed. For example, if you misspell a keyword,

```
octave:13> function y = f (x) y = x^2; endfunction
```

Octave will respond immediately with a message like this:

```
parse error:
```

```
function y = f (x) y = x^2; endfunction
               ^
```

For most parse errors, Octave uses a caret (^) to mark the point on the line where it was unable to make sense of your input. In this case, Octave generated an error message because the keyword `function` was misspelled. Instead of seeing '`function f`', Octave saw two consecutive variable names, which is invalid in this context. It marked the error at `y` because the first name by itself was accepted as valid input.

Another class of error message occurs at evaluation time. These errors are called *run-time errors*, or sometimes *evaluation errors* because they occur when your program is being *run*, or *evaluated*. For example, if after correcting the mistake in the previous function definition, you type

```
octave:13> f ()
```

Octave will respond with

```
error: 'x' undefined near line 1 column 24
error: evaluating expression near line 1, column 24
error: evaluating assignment expression near line 1, column 22
error: called from 'f'
```

This error message has several parts, and gives you quite a bit of information to help you locate the source of the error. The messages are generated from the point of the innermost error, and provide a traceback of enclosing expressions and function calls.

In the example above, the first line indicates that a variable named '`x`' was found to be undefined near line 1 and column 24 of some function or expression. For errors occurring

within functions, lines are counted from the beginning of the file containing the function definition. For errors occurring at the top level, the line number indicates the input line number, which is usually displayed in the prompt string.

The second and third lines in the example indicate that the error occurred within an assignment expression, and the last line of the error message indicates that the error occurred within the function `f`. If the function `f` had been called from another function, for example, `g`, the list of errors would have ended with one more line:

```
error: called from 'g'
```

These lists of function calls usually make it fairly easy to trace the path your program took before the error occurred, and to correct the error before trying again.

2.6 Executable Octave Programs

Once you have learned Octave, you may want to write self-contained Octave scripts, using the `#!` script mechanism. You can do this on GNU systems and on many Unix systems¹.

Self-contained Octave scripts are useful when you want to write a program which users can invoke without knowing that the program is written in the Octave language.

For example, you could create a text file named `hello`, containing the following lines:

```
#! octave-interpreter-name -qf
# a sample Octave program
printf ("Hello, world!\n");
```

(where *octave-interpreter-name* should be replaced with the full file name for your Octave binary). Note that this will only work if `#!` appears at the very beginning of the file. After making this file executable (with the `chmod` command), you can simply type:

```
hello
```

at the shell, and the system will arrange to run Octave as if you had typed:

```
octave hello
```

The line beginning with `#!` lists the full file name of an interpreter to be run, and an optional initial command line argument to pass to that interpreter. The operating system then runs the interpreter with the given argument and the full argument list of the executed program. The first argument in the list is the full file name of the Octave program. The rest of the argument list will either be options to Octave, or data files, or both. The `-qf` option is usually specified in stand-alone Octave programs to prevent them from printing the normal startup message, and to keep them from behaving differently depending on the contents of a particular user's `~/.octaverc` file. See [Section 2.1 \[Invoking Octave from the Command Line\]](#), page 13.

Note that some operating systems may place a limit on the number of characters that are recognized after `#!`. Also, the various shells/systems parse differently the arguments appearing in a `#!` line. The majority of them group together all the arguments in a string and pass it to the interpreter as a single argument. In this case, the following script:

```
#! octave-interpreter-name -q -f # comment
```

is equivalent to type at the command line:

¹ The `#!` mechanism works on Unix systems derived from Berkeley Unix, System V Release 4, and some System V Release 3 systems.

```
octave "-q -f # comment"
```

which would obviously produce an error message. Unfortunately, it is impossible for Octave to know whether it has been called from the command line or from a ‘#!’ script, so some care is needed when using the ‘#!’ mechanism.

Note that when Octave is started from an executable script, the built-in function `argv` returns a cell array containing the command line arguments passed to an executable Octave script, not the arguments passed to the Octave interpreter on the ‘#!’ line of the script. For example, the following program will reproduce the command line that is used to execute script, not ‘-qf’.

```
#!/bin/octave -qf
printf ("%s", program_name ());
arg_list = argv ();
for i = 1:nargin
    printf (" %s", arg_list{i});
endfor
printf ("\n");
```

2.7 Comments in Octave Programs

A *comment* is some text that is included in a program for the sake of human readers, and that is not really part of the program. Comments can explain what the program does, and how it works. Nearly all programming languages have provisions for comments, because programs are typically hard to understand without them.

In the Octave language, a comment starts with either the sharp sign character, ‘#’, or the percent symbol ‘%’ and continues to the end of the line. The Octave interpreter ignores the rest of a line following a sharp sign or percent symbol. For example, we could have put the following into the function `f`:

```
function xdot = f (x, t)

# usage: f (x, t)
#
# This function defines the right hand
# side functions for a set of nonlinear
# differential equations.

r = 0.25;
...
endfunction
```

The `help` command (see [Section 2.3 \[Getting Help\]](#), page 17) is able to find the first block of comments in a function (even those that are composed directly on the command line). This means that users of Octave can use the same commands to get help for built-in functions, and for functions that you have defined. For example, after defining the function `f` above, the command `help f` produces the output


```
usage: f (x, t)
```

```
This function defines the right hand  
side functions for a set of nonlinear  
differential equations.
```

Although it is possible to put comment lines into keyboard-composed throw-away Octave programs, it usually isn't very useful, because the purpose of a comment is to help you or another person understand the program at a later time.

3 Data Types

All versions of Octave include a number of built-in data types, including real and complex scalars and matrices, character strings, a data structure type, and an array that can contain all data types.

It is also possible to define new specialized data types by writing a small amount of C++ code. On some systems, new data types can be loaded dynamically while Octave is running, so it is not necessary to recompile all of Octave just to add a new type. See [Appendix A \[Dynamically Linked Functions\]](#), page 461, for more information about Octave’s dynamic linking capabilities. [Section 3.2 \[User-defined Data Types\]](#), page 32 describes what you must do to define a new data type for Octave.

typeinfo (*expr*) [Built-in Function]
 Return the type of the expression *expr*, as a string. If *EXPR* is omitted, return an array of strings containing all the currently installed data types.

3.1 Built-in Data Types

The standard built-in data types are real and complex scalars and matrices, ranges, character strings, a data structure type, and cell arrays. Additional built-in data types may be added in future versions. If you need a specialized data type that is not currently provided as a built-in type, you are encouraged to write your own user-defined data type and contribute it for distribution in a future release of Octave.

The data type of a variable can be determined and changed through the use of the following functions.

class (*expr*) [Built-in Function]
 Return the class of the expression *expr*, as a string.

isa (*x*, *class*) [Function File]
 Return true if *x* is a value from the class *class*.

cast (*val*, *type*) [Function File]
 Convert *val* to data type *type*.
See also: int8, uint8, int16, uint16, int32, uint32, int64, uint64, double.

3.1.1 Numeric Objects

Octave’s built-in numeric objects include real, complex, and integer scalars and matrices. All built-in floating point numeric data is currently stored as double precision numbers. On systems that use the IEEE floating point format, values in the range of approximately 2.2251×10^{-308} to 1.7977×10^{308} can be stored, and the relative precision is approximately 2.2204×10^{-16} . The exact values are given by the variables **realmin**, **realmax**, and **eps**, respectively.

Matrix objects can be of any size, and can be dynamically reshaped and resized. It is easy to extract individual rows, columns, or submatrices using a variety of powerful indexing features. See [Section 9.1 \[Index Expressions\]](#), page 87.

See [Chapter 4 \[Numeric Data Types\]](#), page 35, for more information.

3.1.2 Missing Data

It is possible to represent missing data explicitly in Octave using `NA` (short for “Not Available”). Missing data can only be represented when data is represented as floating point numbers. In this case missing data is represented as a special case of the representation of `NaN`.

`NA (x)` [Built-in Function]

`NA (n, m)` [Built-in Function]

`NA (n, m, k, ...)` [Built-in Function]

`NA (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all equal to the special constant used to designate missing values.

`isna (x)` [Mapping Function]

Return 1 for elements of `x` that are `NA` (missing) values and zero otherwise. For example,

```
isna ([13, Inf, NA, NaN])
⇒ [ 0, 0, 1, 0 ]
```

3.1.3 String Objects

A character string in Octave consists of a sequence of characters enclosed in either double-quote or single-quote marks. Internally, Octave currently stores strings as matrices of characters. All the indexing operations that work for matrix objects also work for strings.

See [Chapter 5 \[Strings\]](#), page 47, for more information.

3.1.4 Data Structure Objects

Octave’s data structure type can help you to organize related objects of different types. The current implementation uses an associative array with indices limited to strings, but the syntax is more like C-style structures.

See [Chapter 6 \[Data Structures\]](#), page 63, for more information.

3.1.5 Cell Array Objects

A Cell Array in Octave is general array that can hold any number of different data types.

See [Chapter 7 \[Cell Arrays\]](#), page 69, for more information.

3.2 User-defined Data Types

Someday I hope to expand this to include a complete description of Octave’s mechanism for managing user-defined data types. Until this feature is documented here, you will have to make do by reading the code in the ‘`ov.h`’, ‘`ops.h`’, and related files from Octave’s ‘`src`’ directory.

3.3 Object Sizes

The following functions allow you to determine the size of a variable or expression. These functions are defined for all objects. They return `-1` when the operation doesn’t make

sense. For example, Octave's data structure type doesn't have rows or columns, so the `rows` and `columns` functions return `-1` for structure arguments.

ndims (a) [Built-in Function]

Returns the number of dimensions of array *a*. For any array, the result will always be larger than or equal to 2. Trailing singleton dimensions are not counted.

columns (a) [Built-in Function]

Return the number of columns of *a*.

See also: `size`, `numel`, `rows`, `length`, `isscalar`, `isvector`, and `ismatrix`.

rows (a) [Built-in Function]

Return the number of rows of *a*.

See also: `size`, `numel`, `columns`, `length`, `isscalar`, `isvector`, `ismatrix`.

numel (a) [Built-in Function]

Returns the number of elements in the object *a*.

See also: `size`.

length (a) [Built-in Function]

Return the 'length' of the object *a*. For matrix objects, the length is the number of rows or columns, whichever is greater (this odd definition is used for compatibility with MATLAB).

size (a, n) [Built-in Function]

Return the number rows and columns of *a*.

With one input argument and one output argument, the result is returned in a row vector. If there are multiple output arguments, the number of rows is assigned to the first, and the number of columns to the second, etc. For example,

```
size ([1, 2; 3, 4; 5, 6])
⇒ [ 3, 2 ]
```

```
[nr, nc] = size ([1, 2; 3, 4; 5, 6])
⇒ nr = 3
⇒ nc = 2
```

If given a second argument, `size` will return the size of the corresponding dimension. For example

```
size ([1, 2; 3, 4; 5, 6], 2)
⇒ 2
```

returns the number of columns in the given matrix.

See also: `numel`.

isempty (a) [Built-in Function]

Return 1 if *a* is an empty matrix (either the number of rows, or the number of columns, or both are zero). Otherwise, return 0.

sizeof (*val*) [Built-in Function]

Return the size of *val* in bytes

size_equal (*a*, *b*, ...) [Built-in Function]

Return true if the dimensions of all arguments agree. Trailing singleton dimensions are ignored.

See also: `size`, `numel`.

squeeze (*x*) [Built-in Function]

Remove singleton dimensions from *x* and return the result.

4 Numeric Data Types

A *numeric constant* may be a scalar, a vector, or a matrix, and it may contain complex values.

The simplest form of a numeric constant, a scalar, is a single number that can be an integer, a decimal fraction, a number in scientific (exponential) notation, or a complex number. Note that by default numeric constants are represented within Octave in double-precision floating point format (complex constants are stored as pairs of double-precision floating point values). It is however possible to represent real integers as described in [Section 4.3 \[Integer Data Types\], page 40](#). Here are some examples of real-valued numeric constants, which all have the same value:

```
105
1.05e+2
1050e-1
```

To specify complex constants, you can write an expression of the form

```
3 + 4i
3.0 + 4.0i
0.3e1 + 40e-1i
```

all of which are equivalent. The letter ‘i’ in the previous example stands for the pure imaginary constant, defined as $\sqrt{-1}$.

For Octave to recognize a value as the imaginary part of a complex constant, a space must not appear between the number and the ‘i’. If it does, Octave will print an error message, like this:

```
octave:13> 3 + 4 i

parse error:

  3 + 4 i
      ^
```

You may also use ‘j’, ‘I’, or ‘J’ in place of the ‘i’ above. All four forms are equivalent.

double (x) [Built-in Function]
Convert x to double precision type.

single (val) [Function File]
Convert the numeric value *val* to single precision.

Note: this function currently returns its argument converted to double precision because Octave does not yet have a single-precision numeric data type.

complex (val) [Built-in Function]
complex (re, im) [Built-in Function]
Convert x to a complex value.

4.1 Matrices

It is easy to define a matrix of values in Octave. The size of the matrix is determined automatically, so it is not necessary to explicitly state the dimensions. The expression

```
a = [1, 2; 3, 4]
```

results in the matrix

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Elements of a matrix may be arbitrary expressions, provided that the dimensions all make sense when combining the various pieces. For example, given the above matrix, the expression

```
[ a, a ]
```

produces the matrix

```
ans =
```

```
1 2 1 2
3 4 3 4
```

but the expression

```
[ a, 1 ]
```

produces the error

```
error: number of rows must match near line 13, column 6
```

(assuming that this expression was entered as the first thing on line 13, of course).

Inside the square brackets that delimit a matrix expression, Octave looks at the surrounding context to determine whether spaces and newline characters should be converted into element and row separators, or simply ignored, so an expression like

```
a = [ 1 2
      3 4 ]
```

will work. However, some possible sources of confusion remain. For example, in the expression

```
[ 1 - 1 ]
```

the ‘-’ is treated as a binary operator and the result is the scalar 0, but in the expression

```
[ 1 -1 ]
```

the ‘-’ is treated as a unary operator and the result is the vector [1, -1]. Similarly, the expression

```
[ sin (pi) ]
```

will be parsed as

```
[ sin, (pi) ]
```

and will result in an error since the `sin` function will be called with no arguments. To get around this, you must omit the space between `sin` and the opening parenthesis, or enclose the expression in a set of parentheses:

```
[ (sin (pi)) ]
```

Whitespace surrounding the single quote character (‘’), used as a transpose operator and for delimiting character strings) can also cause confusion. Given `a = 1`, the expression


```
[ 1 a' ]
```

results in the single quote character being treated as a transpose operator and the result is the vector `[1, 1]`, but the expression

```
[ 1 a ' ]
```

produces the error message

```
error: unterminated string constant
```

because not doing so would cause trouble when parsing the valid expression

```
[ a 'foo' ]
```

For clarity, it is probably best to always use commas and semicolons to separate matrix elements and rows.

When you type a matrix or the name of a variable whose value is a matrix, Octave responds by printing the matrix in with neatly aligned rows and columns. If the rows of the matrix are too large to fit on the screen, Octave splits the matrix and displays a header before each section to indicate which columns are being displayed. You can use the following variables to control the format of the output.

```
val = output_max_field_width () [Built-in Function]
old_val = output_max_field_width (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the maximum width of a numeric output field.

See also: `format`, `output_precision`.

```
val = output_precision () [Built-in Function]
old_val = output_precision (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the minimum number of significant figures to display for numeric output.

See also: `format`, `output_max_field_width`.

It is possible to achieve a wide range of output styles by using different values of `output_precision` and `output_max_field_width`. Reasonable combinations can be set using the `format` function. See [Section 15.1 \[Basic Input and Output\]](#), page 151.

```
val = split_long_rows () [Built-in Function]
old_val = split_long_rows (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether rows of a matrix may be split when displayed to a terminal window. If the rows are split, Octave will display the matrix in a series of smaller pieces, each of which can fit within the limits of your terminal width and each set of rows is labeled so that you can easily see which columns are currently being displayed. For example:

```
octave:13> rand (2,10)
ans =

Columns 1 through 6:

    0.75883    0.93290    0.40064    0.43818    0.94958    0.16467
    0.75697    0.51942    0.40031    0.61784    0.92309    0.40201

Columns 7 through 10:

    0.90174    0.11854    0.72313    0.73326
    0.44672    0.94303    0.56564    0.82150
```

Octave automatically switches to scientific notation when values become very large or very small. This guarantees that you will see several significant figures for every value in a matrix. If you would prefer to see all values in a matrix printed in a fixed point format, you can set the built-in variable `fixed_point_format` to a nonzero value. But doing so is not recommended, because it can produce output that can easily be misinterpreted.

```
val = fixed_point_format () [Built-in Function]
old_val = fixed_point_format (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave will use a scaled format to print matrix values such that the largest element may be written with a single leading digit with the scaling factor is printed on the first line of output. For example,

```
octave:1> logspace (1, 7, 5)'
ans =

1.0e+07 *

0.00000
0.00003
0.00100
0.03162
1.00000
```

Notice that first value appears to be zero when it is actually 1. For this reason, you should be careful when setting `fixed_point_format` to a nonzero value.

4.1.1 Empty Matrices

A matrix may have one or both dimensions zero, and operations on empty matrices are handled as described by Carl de Boer in *An Empty Exercise*, SIGNUM, Volume 25, pages 2–6, 1990 and C. N. Nett and W. M. Haddad, in *A System-Theoretic Appropriate Realization of the Empty Matrix Concept*, IEEE Transactions on Automatic Control, Volume 38, Number 5, May 1993. Briefly, given a scalar s , an $m \times n$ matrix $M_{m \times n}$, and an $m \times n$ empty matrix $[]_{m \times n}$ (with either one or both dimensions equal to zero), the following are

true:

$$\begin{aligned}
 s \cdot []_{m \times n} &= []_{m \times n} \cdot s = []_{m \times n} \\
 []_{m \times n} + []_{m \times n} &= []_{m \times n} \\
 []_{0 \times m} \cdot M_{m \times n} &= []_{0 \times n} \\
 M_{m \times n} \cdot []_{n \times 0} &= []_{m \times 0} \\
 []_{m \times 0} \cdot []_{0 \times n} &= 0_{m \times n}
 \end{aligned}$$

By default, dimensions of the empty matrix are printed along with the empty matrix symbol, `[]`. The built-in variable `print_empty_dimensions` controls this behavior.

```

val = print_empty_dimensions () [Built-in Function]
old_val = print_empty_dimensions (new_val) [Built-in Function]
    Query or set the internal variable that controls whether the dimensions of empty
    matrices are printed along with the empty matrix symbol, []. For example, the
    expression
        zeros (3, 0)
    will print
        ans = [] (3x0)

```

Empty matrices may also be used in assignment statements as a convenient way to delete rows or columns of matrices. See [Section 9.6 \[Assignment Expressions\]](#), page 96.

When Octave parses a matrix expression, it examines the elements of the list to determine whether they are all constants. If they are, it replaces the list with a single matrix constant.

4.2 Ranges

A *range* is a convenient way to write a row vector with evenly spaced elements. A range expression is defined by the value of the first element in the range, an optional value for the increment between elements, and a maximum value which the elements of the range will not exceed. The base, increment, and limit are separated by colons (the `:` character) and may contain any arithmetic expressions and function calls. If the increment is omitted, it is assumed to be 1. For example, the range

```
1 : 5
```

defines the set of values `[1, 2, 3, 4, 5]`, and the range

```
1 : 3 : 5
```

defines the set of values `[1, 4]`.

Although a range constant specifies a row vector, Octave does *not* convert range constants to vectors unless it is necessary to do so. This allows you to write a constant like `'1 : 10000'` without using 80,000 bytes of storage on a typical 32-bit workstation.

Note that the upper (or lower, if the increment is negative) bound on the range is not always included in the set of values, and that ranges defined by floating point values can produce surprising results because Octave uses floating point arithmetic to compute the values in the range. If it is important to include the endpoints of a range and the number of elements is known, you should use the `linspace` function instead (see [Section 17.4 \[Special Utility Matrices\]](#), page 204).

When Octave parses a range expression, it examines the elements of the expression to determine whether they are all constants. If they are, it replaces the range expression with a single range constant.

4.3 Integer Data Types

Octave supports integer matrices as an alternative to using double precision. It is possible to use both signed and unsigned integers represented by 8, 16, 32, or 64 bits. It should be noted that most computations require floating point data, meaning that integers will often change type when involved in numeric computations. For this reason integers are most often used to store data, and not for calculations.

In general most integer matrices are created by casting existing matrices to integers. The following example shows how to cast a matrix into 32 bit integers.

```
float = rand (2, 2)
      ⇒ float = 0.37569    0.92982
                0.11962    0.50876
integer = int32 (float)
      ⇒ integer = 0    1
                0    1
```

As can be seen, floating point values are rounded to the nearest integer when converted.

isinteger (x) [Built-in Function]

Return true if x is an integer object (int8, uint8, int16, etc.). Note that **isinteger** (14) is false because numeric constants in are double precision floating point values.

See also: isreal, isnumeric, class, isa.

int8 (x) [Built-in Function]

Convert x to 8-bit integer type.

uint8 (x) [Built-in Function]

Convert x to unsigned 8-bit integer type.

int16 (x) [Built-in Function]

Convert x to 16-bit integer type.

uint16 (x) [Built-in Function]

Convert x to unsigned 16-bit integer type.

int32 (x) [Built-in Function]

Convert x to 32-bit integer type.

uint32 (x) [Built-in Function]

Convert x to unsigned 32-bit integer type.

int64 (x) [Built-in Function]

Convert x to 64-bit integer type.

uint64 (x) [Built-in Function]
 Convert *x* to unsigned 64-bit integer type.

intmax (type) [Built-in Function]
 Return the largest integer that can be represented in an integer type. The variable *type* can be

int8	signed 8-bit integer.
int16	signed 16-bit integer.
int32	signed 32-bit integer.
int64	signed 64-bit integer.
uint8	unsigned 8-bit integer.
uint16	unsigned 16-bit integer.
uint32	unsigned 32-bit integer.
uint64	unsigned 64-bit integer.

The default for *type* is **uint32**.

See also: **intmin**, **bitmax**.

intmin (type) [Built-in Function]
 Return the smallest integer that can be represented in an integer type. The variable *type* can be

int8	signed 8-bit integer.
int16	signed 16-bit integer.
int32	signed 32-bit integer.
int64	signed 64-bit integer.
uint8	unsigned 8-bit integer.
uint16	unsigned 16-bit integer.
uint32	unsigned 32-bit integer.
uint64	unsigned 64-bit integer.

The default for *type* is **uint32**.

See also: **intmax**, **bitmax**.

4.3.1 Integer Arithmetic

While many numerical computations can't be carried out in integers, Octave does support basic operations like addition and multiplication on integers. The operators **+**, **-**, **.***, and **./** works on integers of the same type. So, it is possible to add two 32 bit integers, but not to add a 32 bit integer and a 16 bit integer.

The arithmetic operations on integers are performed by casting the integer values to double precision values, performing the operation, and then re-casting the values back

to the original integer type. As the double precision type of Octave is only capable of representing integers with up to 53 bits of precision, it is not possible to perform arithmetic of the 64 bit integer types.

When doing integer arithmetic one should consider the possibility of underflow and overflow. This happens when the result of the computation can't be represented using the chosen integer type. As an example it is not possible to represent the result of $10 - 20$ when using unsigned integers. Octave makes sure that the result of integer computations is the integer that is closest to the true result. So, the result of $10 - 20$ when using unsigned integers is zero.

When doing integer division Octave will round the result to the nearest integer. This is different from most programming languages, where the result is often floored to the nearest integer. So, the result of `int32(5)./int32(8)` is 1.

4.4 Bit Manipulations

Octave provides a number of functions for the manipulation of numeric values on a bit by bit basis. The basic functions to set and obtain the values of individual bits are `bitset` and `bitget`.

```
x = bitset (a, n) [Function File]
x = bitset (a, n, v) [Function File]
```

Set or reset bit(s) *n* of unsigned integers in *a*. *v* = 0 resets and *v* = 1 sets the bits.
The lowest significant bit is: *n* = 1

```
dec2bin (bitset (10, 1))
⇒ 1011
```

See also: `bitand`, `bitor`, `bitxor`, `bitget`, `bitcmp`, `bitshift`, `bitmax`.

```
X = bitget (a,n) [Function File]
Return the status of bit(s) n of unsigned integers in a the lowest significant bit is n = 1.
```

```
bitget (100, 8:-1:1)
⇒ 0 1 1 0 0 1 0 0
```

See also: `bitand`, `bitor`, `bitxor`, `bitset`, `bitcmp`, `bitshift`, `bitmax`.

The arguments to all of Octave's bitwise operations can be scalar or arrays, except for `bitcmp`, whose *k* argument must a scalar. In the case where more than one argument is an array, then all arguments must have the same shape, and the bitwise operator is applied to each of the elements of the argument individually. If at least one argument is a scalar and one an array, then the scalar argument is duplicated. Therefore

```
bitget (100, 8:-1:1)
```

is the same as

```
bitget (100 * ones (1, 8), 8:-1:1)
```

It should be noted that all values passed to the bit manipulation functions of Octave are treated as integers. Therefore, even though the example for `bitset` above passes the floating point value 10, it is treated as the bits [1, 0, 1, 0] rather than the bits of the native floating point format representation of 10.

As the maximum number that can be represented by a number is important for bit manipulation, particularly when forming masks, Octave supplies the function `bitmax`.

`bitmax ()` [Built-in Function]
Return the largest integer that can be represented as a floating point value. On IEEE-754 compatible systems, `bitmax` is $2^{53} - 1$.

This is the double precision version of the functions `intmax`, previously discussed.

Octave also include the basic bitwise 'and', 'or' and 'exclusive or' operators.

`bitand (x, y)` [Built-in Function]
Return the bitwise AND of nonnegative integers. `x`, `y` must be in range `[0..bitmax]`
See also: `bitor`, `bitxor`, `bitset`, `bitget`, `bitcmp`, `bitshift`, `bitmax`.

`bitor (x, y)` [Built-in Function]
Return the bitwise OR of nonnegative integers. `x`, `y` must be in range `[0..bitmax]`
See also: `bitor`, `bitxor`, `bitset`, `bitget`, `bitcmp`, `bitshift`, `bitmax`.

`bitxor (x, y)` [Built-in Function]
Return the bitwise XOR of nonnegative integers. `x`, `y` must be in range `[0..bitmax]`
See also: `bitand`, `bitor`, `bitset`, `bitget`, `bitcmp`, `bitshift`, `bitmax`.

The bitwise 'not' operator is unary operator that performs a logical negation of each of the bits of the value. For this to make sense, the mask against which the value is negated must be defined. Octave's bitwise 'not' operator is `bitcmp`.

`bitcmp (a, k)` [Function File]
Return the `k`-bit complement of integers in `a`. If `k` is omitted `k = log2 (bitmax) + 1` is assumed.

```
bitcmp(7,4)
⇒ 8
dec2bin(11)
⇒ 1011
dec2bin(bitcmp(11, 6))
⇒ 110100
```

See also: `bitand`, `bitor`, `bitxor`, `bitset`, `bitget`, `bitcmp`, `bitshift`, `bitmax`.

Octave also includes the ability to left and right values bitwise.

`bitshift (a, k)` [Built-in Function]
`bitshift (a, k, n)` [Built-in Function]
Return a `k` bit shift of `n`- digit unsigned integers in `a`. A positive `k` leads to a left shift. A negative value to a right shift. If `n` is omitted it defaults to `log2(bitmax)+1`. `n` must be in range `[1,log2(bitmax)+1]` usually `[1,33]`

```

bitshift (eye (3), 1))
⇒
2 0 0
0 2 0
0 0 2

bitshift (10, [-2, -1, 0, 1, 2])
⇒ 2   5  10  20  40

```

See also: `bitand`, `bitor`, `bitxor`, `bitset`, `bitget`, `bitcmp`, `bitmax`.

Bits that are shifted out of either end of the value are lost. Octave also uses arithmetic shifts, where the sign bit of the value is kept during a right shift. For example

```

bitshift (-10, -1)
⇒ -5
bitshift (int8 (-1), -1)
⇒ -1

```

Note that `bitshift (int8 (-1), -1)` is `-1` since the bit representation of `-1` in the `int8` data type is `[1, 1, 1, 1, 1, 1, 1, 1]`.

4.5 Logical Values

Octave has built-in support for logical values, i.e. variables that are either `true` or `false`. When comparing two variables, the result will be a logical value whose value depends on whether or not the comparison is true.

The basic logical operations are `&`, `|`, and `!`, that corresponds to “Logical And”, “Logical Or”, and “Logical Negation”. These operations all follow the rules of logic.

It is also possible to use logical values as part of standard numerical calculations. In this case `true` is converted to 1, and `false` to 0, both represented using double precision floating point numbers. So, the result of `true*22 - false/6` is 22.

Logical values can also be used to index matrices and cell arrays. When indexing with a logical array the result will be a vector containing the values corresponding to `true` parts of the logical array. The following example illustrates this.

```

data = [ 1, 2; 3, 4 ];
idx = (data <= 2);
data(idx)
⇒ ans = [ 1; 4 ]

```

Instead of creating the `idx` array it is possible to replace `data(idx)` with `data(data <= 2)` in the above code.

Besides when doing comparisons, logical values can be constructed by casting numeric objects to logical values, or by using the `true` or `false` functions.

logical (arg) [Function File]

Convert *arg* to a logical value. For example,

```
logical ([-1, 0, 1])
```

is equivalent to

```
[-1, 0, 1] != 0
```


<code>true (x)</code>	[Built-in Function]
<code>true (n, m)</code>	[Built-in Function]
<code>true (n, m, k, ...)</code>	[Built-in Function]
Return a matrix or N-dimensional array whose elements are all logical 1. The arguments are handled the same as the arguments for <code>eye</code> .	
<code>false (x)</code>	[Built-in Function]
<code>false (n, m)</code>	[Built-in Function]
<code>false (n, m, k, ...)</code>	[Built-in Function]
Return a matrix or N-dimensional array whose elements are all logical 0. The arguments are handled the same as the arguments for <code>eye</code> .	

4.6 Predicates for Numeric Objects

Since the type of a variable may change during the execution of a program, it can be necessary to type checking at run-time. Doing this also allows you to change the behaviour of a function depending on the type of the input. As an example, this naive implementation of `abs` return the absolute value of the input if it is a real number, and the length of the input if it is a complex number.

```
function a = abs (x)
    if (isreal (x))
        a = sign (x) .* x;
    elseif (iscomplex (x))
        a = sqrt (real(x).^2 + imag(x).^2);
    endif
endfunction
```

The following functions are available for determining the type of a variable.

<code>isnumeric (x)</code>	[Built-in Function]
Return nonzero if <code>x</code> is a numeric object.	
<code>isreal (x)</code>	[Built-in Function]
Return true if <code>x</code> is a real-valued numeric object.	
<code>iscomplex (x)</code>	[Built-in Function]
Return true if <code>x</code> is a complex-valued numeric object.	
<code>ismatrix (a)</code>	[Built-in Function]
Return 1 if <code>a</code> is a matrix. Otherwise, return 0.	
<code>isvector (a)</code>	[Function File]
Return 1 if <code>a</code> is a vector. Otherwise, return 0.	
See also: size, rows, columns, length, isscalar, ismatrix.	
<code>isscalar (a)</code>	[Function File]
Return 1 if <code>a</code> is a scalar. Otherwise, return 0.	
See also: size, rows, columns, length, isscalar, ismatrix.	

`issquare (x)` [Function File]

If x is a square matrix, then return the dimension of x . Otherwise, return 0.

See also: size, rows, columns, length, ismatrix, isscalar, isvector.

`issymmetric (x, tol)` [Function File]

If x is symmetric within the tolerance specified by tol , then return the dimension of x . Otherwise, return 0. If tol is omitted, use a tolerance equal to the machine precision.

See also: size, rows, columns, length, ismatrix, isscalar, issquare, isvector.

`isdefinite (x, tol)` [Function File]

Return 1 if x is symmetric positive definite within the tolerance specified by tol or 0 if x is symmetric positive semidefinite. Otherwise, return -1. If tol is omitted, use a tolerance equal to 100 times the machine precision.

See also: issymmetric.

`islogical (x)` [Built-in Function]

Return true if x is a logical object.

`isprime (n)` [Function File]

Return true if n is a prime number, false otherwise.

Something like the following is much faster if you need to test a lot of small numbers:

```
t = ismember (n, primes (max (n (:))));
```

If $\max(n)$ is very large, then you should be using special purpose factorization code.

See also: primes, factor, gcd, lcm.

5 Strings

A *string constant* consists of a sequence of characters enclosed in either double-quote or single-quote marks. For example, both of the following expressions

```
"parrot"
'parrot'
```

represent the string whose contents are `'parrot'`. Strings in Octave can be of any length.

Since the single-quote mark is also used for the transpose operator (see [Section 9.3 \[Arithmetic Ops\]](#), [page 91](#)) but double-quote marks have no other purpose in Octave, it is best to use double-quote marks to denote strings.

In double-quoted strings, the backslash character is used to introduce *escape sequences* that represent other characters. For example, `'\n'` embeds a newline character in a double-quoted string and `'\"'` embeds a double quote character.

In single-quoted strings, backslash is not a special character.

Here is an example showing the difference

```
toascii ("\n")
⇒ 10
toascii ('\n')
⇒ [ 92 110 ]
```

You may also insert a single quote character in a single-quoted string by using two single quote characters in succession. For example,

```
'I can''t escape'
⇒ I can't escape
```

Here is a table of all the escape sequences used in Octave. They are the same as those used in the C programming language.

<code>\\</code>	Represents a literal backslash, <code>'\'</code> .
<code>\"</code>	Represents a literal double-quote character, <code>'\"'</code> .
<code>\'</code>	Represents a literal single-quote character, <code>'\''</code> .
<code>\0</code>	Represents the “nul” character, control-@, ASCII code 0.
<code>\a</code>	Represents the “alert” character, control-g, ASCII code 7.
<code>\b</code>	Represents a backspace, control-h, ASCII code 8.
<code>\f</code>	Represents a formfeed, control-l, ASCII code 12.
<code>\n</code>	Represents a newline, control-j, ASCII code 10.
<code>\r</code>	Represents a carriage return, control-m, ASCII code 13.
<code>\t</code>	Represents a horizontal tab, control-i, ASCII code 9.
<code>\v</code>	Represents a vertical tab, control-k, ASCII code 11.

Strings may be concatenated using the notation for defining matrices. For example, the expression

```
[ "foo" , "bar" , "baz" ]
```

produces the string whose contents are `'foobarbaz'`. See [Chapter 4 \[Numeric Data Types\]](#), [page 35](#), for more information about creating matrices.

5.1 Creating Strings

The easiest way to create a string is, as illustrated in the introduction, to enclose a text in double-quotes or single-quotes. It is however possible to create a string without actually writing a text. The function `blanks` creates a string of a given length consisting only of blank characters (ASCII code 32).

`blanks (n)` [Function File]

Return a string of n blanks.

See also: `repmat`.

The string representation used by Octave is an array of characters, so the result of `blanks(10)` is actually a row vector of length 10 containing the value 32 in all places. This lends itself to the obvious generalisation to character matrices. Using a matrix of characters, it is possible to represent a collection of same-length strings in one variable. The convention used in Octave is that each row in a character matrix is a separate string, but letting each column represent a string is equally possible.

The easiest way to create a character matrix is to put several strings together into a matrix.

```
collection = [ "String #1"; "String #2" ];
```

This creates a 2-by-9 character matrix.

One relevant question is, what happens when character matrix is created from strings of different length. The answer is that Octave puts blank characters at the end of strings shorter than the longest string. While it is possible to use a different character than the blank character using the `string_fill_char` function, it shows a problem with character matrices. It simply isn't possible to represent strings of different lengths. The solution is to use a cell array of strings, which is described in [Section 7.3 \[Cell Arrays of Strings\]](#), page 72.

`char (x)` [Built-in Function]

`char (cell_array)` [Built-in Function]

`char (s1, s2, ...)` [Built-in Function]

Create a string array from a numeric matrix, cell array, or list of

If the argument is a numeric matrix, each element of the matrix is converted to the corresponding ASCII character. For example,

```
char ([97, 98, 99])
⇒ "abc"
```

If the argument is a cell array of strings, the result is a string array with each element corresponding to one element of the cell array.

For multiple string arguments, the result is a string array with each element corresponding to the arguments.

The returned values are padded with blanks as needed to make each row of the string array have the same length.

`strcat (s1, s2, ...)` [Function File]

Return a string containing all the arguments concatenated. For example,

```
s = [ "ab"; "cde" ];
strcat (s, s, s)
⇒ "ab ab ab "
    "cdecdecde"
```

strvcat (*s*₁, ..., *s*_{*n*}) [Function File]

Return a matrix containing the strings (and cell-strings) *s*₁, ..., *s*_{*n*} as its rows. Each string is padded with blanks in order to form a valid matrix. Unlike *str2mat*, empty strings are ignored.

See also: *strcat*, *str2mat*.

strtrunc (*s*, *n*) [Function File]

Truncate the character string *s* to length *n*. If *s* is a char matrix, then the number of columns are adjusted.

If *s* is a cell array of strings, then the operation is performed on its members and the new cell array is returned.

val = **string_fill_char** () [Built-in Function]

old_val = **string_fill_char** (*new_val*) [Built-in Function]

Query or set the internal variable used to pad all rows of a character matrix to the same length. It must be a single character. The default value is " " (a single space). For example,

```
string_fill_char ("X");
[ "these"; "are"; "strings" ]
⇒ "theseXX"
   "areXXXX"
   "strings"
```

str2mat (*s*₁, ..., *s*_{*n*}) [Function File]

Return a matrix containing the strings *s*₁, ..., *s*_{*n*} as its rows. Each string is padded with blanks in order to form a valid matrix.

This function is modelled after MATLAB. In Octave, you can create a matrix of strings by [*s*₁; ...; *s*_{*n*}] even if the strings are not all the same length.

ischar (*a*) [Built-in Function]

Return 1 if *a* is a string. Otherwise, return 0.

s = **mat2str** (*x*, *n*) [Function File]

s = **mat2str** (... , 'class') [Function File]

Format real/complex numerical matrices as strings. This function returns values that are suitable for the use of the *eval* function.

The precision of the values is given by *n*. If *n* is a scalar then both real and imaginary parts of the matrix are printed to the same precision. Otherwise *n* (1) defines the precision of the real part and *n* (2) defines the precision of the imaginary part. The default for *n* is 17.

If the argument 'class' is given, then the class of *x* is included in the string in such a way that the eval will result in the construction of a matrix of the same class.

```
mat2str( [ -1/3 + i/7; 1/3 - i/7 ], [4 2] )
⇒ '[-0.3333+0.14i;0.3333-0.14i]'
mat2str( [ -1/3 +i/7; 1/3 -i/7 ], [4 2] )
⇒ '[-0.3333+0i,0+0.14i;0.3333+0i,-0-0.14i]'
mat2str( int16([1 -1]), 'class')
⇒ 'int16([1,-1])'
```

See also: `sprintf`, `int2str`.

<code>num2str (n)</code>	[Function File]
<code>num2str (x, <i>precision</i>)</code>	[Function File]
<code>num2str (x, <i>format</i>)</code>	[Function File]

Convert a number to a string. This function is not very flexible. For better control over the results, use `sprintf` (see [Section 15.2.4 \[Formatted Output\]](#), page 165).

See also: `sprintf`, `int2str`.

<code>int2str (n)</code>	[Function File]
<code>int2str (x, <i>precision</i>)</code>	[Function File]
<code>int2str (x, <i>format</i>)</code>	[Function File]

Convert an integer to a string. This function is not very flexible. For better control over the results, use `sprintf` (see [Section 15.2.4 \[Formatted Output\]](#), page 165).

See also: `sprintf`, `num2str`.

5.2 Comparing Strings

Since a string is a character array comparison between strings work element by element as the following example shows.

```
GNU = "GNU's Not UNIX";
spaces = (GNU == " ")
⇒ spaces =
    0    0    0    0    0    1    0    0    0    1    0    0    0    0
```

To determine if two functions are identical it is therefore necessary to use the `strcmp` or `strncmp` functions. Similar functions exists for doing case-insensitive comparisons.

<code>strcmp (s1, s2)</code>	[Built-in Function]
------------------------------	---------------------

Return 1 if the character strings *s1* and *s2* are the same, and 0 otherwise.

If either *s1* or *s2* is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

Caution: For compatibility with MATLAB, Octave's `strcmp` function returns 1 if the character strings are equal, and 0 otherwise. This is just the opposite of the corresponding C library function.

See also: `strcmpi`, `strncmp`, `strncmpi`.

strcmpi (*s1*, *s2*) [Function File]

Ignoring case, return 1 if the character strings *s1* and *s2* are the same, and 0 otherwise.

If either *s1* or *s2* is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

Caution: For compatibility with MATLAB, Octave's strcmpi function returns 1 if the character strings are equal, and 0 otherwise. This is just the opposite of the corresponding C library function.

See also: strcmp, strncmp, strncmpi.

strncmp (*s1*, *s2*, *n*) [Built-in Function]

Return 1 if the first *n* characters of strings *s1* and *s2* are the same, and 0 otherwise.

```
strncmp ("abce", "abcd", 3)
⇒ 1
```

If either *s1* or *s2* is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

```
strncmp ("abce", {"abcd", "bca", "abc"}, 3)
⇒ [1, 0, 1]
```

Caution: For compatibility with MATLAB, Octave's strncmp function returns 1 if the character strings are equal, and 0 otherwise. This is just the opposite of the corresponding C library function.

See also: strncmpi, strcmp, strcmpi.

strncmpi (*s1*, *s2*, *n*) [Function File]

Ignoring case, return 1 if the first *n* characters of character strings *s1* and *s2* are the same, and 0 otherwise.

If either *s1* or *s2* is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

Caution: For compatibility with MATLAB, Octave's strncmpi function returns 1 if the character strings are equal, and 0 otherwise. This is just the opposite of the corresponding C library function.

See also: strcmp, strcmpi, strncmp.

5.3 Manipulating Strings

Octave supports a wide range of functions for manipulating strings. Since a string is just a matrix, simple manipulations can be accomplished using standard operators. The following example shows how to replace all blank characters with underscores.

```
quote = "First things first, but not necessarily in that order";
quote( quote == " " ) = "_"
⇒ quote = First_things_first,_but_not_necessarily_in_that_order
```

For more complex manipulations, such as searching, replacing, and general regular expressions, the following function come with Octave.

deblank (*s*) [Function File]

Remove trailing blanks and nulls from *s*. If *s* is a matrix, *deblank* trims each row to the length of longest string. If *s* is a cell array, operate recursively on each element of the cell array.

findstr (*s*, *t*, *overlap*) [Function File]

Return the vector of all positions in the longer of the two strings *s* and *t* where an occurrence of the shorter of the two starts. If the optional argument *overlap* is nonzero, the returned vector can include overlapping positions (this is the default). For example,

```
findstr ("ababab", "a")
⇒ [ 1, 3, 5 ]
findstr ("abababa", "aba", 0)
⇒ [ 1, 5 ]
```

index (*s*, *t*) [Function File]

index (*s*, *t*, *direction*) [Function File]

Return the position of the first occurrence of the string *t* in the string *s*, or 0 if no occurrence is found. For example,

```
index ("Teststring", "t")
⇒ 4
```

If *direction* is `"first"`, return the first element found. If *direction* is `"last"`, return the last element found. The **rindex** function is equivalent to **index** with *direction* set to `"last"`.

Caution: This function does not work for arrays of character strings.

See also: **find**, **rindex**.

rindex (*s*, *t*) [Function File]

Return the position of the last occurrence of the character string *t* in the character string *s*, or 0 if no occurrence is found. For example,

```
rindex ("Teststring", "t")
⇒ 6
```

Caution: This function does not work for arrays of character strings.

See also: **find**, **index**.

idx = **strfind** (*str*, *pattern*) [Function File]

idx = **strfind** (*cellstr*, *pattern*) [Function File]

Search for *pattern* in the string *str* and return the starting index of every such occurrence in the vector *idx*. If there is no such occurrence, or if *pattern* is longer than *str*, then *idx* is the empty array `[]`.

If the cell array of strings *cellstr* is specified instead of the string *str*, then *idx* is a cell array of vectors, as specified above.

See also: findstr, strmatch, strcmp, strncmp, strcmpi, strncmpi.

strmatch (*s*, *a*, "exact") [Function File]

Return indices of entries of *a* that match the string *s*. The second argument *a* may be a string matrix or a cell array of strings. If the third argument "exact" is not given, then *s* only needs to match *a* up to the length of *s*. Nul characters match blanks. Results are returned as a column vector.

[tok, rem] = strtok (*str*, *delim*) [Function File]

Find all characters up to but not including the first character which is in the string *delim*. If *rem* is requested, it contains the remainder of the string, starting at the first delimiter. Leading delimiters are ignored. If *delim* is not specified, space is assumed.

split (*s*, *t*, *n*) [Function File]

Divides the string *s* into pieces separated by *t*, returning the result in a string array (padded with blanks to form a valid matrix). If the optional input *n* is supplied, split *s* into at most *n* different pieces.

For example,

```
split ("Test string", "t")
⇒ "Tes "
   " s "
   "ring"

split ("Test string", "t", 2)
⇒ "Tes "
   " string"
```

strrep (*s*, *x*, *y*) [Function File]

Replaces all occurrences of the substring *x* of the string *s* with the string *y*. For example,

```
strrep ("This is a test string", "is", "&%$")
⇒ "Th&%$ &%$ a test string"
```

substr (*s*, *offset*, *len*) [Function File]

Return the substring of *s* which starts at character number *offset* and is *len* characters long.

If *offset* is negative, extraction starts that far from the end of the string. If *len* is omitted, the substring extends to the end of *S*.

For example,

```
substr ("This is a test string", 6, 9)
⇒ "is a test"
```

This function is patterned after AWK. You can get the same result by *s* (*offset* : (*offset* + *len* - 1)).

`[s, e, te, m, t, nm] = regexp (str, pat)` [Loadable Function]
`[...] = regexp (str, pat, opts, ...)` [Loadable Function]

Regular expression string matching. Matches *pat* in *str* and returns the position and matching substrings or empty values if there are none.

The matched pattern *pat* can include any of the standard regex operators, including:

.	Match any character
* + ? {}	Repetition operators, representing
*	Match zero or more times
+	Match one or more times
?	Match zero or one times
{}	Match range operator, which is of the form { <i>n</i> } to match exactly <i>n</i> times, { <i>m</i> ,} to match <i>m</i> or more times, { <i>m</i> , <i>n</i> } to match between <i>m</i> and <i>n</i> times.

[...] [^...]

List operators, where for example `[ab]c` matches `ac` and `bc`

() Grouping operator

| Alternation operator. Match one of a choice of regular expressions. The alternatives must be delimited by the grouping operator () above

^ \$ Anchoring operator. ^ matches the start of the string *str* and \$ the end

In addition the following escaped characters have special meaning. It should be noted that it is recommended to quote *pat* in single quotes rather than double quotes, to avoid the escape sequences being interpreted by octave before being passed to `regexp`.

\b	Match a word boundary
\B	Match within a word
\w	Matches any word character
\W	Matches any non word character
\<	Matches the beginning of a word
\>	Matches the end of a word
\s	Matches any whitespace character
\S	Matches any non whitespace character
\d	Matches any digit
\D	Matches any non-digit

The outputs of `regexp` by default are in the order as given below

<i>s</i>	The start indices of each of the matching substrings
<i>e</i>	The end indices of each matching substring
<i>te</i>	The extents of each of the matched token surrounded by (...) in <i>pat</i> .

<i>m</i>	A cell array of the text of each match.
<i>t</i>	A cell array of the text of each token matched.
<i>nm</i>	A structure containing the text of each matched named token, with the name being used as the fieldname. A named token is denoted as (?<name>...)

Particular output arguments or the order of the output arguments can be selected by additional *opts* arguments. These are strings and the correspondence between the output arguments and the optional argument are

'start'	<i>s</i>
'end'	<i>e</i>
'tokenExtents'	<i>te</i>
'match'	<i>m</i>
'tokens'	<i>t</i>
'names'	<i>nm</i>

A further optional argument is 'once', that limits the number of returned matches to the first match. Additional arguments are

matchcase Make the matching case sensitive.

ignorecase Make the matching case insensitive.

stringanchors

Match the anchor characters at the beginning and end of the string.

lineanchors

Match the anchor characters at the beginning and end of the line.

dotall The character . matches the newline character.

dotexceptnewline

The character . matches all but the newline character.

freespacing

The pattern can include arbitrary whitespace and comments starting with #.

literalspacing

The pattern is taken literally.

```
[s, e, te, m, t, nm] = regexpi (str, pat) [Loadable Function]
[...] = regexpi (str, pat, opts, ...) [Loadable Function]
```

Case insensitive regular expression string matching. Matches *pat* in *str* and returns the position and matching substrings or empty values if there are none. See **regexp** for more details

```
string = regexprep (string, pat, repstr, options) [Loadable Function]
```

Replace matches of *pat* in *string* with *repstr*.

The replacement can contain \$*i*, which subsubstitutes for the *i*th set of parentheses in the match string. E.g.,

```
regexprep("Bill Dunn", '(\w+) (\w+)', '$2, $1')
```

returns "Dunn, Bill"

options may be zero or more of

‘once’ Replace only the first occurrence of *pat* in the result.

‘warnings’
 This option is present for compatibility but is ignored.

‘ignorecase or matchcase’
 Ignore case for the pattern matching (see `regexpi`). Alternatively, use `(?i)` or `(?-i)` in the pattern.

‘lineanchors and stringanchors’
 Whether characters `^` and `$` match the beginning and ending of lines. Alternatively, use `(?m)` or `(?-m)` in the pattern.

‘dotexceptnewline and dotall’
 Whether `.` matches newlines in the string. Alternatively, use `(?s)` or `(?-s)` in the pattern.

‘freespacing or literalspaceing’
 Whether whitespace and `#` comments can be used to make the regular expression more readable. Alternatively, use `(?x)` or `(?-x)` in the pattern.

See also: `regex`, `regexpi`.

5.4 String Conversions

Octave supports various kinds of conversions between strings and numbers. As an example, it is possible to convert a string containing a hexadecimal number to a floating point number.

```
hex2dec ("FF")
⇒ ans = 255
```

`bin2dec (s)` [Function File]

Return the decimal number corresponding to the binary number stored in the string *s*. For example,

```
bin2dec ("1110")
⇒ 14
```

If *s* is a string matrix, returns a column vector of converted numbers, one per row of *s*. Invalid rows evaluate to NaN.

See also: `dec2hex`, `base2dec`, `dec2base`, `hex2dec`, `dec2bin`.

`dec2bin (n, len)` [Function File]

Return a binary number corresponding the nonnegative decimal number *n*, as a string of ones and zeros. For example,

```
dec2bin (14)
⇒ "1110"
```

If n is a vector, returns a string matrix, one row per value, padded with leading zeros to the width of the largest value.

The optional second argument, len , specifies the minimum number of digits in the result.

See also: bin2dec, dec2base, base2dec, hex2dec, dec2hex.

dec2hex (n , len) [Function File]

Return the hexadecimal string corresponding to the nonnegative integer n . For example,

```
dec2hex (2748)
⇒ "ABC"
```

If n is a vector, returns a string matrix, one row per value, padded with leading zeros to the width of the largest value.

The optional second argument, len , specifies the minimum number of digits in the result.

See also: hex2dec, dec2base, base2dec, bin2dec, dec2bin.

hex2dec (s) [Function File]

Return the integer corresponding to the hexadecimal number stored in the string s . For example,

```
hex2dec ("12B")
⇒ 299
hex2dec ("12b")
⇒ 299
```

If s is a string matrix, returns a column vector of converted numbers, one per row of s . Invalid rows evaluate to NaN.

See also: dec2hex, base2dec, dec2base, bin2dec, dec2bin.

dec2base (n , b , len) [Function File]

Return a string of symbols in base b corresponding to the nonnegative integer n .

```
dec2base (123, 3)
⇒ "11120"
```

If n is a vector, return a string matrix with one row per value, padded with leading zeros to the width of the largest value.

If b is a string then the characters of b are used as the symbols for the digits of n . Space (' ') may not be used as a symbol.

```
dec2base (123, "aei")
⇒ "eeeia"
```

The optional third argument, len , specifies the minimum number of digits in the result.

See also: base2dec, dec2bin, bin2dec, hex2dec, dec2hex.

base2dec (*s*, *b*) [Function File]

Convert *s* from a string of digits of base *b* into an integer.

```
base2dec ("11120", 3)
⇒ 123
```

If *s* is a matrix, returns a column vector with one value per row of *s*. If a row contains invalid symbols then the corresponding value will be NaN. Rows are right-justified before converting so that trailing spaces are ignored.

If *b* is a string, the characters of *b* are used as the symbols for the digits of *s*. Space (' ') may not be used as a symbol.

```
base2dec ("yyyzx", "xyz")
⇒ 123
```

See also: dec2base, dec2bin, bin2dec, hex2dec, dec2hex.

[num, status, strarray] = str2double (*str*, *cdelim*, *rdelim*, *ddelim*) [Function File]

Convert strings into numeric values.

str2double can replace **str2num**, but avoids the use of **eval** on unknown data.

str can be the form '[+-]d[.]dd[**[eE]** [+-]ddd]' in which 'd' can be any of digit from 0 to 9, and '[' indicate optional elements.

num is the corresponding numeric value. If the conversion fails, *status* is -1 and *num* is NaN.

status is 0 if the conversion was successful and -1 otherwise.

strarray is a cell array of strings.

Elements which are not defined or not valid return NaN and the *status* becomes -1.

If *str* is a character array or a cell array of strings, then *num* and *status* return matrices of appropriate size.

str can also contain multiple elements separated by row and column delimiters (*cdelim* and *rdelim*).

The parameters *cdelim*, *rdelim*, and *ddelim* are optional column, row, and decimal delimiters.

The default row-delimiters are newline, carriage return and semicolon (ASCII 10, 13 and 59). The default column-delimiters are tab, space and comma (ASCII 9, 32, and 44). The default decimal delimiter is '.' (ASCII 46).

cdelim, *rdelim*, and *ddelim* must contain only nul, newline, carriage return, semicolon, colon, slash, tab, space, comma, or '()'[]{}' (ASCII 0, 9, 10, 11, 12, 13, 14, 32, 33, 34, 40, 41, 44, 47, 58, 59, 91, 93, 123, 124, 125).

Examples:

```
str2double ("-1e-5")
⇒ -1.0000e-006
```

```
str2double (".314e1, 44.44e-1, .7; -1e+1")
⇒
    3.1400    4.4440    0.7000
```

```

-10.0000      NaN      NaN

line = "200,300,400,NaN,-inf,cd,yes,no,999,maybe,NaN";
[x, status] = str2double (line)
x =
    200    300    400    NaN  -Inf    NaN    NaN    NaN    999    NaN    NaN
status =
     0     0     0     0     0    -1    -1    -1     0    -1     0

```

strjust (*s*, ["left"|"right"|"center"]) [Function File]

Shift the non-blank text of *s* to the left, right or center of the string. If *s* is a string array, justify each string in the array. Null characters are replaced by blanks. If no justification is specified, then all rows are right-justified.

str2num (*s*) [Function File]

Convert the string *s* to a number.

toascii (*s*) [Mapping Function]

Return ASCII representation of *s* in a matrix. For example,

```

toascii ("ASCII")
⇒ [ 65, 83, 67, 73, 73 ]

```

tolower (*s*) [Mapping Function]

Return a copy of the string *s*, with each upper-case character replaced by the corresponding lower-case one; nonalphabetic characters are left unchanged. For example,

```

tolower ("MiXeD cAsE 123")
⇒ "mixed case 123"

```

toupper (*s*) [Built-in Function]

Return a copy of the string *s*, with each lower-case character replaced by the corresponding upper-case one; nonalphabetic characters are left unchanged. For example,

```

toupper ("MiXeD cAsE 123")
⇒ "MIXED CASE 123"

```

do_string_escapes (*string*) [Built-in Function]

Convert special characters in *string* to their escaped forms.

undo_string_escapes (*s*) [Built-in Function]

Converts special characters in strings back to their escaped forms. For example, the expression

```
bell = "\a";
```

assigns the value of the alert character (control-g, ASCII code 7) to the string variable *bell*. If this string is printed, the system will ring the terminal bell (if it is possible). This is normally the desired outcome. However, sometimes it is useful to be able to print the original representation of the string, with the special characters replaced by their escape sequences. For example,

```
octave:13> undo_string_escapes (bell)
ans = \a
```

replaces the unprintable alert character with its printable representation.

5.5 Character Class Functions

Octave also provides the following character class test functions patterned after the functions in the standard C library. They all operate on string arrays and return matrices of zeros and ones. Elements that are nonzero indicate that the condition was true for the corresponding character in the string array. For example,

```
isalpha ("!Q@WERT^Y&")
⇒ [ 0, 1, 0, 1, 1, 1, 1, 0, 1, 0 ]
```

isalnum (*s*) [Mapping Function]
Return 1 for characters that are letters or digits (**isalpha** (*s*) or **isdigit** (*s*) is true).

isalpha (*s*) [Mapping Function]
isletter (*s*) [Mapping Function]
Return true for characters that are letters (**isupper** (*s*) or **islower** (*s*) is true).

isasci (*s*) [Mapping Function]
Return 1 for characters that are ASCII (in the range 0 to 127 decimal).

iscntrl (*s*) [Mapping Function]
Return 1 for control characters.

isdigit (*s*) [Mapping Function]
Return 1 for characters that are decimal digits.

isgraph (*s*) [Mapping Function]
Return 1 for printable characters (but not the space character).

isletter (*s*) [Function File]
Returns true if *s* is a letter false otherwise.
See also: **isalpha**.

islower (*s*) [Mapping Function]
Return 1 for characters that are lower case letters.

isprint (*s*) [Mapping Function]
Return 1 for printable characters (including the space character).

ispunct (*s*) [Mapping Function]
Return 1 for punctuation characters.

`isspace (s)` [Mapping Function]
Return 1 for whitespace characters (space, formfeed, newline, carriage return, tab, and vertical tab).

`isupper (s)` [Mapping Function]
Return 1 for upper case letters.

`isxdigit (s)` [Mapping Function]
Return 1 for characters that are hexadecimal digits.

6 Data Structures

Octave includes support for organizing data in structures. The current implementation uses an associative array with indices limited to strings, but the syntax is more like C-style structures. Here are some examples of using data structures in Octave.

Elements of structures can be of any value type. For example, the three expressions

```
x.a = 1
x.b = [1, 2; 3, 4]
x.c = "string"
```

create a structure with three elements. To print the value of the structure, you can type its name, just as for any other variable:

```
octave:2> x
x =
{
  a = 1
  b =

      1  2
      3  4

  c = string
}
```

Note that Octave may print the elements in any order.

Structures may be copied.

```
octave:1> y = x
y =
{
  a = 1
  b =

      1  2
      3  4

  c = string
}
```

Since structures are themselves values, structure elements may reference other structures. The following statements change the value of the element `b` of the structure `x` to be a data structure containing the single element `d`, which has a value of 3.

```

octave:1> x.b.d = 3
x.b.d = 3
octave:2> x.b
ans =
{
  d = 3
}
octave:3> x
x =
{
  a = 1
  b =
  {
    d = 3
  }

  c = string
}

```

Note that when Octave prints the value of a structure that contains other structures, only a few levels are displayed. For example,

```

octave:1> a.b.c.d.e = 1;
octave:2> a
a =
{
  b =
  {
    c =
    {
      d: 1x1 struct
    }
  }
}

```

This prevents long and confusing output from large deeply nested structures.

```

val = struct_levels_to_print () [Built-in Function]
old_val = struct_levels_to_print (new_val) [Built-in Function]

```

Query or set the internal variable that specifies the number of structure levels to display.

Functions can return structures. For example, the following function separates the real and complex parts of a matrix and stores them in two elements of the same structure variable.

```

octave:1> function y = f (x)
> y.re = real (x);
> y.im = imag (x);
> endfunction

```

When called with a complex-valued argument, `f` returns the data structure containing the real and imaginary parts of the original function argument.

```
octave:2> f (rand (2) + rand (2) * I)
ans =
{
  im =

    0.26475  0.14828
    0.18436  0.83669

  re =

    0.040239  0.242160
    0.238081  0.402523
}
```

Function return lists can include structure elements, and they may be indexed like any other variable. For example,

```
octave:1> [ x.u, x.s(2:3,2:3), x.v ] = svd ([1, 2; 3, 4])
x.u =

   -0.40455   -0.91451
   -0.91451    0.40455

x.s =

   0.00000   0.00000   0.00000
   0.00000   5.46499   0.00000
   0.00000   0.00000   0.36597

x.v =

   -0.57605    0.81742
   -0.81742   -0.57605
```

It is also possible to cycle through all the elements of a structure in a loop, using a special form of the `for` statement (see [Section 11.5 \[The for Statement\]](#), page 110)

The following functions are available to give you information about structures.

struct ("field", value, "field", value, ...) [Built-in Function]

Create a structure and initialize its value.

If the values are cell arrays, create a structure array and initialize its values. The dimensions of each cell array of values must match. Singleton cells and non-cell values are repeated so that they fill the entire array. If the cells are empty, create an empty structure array with the specified field names.

isstruct (expr) [Built-in Function]

Return 1 if the value of the expression `expr` is a structure.

fieldnames (*struct*) [Built-in Function]
 Return a cell array of strings naming the elements of the structure *struct*. It is an error to call **fieldnames** with an argument that is not a structure.

isfield (*expr*, *name*) [Built-in Function]
 Return true if the expression *expr* is a structure and it includes an element named *name*. The first argument must be a structure and the second must be a string.

[*v1*, ...] = **getfield** (*s*, *key*, ...) [Function File]
 Extract fields from a structure. For example

```
ss(1,2).fd(3).b=5;
getfield (ss, {1,2}, "fd", {3}, "b")
⇒ ans = 5
```

Note that the function call in the previous example is equivalent to the expression

```
i1= {1,2}; i2= "fd"; i3= {3}; i4= "b";
ss(i1{:}).(i2)(i3{:}).(i4)
```

See also: **setfield**, **rmfield**, **isfield**, **isstruct**, **fieldnames**, **struct**.

[*t*, *p*] = **orderfields** (*s1*, *s2*) [Function File]
 Return a struct with fields arranged alphabetically or as specified by *s2* and a corresponding permutation vector.
 Given one struct, arrange field names in *s1* alphabetically.
 Given two structs, arrange field names in *s1* as they appear in *s2*. The second argument may also specify the order in a permutation vector or a cell array of strings.
See also: **getfield**, **rmfield**, **isfield**, **isstruct**, **fieldnames**, **struct**.

rmfield (*s*, *f*) [Built-in Function]
 Remove field *f* from the structure *s*. If *f* is a cell array of character strings or a character array, remove the named fields.
See also: **cellstr**, **iscellstr**, **setfield**.

[*k1*, ..., *v1*] = **setfield** (*s*, *k1*, *v1*, ...) [Function File]
 Set field members in a structure.

```
oo(1,1).f0= 1;
oo = setfield(oo,{1,2},'fd',{3},'b', 6);
oo(1,2).fd(3).b == 6
⇒ ans = 1
```

Note that this function could be written

```
i1= {1,2}; i2= 'fd'; i3= {3}; i4= 'b';
oo( i1{:} ).( i2 )( i3{:} ).( i4 ) == 6;
```

See also: **getfield**, **rmfield**, **isfield**, **isstruct**, **fieldnames**, **struct**.

struct2cell (*S*) [Built-in Function]

Create a new cell array from the objects stored in the struct object. If *f* is the number of fields in the structure, the resulting cell array will have a dimension vector corresponding to [*F* **size**(*S*)].

See also: cell2struct, fieldnames.

substruct (*type*, *subs*, ...) [Function File]

Create a subscript structure for use with **subsref** or **subsasgn**.

See also: subsref, subsasgn.

7 Cell Arrays

It can be both necessary and convenient to store several variables of different size or type in one variable. A cell array is a container class able to do just that. In general cell arrays work just like N -dimensional arrays, with the exception of the use of ‘{’ and ‘}’ as allocation and indexing operators.

As an example, the following code creates a cell array containing a string and a 2-by-2 random matrix

```
c = {"a string", rand(2, 2)};
```

And a cell array can be indexed with the { and } operators, so the variable created in the previous example can be indexed like this

```
c{1}
⇒ ans = a string
```

As with numerical arrays several elements of a cell array can be extracted by indexing with a vector of indexes

```
c{1:2}
⇒ ans =

    (,
      [1] = a string
      [2] =

          0.593993    0.627732
          0.377037    0.033643

    ,)
```

The indexing operators can also be used to insert or overwrite elements of a cell array. The following code inserts the scalar 3 on the third place of the previously created cell array

```
c{3} = 3
⇒ c =

    {
      [1,1] = a string
      [1,2] =

          0.593993    0.627732
          0.377037    0.033643

      [1,3] = 3
    }
```

7.1 Creating Cell Array

The introductory example showed how to create a cell array containing currently available variables. In many situations, however, it is useful to create a cell array and then fill it with data.

The `cell` function returns a cell array of a given size, containing empty matrices. This function works very similar to the `zeros` function for creating new numerical arrays. The following example creates a 2-by-2 cell array containing empty matrices

```
c = cell(2,2)
⇒ c =

    {
    [1,1] = [] (0x0)
    [2,1] = [] (0x0)
    [1,2] = [] (0x0)
    [2,2] = [] (0x0)
    }
```

Just like numerical arrays, cell arrays can be multidimensional. The `cell` function accepts any number of positive integers to describe the size of the returned cell array. It is also possible to set the size of the cell array through a vector of positive integers. In the following example two cell arrays of equal size is created, and the size of the first one is displayed

```
c1 = cell(3, 4, 5);
c2 = cell( [3, 4, 5] );
size(c1)
⇒ ans =
    3    4    5
```

As can be seen, the `size` function also work for cell arrays. As do the other functions describing the size of an object, such as `length`, `numel`, `rows`, and `columns`.

An alternative to creating empty cell arrays, and then filling them, it is possible to convert numerical arrays into cell arrays using the `num2cell` and `mat2cell` functions.

`cell (x)` [Built-in Function]
`cell (n, m)` [Built-in Function]

Create a new cell array object. If invoked with a single scalar argument, `cell` returns a square cell array with the dimension specified. If you supply two scalar arguments, `cell` takes them to be the number of rows and columns. If given a vector with two elements, `cell` uses the values of the elements as the number of rows and columns, respectively.

`iscell (x)` [Built-in Function]
 Return true if `x` is a cell array object. Otherwise, return false.

`c = num2cell (m)` [Loadable Function]
`c = num2cell (m, d)` [Loadable Function]

Convert to matrix `m` into a cell array. If `d` is defined the value `c` is of dimension 1 in this dimension and the elements of `m` are placed in slices in `c`.

See also: `mat2cell`.

```

b = mat2cell (a, m, n) [Loadable Function]
b = mat2cell (a, d1, d2, ...) [Loadable Function]
b = mat2cell (a, r) [Loadable Function]

```

Converts the matrix *a* to a cell array. If *a* is 2-D, then it is required that `sum(m) == size(a, 1)` and `sum(n) == size(a, 2)`. Similarly, if *a* is a multi-dimensional and the number of dimensional arguments is equal to the dimensions of *a*, then it is required that `sum(di) == size(a, i)`.

Given a single dimensional argument *r*, the other dimensional arguments are assumed to equal `size(a, i)`.

An example of the use of `mat2cell` is

```

mat2cell (reshape(1:16,4,4), [3,1], [3,1])
⇒ {
    [1,1] =

        1     5     9
        2     6    10
        3     7    11

    [2,1] =

        4     8    12

    [1,2] =

        13
        14
        15

    [2,2] = 16
}

```

See also: `num2cell`, `cell2mat`.

7.2 Indexing Cell Arrays

As shown in the introductory example elements can be inserted from cell arrays using the ‘{’ and ‘}’ operators. Besides the change of operators, indexing works for cell arrays like for multidimensional arrays. As an example, all the rows of the first and third column of a cell array can be set to 0 with the following code

```
c{:, [1, 3]} = 0;
```

Accessing values in a cell array is, however, different from the same operation for numerical arrays. Accessing a single element of a cell array is very similar to numerical arrays, for example

```
element = c{1, 2};
```

This will, however, *not* work when accessing multiple elements of a cell array, because it might not be possible to represent all elements with a single variable as is the case with numerical arrays.

Accessing multiple elements of a cell array with the ‘{’ and ‘}’ operators will result in a comma-separated list of all the requested elements. This list can then be used anywhere where a comma-separated list is used, such as in the creation of a new numerical array or cell array, or be passed as arguments to a function. If all the accessed elements of a cell array are scalars or column vectors, they can be concatenated into a new column vector containing the elements, by surrounding the list with [and] as in the following example

```
a = {1, [2, 3], 4};
b = [a{:}]
⇒ b =
    1    2    3    4
```

It is also possible to pass the accessed elements directly to a function. The list of elements from the cell array will be passed as an argument list to a given function as if it is called with the elements as arguments. The two calls to `printf` in the following example are identical but the latter is more simple and handles more situations

```
c = {"GNU", "Octave", "is", "Free", "Software"};
printf ("%s ", c{1}, c{2}, c{3}, c{4}, c{5});
+ GNU Octave is Free Software
printf ("%s ", c{:});
+ GNU Octave is Free Software
```

Just like it is possible to create a numerical array from selected elements of a cell array, it is possible to create a new cell array containing the selected elements. By surrounding the list with ‘{’ and ‘}’ a new cell array will be created, like the following example illustrates

```
a = {1, rand(2, 2), "three"};
b = { a{ [1, 3] } }
⇒ b =
{
    [1,1] = 1
    [1,2] = three
}
```

This syntax is however a bit cumbersome, and since this is a common operation, it is possible to achieve the same using the ‘(’ and ‘)’ operators for indexing. When a cell array is indexed using the ‘(’ and ‘)’ operators a new cell array containing the selected elements. Using this syntax, the previous example can be simplified into the following

```
a = {1, rand(2, 2), "three"};
b = a( [1, 3] )
⇒ b =
{
    [1,1] = 1
    [1,2] = three
}
```

7.3 Cell Arrays of Strings

One common use of cell arrays is to store multiple strings in the same variable. It is possible to store multiple strings in a character matrix by letting each row be a string. This, however, introduces the problem that all strings must be of equal length. Therefore

it is recommended to use cell arrays to store multiple strings. If, however, the character matrix representation is required for an operation, it can be converted to a cell array of strings using the `cellstr` function

```
a = ["hello"; "world"];
c = cellstr (a)
⇒ c =
    {
    [1,1] = hello
    [2,1] = world
    }
```

One further advantage of using cell arrays to store multiple strings, is that most functions for string manipulations included with Octave supports this representation. As an example, it is possible to compare one string with many others using the `strcmp` function. If one of the arguments to this function is a string and the other is a cell array of strings, each element of the cell array will be compared the string argument,

```
c = {"hello", "world"};
strcmp ("hello", c)
⇒ ans =
    1    0
```

The following functions for string manipulation support cell arrays of strings, `strcmp`, `strcmpi`, `strncmp`, `strncmpi`, `str2double`, `str2mat`, `strappend`, `strtrunc`, `strvcats`, `strfind`, and `strmatch`.

`cellstr (string)` [Built-in Function]
Create a new cell array object from the elements of the string array *string*.

`iscellstr (cell)` [Built-in Function]
Return true if every element of the cell array *cell* is a character string

`[idxvec, errmsg] = cellidx (listvar, strlist)` [Function File]
Return indices of string entries in *listvar* that match strings in *strlist*.

Both *listvar* and *strlist* may be passed as strings or string matrices. If they are passed as string matrices, each entry is processed by `deblank` prior to searching for the entries.

The first output is the vector of indices in *listvar*.

If *strlist* contains a string not in *listvar*, then an error message is returned in *errmsg*. If only one output argument is requested, then *cellidx* prints *errmsg* to the screen and exits with an error.

7.4 Processing Data in Cell Arrays

Data that is stored in a cell array can be processed in several ways depending on the actual data. The most simple way to process that data is to iterate through it using one or more `for` loops. The same idea can be implemented easier through the use of the `cellfun` function that calls a user specified function on all elements of a cell array.

<code>cellfun (name, c)</code>	[Lodable Function]
<code>cellfun ("size", c, k)</code>	[Lodable Function]
<code>cellfun ("isclass", c, class)</code>	[Lodable Function]
<code>cellfun (func, c)</code>	[Lodable Function]
<code>cellfun (func, c, d)</code>	[Lodable Function]
<code>[a, b] = cellfun (...)</code>	[Lodable Function]
<code>cellfun (... , 'ErrorHandler', errfunc)</code>	[Lodable Function]
<code>cellfun (... , 'UniformOutput', val)</code>	[Lodable Function]

Evaluate the function named *name* on the elements of the cell array *c*. Elements in *c* are passed on to the named function individually. The function *name* can be one of the functions

`isempty` Return 1 for empty elements.

`islogical`
 Return 1 for logical elements.

`isreal` Return 1 for real elements.

`length` Return a vector of the lengths of cell elements.

`ndims` Return the number of dimensions of each element.

`prodofsize`
 Return the product of dimensions of each element.

`size` Return the size along the *k*-th dimension.

`isclass` Return 1 for elements of *class*.

Additionally, `cellfun` accepts an arbitrary function *func* in the form of an inline function, function handle, or the name of a function (in a character string). In the case of a character string argument, the function must accept a single argument named *x*, and it must return a string value. The function can take one or more arguments, with the inputs args given by *c*, *d*, etc. Equally the function can return one or more output arguments. For example

```
cellfun (@atan2, {1, 0}, {0, 1})
⇒ ans = [1.57080  0.00000]
```

Note that the default output argument is an array of the same size as the input arguments.

If the param 'UniformOutput' is set to true (the default), then the function must return either a single element which will be concatenated into the return value. If 'UniformOutput' is false, the outputs are concatenated in a cell array. For example

```
cellfun ("tolower(x)", {"Foo", "Bar", "FooBar"},
        "UniformOutput", false)
⇒ ans = {"foo", "bar", "foobar"}
```

Given the parameter 'ErrorHandler', then *errfunc* defines a function to call in case *func* generates an error. The form of the function is

```
function [...] = errfunc (s, ...)
```

where there is an additional input argument to *errfunc* relative to *func*, given by *s*. This is a structure with the elements 'identifier', 'message' and 'index', giving respectively the error identifier, the error message, and the index into the input arguments of the element that caused the error. For example

```
function y = foo (s, x), y = NaN; endfunction
cellfun (@factorial, {-1,2},'ErrorHandler',@foo)
⇒ ans = [NaN 2]
```

See also: isempty, islogical, isreal, length, ndims, numel, size, isclass.

An alternative is to convert the data to a different container, such as a matrix or a data structure. Depending on the data this is possible using the `cell2mat` and `cell2struct` functions.

`m = cell2mat (c)` [Function File]

Convert the cell array *c* into a matrix by concatenating all elements of *c* into a hyperrectangle. Elements of *c* must be numeric, logical or char, and `cat` must be able to concatenate them together.

See also: mat2cell, num2cell.

`cell2struct (cell, fields, dim)` [Built-in Function]

Convert *cell* to a structure. The number of fields in *fields* must match the number of elements in *cell* along dimension *dim*, that is `numel (fields) == size (cell, dim)`.

```
A = cell2struct ({'Peter', 'Hannah', 'Robert'; 185, 170, 168},
                {'Name','Height'}, 1);
```

```
A(1)
⇒ ans =
    {
      Height = 185
      Name   = Peter
    }
```


8 Variables

Variables let you give names to values and refer to them later. You have already seen variables in many of the examples. The name of a variable must be a sequence of letters, digits and underscores, but it may not begin with a digit. Octave does not enforce a limit on the length of variable names, but it is seldom useful to have variables with names longer than about 30 characters. The following are all valid variable names

```
x
x15
__foo_bar_baz__
fucnrthsucngtagdjb
```

However, names like `__foo_bar_baz__` that begin and end with two underscores are understood to be reserved for internal use by Octave. You should not use them in code you write, except to access Octave's documented internal variables and built-in symbolic constants.

Case is significant in variable names. The symbols `a` and `A` are distinct variables.

A variable name is a valid expression by itself. It represents the variable's current value. Variables are given new values with *assignment operators* and *increment operators*. See [Section 9.6 \[Assignment Expressions\]](#), page 96.

A number of variables have special built-in meanings. For example, `ans` holds the current working directory, and `pi` names the ratio of the circumference of a circle to its diameter. See [Section 8.4 \[Summary of Built-in Variables\]](#), page 82, for a list of all the predefined variables. Some of these built-in symbols are constants and may not be changed. Others can be used and assigned just like all other variables, but their values are also used or changed automatically by Octave.

Variables in Octave do not have fixed types, so it is possible to first store a numeric value in a variable and then to later use the same name to hold a string value in the same program. Variables may not be used before they have been given a value. Doing so results in an error.

`isvarname (name)` [Built-in Function]
Return true if *name* is a valid variable name

8.1 Global Variables

A variable that has been declared *global* may be accessed from within a function body without having to pass it as a formal parameter.

A variable may be declared global using a `global` declaration statement. The following statements are all global declarations.

```
global a
global a b
global c = 2
global d = 3 e f = 5
```

A global variable may only be initialized once in a `global` statement. For example, after executing the following code

```
global gvar = 1
global gvar = 2
```

the value of the global variable `gvar` is 1, not 2. Issuing a `'clear gvar'` command does not change the above behavior, but `'clear all'` does.

It is necessary declare a variable as global within a function body in order to access it. For example,

```
global x
function f ()
  x = 1;
endfunction
f ()
```

does *not* set the value of the global variable `x` to 1. In order to change the value of the global variable `x`, you must also declare it to be global within the function body, like this

```
function f ()
  global x;
  x = 1;
endfunction
```

Passing a global variable in a function parameter list will make a local copy and not modify the global value. For example, given the function

```
function f (x)
  x = 0
endfunction
```

and the definition of `x` as a global variable at the top level,

```
global x = 13
```

the expression

```
f (x)
```

will display the value of `x` from inside the function as 0, but the value of `x` at the top level remains unchanged, because the function works with a *copy* of its argument.

`isglobal (name)` [Built-in Function]

Return 1 if `name` is globally visible. Otherwise, return 0. For example,

```
global x
isglobal ("x")
⇒ 1
```

8.2 Persistent Variables

A variable that has been declared *persistent* within a function will retain its contents in memory between subsequent calls to the same function. The difference between persistent variables and global variables is that persistent variables are local in scope to a particular function and are not visible elsewhere.

A variable may be declared persistent using a **`persistent`** declaration statement. The following statements are all persistent declarations.

```

persistent a
persistent a b
persistent c = 2
persistent d = 3 e f = 5

```

The behavior of persistent variables is equivalent to the behavior of static variables in C. The command `static` in octave is also recognized and is equivalent to `persistent`. Unlike global variables, every initialization statement will re-initialize the variable. For example, after executing the following code

```

persistent pvar = 1
persistent pvar = 2

```

the value of the persistent variable `pvar` is 2.

8.3 Status of Variables

When creating simple one-shot programs it can be very convenient to see which variables are available at the prompt. The function `who` and its siblings `whos` and `whos_line_format` will show different information about what is in memory, as the following shows.

```

str = "A random string";
who -variables
  + *** local user variables:
  +
  + __nargin__  str

```

`who options pattern ...` [Command]

`whos options pattern ...` [Command]

List currently defined symbols matching the given patterns. The following are valid options. They may be shortened to one character but may not be combined.

`-all` List all currently defined symbols.

`-builtins`

List built-in functions. This includes all currently compiled function files, but does not include all function files that are in the search path.

`-functions`

List user-defined functions.

`-long`

Print a long listing including the type and dimensions of any symbols. The symbols in the first column of output indicate whether it is possible to redefine the symbol, and whether it is possible for it to be cleared.

`-variables`

List user-defined variables.

Valid patterns are the same as described for the `clear` command above. If no patterns are supplied, all symbols from the given category are listed. By default, only user defined functions and variables visible in the local scope are displayed.

The command `whos` is equivalent to `who -long`.

whos *options pattern* ... [Command]
 See who.

val = **whos_line_format** () [Built-in Function]
old_val = **whos_line_format** (*new_val*) [Built-in Function]

Query or set the format string used by the **whos**.

The following escape sequences may be used in the format:

%b	Prints number of bytes occupied by variables.
%c	Prints class names of variables.
%e	Prints elements held by variables.
%n	Prints variable names.
%p	Prints protection attributes of variables.
%s	Prints dimensions of variables.
%t	Prints type names of variables.

Every command may also have a modifier:

l	Left alignment.
r	Right alignment (this is the default).
c	Centered (may only be applied to command %s).

A command is composed like this: **%[modifier]<command>[:size_of_parameter[:center-specific[:print_dims[:balance]]]]**;

Command and modifier is already explained. **Size_of_parameter** tells how many columns the parameter will need for printing. **print_dims** tells how many dimensions to print. If number of dimensions exceeds **print_dims**, dimensions will be printed like x-D. **center-specific** and **print_dims** may only be applied to command **%s**. A negative value for **print_dims** will cause Octave to print all dimensions whatsoever. **balance** specifies the offset for printing of the dimensions string.

The default format is " **%p:4; %ln:6; %cs:16:6:8:1; %rb:12; %lc:-1;\n**".

Instead of displaying which variables are in memory, it is possible to determine if a given variable is available. That way it is possible to alter the behaviour of a program depending on the existence of a variable. The following example illustrates this.

```
if (! exist ("meaning", "var"))
    disp ("The program has no 'meaning'");
endif
```

exist (*name, type*) [Built-in Function]

Return 1 if the name exists as a variable, 2 if the name (after appending **'.m'**) is a function file in Octave's **path**, 3 if the name is a **'.oct'** or **'.mex'** file in Octave's **path**, 5 if the name is a built-in function, 7 if the name is a directory, or 103 if the name is a function not associated with a file (entered on the command line).

Otherwise, return 0.

This function also returns 2 if a regular file called *name* exists in Octave's search path. If you want information about other types of files, you should use some combination of the functions `file_in_path` and `stat` instead.

If the optional argument *type* is supplied, check only for symbols of the specified type. Valid types are

```
"var"      Check only for variables.
"builtin"   Check only for built-in functions.
"file"      Check only for files.
"dir"       Check only for directories.
```

Usually Octave will manage the memory, but sometimes it can be practical to remove variables from memory manually. This is usually needed when working with large variables that fill a substantial part of the memory. On a computer that uses the IEEE floating point format, the following program allocates a matrix that requires around 128 MB memory.

```
large_matrix = zeros (4000, 4000);
```

Since having this variable in memory might slow down other computations, it can be necessary to remove it manually from memory. The `clear` function allows this.

`clear [-x] pattern ...` [Command]

Delete the names matching the given patterns from the symbol table. The pattern may contain the following special characters:

```
?      Match any single character.
*      Match zero or more characters.
[ list ] Match the list of characters specified by list. If the first character is !
or ^, match all characters except those specified by list. For example,
the pattern '[a-zA-Z]' will match all lower and upper case alphabetic
characters.
```

For example, the command

```
clear foo b*r
```

clears the name `foo` and all names that begin with the letter `b` and end with the letter `r`.

If `clear` is called without any arguments, all user-defined variables (local and global) are cleared from the symbol table. If `clear` is called with at least one argument, only the visible names matching the arguments are cleared. For example, suppose you have defined a function `foo`, and then hidden it by performing the assignment `foo = 2`. Executing the command `clear foo` once will clear the variable definition and restore the definition of `foo` as a function. Executing `clear foo` a second time will clear the function definition.

With `-x`, clear the variables that don't match the patterns.

Information about a function or variable such as its location in the file system can also be acquired from within Octave. This is usually only useful during development of programs, and not within a program.

document (*symbol*, *text*) [Built-in Function]
 Set the documentation string for *symbol* to *text*.

type *options name* ... [Command]
 Display the definition of each *name* that refers to a function.
 Normally also displays whether each *name* is user-defined or built-in; the `-q` option suppresses this behaviour.

which *name* ... [Command]
 Display the type of each *name*. If *name* is defined from a function file, the full name of the file is also displayed.
See also: `help`, `lookfor`.

8.4 Summary of Built-in Variables

Here is a summary of all of Octave's built-in variables along with cross references to additional information and their default values. In the following table *octave-home* stands for the root directory where all of Octave is installed (the default is `'/usr'`, *version* stands for the Octave version number (for example, 2.9.13) and *arch* stands for the type of system for which Octave was compiled (for example, `x86_64-unknown-linux-gnu`).

EDITOR See [Section 2.4.5 \[Commands For History\]](#), page 21.
 Default value: `"emacs"`.

EXEC_PATH
 See [Section 36.5 \[Controlling Subprocesses\]](#), page 443.
 Default value: `":$PATH"`.

OCTAVE_HOME
 Default value: `"/usr"`.

PAGER See [Chapter 15 \[Input and Output\]](#), page 151.
 Default value: `"less"`, or `"more"`.

PS1 See [Section 2.4.7 \[Customizing the Prompt\]](#), page 24.
 Default value: `"\s:\#> "`.

PS2 See [Section 2.4.7 \[Customizing the Prompt\]](#), page 24.
 Default value: `> "`.

PS4 See [Section 2.4.7 \[Customizing the Prompt\]](#), page 24.
 Default value: `"+ "`.

beep_on_error
 See [Chapter 13 \[Errors and Warnings\]](#), page 137.
 Default value: 0.

completion_append_char
 See [Section 2.4.4 \[Commands For Completion\]](#), page 21.
 Default value: `" "`.

default_save_options

See [Section 15.1.3 \[Simple File I/O\]](#), page 156.

Default value: "ascii".

crash_dumps_octave_core

See [Section 15.1.3 \[Simple File I/O\]](#), page 156.

Default value: 1.

fixed_point_format

See [Section 4.1 \[Matrices\]](#), page 36.

Default value: 0.

gnuplot_binary

See [Section 16.3 \[Three-Dimensional Plotting\]](#), page 187.

Default value: "gnuplot".

history_file

See [Section 2.4.5 \[Commands For History\]](#), page 21.

Default value: "~/.octave_hist".

history_size

See [Section 2.4.5 \[Commands For History\]](#), page 21.

Default value: 1024.

ignore_function_time_stamp

See [Section 12.7 \[Function Files\]](#), page 124.

Default value: "system".

max_recursion_depth

See [Section 9.2.2 \[Recursion\]](#), page 91.

Default value: 256.

output_max_field_width

See [Section 4.1 \[Matrices\]](#), page 36.

Default value: 10.

output_precision

See [Section 4.1 \[Matrices\]](#), page 36.

Default value: 5.

page_screen_output

See [Chapter 15 \[Input and Output\]](#), page 151.

Default value: 1.

print_answer_id_name

See [Section 15.1.1 \[Terminal Output\]](#), page 151.

Default value: 1.

print_empty_dimensions

See [Section 4.1.1 \[Empty Matrices\]](#), page 38.

Default value: 1.

`return_last_computed_value`
See [Section 12.5 \[Returning From a Function\]](#), page 123.
Default value: 0.

`save_precision`
See [Section 15.1.3 \[Simple File I/O\]](#), page 156.
Default value: 17.

`saving_history`
See [Section 2.4.5 \[Commands For History\]](#), page 21.
Default value: 1.

`sigdump_dumps_octave_core`
See [Section 15.1.3 \[Simple File I/O\]](#), page 156.
Default value: 1.

`sigterm_dumps_octave_core`
See [Section 15.1.3 \[Simple File I/O\]](#), page 156.
Default value: 1.

`silent_functions`
See [Section 12.1 \[Defining Functions\]](#), page 117.
Default value: 0.

`split_long_rows`
See [Section 4.1 \[Matrices\]](#), page 36.
Default value: 1.

`struct_levels_to_print`
See [Chapter 6 \[Data Structures\]](#), page 63.
Default value: 2.

`suppress_verbose_help_message`
See [Section 2.3 \[Getting Help\]](#), page 17.
Default value: 1.

8.5 Defaults from the Environment

Octave uses the values of the following environment variables to set the default values for the corresponding built-in or internal variables. In addition, the values from the environment may be overridden by command-line arguments. See [Section 2.1.1 \[Command Line Options\]](#), page 13.

`EDITOR` See [Section 2.4.5 \[Commands For History\]](#), page 21.
Built-in variable: `EDITOR`.

`OCTAVE_EXEC_PATH`
See [Section 36.5 \[Controlling Subprocesses\]](#), page 443.
Built-in variable: `EXEC_PATH`. Command-line argument: `--exec-path`.

OCTAVE_PATH

See [Section 12.7 \[Function Files\]](#), page 124.

Internal variable changed by function `path`. Command-line argument: `--path`.

OCTAVE_INFO_FILE

See [Section 2.3 \[Getting Help\]](#), page 17.

Internal variable changed by function `info_file`. Command-line argument: `--info-file`.

OCTAVE_INFO_PROGRAM

See [Section 2.3 \[Getting Help\]](#), page 17.

Internal variable changed by function `info_program`. Command-line argument: `--info-program`.

OCTAVE_HISTSIZE

See [Section 2.4.5 \[Commands For History\]](#), page 21.

Built-in variable: `history_size`.

OCTAVE_HISTFILE

See [Section 2.4.5 \[Commands For History\]](#), page 21.

Built-in variable: `history_file`.

9 Expressions

Expressions are the basic building block of statements in Octave. An expression evaluates to a value, which you can print, test, store in a variable, pass to a function, or assign a new value to a variable with an assignment operator.

An expression can serve as a statement on its own. Most other kinds of statements contain one or more expressions which specify data to be operated on. As in other languages, expressions in Octave include variables, array references, constants, and function calls, as well as combinations of these with various operators.

9.1 Index Expressions

An *index expression* allows you to reference or extract selected elements of a matrix or vector.

Indices may be scalars, vectors, ranges, or the special operator ‘:’, which may be used to select entire rows or columns.

Vectors are indexed using a single index expression. Matrices may be indexed using one or two indices. When using a single index expression, the elements of the matrix are taken in column-first order; the dimensions of the output match those of the index expression. For example,

```
a (2)      # a scalar
a (1:2)     # a row vector
a ([1; 2])  # a column vector
```

As a special case, when a colon is used as a single index, the output is a column vector containing all the elements of the vector or matrix. For example

```
a (:)      # a column vector
```

Given the matrix

```
a = [1, 2; 3, 4]
```

all of the following expressions are equivalent

```
a (1, [1, 2])
a (1, 1:2)
a (1, :)
```

and select the first row of the matrix.

Indexing a scalar with a vector of ones can be used to create a vector the same size as the index vector, with each element equal to the value of the original scalar. For example, the following statements

```
a = 13;
a ([1, 1, 1, 1])
```

produce a vector whose four elements are all equal to 13.

Similarly, indexing a scalar with two vectors of ones can be used to create a matrix. For example the following statements

```
a = 13;
a ([1, 1], [1, 1, 1])
```

create a 2 by 3 matrix with all elements equal to 13.

This is an obscure notation and should be avoided. It is better to use the function `ones` to generate a matrix of the appropriate size whose elements are all one, and then to scale it to produce the desired result. See [Section 17.4 \[Special Utility Matrices\]](#), page 204.

It is also possible to create a matrix with different values. The following example create a 10 dimensional row vector a containing the values $a_i = \sqrt{i}$.

```
for i = 1:10
    a(i) = sqrt (i);
endfor
```

Note that it is quite inefficient to create a vector using a loop like the one shown in the example above. In this particular case, it would have been much more efficient to use the expression

```
a = sqrt (1:10);
```

thus avoiding the loop entirely. In cases where a loop is still required, or a number of values must be combined to form a larger matrix, it is generally much faster to set the size of the matrix first, and then insert elements using indexing commands. For example, given a matrix a ,

```
[nr, nc] = size (a);
x = zeros (nr, n * nc);
for i = 1:n
    x(:,(i-1)*nc+1:i*nc) = a;
endfor
```

is considerably faster than

```
x = a;
for i = 1:n-1
    x = [x, a];
endfor
```

particularly for large matrices because Octave does not have to repeatedly resize the result.

subsref (val, idx) [Built-in Function]

Perform the subscripted element selection operation according to the subscript specified by idx .

The subscript idx is expected to be a structure array with fields `'type'` and `'subs'`. Valid values for `'type'` are `"()"`, `"{}"`, and `"."`. The `'subs'` field may be either `":"` or a cell array of index values.

The following example shows how to extract the two first columns of a matrix

```
val = magic(3)
⇒ val = [ 8    1    6
          3    5    7
          4    9    2 ]

idx.type = "()";
idx.subs = {":", 1:2};
subsref(val, idx)
⇒ [ 8    1
```

```

      3   5
      4   9 ]

```

Note that this is the same as writing `val(:,1:2)`.

See also: `subsasgn`, `substruct`.

```
ind = sub2ind (dims, i, j) [Function File]
```

```
ind = sub2ind (dims, s1, s2, ..., sN) [Function File]
```

Convert subscripts into a linear index.

The following example shows how to convert the two-dimensional index (2,3) of a 3-by-3 matrix to a linear index.

```

linear_index = sub2ind ([3, 3], 2, 3)
⇒ 8

```

See also: `ind2sub`.

```
[s1, s2, ..., sN] = ind2sub (dims, ind) [Function File]
```

Convert a linear index into subscripts.

The following example shows how to convert the linear index 8 in a 3-by-3 matrix into a subscript.

```

[r, c] = ind2sub ([3, 3], 8)
⇒ r =  2
   c =  3

```

See also: `sub2ind`.

9.2 Calling Functions

A *function* is a name for a particular calculation. Because it has a name, you can ask for it by name at any point in the program. For example, the function `sqrt` computes the square root of a number.

A fixed set of functions are *built-in*, which means they are available in every Octave program. The `sqrt` function is one of these. In addition, you can define your own functions. See [Chapter 12 \[Functions and Scripts\]](#), page 117, for information about how to do this.

The way to use a function is with a *function call* expression, which consists of the function name followed by a list of *arguments* in parentheses. The arguments are expressions which give the raw materials for the calculation that the function will do. When there is more than one argument, they are separated by commas. If there are no arguments, you can omit the parentheses, but it is a good idea to include them anyway, to clearly indicate that a function call was intended. Here are some examples:

```

sqrt (x^2 + y^2)    # One argument
ones (n, m)         # Two arguments
rand ()             # No arguments

```

Each function expects a particular number of arguments. For example, the `sqrt` function must be called with a single argument, the number to take the square root of:

```
sqrt (argument)
```

Some of the built-in functions take a variable number of arguments, depending on the particular usage, and their behavior is different depending on the number of arguments supplied.

Like every other expression, the function call has a value, which is computed by the function based on the arguments you give it. In this example, the value of `sqrt (argument)` is the square root of the argument. A function can also have side effects, such as assigning the values of certain variables or doing input or output operations.

Unlike most languages, functions in Octave may return multiple values. For example, the following statement

```
[u, s, v] = svd (a)
```

computes the singular value decomposition of the matrix `a` and assigns the three result matrices to `u`, `s`, and `v`.

The left side of a multiple assignment expression is itself a list of expressions, and is allowed to be a list of variable names or index expressions. See also [Section 9.1 \[Index Expressions\]](#), page 87, and [Section 9.6 \[Assignment Ops\]](#), page 96.

9.2.1 Call by Value

In Octave, unlike Fortran, function arguments are passed by value, which means that each argument in a function call is evaluated and assigned to a temporary location in memory before being passed to the function. There is currently no way to specify that a function parameter should be passed by reference instead of by value. This means that it is impossible to directly alter the value of function parameter in the calling function. It can only change the local copy within the function body. For example, the function

```
function f (x, n)
  while (n-- > 0)
    disp (x);
  endwhile
endfunction
```

displays the value of the first argument `n` times. In this function, the variable `n` is used as a temporary variable without having to worry that its value might also change in the calling function. Call by value is also useful because it is always possible to pass constants for any function parameter without first having to determine that the function will not attempt to modify the parameter.

The caller may use a variable as the expression for the argument, but the called function does not know this: it only knows what value the argument had. For example, given a function called as

```
foo = "bar";
fcn (foo)
```

you should not think of the argument as being “the variable `foo`.” Instead, think of the argument as the string value, `"bar"`.

Even though Octave uses pass-by-value semantics for function arguments, values are not copied unnecessarily. For example,

```
x = rand (1000);
f (x);
```

does not actually force two 1000 by 1000 element matrices to exist *unless* the function `f` modifies the value of its argument. Then Octave must create a copy to avoid changing the value outside the scope of the function `f`, or attempting (and probably failing!) to modify the value of a constant or the value of a temporary result.

9.2.2 Recursion

With some restrictions¹, recursive function calls are allowed. A *recursive function* is one which calls itself, either directly or indirectly. For example, here is an inefficient² way to compute the factorial of a given integer:

```
function retval = fact (n)
  if (n > 0)
    retval = n * fact (n-1);
  else
    retval = 1;
  endif
endfunction
```

This function is recursive because it calls itself directly. It eventually terminates because each time it calls itself, it uses an argument that is one less than was used for the previous call. Once the argument is no longer greater than zero, it does not call itself, and the recursion ends.

The built-in variable `max_recursion_depth` specifies a limit to the recursion depth and prevents Octave from recursing infinitely.

```
val = max_recursion_depth () [Built-in Function]
old_val = max_recursion_depth (new_val) [Built-in Function]
```

Query or set the internal limit on the number of times a function may be called recursively. If the limit is exceeded, an error message is printed and control returns to the top level.

9.3 Arithmetic Operators

The following arithmetic operators are available, and work on scalars and matrices.

<code>x + y</code>	Addition. If both operands are matrices, the number of rows and columns must both agree. If one operand is a scalar, its value is added to all the elements of the other operand.
<code>x .+ y</code>	Element by element addition. This operator is equivalent to <code>+</code> .
<code>x - y</code>	Subtraction. If both operands are matrices, the number of rows and columns of both must agree.

¹ Some of Octave's function are implemented in terms of functions that cannot be called recursively. For example, the ODE solver `lsode` is ultimately implemented in a Fortran subroutine that cannot be called recursively, so `lsode` should not be called either directly or indirectly from within the user-supplied function that `lsode` requires. Doing so will result in an error.

² It would be much better to use `prod (1:n)`, or `gamma (n+1)` instead, after first checking to ensure that the value `n` is actually a positive integer.

<code>x .- y</code>	Element by element subtraction. This operator is equivalent to <code>-</code> .
<code>x * y</code>	Matrix multiplication. The number of columns of <code>x</code> must agree with the number of rows of <code>y</code> .
<code>x .* y</code>	Element by element multiplication. If both operands are matrices, the number of rows and columns must both agree.
<code>x / y</code>	Right division. This is conceptually equivalent to the expression $(\text{inverse}(y') * x')'$ but it is computed without forming the inverse of <code>y'</code> . If the system is not square, or if the coefficient matrix is singular, a minimum norm solution is computed.
<code>x ./ y</code>	Element by element right division.
<code>x \ y</code>	Left division. This is conceptually equivalent to the expression $\text{inverse}(x) * y$ but it is computed without forming the inverse of <code>x</code> . If the system is not square, or if the coefficient matrix is singular, a minimum norm solution is computed.
<code>x .\ y</code>	Element by element left division. Each element of <code>y</code> is divided by each corresponding element of <code>x</code> .
<code>x ^ y</code> <code>x ** y</code>	Power operator. If <code>x</code> and <code>y</code> are both scalars, this operator returns <code>x</code> raised to the power <code>y</code> . If <code>x</code> is a scalar and <code>y</code> is a square matrix, the result is computed using an eigenvalue expansion. If <code>x</code> is a square matrix. the result is computed by repeated multiplication if <code>y</code> is an integer, and by an eigenvalue expansion if <code>y</code> is not an integer. An error results if both <code>x</code> and <code>y</code> are matrices. The implementation of this operator needs to be improved.
<code>x .^ y</code>	
<code>x .** y</code>	Element by element power operator. If both operands are matrices, the number of rows and columns must both agree.
<code>-x</code>	Negation.
<code>+x</code>	Unary plus. This operator has no effect on the operand.
<code>x'</code>	Complex conjugate transpose. For real arguments, this operator is the same as the transpose operator. For complex arguments, this operator is equivalent to the expression $\text{conj}(x.')$
<code>x.'</code>	Transpose.

Note that because Octave's element by element operators begin with a `'.'`, there is a possible ambiguity for statements like

```
1./m
```

because the period could be interpreted either as part of the constant or as part of the operator. To resolve this conflict, Octave treats the expression as if you had typed

(1) ./ m

and not

(1.) / m

Although this is inconsistent with the normal behavior of Octave's lexer, which usually prefers to break the input into tokens by preferring the longest possible match at any given point, it is more useful in this case.

9.4 Comparison Operators

Comparison operators compare numeric values for relationships such as equality. They are written using *relational operators*.

All of Octave's comparison operators return a value of 1 if the comparison is true, or 0 if it is false. For matrix values, they all work on an element-by-element basis. For example,

```
[1, 2; 3, 4] == [1, 3; 2, 4]
⇒  1  0
   0  1
```

If one operand is a scalar and the other is a matrix, the scalar is compared to each element of the matrix in turn, and the result is the same size as the matrix.

<code>x < y</code>	True if x is less than y.
<code>x <= y</code>	True if x is less than or equal to y.
<code>x == y</code>	True if x is equal to y.
<code>x >= y</code>	True if x is greater than or equal to y.
<code>x > y</code>	True if x is greater than y.
<code>x != y</code>	
<code>x ~= y</code>	
<code>x <> y</code>	True if x is not equal to y.

String comparisons may also be performed with the `strcmp` function, not with the comparison operators listed above. See [Chapter 5 \[Strings\]](#), page 47.

`isequal (x1, x2, ...)` [Function File]

Return true if all of `x1`, `x2`, ... are equal.

See also: `isequalwithequalnans`.

`isequalwithequalnans (x1, x2, ...)` [Function File]

Assuming `NaN == NaN`, return true if all of `x1`, `x2`, ... are equal.

See also: `isequal`.

9.5 Boolean Expressions

9.5.1 Element-by-element Boolean Operators

An *element-by-element boolean expression* is a combination of comparison expressions using the boolean operators “or” (`|`), “and” (`&`), and “not” (`!`), along with parentheses to control nesting. The truth of the boolean expression is computed by combining the truth values of the corresponding elements of the component expressions. A value is considered to be false if it is zero, and true otherwise.

Element-by-element boolean expressions can be used wherever comparison expressions can be used. They can be used in `if` and `while` statements. However, if a matrix value used as the condition in an `if` or `while` statement is only true if *all* of its elements are nonzero.

Like comparison operations, each element of an element-by-element boolean expression also has a numeric value (1 if true, 0 if false) that comes into play if the result of the boolean expression is stored in a variable, or used in arithmetic.

Here are descriptions of the three element-by-element boolean operators.

`boolean1 & boolean2`

Elements of the result are true if both corresponding elements of *boolean1* and *boolean2* are true.

`boolean1 | boolean2`

Elements of the result are true if either of the corresponding elements of *boolean1* or *boolean2* is true.

`! boolean`

`~ boolean`

Each element of the result is true if the corresponding element of *boolean* is false.

For matrix operands, these operators work on an element-by-element basis. For example, the expression

```
[1, 0; 0, 1] & [1, 0; 2, 3]
```

returns a two by two identity matrix.

For the binary operators, the dimensions of the operands must conform if both are matrices. If one of the operands is a scalar and the other a matrix, the operator is applied to the scalar and each element of the matrix.

For the binary element-by-element boolean operators, both subexpressions *boolean1* and *boolean2* are evaluated before computing the result. This can make a difference when the expressions have side effects. For example, in the expression

```
a & b++
```

the value of the variable *b* is incremented even if the variable *a* is zero.

This behavior is necessary for the boolean operators to work as described for matrix-valued operands.

9.5.2 Short-circuit Boolean Operators

Combined with the implicit conversion to scalar values in `if` and `while` conditions, Octave's element-by-element boolean operators are often sufficient for performing most logical operations. However, it is sometimes desirable to stop evaluating a boolean expression as soon as the overall truth value can be determined. Octave's *short-circuit* boolean operators work this way.

`boolean1 && boolean2`

The expression `boolean1` is evaluated and converted to a scalar using the equivalent of the operation `all (boolean1 (:))`. If it is false, the result of the overall expression is 0. If it is true, the expression `boolean2` is evaluated and converted to a scalar using the equivalent of the operation `all (boolean1 (:))`. If it is true, the result of the overall expression is 1. Otherwise, the result of the overall expression is 0.

Warning: there is one exception to the rule of evaluating `all (boolean1 (:))`, which is when `boolean1` is the empty matrix. The truth value of an empty matrix is always `false` so `[] && true` evaluates to `false` even though `all ([])` is `true`.

`boolean1 || boolean2`

The expression `boolean1` is evaluated and converted to a scalar using the equivalent of the operation `all (boolean1 (:))`. If it is true, the result of the overall expression is 1. If it is false, the expression `boolean2` is evaluated and converted to a scalar using the equivalent of the operation `all (boolean1 (:))`. If it is true, the result of the overall expression is 1. Otherwise, the result of the overall expression is 0.

Warning: the truth value of an empty matrix is always `false`, see the previous list item for details.

The fact that both operands may not be evaluated before determining the overall truth value of the expression can be important. For example, in the expression

```
a && b++
```

the value of the variable `b` is only incremented if the variable `a` is nonzero.

This can be used to write somewhat more concise code. For example, it is possible write

```
function f (a, b, c)
  if (nargin > 2 && isstr (c))
    ...
```

instead of having to use two `if` statements to avoid attempting to evaluate an argument that doesn't exist. For example, without the short-circuit feature, it would be necessary to write

```
function f (a, b, c)
  if (nargin > 2)
    if (isstr (c))
      ...
```

Writing

```
function f (a, b, c)
  if (nargin > 2 & isstr (c))
    ...
```

would result in an error if `f` were called with one or two arguments because Octave would be forced to try to evaluate both of the operands for the operator `'&'`.

9.6 Assignment Expressions

An *assignment* is an expression that stores a new value into a variable. For example, the following expression assigns the value 1 to the variable `z`:

```
z = 1
```

After this expression is executed, the variable `z` has the value 1. Whatever old value `z` had before the assignment is forgotten. The `'='` sign is called an *assignment operator*.

Assignments can store string values also. For example, the following expression would store the value `"this food is good"` in the variable `message`:

```
thing = "food"
predicate = "good"
message = [ "this " , thing , " is " , predicate ]
```

(This also illustrates concatenation of strings.)

Most operators (addition, concatenation, and so on) have no effect except to compute a value. If you ignore the value, you might as well not use the operator. An assignment operator is different. It does produce a value, but even if you ignore the value, the assignment still makes itself felt through the alteration of the variable. We call this a *side effect*.

The left-hand operand of an assignment need not be a variable (see [Chapter 8 \[Variables\]](#), [page 77](#)). It can also be an element of a matrix (see [Section 9.1 \[Index Expressions\]](#), [page 87](#)) or a list of return values (see [Section 9.2 \[Calling Functions\]](#), [page 89](#)). These are all called *lvalues*, which means they can appear on the left-hand side of an assignment operator. The right-hand operand may be any expression. It produces the new value which the assignment stores in the specified variable, matrix element, or list of return values.

It is important to note that variables do *not* have permanent types. The type of a variable is simply the type of whatever value it happens to hold at the moment. In the following program fragment, the variable `foo` has a numeric value at first, and a string value later on:

```
octave:13> foo = 1
foo = 1
octave:13> foo = "bar"
foo = bar
```

When the second assignment gives `foo` a string value, the fact that it previously had a numeric value is forgotten.

Assignment of a scalar to an indexed matrix sets all of the elements that are referenced by the indices to the scalar value. For example, if `a` is a matrix with at least two columns,

```
a(:, 2) = 5
```

sets all the elements in the second column of `a` to 5.

Assigning an empty matrix ‘[]’ works in most cases to allow you to delete rows or columns of matrices and vectors. See [Section 4.1.1 \[Empty Matrices\]](#), page 38. For example, given a 4 by 5 matrix *A*, the assignment

```
A (3, :) = []
```

deletes the third row of *A*, and the assignment

```
A (:, 1:2:5) = []
```

deletes the first, third, and fifth columns.

An assignment is an expression, so it has a value. Thus, *z* = 1 as an expression has the value 1. One consequence of this is that you can write multiple assignments together:

```
x = y = z = 0
```

stores the value 0 in all three variables. It does this because the value of *z* = 0, which is 0, is stored into *y*, and then the value of *y* = *z* = 0, which is 0, is stored into *x*.

This is also true of assignments to lists of values, so the following is a valid expression

```
[a, b, c] = [u, s, v] = svd (a)
```

that is exactly equivalent to

```
[u, s, v] = svd (a)
```

```
a = u
```

```
b = s
```

```
c = v
```

In expressions like this, the number of values in each part of the expression need not match. For example, the expression

```
[a, b] = [u, s, v] = svd (a)
```

is equivalent to

```
[u, s, v] = svd (a)
```

```
a = u
```

```
b = s
```

The number of values on the left side of the expression can, however, not exceed the number of values on the right side. For example, the following will produce an error.

```
[a, b, c, d] = [u, s, v] = svd (a)
```

```
    error: element number 4 undefined in return list
```

```
    error: evaluating assignment expression near line 8, column 15
```

A very common programming pattern is to increment an existing variable with a given value, like this

```
a = a + 2;
```

This can be written in a clearer and more condensed form using the += operator

```
a += 2;
```

Similar operators also exist for subtraction (–=), multiplication (*=), and division (/=). An expression of the form

```
expr1 op= expr2
```

is evaluated as

```
expr1 = (expr1) op (expr2)
```

where *op* can be either +, -, *, or /. So, the expression

```
a *= b+1
```

is evaluated as

```
a = a * (b+1)
```

and *not*

```
a = a * b + 1
```

You can use an assignment anywhere an expression is called for. For example, it is valid to write `x != (y = 1)` to set `y` to 1 and then test whether `x` equals 1. But this style tends to make programs hard to read. Except in a one-shot program, you should rewrite it to get rid of such nesting of assignments. This is never very hard.

subsasgn (*val*, *idx*, *rhs*) [Built-in Function]

Perform the subscripted assignment operation according to the subscript specified by *idx*.

The subscript *idx* is expected to be a structure array with fields ‘*type*’ and ‘*subs*’. Valid values for ‘*type*’ are ‘“()”’, ‘“{ }”’, and ‘“.”’’. The ‘*subs*’ field may be either ‘“:”’ or a cell array of index values.

The following example shows how to set the two first columns of a 3-by-3 matrix to zero.

```
val = magic(3);
idx.type = "()";
idx.subs = {":", 1:2};
subsasgn (val, idx, 0)
⇒ [ 0  0  6
    0  0  7
    0  0  2 ]
```

Note that this is the same as writing `val(:,1:2) = 0`.

See also: `subsref`, `substruct`.

9.7 Increment Operators

Increment operators increase or decrease the value of a variable by 1. The operator to increment a variable is written as ‘++’. It may be used to increment a variable either before or after taking its value.

For example, to pre-increment the variable `x`, you would write `++x`. This would add one to `x` and then return the new value of `x` as the result of the expression. It is exactly the same as the expression `x = x + 1`.

To post-increment a variable `x`, you would write `x++`. This adds one to the variable `x`, but returns the value that `x` had prior to incrementing it. For example, if `x` is equal to 2, the result of the expression `x++` is 2, and the new value of `x` is 3.

For matrix and vector arguments, the increment and decrement operators work on each element of the operand.

Here is a list of all the increment and decrement expressions.

<code>++x</code>	This expression increments the variable <code>x</code> . The value of the expression is the <i>new</i> value of <code>x</code> . It is equivalent to the expression <code>x = x + 1</code> .
<code>--x</code>	This expression decrements the variable <code>x</code> . The value of the expression is the <i>new</i> value of <code>x</code> . It is equivalent to the expression <code>x = x - 1</code> .
<code>x++</code>	This expression causes the variable <code>x</code> to be incremented. The value of the expression is the <i>old</i> value of <code>x</code> .
<code>x--</code>	This expression causes the variable <code>x</code> to be decremented. The value of the expression is the <i>old</i> value of <code>x</code> .

9.8 Operator Precedence

Operator precedence determines how operators are grouped, when different operators appear close by in one expression. For example, `*` has higher precedence than `+`. Thus, the expression `a + b * c` means to multiply `b` and `c`, and then add `a` to the product (i.e., `a + (b * c)`).

You can overrule the precedence of the operators by using parentheses. You can think of the precedence rules as saying where the parentheses are assumed if you do not write parentheses yourself. In fact, it is wise to use parentheses whenever you have an unusual combination of operators, because other people who read the program may not remember what the precedence is in this case. You might forget as well, and then you too could make a mistake. Explicit parentheses will help prevent any such mistake.

When operators of equal precedence are used together, the leftmost operator groups first, except for the assignment and exponentiation operators, which group in the opposite order. Thus, the expression `a - b + c` groups as `(a - b) + c`, but the expression `a = b = c` groups as `a = (b = c)`.

The precedence of prefix unary operators is important when another operator follows the operand. For example, `-x^2` means `-(x^2)`, because `-` has lower precedence than `^`.

Here is a table of the operators in Octave, in order of increasing precedence.

statement separators

`;', '.,`

assignment

`=, '+=', '-=', '*=', '/='`. This operator groups right to left.

logical "or" and "and"

`||, &&`.

element-wise "or" and "and"

`|, &`.

relational

`<, <=, ==, >=, >, !=, ~=, <>`.

colon

`:`.

add, subtract

`+, -`.

```
multiply, divide
    '*', '/', '\', '.\', '.*', './'.

transpose
    ',', '.,'

unary plus, minus, increment, decrement, and 'not'
    '+', '-', '++', '--', '!', '~'.

exponentiation
    '^', '**', '.^', '**'.
```


10 Evaluation

Normally, you evaluate expressions simply by typing them at the Octave prompt, or by asking Octave to interpret commands that you have saved in a file.

Sometimes, you may find it necessary to evaluate an expression that has been computed and stored in a string, which is exactly what the `eval` lets you do.

`eval (try, catch)` [Built-in Function]

Parse the string *try* and evaluate it as if it were an Octave program. If that fails, evaluate the optional string *catch*. The string *try* is evaluated in the current context, so any results remain available after `eval` returns.

The following example makes the variable *a* with the approximate value 3.1416 available.

```
eval("a = acos(-1);");
```

If an error occurs during the evaluation of *try* the *catch* string is evaluated, as the following example shows.

```
eval ('error ("This is a bad example");',
      'printf ("This error occurred:\n%s", lasterr ());');
+ This error occurred:
  error: This is a bad example
```

10.1 Calling a Function by its Name

The `feval` function allow you to call a function from a string containing its name. This is useful when writing a function that need to call user-supplied functions. The `feval` function takes the name of the function to call as its first argument, and the remaining arguments are given to the function.

The following example is a simple-minded function using `feval` that finds the root of a user-supplied function of one variable using Newton's method.

```

function result = newtroot (fname, x)

# usage: newtroot (fname, x)
#
#   fname : a string naming a function f(x).
#   x      : initial guess

delta = tol = sqrt (eps);
maxit = 200;
fx = feval (fname, x);
for i = 1:maxit
  if (abs (fx) < tol)
    result = x;
    return;
  else
    fx_new = feval (fname, x + delta);
    deriv = (fx_new - fx) / delta;
    x = x - fx / deriv;
    fx = fx_new;
  endif
endfor

result = x;

endfunction

```

Note that this is only meant to be an example of calling user-supplied functions and should not be taken too seriously. In addition to using a more robust algorithm, any serious code would check the number and type of all the arguments, ensure that the supplied function really was a function, etc. See [Section 4.6 \[Predicates for Numeric Objects\]](#), page 45, for example, for a list of predicates for numeric objects, and see [Section 8.3 \[Status of Variables\]](#), page 79, for a description of the `exist` function.

feval (*name*, ...) [Built-in Function]

Evaluate the function named *name*. Any arguments after the first are passed on to the named function. For example,

```

feval ("acos", -1)
⇒ 3.1416

```

calls the function `acos` with the argument `-1`.

The function `feval` is necessary in order to be able to write functions that call user-supplied functions, because Octave does not have a way to declare a pointer to a function (like C) or to declare a special kind of variable that can be used to hold the name of a function (like `EXTERNAL` in Fortran). Instead, you must refer to functions by name, and use `feval` to call them.

10.2 Evaluation in a Different Context

Before you evaluate an expression you need to substitute the values of the variables used in the expression. These are stored in the symbol table. Whenever the interpreter starts a new function it saves the current symbol table and creates a new one, initializing it with the list of function parameters and a couple of predefined variables such as `nargin`. Expressions inside the function use the new symbol table.

Sometimes you want to write a function so that when you call it, it modifies variables in your own context. This allows you to use a pass-by-name style of function, which is similar to using a pointer in programming languages such as C.

Consider how you might write `save` and `load` as m-files. For example,

```
function create_data
    x = linspace (0, 10, 10);
    y = sin (x);
    save mydata x y
endfunction
```

With `evalin`, you could write `save` as follows:

```
function save (file, name1, name2)
    f = open_save_file (file);
    save_var(f, name1, evalin ("caller", name1));
    save_var(f, name2, evalin ("caller", name2));
endfunction
```

Here, `'caller'` is the `create_data` function and `name1` is the string `"x"`, which evaluates simply as the value of `x`.

You later want to load the values back from `mydata` in a different context:

```
function process_data
    load mydata
    ... do work ...
endfunction
```

With `assignin`, you could write `load` as follows:

```
function load (file)
    f = open_load_file (file);
    [name, val] = load_var (f);
    assignin ("caller", name, val);
    [name, val] = load_var (f);
    assignin ("caller", name, val);
endfunction
```

Here, `'caller'` is the `process_data` function.

You can set and use variables at the command prompt using the context `'base'` rather than `'caller'`.

These functions are rarely used in practice. One example is the `fail ('code', 'pattern')` function which evaluates `'code'` in the caller's context and checks that the error message it produces matches the given pattern. Other examples such as `save` and `load` are written in C++ where all octave variables are in the `'caller'` context and `evalin` is not needed.

`evalin (context, try, catch)` [Built-in Function]

Like `eval`, except that the expressions are evaluated in the context *context*, which may be either `"caller"` or `"base"`.

`assignin (context, varname, value)` [Built-in Function]

Assign *value* to *varname* in context *context*, which may be either `"base"` or `"caller"`.

11 Statements

Statements may be a simple constant expression or a complicated list of nested loops and conditional statements.

Control statements such as `if`, `while`, and so on control the flow of execution in Octave programs. All the control statements start with special keywords such as `if` and `while`, to distinguish them from simple expressions. Many control statements contain other statements; for example, the `if` statement contains another statement which may or may not be executed.

Each control statement has a corresponding *end* statement that marks the end of the end of the control statement. For example, the keyword `endif` marks the end of an `if` statement, and `endwhile` marks the end of a `while` statement. You can use the keyword `end` anywhere a more specific end keyword is expected, but using the more specific keywords is preferred because if you use them, Octave is able to provide better diagnostics for mismatched or missing end tokens.

The list of statements contained between keywords like `if` or `while` and the corresponding end statement is called the *body* of a control statement.

11.1 The if Statement

The `if` statement is Octave's decision-making statement. There are three basic forms of an `if` statement. In its simplest form, it looks like this:

```
if (condition)
    then-body
endif
```

condition is an expression that controls what the rest of the statement will do. The *then-body* is executed only if *condition* is true.

The condition in an `if` statement is considered true if its value is non-zero, and false if its value is zero. If the value of the conditional expression in an `if` statement is a vector or a matrix, it is considered true only if it is non-empty and *all* of the elements are non-zero.

The second form of an `if` statement looks like this:

```
if (condition)
    then-body
else
    else-body
endif
```

If *condition* is true, *then-body* is executed; otherwise, *else-body* is executed.

Here is an example:

```
if (rem (x, 2) == 0)
    printf ("x is even\n");
else
    printf ("x is odd\n");
endif
```

In this example, if the expression `rem (x, 2) == 0` is true (that is, the value of `x` is divisible by 2), then the first `printf` statement is evaluated, otherwise the second `printf` statement is evaluated.

The third and most general form of the `if` statement allows multiple decisions to be combined in a single statement. It looks like this:

```
if (condition)
  then-body
elseif (condition)
  elseif-body
else
  else-body
endif
```

Any number of `elseif` clauses may appear. Each condition is tested in turn, and if one is found to be true, its corresponding *body* is executed. If none of the conditions are true and the `else` clause is present, its body is executed. Only one `else` clause may appear, and it must be the last part of the statement.

In the following example, if the first condition is true (that is, the value of `x` is divisible by 2), then the first `printf` statement is executed. If it is false, then the second condition is tested, and if it is true (that is, the value of `x` is divisible by 3), then the second `printf` statement is executed. Otherwise, the third `printf` statement is performed.

```
if (rem (x, 2) == 0)
  printf ("x is even\n");
elseif (rem (x, 3) == 0)
  printf ("x is odd and divisible by 3\n");
else
  printf ("x is odd\n");
endif
```

Note that the `elseif` keyword must not be spelled `else if`, as is allowed in Fortran. If it is, the space between the `else` and `if` will tell Octave to treat this as a new `if` statement within another `if` statement's `else` clause. For example, if you write

```
if (c1)
  body-1
else if (c2)
  body-2
endif
```

Octave will expect additional input to complete the first `if` statement. If you are using Octave interactively, it will continue to prompt you for additional input. If Octave is reading this input from a file, it may complain about missing or mismatched `end` statements, or, if you have not used the more specific `endif`, `endfor`, etc.), it may simply produce incorrect results, without producing any warning messages.

It is much easier to see the error if we rewrite the statements above like this,

```
if (c1)
    body-1
else
    if (c2)
        body-2
    endif
endif
```

using the indentation to show how Octave groups the statements. See [Chapter 12 \[Functions and Scripts\]](#), page 117.

11.2 The switch Statement

It is very common to take different actions depending on the value of one variable. This is possible using the `if` statement in the following way

```
if (X == 1)
    do_something ();
elseif (X == 2)
    do_something_else ();
else
    do_something_completely_different ();
endif
```

This kind of code can however be very cumbersome to both write and maintain. To overcome this problem Octave supports the `switch` statement. Using this statement, the above example becomes

```
switch (X)
    case 1
        do_something ();
    case 2
        do_something_else ();
    otherwise
        do_something_completely_different ();
endswitch
```

This code makes the repetitive structure of the problem more explicit, making the code easier to read, and hence maintain. Also, if the variable `X` should change its name, only one line would need changing compared to one line per case when `if` statements are used.

The general form of the `switch` statement is

```
switch expression
    case label
        command_list
    case label
        command_list
    ...

    otherwise
        command_list
endswitch
```

where *label* can be any expression. However, duplicate *label* values are not detected, and only the *command_list* corresponding to the first match will be executed. For the **switch** statement to be meaningful at least one **case label command_list** clause must be present, while the **otherwise command_list** clause is optional.

If *label* is a cell array the corresponding *command_list* is executed if *any* of the elements of the cell array match *expression*. As an example, the following program will print ‘Variable is either 6 or 7’.

```
A = 7;
switch A
  case { 6, 7 }
    printf ("variable is either 6 or 7\n");
  otherwise
    printf ("variable is neither 6 nor 7\n");
endswitch
```

As with all other specific end keywords, **endswitch** may be replaced by **end**, but you can get better diagnostics if you use the specific forms.

One advantage of using the **switch** statement compared to using **if** statements is that the *labels* can be strings. If an **if** statement is used it is *not* possible to write

```
if (X == "a string") # This is NOT valid
```

since a character-to-character comparison between *X* and the string will be made instead of evaluating if the strings are equal. This special-case is handled by the **switch** statement, and it is possible to write programs that look like this

```
switch (X)
  case "a string"
    do_something
  ...
endswitch
```

11.2.1 Notes for the C programmer

The **switch** statement is also available in the widely used C programming language. There are, however, some differences between the statement in Octave and C

- Cases are exclusive, so they don’t ‘fall through’ as do the cases in the **switch** statement of the C language.
- The *command_list* elements are not optional. Making the list optional would have meant requiring a separator between the label and the command list. Otherwise, things like

```
switch (foo)
  case (1) -2
  ...
```

would produce surprising results, as would

```
switch (foo)
  case (1)
  case (2)
    doit ();
  ...
```


particularly for C programmers. If `doit()` should be executed if *foo* is either 1 or 2, the above code should be written with a cell array like this

```
switch (foo)
  case { 1, 2 }
    doit ();
  ...
```

11.3 The while Statement

In programming, a *loop* means a part of a program that is (or at least can be) executed two or more times in succession.

The **while** statement is the simplest looping statement in Octave. It repeatedly executes a statement as long as a condition is true. As with the condition in an **if** statement, the condition in a **while** statement is considered true if its value is non-zero, and false if its value is zero. If the value of the conditional expression in a **while** statement is a vector or a matrix, it is considered true only if it is non-empty and *all* of the elements are non-zero.

Octave's **while** statement looks like this:

```
while (condition)
  body
endwhile
```

Here *body* is a statement or list of statements that we call the *body* of the loop, and *condition* is an expression that controls how long the loop keeps running.

The first thing the **while** statement does is test *condition*. If *condition* is true, it executes the statement *body*. After *body* has been executed, *condition* is tested again, and if it is still true, *body* is executed again. This process repeats until *condition* is no longer true. If *condition* is initially false, the body of the loop is never executed.

This example creates a variable **fib** that contains the first ten elements of the Fibonacci sequence.

```
fib = ones (1, 10);
i = 3;
while (i <= 10)
  fib (i) = fib (i-1) + fib (i-2);
  i++;
endwhile
```

Here the body of the loop contains two statements.

The loop works like this: first, the value of *i* is set to 3. Then, the **while** tests whether *i* is less than or equal to 10. This is the case when *i* equals 3, so the value of the *i*-th element of **fib** is set to the sum of the previous two values in the sequence. Then the **i++** increments the value of *i* and the loop repeats. The loop terminates when *i* reaches 11.

A newline is not required between the condition and the body; but using one makes the program clearer unless the body is very simple.

11.4 The do-until Statement

The **do-until** statement is similar to the **while** statement, except that it repeatedly executes a statement until a condition becomes true, and the test of the condition is at the end of the loop, so the body of the loop is always executed at least once. As with the condition in an **if** statement, the condition in a **do-until** statement is considered true if its value is non-zero, and false if its value is zero. If the value of the conditional expression in a **do-until** statement is a vector or a matrix, it is considered true only if it is non-empty and *all* of the elements are non-zero.

Octave's **do-until** statement looks like this:

```
do
  body
until (condition)
```

Here *body* is a statement or list of statements that we call the *body* of the loop, and *condition* is an expression that controls how long the loop keeps running.

This example creates a variable **fib** that contains the first ten elements of the Fibonacci sequence.

```
fib = ones (1, 10);
i = 2;
do
  i++;
  fib (i) = fib (i-1) + fib (i-2);
until (i == 10)
```

A newline is not required between the **do** keyword and the body; but using one makes the program clearer unless the body is very simple.

11.5 The for Statement

The **for** statement makes it more convenient to count iterations of a loop. The general form of the **for** statement looks like this:

```
for var = expression
  body
endfor
```

where *body* stands for any statement or list of statements, *expression* is any valid expression, and *var* may take several forms. Usually it is a simple variable name or an indexed variable. If the value of *expression* is a structure, *var* may also be a vector with two elements. See [Section 11.5.1 \[Looping Over Structure Elements\]](#), page 111, below.

The assignment expression in the **for** statement works a bit differently than Octave's normal assignment statement. Instead of assigning the complete result of the expression, it assigns each column of the expression to *var* in turn. If *expression* is a range, a row vector, or a scalar, the value of *var* will be a scalar each time the loop body is executed. If *var* is a column vector or a matrix, *var* will be a column vector each time the loop body is executed.

The following example shows another way to create a vector containing the first ten elements of the Fibonacci sequence, this time using the **for** statement:

```

fib = ones (1, 10);
for i = 3:10
    fib (i) = fib (i-1) + fib (i-2);
endfor

```

This code works by first evaluating the expression `3:10`, to produce a range of values from 3 to 10 inclusive. Then the variable `i` is assigned the first element of the range and the body of the loop is executed once. When the end of the loop body is reached, the next value in the range is assigned to the variable `i`, and the loop body is executed again. This process continues until there are no more elements to assign.

Within Octave it is also possible to iterate over matrices or cell arrays using the `for` statement. For example consider

```

disp("Loop over a matrix")
for i = [1,3;2,4]
    i
endfor
disp("Loop over a cell array")
for i = {1,"two";"three",4}
    i
endfor

```

In this case the variable `i` takes on the value of the columns of the matrix or cell matrix. So the first loop iterates twice, producing two column vectors `[1;2]`, followed by `[3;4]`, and likewise for the loop over the cell array. This can be extended to loops over multidimensional arrays. For example

```

a = [1,3;2,4]; b = cat(3, a, 2*a);
for i = c
    i
endfor

```

In the above case, the multidimensional matrix `c` is reshaped to a two dimensional matrix as `reshape(c, rows(c), prod(size(c)(2:end)))` and then the same behavior as a loop over a two dimensional matrix is produced.

Although it is possible to rewrite all `for` loops as `while` loops, the Octave language has both statements because often a `for` loop is both less work to type and more natural to think of. Counting the number of iterations is very common in loops and it can be easier to think of this counting as part of looping rather than as something to do inside the loop.

11.5.1 Looping Over Structure Elements

A special form of the `for` statement allows you to loop over all the elements of a structure:

```

for [ val, key ] = expression
    body
endfor

```

In this form of the `for` statement, the value of *expression* must be a structure. If it is, *key* and *val* are set to the name of the element and the corresponding value in turn, until there are no more elements. For example,

```

x.a = 1
x.b = [1, 2; 3, 4]
x.c = "string"
for [val, key] = x
    key
    val
endfor

    ↪ key = a
    ↪ val = 1
    ↪ key = b
    ↪ val =
    ↪
    ↪   1  2
    ↪   3  4
    ↪
    ↪ key = c
    ↪ val = string

```

The elements are not accessed in any particular order. If you need to cycle through the list in a particular way, you will have to use the function `fieldnames` and sort the list yourself.

The key variable may also be omitted. If it is, the brackets are also optional. This is useful for cycling through the values of all the structure elements when the names of the elements do not need to be known.

11.6 The break Statement

The `break` statement jumps out of the innermost `for` or `while` loop that encloses it. The `break` statement may only be used within the body of a loop. The following example finds the smallest divisor of a given integer, and also identifies prime numbers:

```

num = 103;
div = 2;
while (div*div <= num)
    if (rem (num, div) == 0)
        break;
    endif
    div++;
endwhile
if (rem (num, div) == 0)
    printf ("Smallest divisor of %d is %d\n", num, div)
else
    printf ("%d is prime\n", num);
endif

```

When the remainder is zero in the first `while` statement, Octave immediately *breaks out* of the loop. This means that Octave proceeds immediately to the statement following

the loop and continues processing. (This is very different from the `exit` statement which stops the entire Octave program.)

Here is another program equivalent to the previous one. It illustrates how the *condition* of a `while` statement could just as well be replaced with a `break` inside an `if`:

```
num = 103;
div = 2;
while (1)
  if (rem (num, div) == 0)
    printf ("Smallest divisor of %d is %d\n", num, div);
    break;
  endif
  div++;
  if (div*div > num)
    printf ("%d is prime\n", num);
    break;
  endif
endwhile
```

11.7 The continue Statement

The `continue` statement, like `break`, is used only inside `for` or `while` loops. It skips over the rest of the loop body, causing the next cycle around the loop to begin immediately. Contrast this with `break`, which jumps out of the loop altogether. Here is an example:

```
# print elements of a vector of random
# integers that are even.

# first, create a row vector of 10 random
# integers with values between 0 and 100:

vec = round (rand (1, 10) * 100);

# print what we're interested in:

for x = vec
  if (rem (x, 2) != 0)
    continue;
  endif
  printf ("%d\n", x);
endfor
```

If one of the elements of `vec` is an odd number, this example skips the print statement for that element, and continues back to the first statement in the loop.

This is not a practical example of the `continue` statement, but it should give you a clear understanding of how it works. Normally, one would probably write the loop like this:

```

for x = vec
  if (rem (x, 2) == 0)
    printf ("%d\n", x);
  endif
endfor

```

11.8 The `unwind_protect` Statement

Octave supports a limited form of exception handling modelled after the `unwind-protect` form of Lisp.

The general form of an `unwind_protect` block looks like this:

```

unwind_protect
  body
unwind_protect_cleanup
  cleanup
end_unwind_protect

```

where *body* and *cleanup* are both optional and may contain any Octave expressions or commands. The statements in *cleanup* are guaranteed to be executed regardless of how control exits *body*.

This is useful to protect temporary changes to global variables from possible errors. For example, the following code will always restore the original value of the global variable `frobnosticate` even if an error occurs while performing the indexing operation.

```

save_frobnosticate = frobnosticate;
unwind_protect
  frobnosticate = true;
  ...
unwind_protect_cleanup
  frobnosticate = save_frobnosticate;
end_unwind_protect

```

Without `unwind_protect`, the value of *frobnosticate* would not be restored if an error occurs while performing the indexing operation because evaluation would stop at the point of the error and the statement to restore the value would not be executed.

11.9 The `try` Statement

In addition to `unwind_protect`, Octave supports another limited form of exception handling.

The general form of a `try` block looks like this:

```

try
  body
catch
  cleanup
end_try_catch

```

where *body* and *cleanup* are both optional and may contain any Octave expressions or commands. The statements in *cleanup* are only executed if an error occurs in *body*.

No warnings or error messages are printed while *body* is executing. If an error does occur during the execution of *body*, *cleanup* can use the function `lasterr` to access the

text of the message that would have been printed. This is the same as `eval (try, catch)` but it is more efficient since the commands do not need to be parsed each time the *try* and *catch* statements are evaluated. See [Chapter 13 \[Errors and Warnings\]](#), page 137, for more information about the `lasterr` function.

11.10 Continuation Lines

In the Octave language, most statements end with a newline character and you must tell Octave to ignore the newline character in order to continue a statement from one line to the next. Lines that end with the characters `...` or `\` are joined with the following line before they are divided into tokens by Octave's parser. For example, the lines

```
x = long_variable_name ...  
  + longer_variable_name \  
  - 42
```

form a single statement. The backslash character on the second line above is interpreted as a continuation character, *not* as a division operator.

For continuation lines that do not occur inside string constants, whitespace and comments may appear between the continuation marker and the newline character. For example, the statement

```
x = long_variable_name ...      # comment one  
  + longer_variable_name \  
  - 42                          # last comment
```

is equivalent to the one shown above. Inside string constants, the continuation marker must appear at the end of the line just before the newline character.

Input that occurs inside parentheses can be continued to the next line without having to use a continuation marker. For example, it is possible to write statements like

```
if (fine_dining_destination == on_a_boat  
    || fine_dining_destination == on_a_train)  
    seuss (i, will, not, eat, them, sam, i, am, i,  
          will, not, eat, green, eggs, and, ham);  
endif
```

without having to add to the clutter with continuation markers.

12 Functions and Script Files

Complicated Octave programs can often be simplified by defining functions. Functions can be defined directly on the command line during interactive Octave sessions, or in external files, and can be called just like built-in functions.

12.1 Defining Functions

In its simplest form, the definition of a function named *name* looks like this:

```
function name
    body
endfunction
```

A valid function name is like a valid variable name: a sequence of letters, digits and under-scores, not starting with a digit. Functions share the same pool of names as variables.

The function *body* consists of Octave statements. It is the most important part of the definition, because it says what the function should actually *do*.

For example, here is a function that, when executed, will ring the bell on your terminal (assuming that it is possible to do so):

```
function wakeup
    printf ("\a");
endfunction
```

The `printf` statement (see [Chapter 15 \[Input and Output\]](#), page 151) simply tells Octave to print the string `"\a"`. The special character ‘`\a`’ stands for the alert character (ASCII 7). See [Chapter 5 \[Strings\]](#), page 47.

Once this function is defined, you can ask Octave to evaluate it by typing the name of the function.

Normally, you will want to pass some information to the functions you define. The syntax for passing parameters to a function in Octave is

```
function name (arg-list)
    body
endfunction
```

where *arg-list* is a comma-separated list of the function’s arguments. When the function is called, the argument names are used to hold the argument values given in the call. The list of arguments may be empty, in which case this form is equivalent to the one shown above.

To print a message along with ringing the bell, you might modify the `wakeup` to look like this:

```
function wakeup (message)
    printf ("\a%s\n", message);
endfunction
```

Calling this function using a statement like this

```
wakeup ("Rise and shine!");
```

will cause Octave to ring your terminal’s bell and print the message ‘`Rise and shine!`’, followed by a newline character (the ‘`\n`’ in the first argument to the `printf` statement).

In most cases, you will also want to get some information back from the functions you define. Here is the syntax for writing a function that returns a single value:

```
function ret-var = name (arg-list)
  body
endfunction
```

The symbol *ret-var* is the name of the variable that will hold the value to be returned by the function. This variable must be defined before the end of the function body in order for the function to return a value.

Variables used in the body of a function are local to the function. Variables named in *arg-list* and *ret-var* are also local to the function. See [Section 8.1 \[Global Variables\]](#), [page 77](#), for information about how to access global variables inside a function.

For example, here is a function that computes the average of the elements of a vector:

```
function retval = avg (v)
  retval = sum (v) / length (v);
endfunction
```

If we had written `avg` like this instead,

```
function retval = avg (v)
  if (isvector (v))
    retval = sum (v) / length (v);
  endif
endfunction
```

and then called the function with a matrix instead of a vector as the argument, Octave would have printed an error message like this:

```
error: 'retval' undefined near line 1 column 10
error: evaluating index expression near line 7, column 1
```

because the body of the `if` statement was never executed, and `retval` was never defined. To prevent obscure errors like this, it is a good idea to always make sure that the return variables will always have values, and to produce meaningful error messages when problems are encountered. For example, `avg` could have been written like this:

```
function retval = avg (v)
  retval = 0;
  if (isvector (v))
    retval = sum (v) / length (v);
  else
    error ("avg: expecting vector argument");
  endif
endfunction
```

There is still one additional problem with this function. What if it is called without an argument? Without additional error checking, Octave will probably print an error message that won't really help you track down the source of the error. To allow you to catch errors like this, Octave provides each function with an automatic variable called `nargin`. Each time a function is called, `nargin` is automatically initialized to the number of arguments that have actually been passed to the function. For example, we might rewrite the `avg` function like this:

```

function retval = avg (v)
    retval = 0;
    if (nargin != 1)
        usage ("avg (vector)");
    endif
    if (isvector (v))
        retval = sum (v) / length (v);
    else
        error ("avg: expecting vector argument");
    endif
endfunction

```

Although Octave does not automatically report an error if you call a function with more arguments than expected, doing so probably indicates that something is wrong. Octave also does not automatically report an error if a function is called with too few arguments, but any attempt to use a variable that has not been given a value will result in an error. To avoid such problems and to provide useful messages, we check for both possibilities and issue our own error message.

nargin () [Built-in Function]
nargin (fcn_name) [Built-in Function]

Within a function, return the number of arguments passed to the function. At the top level, return the number of command line arguments passed to Octave. If called with the optional argument *fcn_name*, return the maximum number of arguments the named function can accept, or -1 if the function accepts a variable number of arguments.

See also: nargout, varargin, varargout.

inputname (n) [Function File]
 Return the text defining *n*-th input to the function.

val = silent_functions () [Built-in Function]
old_val = silent_functions (new_val) [Built-in Function]

Query or set the internal variable that controls whether internal output from a function is suppressed. If this option is disabled, Octave will display the results produced by evaluating expressions within a function body that are not terminated with a semicolon.

12.2 Multiple Return Values

Unlike many other computer languages, Octave allows you to define functions that return more than one value. The syntax for defining functions that return multiple values is

```

function [ret-list] = name (arg-list)
    body
endfunction

```

where *name*, *arg-list*, and *body* have the same meaning as before, and *ret-list* is a comma-separated list of variable names that will hold the values returned from the function. The

list of return values must have at least one element. If *ret-list* has only one element, this form of the **function** statement is equivalent to the form described in the previous section.

Here is an example of a function that returns two values, the maximum element of a vector and the index of its first occurrence in the vector.

```
function [max, idx] = vmax (v)
  idx = 1;
  max = v (idx);
  for i = 2:length (v)
    if (v (i) > max)
      max = v (i);
      idx = i;
    endif
  endfor
endfunction
```

In this particular case, the two values could have been returned as elements of a single array, but that is not always possible or convenient. The values to be returned may not have compatible dimensions, and it is often desirable to give the individual return values distinct names.

In addition to setting **nargin** each time a function is called, Octave also automatically initializes **nargout** to the number of values that are expected to be returned. This allows you to write functions that behave differently depending on the number of values that the user of the function has requested. The implicit assignment to the built-in variable **ans** does not figure in the count of output arguments, so the value of **nargout** may be zero.

The **svd** and **lu** functions are examples of built-in functions that behave differently depending on the value of **nargout**.

It is possible to write functions that only set some return values. For example, calling the function

```
function [x, y, z] = f ()
  x = 1;
  z = 2;
endfunction
```

as

```
[a, b, c] = f ()
```

produces:

```
a = 1
```

```
b = [] (0x0)
```

```
c = 2
```

along with a warning.

nargout () [Built-in Function]

nargout (*fcn_name*) [Built-in Function]

Within a function, return the number of values the caller expects to receive. If called with the optional argument *fcn_name*, return the maximum number of values the

named function can produce, or -1 if the function can produce a variable number of values.

For example,

```
f ()
```

will cause `nargout` to return 0 inside the function `f` and

```
[s, t] = f ()
```

will cause `nargout` to return 2 inside the function `f`.

At the top level, `nargout` is undefined.

See also: `nargin`, `varargin`, `varargout`.

`nargchk (nargin_min, nargin_max, n)` [Function File]

If `n` is in the range `nargin_min` through `nargin_max` inclusive, return the empty matrix. Otherwise, return a message indicating whether `n` is too large or too small.

This is useful for checking to see that the number of arguments supplied to a function is within an acceptable range.

12.3 Variable-length Argument Lists

Sometimes the number of input arguments is not known when the function is defined. As an example think of a function that returns the smallest of all its input arguments. For example,

```
a = smallest (1, 2, 3);
b = smallest (1, 2, 3, 4);
```

In this example both `a` and `b` would be 1. One way to write the `smallest` function is

```
function val = smallest (arg1, arg2, arg3, arg4, arg5)
    body
endfunction
```

and then use the value of `nargin` to determine which of the input arguments should be considered. The problem with this approach is that it can only handle a limited number of input arguments.

Octave supports the `varargin` keyword for handling a variable number of input arguments. Using `varargin` the function looks like this

```
function val = smallest (varargin)
    body
endfunction
```

In the function body the input arguments can be accessed through the variable `varargin`. This variable is a cell array containing all the input arguments. See [Chapter 7 \[Cell Arrays\]](#), [page 69](#), for details on working with cell arrays. The `smallest` function can now be defined like this

```
function val = smallest (varargin)
    val = min ([varargin{:}]);
endfunction
```

This implementation handles any number of input arguments, but it's also a very simple solution to the problem.

A slightly more complex example of `varargin` is a function `print_arguments` that prints all input arguments. Such a function can be defined like this

```
function print_arguments (varargin)
  for i = 1:length (varargin)
    printf ("Input argument %d: ", i);
    disp (varargin{i});
  endfor
endfunction
```

This function produces output like this

```
print_arguments (1, "two", 3);
  + Input argument 1:  1
  + Input argument 2: two
  + Input argument 3:  3
```

`[reg, prop] = parseparams (params)` [Function File]

Return in `reg` the cell elements of `param` up to the first string element and in `prop` all remaining elements beginning with the first string element. For example

```
[reg, prop] = parseparams ({1, 2, "linewidth", 10})
reg =
{
  [1,1] = 1
  [1,2] = 2
}
prop =
{
  [1,1] = linewidth
  [1,2] = 10
}
```

The `parseparams` function may be used to separate 'regular' arguments and additional arguments given as property/value pairs of the `varargin` cell array.

See also: `varargin`.

12.4 Variable-length Return Lists

It is possible to return a variable number of output arguments from a function using a syntax that's similar to the one used with the `varargin` keyword. To let a function return a variable number of output arguments the `varargout` keyword is used. As with `varargin` `varargout` is a cell array that will contain the requested output arguments.

As an example the following function sets the first output argument to 1, the second to 2, and so on.

```
function varargout = one_to_n ()
  for i = 1:nargout
    varargout{i} = i;
  endfor
endfunction
```

When called this function returns values like this

```
[a, b, c] = one_to_n ()
⇒ a = 1
⇒ b = 2
⇒ c = 3
```

```
[r1, r2, ..., rn] = deal (a) [Function File]
[r1, r2, ..., rn] = deal (a1, a2, ..., an) [Function File]
```

Copy the input parameters into the corresponding output parameters. If only one input parameter is supplied, its value is copied to each of the outputs.

For example,

```
[a, b, c] = deal (x, y, z);
```

is equivalent to

```
a = x;
b = y;
c = z;
```

and

```
[a, b, c] = deal (x);
```

is equivalent to

```
a = b = c = x;
```

12.5 Returning From a Function

The body of a user-defined function can contain a **return** statement. This statement returns control to the rest of the Octave program. It looks like this:

```
return
```

Unlike the **return** statement in C, Octave's **return** statement cannot be used to return a value from a function. Instead, you must assign values to the list of return variables that are part of the **function** statement. The **return** statement simply makes it easier to exit a function from a deeply nested loop or conditional statement.

Here is an example of a function that checks to see if any elements of a vector are nonzero.

```
function retval = any_nonzero (v)
    retval = 0;
    for i = 1:length (v)
        if (v (i) != 0)
            retval = 1;
            return;
        endif
    endfor
    printf ("no nonzero elements found\n");
endfunction
```

Note that this function could not have been written using the **break** statement to exit the loop once a nonzero value is found without adding extra logic to avoid printing the message if the vector does contain a nonzero element.

return [Keyword]

When Octave encounters the keyword **return** inside a function or script, it returns control to the caller immediately. At the top level, the return statement is ignored. A **return** statement is assumed at the end of every function definition.

12.6 Default Arguments

Since Octave supports variable number of input arguments, it is very useful to assign default values to some input arguments. When an input argument is declared in the argument list it is possible to assign a default value to the argument like this

```
function name (arg1 = val1, ...)
    body
endfunction
```

If no value is assigned to *arg1* by the user, it will have the value *val1*.

As an example, the following function implements a variant of the classic “Hello, World” program.

```
function hello (who = "World")
    printf ("Hello, %s!\n", who);
endfunction
```

When called without an input argument the function prints the following

```
hello ();
    ↪ Hello, World!
```

and when it’s called with an input argument it prints the following

```
hello ("Beautiful World of Free Software");
    ↪ Hello, Beautiful World of Free Software!
```

Sometimes it is useful to explicitly tell Octave to use the default value of an input argument. This can be done writing a ‘:’ as the value of the input argument when calling the function.

```
hello (:);
    ↪ Hello, World!
```

12.7 Function Files

Except for simple one-shot programs, it is not practical to have to define all the functions you need each time you need them. Instead, you will normally want to save them in a file so that you can easily edit them, and save them for use at a later time.

Octave does not require you to load function definitions from files before using them. You simply need to put the function definitions in a place where Octave can find them.

When Octave encounters an identifier that is undefined, it first looks for variables or functions that are already compiled and currently listed in its symbol table. If it fails to find a definition there, it searches a list of directories (the *path*) for files ending in ‘.m’ that have the same base name as the undefined identifier.¹ Once Octave finds a file with a name that matches, the contents of the file are read. If it defines a *single* function, it is compiled

¹ The ‘.m’ suffix was chosen for compatibility with MATLAB.

and executed. See [Section 12.8 \[Script Files\], page 129](#), for more information about how you can define more than one function in a single file.

When Octave defines a function from a function file, it saves the full name of the file it read and the time stamp on the file. If the time stamp on the file changes, Octave may reload the file. When Octave is running interactively, time stamp checking normally happens at most once each time Octave prints the prompt. Searching for new function definitions also occurs if the current working directory changes.

Checking the time stamp allows you to edit the definition of a function while Octave is running, and automatically use the new function definition without having to restart your Octave session.

To avoid degrading performance unnecessarily by checking the time stamps on functions that are not likely to change, Octave assumes that function files in the directory tree ‘`octave-home/share/octave/version/m`’ will not change, so it doesn’t have to check their time stamps every time the functions defined in those files are used. This is normally a very good assumption and provides a significant improvement in performance for the function files that are distributed with Octave.

If you know that your own function files will not change while you are running Octave, you can improve performance by calling `ignore_function_time_stamp ("all")`, so that Octave will ignore the time stamps for all function files. Passing `"system"` to this function resets the default behavior.

```
mfilename () [Built-in Function]
mfilename ("fullpath") [Built-in Function]
mfilename ("fullpathext") [Built-in Function]
```

Return the name of the currently executing file. At the top-level, return the empty string. Given the argument `"fullpath"`, include the directory part of the file name, but not the extension. Given the argument `"fullpathext"`, include the directory part of the file name and the extension.

```
val = ignore_function_time_stamp () [Built-in Function]
old_val = ignore_function_time_stamp (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave checks the time stamp on files each time it looks up functions defined in function files. If the internal variable is set to `"system"`, Octave will not automatically recompile function files in subdirectories of ‘`octave-home/lib/version`’ if they have changed since they were last compiled, but will recompile other function files in the search path if they change. If set to `"all"`, Octave will not recompile any function files unless their definitions are removed with `clear`. If set to `"none"`, Octave will always check time stamps on files to determine whether functions defined in function files need to be recompiled.

12.7.1 Manipulating the load path

When a function is called Octave searches a list of directories for a file that contains the function declaration. This list of directories is known as the load path. By default the load path contains a list of directories distributed with Octave plus the current working directory. To see your current load path call the `path` function without any input or output arguments.

It is possible to add or remove directories to or from the load path using the `addpath` and `rmpath`. As an example, the following code adds ‘~/Octave’ to the load path.

```
addpath("~/Octave")
```

After this the directory ‘~/Octave’ will be searched for functions.

`addpath (dir1, ...)` [Built-in Function]

`addpath (dir1, ..., option)` [Built-in Function]

Add *dir1, ...* to the current function search path. If *option* is “-begin” or 0 (the default), prepend the directory name to the current path. If *option* is “-end” or 1, append the directory name to the current path. Directories added to the path must exist.

See also: path, rmpath, genpath, pathdef, savepath, pathsep.

`genpath (dir)` [Built-in Function]

Return a path constructed from *dir* and all its subdirectories.

`rmpath (dir1, ...)` [Built-in Function]

Remove *dir1, ...* from the current function search path.

See also: path, addpath, genpath, pathdef, savepath, pathsep.

`savepath (file)` [Function File]

Save the current function search path to *file*. If *file* is omitted, ‘~/octaverc’ is used. If successful, `savepath` returns 0.

See also: path, addpath, rmpath, genpath, pathdef, pathsep.

`path (...)` [Built-in Function]

Modify or display Octave’s load path.

If *nargin* and *nargout* are zero, display the elements of Octave’s load path in an easy to read format.

If *nargin* is zero and *nargout* is greater than zero, return the current load path.

If *nargin* is greater than zero, concatenate the arguments, separating them with `pathsep()`. Set the internal search path to the result and return it.

No checks are made for duplicate elements.

See also: addpath, rmpath, genpath, pathdef, savepath, pathsep.

`val = pathdef ()` [Built-in Function]

Return the default list of directories in which to search for function files.

See also: path, addpath, rmpath, genpath, savepath, pathsep.

`pathsep ()` [Built-in Function]

Return the system-dependent character used to separate directories in a path.

See also: filesep, dir, ls.

rehash () [Built-in Function]

Reinitialize Octave's load path directory cache.

file_in_loadpath (file) [Built-in Function]

file_in_loadpath (file, "all") [Built-in Function]

Return the absolute name of *file* if it can be found in the list of directories specified by *path*. If no file is found, return an empty matrix.

If the first argument is a cell array of strings, search each directory of the loadpath for element of the cell array and return the first that matches.

If the second optional argument "all" is supplied, return a cell array containing the list of all files that have the same name in the path. If no files are found, return an empty cell array.

See also: file_in_path, path.

12.7.2 Subfunctions

A function file may contain secondary functions called *subfunctions*. These secondary functions are only visible to the other functions in the same function file. For example, a file 'f.m' containing

```
function f ()
    printf ("in f, calling g\n");
    g ()
endfunction
function g ()
    printf ("in g, calling h\n");
    h ()
endfunction
function h ()
    printf ("in h\n")
endfunction
```

defines a main function **f** and two subfunctions. The subfunctions **g** and **h** may only be called from the main function **f** or from the other subfunctions, but not from outside the file 'f.m'.

12.7.3 Overloading and Autoloading

The **dispatch** function can be used to alias one function name to another. It can be used to alias all calls to a particular function name to another function, or the alias can be limited to only a particular variable type. Consider the example

```
function y = spsin (x)
    printf ("Calling spsin\n");
    fflush(stdout);
    y = spfun ("sin", x);
endfunction

dispatch ("sin", "spsin", "sparse matrix");
y0 = sin(eye(3));
y1 = sin(speye(3));
```

Which aliases the `spsin` to `sin`, but only for real sparse matrices. Note that the builtin `sin` already correctly treats sparse matrices and so this example is only illustrative.

dispatch (*f*, *r*, *type*) [Loadable Function]

Replace the function *f* with a dispatch so that function *r* is called when *f* is called with the first argument of the named *type*. If the *type* is *any* then call *r* if no other type matches. The original function *f* is accessible using `builtin (f, ...)`.

If *r* is omitted, clear dispatch function associated with *type*.

If both *r* and *type* are omitted, list dispatch functions for *f*.

See also: `builtin`.

[...] **builtin** (*f*, ...) [Loadable Function]

Call the base function *f* even if *f* is overloaded to some other function for the given type signature.

See also: `dispatch`.

A single dynamically linked file might define several functions. However, as Octave searches for functions based on the functions filename, Octave needs a manner in which to find each of the functions in the dynamically linked file. On operating systems that support symbolic links, it is possible to create a symbolic link to the original file for each of the functions which it contains.

However, there is at least one well known operating system that doesn't support symbolic links. Making copies of the original file for each of the functions is also possible, but is undesirable as it multiplies the amount of disk space used by Octave. Instead Octave supplies the `autoload` function, that permits the user to define in which file a certain function will be found.

autoload (*function*, *file*) [Built-in Function]

Define *function* to autoload from *file*.

The second argument, *file*, should be an absolute file name and should not depend on the Octave load path.

Normally, calls to `autoload` appear in `PKG_ADD` script files that are evaluated when a directory is added to the Octave's load path. To avoid having to hardcode directory names in *file*, it is customary to use

```
autoload ("foo",
          fullfile (fileparts (mfilename ("fullpath")),
                  "bar.oct"));
```

Uses like

```
autoload ("foo", file_in_loadpath ("bar.oct"))
```

are strongly discouraged.

With no arguments, return a structure containing the current autoload map.

See also: `PKG_ADD`.

12.7.4 Function Locking

It is sometime desirable to lock a function into memory with the `mlock` function. This is typically used for dynamically linked functions in Oct-files or mex-files, that contain some initialization, and it is desirable that a `clear` does not remove this initialization.

This might equally be used to prevent changes to a function from having effect in Octave, though a similar effect can be had with the `ignore_function_time_stamp` function.

mlock (*name*) [Built-in Function]

Lock the named function into memory. If no function is named then lock in the current function.

See also: `munlock`, `mislocked`, `persistent`.

munlock (*fcn*) [Built-in Function]

Unlock the named function. If no function is named then unlock the current function.

See also: `mlock`, `mislocked`, `persistent`.

mislocked (*fcn*) [Built-in Function]

Return true if the named function is locked. If no function is named then return true if the current function is locked.

See also: `mlock`, `munlock`, `persistent`.

12.8 Script Files

A script file is a file containing (almost) any sequence of Octave commands. It is read and evaluated just as if you had typed each command at the Octave prompt, and provides a convenient way to perform a sequence of commands that do not logically belong inside a function.

Unlike a function file, a script file must *not* begin with the keyword `function`. If it does, Octave will assume that it is a function file, and that it defines a single function that should be evaluated as soon as it is defined.

A script file also differs from a function file in that the variables named in a script file are not local variables, but are in the same scope as the other variables that are visible on the command line.

Even though a script file may not begin with the `function` keyword, it is possible to define more than one function in a single script file and load (but not execute) all of them at once. To do this, the first token in the file (ignoring comments and other white space) must be something other than `function`. If you have no other statements to evaluate, you can use a statement that has no effect, like this:

```
# Prevent Octave from thinking that this
# is a function file:

1;

# Define function one:

function one ()
    ...
```

To have Octave read and compile these functions into an internal form, you need to make sure that the file is in Octave's load path (accessible through the `path` function), then simply type the base name of the file that contains the commands. (Octave uses the same rules to search for script files as it does to search for function files.)

If the first token in a file (ignoring comments) is **function**, Octave will compile the function and try to execute it, printing a message warning about any non-whitespace characters that appear after the function definition.

Note that Octave does not try to look up the definition of any identifier until it needs to evaluate it. This means that Octave will compile the following statements if they appear in a script file, or are typed at the command line,

```
# not a function file:
1;
function foo ()
    do_something ();
endfunction
function do_something ()
    do_something_else ();
endfunction
```

even though the function `do_something` is not defined before it is referenced in the function `foo`. This is not an error because Octave does not need to resolve all symbols that are referenced by a function until the function is actually evaluated.

Since Octave doesn't look for definitions until they are needed, the following code will always print `'bar = 3'` whether it is typed directly on the command line, read from a script file, or is part of a function body, even if there is a function or script file called `'bar.m'` in Octave's path.

```
eval ("bar = 3");
bar
```

Code like this appearing within a function body could fool Octave if definitions were resolved as the function was being compiled. It would be virtually impossible to make Octave clever enough to evaluate this code in a consistent fashion. The parser would have to be able to perform the call to `eval` at compile time, and that would be impossible unless all the references in the string to be evaluated could also be resolved, and requiring that would be too restrictive (the string might come from user input, or depend on things that are not known until the function is evaluated).

Although Octave normally executes commands from script files that have the name `'file.m'`, you can use the function `source` to execute commands from any file.

source (*file*) [Built-in Function]
 Parse and execute the contents of *file*. This is equivalent to executing commands from a script file, but without requiring the file to be named '*file.m*'.

12.9 Function Handles, Inline Functions, and Anonymous Functions

It can be very convenient store a function in a variable so that it can be passed to a different function. For example, a function that performs numerical minimisation needs access to the function that should be minimised.

12.9.1 Function Handles

A function handle is a pointer to another function and is defined with the syntax

```
@function-name
```

For example

```
f = @sin;
```

Creates a function handle called **f** that refers to the function **sin**.

Function handles are used to call other functions indirectly, or to pass a function as an argument to another function like **quad** or **fsolve**. For example

```
f = @sin;
quad (f, 0, pi)
⇒ 1.8391
```

You may use **feval** to call a function using function handle, or simply write the name of the function handle followed by an argument list. If there are no arguments, you must use an empty argument list '()'. For example

```
f = @sin;
feval (f, pi/4)
⇒ 0.70711
f (pi/4)
⇒ 0.70711
```

functions (*fcn_handle*) [Built-in Function]
 Return a struct containing information about the function handle *fcn_handle*.

func2str (*fcn_handle*) [Built-in Function]
 Return a string containing the name of the function referenced by the function handle *fcn_handle*.

str2func (*fcn_name*) [Built-in Function]
 Return a function handle constructed from the string *fcn_name*.

12.9.2 Anonymous Functions

Anonymous functions are defined using the syntax

`@(argument-list) expression`

Any variables that are not found in the argument list are inherited from the enclosing scope. Anonymous functions are useful for creating simple unnamed functions from expressions or for wrapping calls to other functions to adapt them for use by functions like `quad`. For example,

```
f = @(x) x.^2;
quad (f, 0, 10)
⇒ 333.33
```

creates a simple unnamed function from the expression $x.^2$ and passes it to `quad`,

```
quad (@(x) sin (x), 0, pi)
⇒ 1.8391
```

wraps another function, and

```
a = 1;
b = 2;
quad (@(x) betainc (x, a, b), 0, 0.4)
⇒ 1.3867
```

adapts a function with several parameters to the form required by `quad`. In this example, the values of a and b that are passed to `betainc` are inherited from the current environment.

12.9.3 Inline Functions

An inline function is created from a string containing the function body using the `inline` function. The following code defines the function $f(x) = x^2 + 2$.

```
f = inline("x^2 + 2");
```

After this it is possible to evaluate f at any x by writing `f(x)`.

<code>inline (str)</code>	[Built-in Function]
<code>inline (str, arg1, ...)</code>	[Built-in Function]
<code>inline (str, n)</code>	[Built-in Function]

Create an inline function from the character string *str*. If called with a single argument, the arguments of the generated function are extracted from the function itself. The generated function arguments will then be in alphabetical order. It should be noted that *i*, and *j* are ignored as arguments due to the ambiguity between their use as a variable or their use as an inbuilt constant. All arguments followed by a parentheses are considered to be functions.

If the second and subsequent arguments are character strings, they are the names of the arguments of the function.

If the second argument is an integer n , the arguments are " x ", " $P1$ ", ..., " PN ".

See also: `argnames`, `formula`, `vectorize`.

<code>argnames (fun)</code>	[Built-in Function]
-----------------------------	---------------------

Return a cell array of character strings containing the names of the arguments of the inline function *fun*.

See also: `inline`, `formula`, `vectorize`.

formula (*fun*) [Built-in Function]

Return a character string representing the inline function *fun*. Note that **char** (*fun*) is equivalent to **formula** (*fun*).

See also: argnames, inline, vectorize.

vectorize (*fun*) [Built-in Function]

Create a vectorized version of the inline function *fun* by replacing all occurrences of ***, */*, etc., with *.**, *./*, etc.

12.10 Commands

Commands are a special class of functions that only accept string input arguments. A command can be called as an ordinary function, but it can also be called without the parentheses like the following example shows

```
my_command hello world
```

which is the same as

```
my_command("hello", "world")
```

The general form of a command call is

```
name arg1 arg2 ...
```

which translates directly to

```
name ("arg1", "arg2", ...)
```

A function can be used as a command if it accept string input arguments. To do this, the function must be marked as a command, which can be done with the **mark_as_command** command like this

```
mark_as_command name
```

where **name** is the function to be marked as a command.

One difficulty of commands occurs when one of the string input arguments are stored in a variable. Since Octave can't tell the difference between a variable name, and an ordinary string, it is not possible to pass a variable as input to a command. In such a situation a command must be called as a function.

mark_as_command (*name*) [Built-in Function]

Enter *name* into the list of commands.

See also: unmark_command, iscommand.

unmark_command (*name*) [Built-in Function]

Remove *name* from the list of commands.

See also: mark_as_command, iscommand.

iscommand (*name*) [Built-in Function]

Return true if *name* is a command style function. If *name* is omitted, return a list of identifiers which are marked as commands with **mark_as_command**.

See also: mark_as_command, unmark_command.

mark_as_rawcommand (*name*) [Built-in Function]

Enter *name* into the list of raw input commands and to the list of command style functions. Raw input commands are like normal command style functions, but they receive their input unprocessed (ie. strings still contain the quotes and escapes they had when input). However, comments and continuations are handled as usual, you cannot pass a token starting with a comment character ('#' or '%') to your function, and the last token cannot be a continuation token ('\ or '...').

See also: unmark_rawcommand, israwcommand, iscommand, mark_as_command.

unmark_rawcommand (*name*) [Built-in Function]

Remove *name* from the list of raw input commands. Note that this does not remove *name* from the list of command style functions.

See also: mark_as_rawcommand, israwcommand, iscommand, unmark_command.

israwcommand (*name*) [Built-in Function]

Return true if *name* is a raw input command function. If *name* is omitted, return a list of identifiers which are marked as raw input commands with mark_as_rawcommand.

See also: mark_as_rawcommand, unmark_rawcommand.

12.11 Organization of Functions Distributed with Octave

Many of Octave's standard functions are distributed as function files. They are loosely organized by topic, in subdirectories of '*octave-home/lib/octave/version/m*', to make it easier to find them.

The following is a list of all the function file subdirectories, and the types of functions you will find there.

- 'audio' Functions for playing and recording sounds.
- 'control' Functions for design and simulation of automatic control systems.
- 'elfun' Elementary functions.
- 'finance' Functions for computing interest payments, investment values, and rates of return.
- 'general' Miscellaneous matrix manipulations, like `flipud`, `rot90`, and `triu`, as well as other basic functions, like `ismatrix`, `nargchk`, etc.
- 'image' Image processing tools. These functions require the X Window System.
- 'io' Input-output functions.
- 'linear-algebra' Functions for linear algebra.
- 'miscellaneous' Functions that don't really belong anywhere else.
- 'optimization' Minimization of functions.

<code>'path'</code>	Functions to manage the directory path Octave uses to find functions.
<code>'pkg'</code>	Install external packages of functions in Octave.
<code>'plot'</code>	Functions for displaying and printing two- and three-dimensional graphs.
<code>'polynomial'</code>	Functions for manipulating polynomials.
<code>'set'</code>	Functions for creating and manipulating sets of unique values.
<code>'signal'</code>	Functions for signal processing applications.
<code>'sparse'</code>	Functions for handling sparse matrices.
<code>'specfun'</code>	Special functions.
<code>'special-matrix'</code>	Functions that create special matrix forms.
<code>'startup'</code>	Octave's system-wide startup file.
<code>'statistics'</code>	Statistical functions.
<code>'strings'</code>	Miscellaneous string-handling functions.
<code>'testfun'</code>	Perform unit tests on other functions.
<code>'time'</code>	Functions related to time keeping.

13 Errors and Warnings

Octave includes several functions for printing error and warning messages. When you write functions that need to take special action when they encounter abnormal conditions, you should print the error messages using the functions described in this chapter.

Since many of Octave's functions use these functions, it is also useful to understand them, so that errors and warnings can be handled.

13.1 Handling Errors

An error is something that occurs when a program is in a state where it doesn't make sense to continue. An example is when a function is called with too few input arguments. In this situation the function should abort with an error message informing the user of the lacking input arguments.

Since an error can occur during the evaluation of a program, it is very convenient to be able to detect that an error occurred, so that the error can be fixed. This is possible with the `try` statement described in [Section 11.9 \[The try Statement\]](#), page 114.

13.1.1 Raising Errors

The most common use of errors is for checking input arguments to functions. The following example calls the `error` function if the function `f` is called without any input arguments.

```
function f (arg1)
  if (nargin == 0)
    error("not enough input arguments");
  endif
endfunction
```

When the `error` function is called, it prints the given message and returns to the Octave prompt. This means that no code following a call to `error` will be executed.

error (*template*, ...) [Built-in Function]

Format the optional arguments under the control of the template string *template* using the same rules as the `printf` family of functions (see [Section 15.2.4 \[Formatted Output\]](#), page 165) and print the resulting message on the `stderr` stream. The message is prefixed by the character string `'error: '`.

Calling `error` also sets Octave's internal error state such that control will return to the top level without evaluating any more commands. This is useful for aborting from functions or scripts.

If the error message does not end with a new line character, Octave will print a traceback of all the function calls leading to the error. For example, given the following function definitions:

```
function f () g (); end
function g () h (); end
function h () nargin == 1 || error ("nargin != 1"); end
```

calling the function `f` will result in a list of messages that can help you to quickly locate the exact location of the error:

```

f ()
error: nargin != 1
error: evaluating index expression near line 1, column 30
error: evaluating binary operator '||' near line 1, column 27
error: called from 'h'
error: called from 'g'
error: called from 'f'

```

If the error message ends in a new line character, Octave will print the message but will not display any traceback messages as it returns control to the top level. For example, modifying the error message in the previous example to end in a new line causes Octave to only print a single message:

```

function h () nargin == 1 || error ("nargin != 1\n"); end
f ()
error: nargin != 1

```

Since it is common to use errors when there is something wrong with the input to a function, Octave supports functions to simplify such code. When the `print_usage` function is called, it reads the help text of the function calling `print_usage`, and presents a useful error. If the help text is written in Texinfo it is possible to present an error message that only contains the function prototypes as described by the `@deftypefn` parts of the help text. When the help text isn't written in Texinfo, the error message contains the entire help message.

Consider the following function.

```

## -*- texinfo -*-
## @deftypefn {Function File} f (@var{arg1})
## Function help text goes here...
## @end deftypefn
function f (arg1)
  if (nargin == 0)
    print_usage ();
  endif
endfunction

```

When it is called with no input arguments it produces the following error.

```

f ()
+ Invalid call to f.  Correct usage is:
+
+ -- Function File: f (ARG1)
+
+
+ error: evaluating if command near line 6, column 3
+ error: called from 'f' in file '/home/jwe/octave/f.m'

```

`print_usage ()` [Loadable Function]

Print the usage message for the currently executing function. The `print_usage` function is only intended to work inside a user-defined function.

See also: `help`.

`usage (msg)` [Built-in Function]

Print the message *msg*, prefixed by the string ‘**usage:** ’, and set Octave’s internal error state such that control will return to the top level without evaluating any more commands. This is useful for aborting from functions.

After `usage` is evaluated, Octave will print a traceback of all the function calls leading to the usage message.

You should use this function for reporting problems errors that result from an improper call to a function, such as calling a function with an incorrect number of arguments, or with arguments of the wrong type. For example, most functions distributed with Octave begin with code like this

```
if (nargin != 2)
    usage ("foo (a, b)");
endif
```

to check for the proper number of arguments.

`beep ()` [Function File]

Produce a beep from the speaker (or visual bell).

See also: `puts`, `fputs`, `printf`, `fprintf`.

`val = beep_on_error ()` [Built-in Function]

`old_val = beep_on_error (new_val)` [Built-in Function]

Query or set the internal variable that controls whether Octave will try to ring the terminal bell before printing an error message.

13.1.2 Catching Errors

When an error occurs, it can be detected and handled using the `try` statement as described in [Section 11.9 \[The try Statement\]](#), [page 114](#). As an example, the following piece of code counts the number of errors that occurs during a `for` loop.

```
number_of_errors = 0;
for n = 1:100
    try
        ...
    catch
        number_of_errors++;
    end_try_catch
endfor
```

The above example treats all errors the same. In many situations it can however be necessary to discriminate between errors, and take different actions depending on the error. The `lasterror` function returns a structure containing information about the last error that occurred. As an example, the code above could be changed to count the number of errors related to the ‘`*`’ operator.

```

number_of_errors = 0;
for n = 1:100
    try
        ...
    catch
        msg = lasterror.message;
        if (strfind (msg, "operator *"))
            number_of_errors++;
        endif
    end_try_catch
endfor

```

`err = lasterror (err)` [Built-in Function]

`lasterror ('reset')` [Built-in Function]

Returns or sets the last error message. Called without any arguments returns a structure containing the last error message, as well as other information related to this error. The elements of this structure are:

<code>'message'</code>	The text of the last error message								
<code>'identifier'</code>	The message identifier of this error message								
<code>'stack'</code>	A structure containing information on where the message occurred. This might be an empty structure if this in the case where this information can not be obtained. The fields of this structure are: <table> <tbody> <tr> <td><code>'file'</code></td> <td>The name of the file where the error occurred</td> </tr> <tr> <td><code>'name'</code></td> <td>The name of function in which the error occurred</td> </tr> <tr> <td><code>'line'</code></td> <td>The line number at which the error occurred</td> </tr> <tr> <td><code>'column'</code></td> <td>An optional field with the column number at which the error occurred</td> </tr> </tbody> </table>	<code>'file'</code>	The name of the file where the error occurred	<code>'name'</code>	The name of function in which the error occurred	<code>'line'</code>	The line number at which the error occurred	<code>'column'</code>	An optional field with the column number at which the error occurred
<code>'file'</code>	The name of the file where the error occurred								
<code>'name'</code>	The name of function in which the error occurred								
<code>'line'</code>	The line number at which the error occurred								
<code>'column'</code>	An optional field with the column number at which the error occurred								

The `err` structure may also be passed to `lasterror` to set the information about the last error. The only constraint on `err` in that case is that it is a scalar structure. Any fields of `err` that match the above are set to the value passed in `err`, while other fields are set to their default values.

If `lasterror` is called with the argument `'reset'`, all values take their default values.

`[msg, msgid] = lasterr (msg, msgid)` [Built-in Function]

Without any arguments, return the last error message. With one argument, set the last error message to `msg`. With two arguments, also set the last message identifier.

When an error has been handled it is possible to raise it again. This can be useful when an error needs to be detected, but the program should still abort. This is possible using the `rethrow` function. The previous example can now be changed to count the number of errors related to the `'*'` operator, but still abort if another kind of error occurs.

```

number_of_errors = 0;
for n = 1:100

```



```

try
  ...
catch
  msg = lasterror.message;
  if (strfind (msg, "operator *"))
    number_of_errors++;
  else
    rethrow (lasterror);
  endif
end_try_catch
endfor

```

rethrow (err) [Built-in Function]

Reissues a previous error as defined by *err*. *err* is a structure that must contain at least the 'message' and 'identifier' fields. *err* can also contain a field 'stack' that gives information on the assumed location of the error. Typically *err* is returned from *lasterror*.

See also: *lasterror*, *lasterr*, *error*.

err = errno () [Built-in Function]

err = errno (val) [Built-in Function]

err = errno (name) [Built-in Function]

Return the current value of the system-dependent variable *errno*, set its value to *val* and return the previous value, or return the named error code given *name* as a character string, or -1 if *name* is not found.

errno_list () [Built-in Function]

Return a structure containing the system-dependent *errno* values.

13.2 Handling Warnings

Like an error, a warning is issued when something unexpected happens. Unlike an error, a warning doesn't abort the currently running program. A simple example of a warning is when a number is divided by zero. In this case Octave will issue a warning and assign the value *Inf* to the result.

```

a = 1/0
    + warning: division by zero
    => a = Inf

```

13.2.1 Issuing Warnings

It is possible to issue warnings from any code using the **warning** function. In its most simple form, the **warning** function takes a string describing the warning as its input argument. As an example, the following code controls if the variable 'a' is non-negative, and if not issues a warning and sets 'a' to zero.

```

a = -1;
if (a < 0)

```

```

    warning("'a' must be non-negative number. Setting 'a' to zero.");
    a = 0;
endif
    └ 'a' must be non-negative number. Setting 'a' to zero.

```

Since warnings aren't fatal to a running program, it is not possible to catch a warning using the `try` statement or something similar. It is however possible to access the last warning as a string using the `lastwarn` function.

It is also possible to assign an identification string to a warning. If a warning has such an ID the user can enable and disable this warning as will be described in the next section. To assign an ID to a warning, simply call `warning` with two string arguments, where the first is the identification string, and the second is the actual warning.

`warning (template, ...)` [Built-in Function]
`warning (id, template)` [Built-in Function]

Format the optional arguments under the control of the template string *template* using the same rules as the `printf` family of functions (see [Section 15.2.4 \[Formatted Output\]](#), page 165) and print the resulting message on the `stderr` stream. The message is prefixed by the character string `'warning: '`. You should use this function when you want to notify the user of an unusual condition, but only when it makes sense for your program to go on.

The optional message identifier allows users to enable or disable warnings tagged by *id*. The special identifier `"all"` may be used to set the state of all warnings.

`warning ("on", id)` [Built-in Function]
`warning ("off", id)` [Built-in Function]
`warning ("error", id)` [Built-in Function]
`warning ("query", id)` [Built-in Function]

Set or query the state of a particular warning using the identifier *id*. If the identifier is omitted, a value of `"all"` is assumed. If you set the state of a warning to `"error"`, the warning named by *id* is handled as if it were an error instead.

See also: `warning_ids`.

`[msg, msgid] = lastwarn (msg, msgid)` [Built-in Function]

Without any arguments, return the last warning message. With one argument, set the last warning message to *msg*. With two arguments, also set the last message identifier.

13.2.2 Enabling and Disabling Warnings

The `warning` function also allows you to control which warnings are actually printed to the screen. If the `warning` function is called with a string argument that is either `"on"` or `"off"` all warnings will be enabled or disabled.

It is also possible to enable and disable individual warnings through their string identifications. The following code will issue a warning

```

warning ("non-negative-variable",
        "'a' must be non-negative number. Setting 'a' to zero.");

```

while the following won't issue a warning

```
warning ("off", "non-negative-variable");
warning ("non-negative-variable",
        "'a' must be non-negative number. Setting 'a' to zero.");
```

The functions distributed with Octave can issue one of the following warnings.

Octave:array-to-scalar

If the `Octave:array-to-scalar` warning is enabled, Octave will warn when an implicit conversion from an array to a scalar value is attempted. By default, the `Octave:array-to-scalar` warning is disabled.

Octave:array-to-vector

If the `Octave:array-to-vector` warning is enabled, Octave will warn when an implicit conversion from an array to a vector value is attempted. By default, the `Octave:array-to-vector` warning is disabled.

Octave:assign-as-truth-value

If the `Octave:assign-as-truth-value` warning is enabled, a warning is issued for statements like

```
if (s = t)
  ...
```

since such statements are not common, and it is likely that the intent was to write

```
if (s == t)
  ...
```

instead.

There are times when it is useful to write code that contains assignments within the condition of a `while` or `if` statement. For example, statements like

```
while (c = getc())
  ...
```

are common in C programming.

It is possible to avoid all warnings about such statements by disabling the `Octave:assign-as-truth-value` warning, but that may also let real errors like

```
if (x = 1) # intended to test (x == 1)!
  ...
```

slip by.

In such cases, it is possible suppress errors for specific statements by writing them with an extra set of parentheses. For example, writing the previous example as

```
while ((c = getc()))
  ...
```

will prevent the warning from being printed for this statement, while allowing Octave to warn about other assignments used in conditional contexts.

By default, the `Octave:assign-as-truth-value` warning is enabled.

Octave:associativity-change

If the `Octave:associativity-change` warning is enabled, Octave will warn about possible changes in the meaning of some code due to changes in associativity for some operators. Associativity changes have typically been made for MATLAB compatibility. By default, the `Octave:associativity-change` warning is enabled.

Octave:divide-by-zero

If the `Octave:divide-by-zero` warning is enabled, a warning is issued when Octave encounters a division by zero. By default, the `Octave:divide-by-zero` warning is enabled.

Octave:empty-list-elements

If the `Octave:empty-list-elements` warning is enabled, a warning is issued when an empty matrix is found in a matrix list. For example,

```
a = [1, [], 3, [], 5]
```

By default, the `Octave:empty-list-elements` warning is enabled.

Octave:fortran-indexing

If the `Octave:fortran-indexing` warning is enabled, a warning is printed for expressions which select elements of a two-dimensional matrix using a single index. By default, the `Octave:fortran-indexing` warning is disabled.

Octave:function-name-clash

If the `Octave:function-name-clash` warning is enabled, a warning is issued when Octave finds that the name of a function defined in a function file differs from the name of the file. (If the names disagree, the name declared inside the file is ignored.) By default, the `Octave:function-name-clash` warning is enabled.

Octave:future-time-stamp

If the `Octave:future-time-stamp` warning is enabled, Octave will print a warning if it finds a function file with a time stamp that is in the future. By default, the `Octave:future-time-stamp` warning is enabled.

Octave:imag-to-real

If the `Octave:imag-to-real` warning is enabled, a warning is printed for implicit conversions of complex numbers to real numbers. By default, the `Octave:imag-to-real` warning is disabled.

Octave:matlab-incompatible

Print warnings for Octave language features that may cause compatibility problems with MATLAB.

Octave:missing-semicolon

If the `Octave:missing-semicolon` warning is enabled, Octave will warn when statements in function definitions don't end in semicolons. By default the `Octave:missing-semicolon` warning is disabled.

Octave:neg-dim-as-zero

If the `Octave:neg-dim-as-zero` warning is enabled, print a warning for expressions like

```
eye (-1)
```

By default, the `Octave:neg-dim-as-zero` warning is disabled.

`Octave:num-to-str`

If the `Octave:num-to-str` warning is enable, a warning is printed for implicit conversions of numbers to their ASCII character equivalents when strings are constructed using a mixture of strings and numbers in matrix notation. For example,

```
[ "f", 111, 111 ]  
⇒ "foo"
```

elicits a warning if the `Octave:num-to-str` warning is enabled. By default, the `Octave:num-to-str` warning is enabled.

`Octave:precedence-change`

If the `Octave:precedence-change` warning is enabled, Octave will warn about possible changes in the meaning of some code due to changes in precedence for some operators. Precedence changes have typically been made for MATLAB compatibility. By default, the `Octave:precedence-change` warning is enabled.

`Octave:reload-forces-clear`

If several functions have been loaded from the same file, Octave must clear all the functions before any one of them can be reloaded. If the `Octave:reload-forces-clear` warning is enabled, Octave will warn you when this happens, and print a list of the additional functions that it is forced to clear. By default, the `Octave:reload-forces-clear` warning is enabled.

`Octave:resize-on-range-error`

If the `Octave:resize-on-range-error` warning is enabled, print a warning when a matrix is resized by an indexed assignment with indices outside the current bounds. By default, the `Octave:resize-on-range-error` warning is disabled.

`Octave:separator-insert`

Print warning if commas or semicolons might be inserted automatically in literal matrices.

`Octave:single-quote-string`

Print warning if a single quote character is used to introduce a string constant.

`Octave:str-to-num`

If the `Octave:str-to-num` warning is enabled, a warning is printed for implicit conversions of strings to their numeric ASCII equivalents. For example,

```
"abc" + 0  
⇒ 97 98 99
```

elicits a warning if the `Octave:str-to-num` warning is enabled. By default, the `Octave:str-to-num` warning is disabled.

`Octave:string-concat`

If the `Octave:string-concat` warning is enabled, print a warning when concatenating a mixture of double and single quoted strings. By default, the `Octave:string-concat` warning is disabled.

Octave:undefined-return-values

If the `Octave:undefined-return-values` warning is disabled, print a warning if a function does not define all the values in the return list which are expected. By default, the `Octave:undefined-return-values` warning is enabled.

Octave:variable-switch-label

If the `Octave:variable-switch-label` warning is enabled, Octave will print a warning if a switch label is not a constant or constant expression. By default, the `Octave:variable-switch-label` warning is disabled.

14 Debugging

Octave includes a built-in debugger to aid in the development of scripts. This can be used to interrupt the execution of an Octave script at a certain point, or when certain conditions are met. Once execution has stopped, and debug mode is entered, the symbol table at the point where execution has stopped can be examined and modified to check for errors.

The normal commandline editing and history functions are available in debug mode. However, one limitation on the debug mode is that commands entered at the debug prompt are evaluated as strings, rather than being handled by the Octave parser. This means that all commands in debug mode must be contained on a single line. That is, it is alright to write

```
debug> for i = 1:n, foo(i); endfor
```

in debug mode. However, writing the above in three lines will not be correctly evaluated. To leave the debug mode, you should simply type either `quit`, `exit`, `return` or `dbcont`.

14.1 Entering Debug Mode

There are two basic means of interrupting the execution of an Octave script. These are breakpoints [Section 14.2 \[Breakpoints\]](#), [page 147](#), discussed in the next section and interruption based on some condition.

Octave supports three means to stop execution based on the values set in the functions `debug_on_interrupt`, `debug_on_warning` and `debug_on_error`.

```
val = debug_on_interrupt () [Built-in Function]
old_val = debug_on_interrupt (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave will try to enter debugging mode when it receives an interrupt signal (typically generated with `C-c`). If a second interrupt signal is received before reaching the debugging mode, a normal interrupt will occur.

```
val = debug_on_warning () [Built-in Function]
old_val = debug_on_warning (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave will try to enter the debugger when a warning is encountered.

```
val = debug_on_error () [Built-in Function]
old_val = debug_on_error (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave will try to enter the debugger when an error is encountered. This will also inhibit printing of the normal traceback message (you will only see the top-level error message).

14.2 Breakpoints

Breakpoints can be set in any Octave function, using the `dbstop` function.

```
rline = dbstop (func, line, ...) [Loadable Function]
```

Set a breakpoint in a function

- func** String representing the function name. When already in debug mode this should be left out and only the line should be given.
- line** Line you would like the breakpoint to be set on. Multiple lines might be given as separate arguments or as a vector.

The rline returned is the real line that the breakpoint was set at.

See also: `dbclear`, `dbstatus`, `dbnext`.

Note that breakpoints can not be set in built-in functions (eg. `sin`, etc) or dynamically loaded function (ie. oct-files). To set a breakpoint immediately on entering a function, the breakpoint should be set to line 1. The leading comment block will be ignored and the breakpoint will be set to the first executable statement in the function. For example

```
dbstop ("asind", 1)
⇒ 27
```

Note that the return value of 27 means that the breakpoint was effectively set to line 27. The status of breakpoints in a function can be queried with the `dbstatus` function.

```
lst = dbstatus ([func]) [Loadable Function]
Return a vector containing the lines on which a function has breakpoints set.
```

- func** String representing the function name. When already in debug mode this should be left out.

See also: `dbclear`, `dbwhere`.

Taking the above as an example, `dbstatus ("asind")` should return 27. The breakpoints can then be cleared with the `dbclear` function

```
dbclear (func, line, ...) [Loadable Function]
Delete a breakpoint in a function
```

- func** String representing the function name. When already in debug mode this should be left out and only the line should be given.
- line** Line where you would like to remove the breakpoint. Multiple lines might be given as separate arguments or as a vector.

No checking is done to make sure that the line you requested is really a breakpoint. If you get the wrong line nothing will happen.

See also: `dbstop`, `dbstatus`, `dbwhere`.

To clear all of the breakpoints in a function the recommended means, following the above example, is then

```
dbclear ("asind", dbstatus ("asind"));
```

Another simple means of setting a breakpoint in an Octave script is the use of the `keyboard` function.

keyboard (*prompt*) [Built-in Function]

This function is normally used for simple debugging. When the **keyboard** function is executed, Octave prints a prompt and waits for user input. The input strings are then evaluated and the results are printed. This makes it possible to examine the values of variables within a function, and to assign new values to variables. No value is returned from the **keyboard** function, and it continues to prompt for input until the user types 'quit', or 'exit'.

If **keyboard** is invoked without any arguments, a default prompt of 'debug> ' is used.

The **keyboard** function is typically placed in a script at the point where the user desires that the execution is stopped. It automatically sets the running script into the debug mode.

14.3 Debug Mode

There are two additional support functions that allow the user to interrogate where in the execution of a script Octave entered the debug mode and to print the code in the script surrounding the point where Octave entered debug mode.

dbwhere () [Loadable Function]

Show where we are in the code

See also: **dbclear**, **dbstatus**, **dbstop**.

dbtype () [Loadable Function]

List script file with line numbers.

See also: **dbclear**, **dbstatus**, **dbstop**.

Debug mode equally allows single line stepping through a function using the commands **dbstep** and **dbnext**. These differ slightly in the way they treat the next executable line if the next line itself is a function defined in an m-file. The **dbnext** command will execute the next line, while staying in the existing function being debugged. The **dbstep** command will step in to the new function.

15 Input and Output

Octave supports several ways of reading and writing data to or from the prompt or a file. The most simple functions for data Input and Output (I/O) are easy to use, but only provides a limited control of how data is processed. For more control, a set of functions modelled after the C standard library are also provided by Octave.

15.1 Basic Input and Output

15.1.1 Terminal Output

Since Octave normally prints the value of an expression as soon as it has been evaluated, the simplest of all I/O functions is a simple expression. For example, the following expression will display the value of 'pi'

```
pi
⇒ pi = 3.1416
```

This works well as long as it is acceptable to have the name of the variable (or 'ans') printed along with the value. To print the value of a variable without printing its name, use the function `disp`.

The `format` command offers some control over the way Octave prints values with `disp` and through the normal echoing mechanism.

ans [Automatic Variable]

The the most recently computed result that was not explicitly assigned to a variable. For example, after the expression

```
3^2 + 4^2
```

is evaluated, the value returned by `ans` is 25.

disp (x) [Built-in Function]

Display the value of x. For example,

```
disp ("The value of pi is:"), disp (pi)
```

```
⇒ the value of pi is:
⇒ 3.1416
```

Note that the output from `disp` always ends with a newline.

If an output value is requested, `disp` prints nothing and returns the formatted output in a string.

See also: `fdisp`.

format options [Command]

Control the format of the output produced by `disp` and Octave's normal echoing mechanism. Valid options are listed in the following table.

short Octave will try to print numbers with at least 5 significant figures within a field that is a maximum of 10 characters wide (not counting additional spacing that is added between columns of a matrix).

If Octave is unable to format a matrix so that columns line up on the decimal point and all the numbers fit within the maximum field width, it switches to an ‘e’ format.

long Octave will try to print numbers with at least 15 significant figures within a field that is a maximum of 20 characters wide (not counting additional spacing that is added between columns of a matrix).

As will the ‘short’ format, Octave will switch to an ‘e’ format if it is unable to format a matrix so that columns line up on the decimal point and all the numbers fit within the maximum field width.

long e
short e

The same as ‘format long’ or ‘format short’ but always display output with an ‘e’ format. For example, with the ‘short e’ format, pi is displayed as 3.14e+00.

long E
short E

The same as ‘format long e’ or ‘format short e’ but always display output with an uppercase ‘E’ format. For example, with the ‘long E’ format, pi is displayed as 3.14159265358979E+00.

long g
short g

Choose between normal ‘long’ (or ‘short’) and ‘long e’ (or ‘short e’) formats based on the magnitude of the number. For example, with the ‘short g’ format, pi .^ [2; 4; 8; 16; 32] is displayed as

ans =

```

      3.1416
      9.8696
     97.409
    9488.5
   9.0032e+07
  8.1058e+15

```

long G
short G

The same as ‘format long g’ or ‘format short g’ but use an uppercase ‘E’ format. For example, with the ‘short G’ format, pi .^ [2; 4; 8; 16; 32] is displayed as

ans =

```

      3.1416
      9.8696
     97.409
    9488.5
   9.0032E+07
  8.1058E+15

```

free
none

Print output in free format, without trying to line up columns of matrices on the decimal point. This also causes complex numbers to be

formatted like this ‘(0.604194, 0.607088)’ instead of like this ‘0.60419 + 0.60709i’.

bank Print in a fixed format with two places to the right of the decimal point.

+

+ *chars*

plus

plus *chars*

Print a ‘+’ symbol for nonzero matrix elements and a space for zero matrix elements. This format can be very useful for examining the structure of a large matrix.

The optional argument *chars* specifies a list of 3 characters to use for printing values greater than zero, less than zero and equal to zero. For example, with the ‘+ "+-."’ format, [1, 0, -1; -1, 0, 1] is displayed as

```
ans =
```

```
+. -
```

```
- . +
```

native-hex

Print the hexadecimal representation numbers as they are stored in memory. For example, on a workstation which stores 8 byte real values in IEEE format with the least significant byte first, the value of `pi` when printed in **hex** format is 400921fb54442d18. This format only works for numeric values.

hex The same as **native-hex**, but always print the most significant byte first.

native-bit

Print the bit representation of numbers as stored in memory. For example, the value of `pi` is

```
01000000000010010010000111111011
```

```
01010100010001000010110100011000
```

(shown here in two 32 bit sections for typesetting purposes) when printed in bit format on a workstation which stores 8 byte real values in IEEE format with the least significant byte first. This format only works for numeric types.

bit The same as **native-bit**, but always print the most significant bits first.

compact Remove extra blank space around column number labels.

loose Insert blank lines above and below column number labels (this is the default).

rat Print a rational approximation. That is the values are approximated by one small integer divided by another.

By default, Octave will try to print numbers with at least 5 significant figures within a field that is a maximum of 10 characters wide.

If Octave is unable to format a matrix so that columns line up on the decimal point and all the numbers fit within the maximum field width, it switches to an ‘e’ format.

If `format` is invoked without any options, the default format state is restored.

```
val = print_answer_id_name () [Built-in Function]
old_val = print_answer_id_name (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether variable names are printed along with results produced by evaluating an expression.

15.1.1.1 Paging Screen Output

When running interactively, Octave normally sends any output intended for your terminal that is more than one screen long to a paging program, such as `less` or `more`. This avoids the problem of having a large volume of output stream by before you can read it. With `less` (and some versions of `more`) you can also scan forward and backward, and search for specific items.

Normally, no output is displayed by the pager until just before Octave is ready to print the top level prompt, or read from the standard input (for example, by using the `fscanf` or `scanf` functions). This means that there may be some delay before any output appears on your screen if you have asked Octave to perform a significant amount of work with a single command statement. The function `fflush` may be used to force output to be sent to the pager (or any other stream) immediately.

You can select the program to run as the pager using the `PAGER` function, and you can turn paging off by using the function `more`.

```
more [Command]
more on [Command]
more off [Command]
```

Turn output pagination on or off. Without an argument, `more` toggles the current state.

```
val = PAGER () [Built-in Function]
old_val = PAGER (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the program to use to display terminal output on your system. The default value is normally "less", "more", or "pg", depending on what programs are installed on your system. See [Appendix E \[Installation\]](#), page 525.

See also: `more`, `page_screen_output`, `page_output_immediately`, `PAGER_FLAGS`.

```
val = PAGER_FLAGS () [Built-in Function]
old_val = PAGER_FLAGS (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the options to pass to the pager.

See also: `PAGER`.

```
val = page_screen_output () [Built-in Function]
```

`old_val = page_screen_output (new_val)` [Built-in Function]

Query or set the internal variable that controls whether output intended for the terminal window that is longer than one page is sent through a pager. This allows you to view one screenful at a time. Some pagers (such as `less`—see [Appendix E \[Installation\]](#), page 525) are also capable of moving backward on the output.

`val = page_output_immediately ()` [Built-in Function]

`val = page_output_immediately (new_val)` [Built-in Function]

Query or set the internal variable that controls whether Octave sends output to the pager as soon as it is available. Otherwise, Octave buffers its output and waits until just before the prompt is printed to flush it to the pager.

`fflush (fid)` [Built-in Function]

Flush output to *fid*. This is useful for ensuring that all pending output makes it to the screen before some other event occurs. For example, it is always a good idea to flush the standard output stream before calling `input`.

`fflush` returns 0 on success and an OS dependent error value (−1 on unix) on error.

See also: `fopen`, `fclose`.

15.1.2 Terminal Input

Octave has three functions that make it easy to prompt users for input. The `input` and `menu` functions are normally used for managing an interactive dialog with a user, and the `keyboard` function is normally used for doing simple debugging.

`input (prompt)` [Built-in Function]

`input (prompt, "s")` [Built-in Function]

Print a prompt and wait for user input. For example,

```
input ("Pick a number, any number! ")
```

prints the prompt

```
Pick a number, any number!
```

and waits for the user to enter a value. The string entered by the user is evaluated as an expression, so it may be a literal constant, a variable name, or any other valid expression.

Currently, `input` only returns one value, regardless of the number of values produced by the evaluation of the expression.

If you are only interested in getting a literal string value, you can call `input` with the character string `"s"` as the second argument. This tells Octave to return the string entered by the user directly, without evaluating it first.

Because there may be output waiting to be displayed by the pager, it is a good idea to always call `fflush (stdout)` before calling `input`. This will ensure that all pending output is written to the screen before your prompt. See [Chapter 15 \[Input and Output\]](#), page 151.

menu (*title*, *opt1*, ...) [Function File]

Print a title string followed by a series of options. Each option will be printed along with a number. The return value is the number of the option selected by the user. This function is useful for interactive programs. There is no limit to the number of options that may be passed in, but it may be confusing to present more than will fit easily on one screen.

See also: disp, printf, input.

For **input**, the normal command line history and editing functions are available at the prompt.

Octave also has a function that makes it possible to get a single character from the keyboard without requiring the user to type a carriage return.

kbhit () [Built-in Function]

Read a single keystroke from the keyboard. If called with one argument, don't wait for a keypress. For example,

```
x = kbhit ();
```

will set *x* to the next character typed at the keyboard as soon as it is typed.

```
x = kbhit (1);
```

identical to the above example, but don't wait for a keypress, returning the empty string if no key is available.

15.1.3 Simple File I/O

The **save** and **load** commands allow data to be written to and read from disk files in various formats. The default format of files written by the **save** command can be controlled using the functions **default_save_options** and **save_precision**.

As an example the following code creates a 3-by-3 matrix and saves it to the file 'myfile.mat'.

```
A = [ 1:3; 4:6; 7:9 ];
save myfile.mat A
```

Once one or more variables have been saved to a file, they can be read into memory using the **load** command.

```
load myfile.mat
A
```

```

+ A =
+
+   1   2   3
+   4   5   6
+   7   8   9
```

save *options file v1 v2 ...* [Command]

Save the named variables *v1*, *v2*, ... in the file *file*. The special filename '-' can be used to write the output to your terminal. If no variable names are listed, Octave saves all the variables in the current scope. Valid options for the **save** command are listed in the following table. Options that modify the output format override the format specified by **default_save_options**.

If `save` is invoked using the functional form

```
save ("-text", "file.txt", "a")
```

then the *options*, *file*, and variable name arguments (*vname1*, ...) must be specified as character strings.

-ascii Save a single matrix in a text file.

-binary Save the data in Octave's binary data format.

-float-binary

Save the data in Octave's binary data format but only using single precision. You should use this format only if you know that all the values to be saved can be represented in single precision.

-V7

-v7

-7

-mat7-binary

Save the data in MATLAB's v7 binary data format.

-V6

-v6

-6

-mat

-mat-binary

Save the data in MATLAB's v6 binary data format.

-V4

-v4

-4

-mat4-binary

Save the data in the binary format written by MATLAB version 4.

-hdf5 Save the data in HDF5 format. (HDF5 is a free, portable binary format developed by the National Center for Supercomputing Applications at the University of Illinois.)

HDF5 load and save are not available, as this Octave executable was not linked with the HDF5 library.

-float-hdf5

Save the data in HDF5 format but only using single precision. You should use this format only if you know that all the values to be saved can be represented in single precision.

-zip

-z

Use the gzip algorithm to compress the file. This works equally on files that are compressed with gzip outside of octave, and gzip can equally be used to convert the files for backward compatibility.

The list of variables to save may include wildcard patterns containing the following special characters:

? Match any single character.

- *** Match zero or more characters.
- [list]** Match the list of characters specified by *list*. If the first character is **!** or **^**, match all characters except those specified by *list*. For example, the pattern **'[a-zA-Z]'** will match all lower and upper case alphabetic characters.
- text** Save the data in Octave's text data format.

Except when using the MATLAB binary data file format, saving global variables also saves the global status of the variable, so that if it is restored at a later time using **'load'**, it will be restored as a global variable.

The command

```
save -binary data a b*
```

saves the variable **'a'** and all variables beginning with **'b'** to the file **'data'** in Octave's binary format.

load options file v1 v2 ... [Command]

Load the named variables from the file *file*. As with **save**, you may specify a list of variables and **load** will only extract those variables with names that match. For example, to restore the variables saved in the file **'data'**, use the command

```
load data
```

If **load** is invoked using the functional form

```
load ("-text", "file.txt", "a")
```

then the *options*, *file*, and variable name arguments (*v1*, ...) must be specified as character strings.

If a variable that is not marked as global is loaded from a file when a global symbol with the same name already exists, it is loaded in the global symbol table. Also, if a variable is marked as global in a file and a local symbol exists, the local symbol is moved to the global symbol table and given the value from the file. Since it seems that both of these cases are likely to be the result of some sort of error, they will generate warnings.

If invoked with a single output argument, Octave returns data instead of inserting variables in the symbol table. If the data file contains only numbers (TAB- or space-delimited columns), a matrix of values is returned. Otherwise, **load** returns a structure with members corresponding to the names of the variables in the file.

The **load** command can read data stored in Octave's text and binary formats, and MATLAB's binary format. It will automatically detect the type of file and do conversion from different floating point formats (currently only IEEE big and little endian, though other formats may be added in the future).

Valid options for **load** are listed in the following table.

- force** The **'-force'** option is accepted but ignored for backward compatibility. Octave now overwrites variables currently in memory with the same name as those found in the file.

- `-ascii` Force Octave to assume the file contains columns of numbers in text format without any header or other information. Data in the file will be loaded as a single numeric matrix with the name of the variable derived from the name of the file.
- `-binary` Force Octave to assume the file is in Octave's binary format.
- `-mat`
- `-mat-binary`
- `-6`
- `-v6`
- `-7`
- `-v7` Force Octave to assume the file is in MATLAB's version 6 or 7 binary format.
- `-V4`
- `-v4`
- `-4`
- `-mat4-binary` Force Octave to assume the file is in the binary format written by MATLAB version 4.
- `-hdf5` Force Octave to assume the file is in HDF5 format. (HDF5 is a free, portable binary format developed by the National Center for Supercomputing Applications at the University of Illinois.) Note that Octave can read HDF5 files not created by itself, but may skip some datasets in formats that it cannot support.
HDF5 load and save are not available, as this Octave executable was not linked with the HDF5 library.
- `-import` The `'-import'` is accepted but ignored for backward compatibility. Octave can now support multi-dimensional HDF data and automatically modifies variable names if they are invalid Octave identifiers.
- `-text` Force Octave to assume the file is in Octave's text format.

There are three functions that modify the behavior of `save`.

`val = default_save_options ()` [Built-in Function]

`old_val = default_save_options (new_val)` [Built-in Function]

Query or set the internal variable that specifies the default options for the `save` command, and defines the default format. Typical values include `"-ascii"`, `"-ascii-zip"`. The default value is `-ascii`.

See also: `save`.

`val = save_precision ()` [Built-in Function]

`old_val = save_precision (new_val)` [Built-in Function]

Query or set the internal variable that specifies the number of digits to keep when saving data in text format.

```
val = save_header_format_string () [Built-in Function]
old_val = save_header_format_string (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the format string used for the comment line written at the beginning of text-format data files saved by Octave. The format string is passed to `strftime` and should begin with the character ‘#’ and contain no newline characters. If the value of `save_header_format_string` is the empty string, the header comment is omitted from text-format data files. The default value is

```
"# Created by Octave VERSION, %a %b %d %H:%M:%S %Y %Z <USER@HOST>"
```

See also: `strftime`.

```
native_float_format () [Built-in Function]
Return the native floating point format as a string
```

It is possible to write data to a file in a way much similar to the `disp` function for writing data to the screen. The `fdisp` works just like `disp` except its first argument is a file pointer as created by `fopen`. As an example, the following code writes to data ‘myfile.txt’.

```
fid = fopen ("myfile.txt", "w");
fdisp (fid, "3/8 is ");
fdisp (fid, 3/8);
fclose (fid);
```

See [Section 15.2.1 \[Opening and Closing Files\], page 162](#), for details on how to use `fopen` and `fclose`.

```
fdisp (fid, x) [Built-in Function]
Display the value of x on the stream fid. For example,
fdisp (stdout, "The value of pi is:"), fdisp (stdout, pi)
```

```
→ the value of pi is:
→ 3.1416
```

Note that the output from `fdisp` always ends with a newline.

See also: `disp`.

15.1.3.1 Saving Data on Unexpected Exits

If Octave for some reason exits unexpectedly it will by default save the variables available in the workspace to a file in the current directory. By default this file is named ‘octave-core’ and can be loaded into memory with the `load` command. While the default behaviour most often is reasonable it can be changed through the following functions.

```
val = crash_dumps_octave_core () [Built-in Function]
old_val = crash_dumps_octave_core (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave tries to save all current variables to the file “octave-core” if it crashes or receives a hangup, terminate or similar signal.

See also: `octave_core_file_limit`, `octave_core_file_name`, `octave_core_file_options`.

```
val = sighup_dumps_octave_core () [Built-in Function]
old_val = sighup_dumps_octave_core (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave tries to save all current variables to the file "octave-core" if it receives a hangup signal.

```
val = sigterm_dumps_octave_core () [Built-in Function]
old_val = sigterm_dumps_octave_core (new_val) [Built-in Function]
```

Query or set the internal variable that controls whether Octave tries to save all current variables to the file "octave-core" if it receives a terminate signal.

```
val = octave_core_file_options () [Built-in Function]
old_val = octave_core_file_options (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the options used for saving the workspace data if Octave aborts. The value of `octave_core_file_options` should follow the same format as the options for the `save` function. The default value is Octave's binary format.

See also: `crash_dumps_octave_core`, `octave_core_file_name`, `octave_core_file_limit`.

```
val = octave_core_file_limit () [Built-in Function]
old_val = octave_core_file_limit (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the maximum amount of memory (in kilobytes) of the top-level workspace that Octave will attempt to save when writing data to the crash dump file (the name of the file is specified by `octave_core_file_name`). If `octave_core_file_options` flags specifies a binary format, then `octave_core_file_limit` will be approximately the maximum size of the file. If a text file format is used, then the file could be much larger than the limit. The default value is -1 (unlimited)

See also: `crash_dumps_octave_core`, `octave_core_file_name`, `octave_core_file_options`.

```
val = octave_core_file_name () [Built-in Function]
old_val = octave_core_file_name (new_val) [Built-in Function]
```

Query or set the internal variable that specifies the name of the file used for saving data from the top-level workspace if Octave aborts. The default value is "octave-core"

See also: `crash_dumps_octave_core`, `octave_core_file_name`, `octave_core_file_options`.

15.1.4 Rational Approximations

```
s = rat (x, tol) [Function File]
[n, d] = rat (x, tol) [Function File]
```

Find a rational approximation to x within tolerance defined by tol using a continued fraction expansion. E.g,

$$\text{rat}(\pi) = 3 + 1/(7 + 1/16) = 355/113$$

$$\text{rat}(e) = 3 + 1/(-4 + 1/(2 + 1/(5 + 1/(-2 + 1/(-7)))))) = 1457/536$$

Called with two arguments returns the numerator and denominator separately as two matrices.

See also: `rats`.

`rats (x, len)` [Built-in Function]

Convert `x` into a rational approximation represented as a string. You can convert the string back into a matrix as follows:

```
eval([' ',rats(hilb(4)),'],';')
```

The optional second argument defines the maximum length of the string representing the elements of `x`. By default `len` is 9.

See also: `format`, `rat`.

15.2 C-Style I/O Functions

Octave's C-style input and output functions provide most of the functionality of the C programming language's standard I/O library. The argument lists for some of the input functions are slightly different, however, because Octave has no way of passing arguments by reference.

In the following, *file* refers to a file name and `fid` refers to an integer file number, as returned by `fopen`.

There are three files that are always available. Although these files can be accessed using their corresponding numeric file ids, you should always use the symbolic names given in the table below, since it will make your programs easier to understand.

`stdin ()` [Built-in Function]

Return the numeric value corresponding to the standard input stream. When Octave is used interactively, this is filtered through the command line editing functions.

See also: `stdout`, `stderr`.

`stdout ()` [Built-in Function]

Return the numeric value corresponding to the standard output stream. Data written to the standard output is normally filtered through the pager.

See also: `stdin`, `stderr`.

`stderr ()` [Built-in Function]

Return the numeric value corresponding to the standard error stream. Even if paging is turned on, the standard error is not sent to the pager. It is useful for error messages and prompts.

See also: `stdin`, `stdout`.

15.2.1 Opening and Closing Files

When reading data from a file it must be opened for reading first, and likewise when writing to a file. The `fopen` function returns a pointer to an open file that is ready to be read or written. Once all data has been read from or written to the opened file it should be closed. The `fclose` function does this. The following code illustrates the basic pattern for writing to a file, but a very similar pattern is used when reading a file.

```
filename = "myfile.txt";
fid = fopen (filename, "w");
# Do the actual I/O here...
fclose (fid);
```

```
[fid, msg] = fopen (name, mode, arch) [Built-in Function]
fid_list = fopen ("all") [Built-in Function]
[file, mode, arch] = fopen (fid) [Built-in Function]
```

The first form of the `fopen` function opens the named file with the specified mode (read-write, read-only, etc.) and architecture interpretation (IEEE big endian, IEEE little endian, etc.), and returns an integer value that may be used to refer to the file later. If an error occurs, `fid` is set to `-1` and `msg` contains the corresponding system error message. The `mode` is a one or two character string that specifies whether the file is to be opened for reading, writing, or both.

The second form of the `fopen` function returns a vector of file ids corresponding to all the currently open files, excluding the `stdin`, `stdout`, and `stderr` streams.

The third form of the `fopen` function returns information about the open file given its file id.

For example,

```
myfile = fopen ("splat.dat", "r", "ieee-le");
```

opens the file `'splat.dat'` for reading. If necessary, binary numeric values will be read assuming they are stored in IEEE format with the least significant bit first, and then converted to the native representation.

Opening a file that is already open simply opens it again and returns a separate file id. It is not an error to open a file several times, though writing to the same file through several different file ids may produce unexpected results.

The possible values `'mode'` may have are

<code>'r'</code>	Open a file for reading.
<code>'w'</code>	Open a file for writing. The previous contents are discarded.
<code>'a'</code>	Open or create a file for writing at the end of the file.
<code>'r+'</code>	Open an existing file for reading and writing.
<code>'w+'</code>	Open a file for reading or writing. The previous contents are discarded.
<code>'a+'</code>	Open or create a file for reading or writing at the end of the file.

Append a `"t"` to the mode string to open the file in text mode or a `"b"` to open in binary mode. On Windows and Macintosh systems, text mode reading and writing automatically converts linefeeds to the appropriate line end character for the system (carriage-return linefeed on Windows, carriage-return on Macintosh). The default if no mode is specified is binary mode.

Additionally, you may append a `"z"` to the mode string to open a gzipped file for reading or writing. For this to be successful, you must also open the file in binary mode.

The parameter `arch` is a string specifying the default data format for the file. Valid values for `arch` are:

‘native’ The format of the current machine (this is the default).
 ‘ieee-be’ IEEE big endian format.
 ‘ieee-le’ IEEE little endian format.
 ‘vaxd’ VAX D floating format.
 ‘vaxg’ VAX G floating format.
 ‘cray’ Cray floating format.

however, conversions are currently only supported for ‘native’ ‘ieee-be’, and ‘ieee-le’ formats.

See also: `fclose`, `fread`, `fseek`.

`fclose (fid)` [Built-in Function]

Closes the specified file. If successful, `fclose` returns 0, otherwise, it returns -1.

See also: `fopen`, `fseek`, `ftell`.

15.2.2 Simple Output

Once a file has been opened for writing a string can be written to the file using the `fputs` function. The following example shows how to write the string ‘Free Software is needed for Free Science’ to the file ‘free.txt’.

```
filename = "free.txt";
fid = fopen (filename, "w");
fputs (fid, "Free Software is needed for Free Science");
fclose (fid);
```

`fputs (fid, string)` [Built-in Function]

Write a string to a file with no formatting.

Return a non-negative number on success and EOF on error.

A function much similar to `fputs` is available for writing data to the screen. The `puts` function works just like `fputs` except it doesn’t take a file pointer as its input.

`puts (string)` [Built-in Function]

Write a string to the standard output with no formatting.

Return a non-negative number on success and EOF on error.

15.2.3 Line-Oriented Input

To read from a file it must be opened for reading using `fopen`. Then a line can be read from the file using `fgetl` as the following code illustrates

```
fid = fopen ("free.txt");
txt = fgetl (fid)
    └ Free Software is needed for Free Science
fclose (fid);
```

This of course assumes that the file ‘free.txt’ exists and contains the line ‘Free Software is needed for Free Science’.

fgetl (*fid*, *len*) [Built-in Function]

Read characters from a file, stopping after a newline, or EOF, or *len* characters have been read. The characters read, excluding the possible trailing newline, are returned as a string.

If *len* is omitted, **fgetl** reads until the next newline character.

If there are no more characters to read, **fgetl** returns `-1`.

See also: `fread`, `fscanf`.

fgets (*fid*, *len*) [Built-in Function]

Read characters from a file, stopping after a newline, or EOF, or *len* characters have been read. The characters read, including the possible trailing newline, are returned as a string.

If *len* is omitted, **fgets** reads until the next newline character.

If there are no more characters to read, **fgets** returns `-1`.

See also: `fread`, `fscanf`.

15.2.4 Formatted Output

This section describes how to call **printf** and related functions.

The following functions are available for formatted output. They are modelled after the C language functions of the same name, but they interpret the format template differently in order to improve the performance of printing vector and matrix values.

printf (*template*, ...) [Built-in Function]

Print optional arguments under the control of the template string *template* to the stream `stdout` and return the number of characters printed.

See also: `fprintf`, `sprintf`, `scanf`.

fprintf (*fid*, *template*, ...) [Built-in Function]

This function is just like **printf**, except that the output is written to the stream *fid* instead of `stdout`.

See also: `printf`, `sprintf`, `fread`, `fscanf`, `fopen`, `fclose`.

sprintf (*template*, ...) [Built-in Function]

This is like **printf**, except that the output is returned as a string. Unlike the C library function, which requires you to provide a suitably sized string as an argument, Octave's **sprintf** function returns the string, automatically sized to hold all of the items converted.

See also: `printf`, `fprintf`, `sscanf`.

The **printf** function can be used to print any number of arguments. The template string argument you supply in a call provides information not only about the number of additional arguments, but also about their types and what style should be used for printing them.

Ordinary characters in the template string are simply written to the output stream as-is, while *conversion specifications* introduced by a '%' character in the template cause subsequent arguments to be formatted and written to the output stream. For example,

```
pct = 37;
filename = "foo.txt";
printf ("Processing of '%s' is %d%% finished.\nPlease be patient.\n",
        filename, pct);
```

produces output like

```
Processing of 'foo.txt' is 37% finished.
Please be patient.
```

This example shows the use of the '%d' conversion to specify that a scalar argument should be printed in decimal notation, the '%s' conversion to specify printing of a string argument, and the '%%' conversion to print a literal '%' character.

There are also conversions for printing an integer argument as an unsigned value in octal, decimal, or hexadecimal radix ('%o', '%u', or '%x', respectively); or as a character value ('%c').

Floating-point numbers can be printed in normal, fixed-point notation using the '%f' conversion or in exponential notation using the '%e' conversion. The '%g' conversion uses either '%e' or '%f' format, depending on what is more appropriate for the magnitude of the particular number.

You can control formatting more precisely by writing *modifiers* between the '%' and the character that indicates which conversion to apply. These slightly alter the ordinary behavior of the conversion. For example, most conversion specifications permit you to specify a minimum field width and a flag indicating whether you want the result left- or right-justified within the field.

The specific flags and modifiers that are permitted and their interpretation vary depending on the particular conversion. They're all described in more detail in the following sections.

15.2.5 Output Conversion for Matrices

When given a matrix value, Octave's formatted output functions cycle through the format template until all the values in the matrix have been printed. For example,

```
printf ("%4.2f %10.2e %8.4g\n", hilb (3));
```

```
→ 1.00    5.00e-01    0.3333
→ 0.50    3.33e-01     0.25
→ 0.33    2.50e-01     0.2
```

If more than one value is to be printed in a single call, the output functions do not return to the beginning of the format template when moving on from one value to the next. This can lead to confusing output if the number of elements in the matrices are not exact multiples of the number of conversions in the format template. For example,

```
printf ("%4.2f %10.2e %8.4g\n", [1, 2], [3, 4]);
```

```
→ 1.00    2.00e+00     3
→ 4.00
```

If this is not what you want, use a series of calls instead of just one.

15.2.6 Output Conversion Syntax

This section provides details about the precise syntax of conversion specifications that can appear in a `printf` template string.

Characters in the template string that are not part of a conversion specification are printed as-is to the output stream.

The conversion specifications in a `printf` template string have the general form:

`% flags width [. precision] type conversion`

For example, in the conversion specifier `%-10.8ld`, the `-` is a flag, `10` specifies the field width, the precision is `8`, the letter `l` is a type modifier, and `d` specifies the conversion style. (This particular type specifier says to print a numeric argument in decimal notation, with a minimum of 8 digits left-justified in a field at least 10 characters wide.)

In more detail, output conversion specifications consist of an initial `%` character followed in sequence by:

- Zero or more *flag characters* that modify the normal behavior of the conversion specification.
- An optional decimal integer specifying the *minimum field width*. If the normal conversion produces fewer characters than this, the field is padded with spaces to the specified width. This is a *minimum* value; if the normal conversion produces more characters than this, the field is *not* truncated. Normally, the output is right-justified within the field.

You can also specify a field width of `*`. This means that the next argument in the argument list (before the actual value to be printed) is used as the field width. The value is rounded to the nearest integer. If the value is negative, this means to set the `-` flag (see below) and to use the absolute value as the field width.

- An optional *precision* to specify the number of digits to be written for the numeric conversions. If the precision is specified, it consists of a period (`.`) followed optionally by a decimal integer (which defaults to zero if omitted).

You can also specify a precision of `*`. This means that the next argument in the argument list (before the actual value to be printed) is used as the precision. The value must be an integer, and is ignored if it is negative.

- An optional *type modifier character*. This character is ignored by Octave's `printf` function, but is recognized to provide compatibility with the C language `printf`.
- A character that specifies the conversion to be applied.

The exact options that are permitted and how they are interpreted vary between the different conversion specifiers. See the descriptions of the individual conversions for information about the particular options that they use.

15.2.7 Table of Output Conversions

Here is a table summarizing what all the different conversions do:

`'%d', '%i'` Print an integer as a signed decimal number. See [Section 15.2.8 \[Integer Conversions\]](#), [page 168](#), for details. `'%d'` and `'%i'` are synonymous for output, but are

different when used with `scanf` for input (see [Section 15.2.13 \[Table of Input Conversions\]](#), page 172).

<code>'%o'</code>	Print an integer as an unsigned octal number. See Section 15.2.8 [Integer Conversions] , page 168, for details.
<code>'%u'</code>	Print an integer as an unsigned decimal number. See Section 15.2.8 [Integer Conversions] , page 168, for details.
<code>'%x', '%X'</code>	Print an integer as an unsigned hexadecimal number. <code>'%x'</code> uses lower-case letters and <code>'%X'</code> uses upper-case. See Section 15.2.8 [Integer Conversions] , page 168, for details.
<code>'%f'</code>	Print a floating-point number in normal (fixed-point) notation. See Section 15.2.9 [Floating-Point Conversions] , page 169, for details.
<code>'%e', '%E'</code>	Print a floating-point number in exponential notation. <code>'%e'</code> uses lower-case letters and <code>'%E'</code> uses upper-case. See Section 15.2.9 [Floating-Point Conversions] , page 169, for details.
<code>'%g', '%G'</code>	Print a floating-point number in either normal (fixed-point) or exponential notation, whichever is more appropriate for its magnitude. <code>'%g'</code> uses lower-case letters and <code>'%G'</code> uses upper-case. See Section 15.2.9 [Floating-Point Conversions] , page 169, for details.
<code>'%c'</code>	Print a single character. See Section 15.2.10 [Other Output Conversions] , page 170.
<code>'%s'</code>	Print a string. See Section 15.2.10 [Other Output Conversions] , page 170.
<code>'%%'</code>	Print a literal <code>'%'</code> character. See Section 15.2.10 [Other Output Conversions] , page 170.

If the syntax of a conversion specification is invalid, unpredictable things will happen, so don't do this. If there aren't enough function arguments provided to supply values for all the conversion specifications in the template string, or if the arguments are not of the correct types, the results are unpredictable. If you supply more arguments than conversion specifications, the extra argument values are simply ignored; this is sometimes useful.

15.2.8 Integer Conversions

This section describes the options for the `'%d'`, `'%i'`, `'%o'`, `'%u'`, `'%x'`, and `'%X'` conversion specifications. These conversions print integers in various formats.

The `'%d'` and `'%i'` conversion specifications both print an numeric argument as a signed decimal number; while `'%o'`, `'%u'`, and `'%x'` print the argument as an unsigned octal, decimal, or hexadecimal number (respectively). The `'%X'` conversion specification is just like `'%x'` except that it uses the characters `'ABCDEF'` as digits instead of `'abcdef'`.

The following flags are meaningful:

<code>'-'</code>	Left-justify the result in the field (instead of the normal right-justification).
<code>'+'</code>	For the signed <code>'%d'</code> and <code>'%i'</code> conversions, print a plus sign if the value is positive.
<code>' '</code>	For the signed <code>'%d'</code> and <code>'%i'</code> conversions, if the result doesn't start with a plus or minus sign, prefix it with a space character instead. Since the <code>'+'</code> flag ensures that the result includes a sign, this flag is ignored if you supply both of them.

- '#' For the '%o' conversion, this forces the leading digit to be '0', as if by increasing the precision. For '%x' or '%X', this prefixes a leading '0x' or '0X' (respectively) to the result. This doesn't do anything useful for the '%d', '%i', or '%u' conversions.
- '0' Pad the field with zeros instead of spaces. The zeros are placed after any indication of sign or base. This flag is ignored if the '-' flag is also specified, or if a precision is specified.

If a precision is supplied, it specifies the minimum number of digits to appear; leading zeros are produced if necessary. If you don't specify a precision, the number is printed with as many digits as it needs. If you convert a value of zero with an explicit precision of zero, then no characters at all are produced.

15.2.9 Floating-Point Conversions

This section discusses the conversion specifications for floating-point numbers: the '%f', '%e', '%E', '%g', and '%G' conversions.

The '%f' conversion prints its argument in fixed-point notation, producing output of the form `[-]ddd.ddd`, where the number of digits following the decimal point is controlled by the precision you specify.

The '%e' conversion prints its argument in exponential notation, producing output of the form `[-]d.ddde[+|-]dd`. Again, the number of digits following the decimal point is controlled by the precision. The exponent always contains at least two digits. The '%E' conversion is similar but the exponent is marked with the letter 'E' instead of 'e'.

The '%g' and '%G' conversions print the argument in the style of '%e' or '%E' (respectively) if the exponent would be less than -4 or greater than or equal to the precision; otherwise they use the '%f' style. Trailing zeros are removed from the fractional portion of the result and a decimal-point character appears only if it is followed by a digit.

The following flags can be used to modify the behavior:

- '-' Left-justify the result in the field. Normally the result is right-justified.
- '+' Always include a plus or minus sign in the result.
- ' ' If the result doesn't start with a plus or minus sign, prefix it with a space instead. Since the '+' flag ensures that the result includes a sign, this flag is ignored if you supply both of them.
- '#' Specifies that the result should always include a decimal point, even if no digits follow it. For the '%g' and '%G' conversions, this also forces trailing zeros after the decimal point to be left in place where they would otherwise be removed.
- '0' Pad the field with zeros instead of spaces; the zeros are placed after any sign. This flag is ignored if the '-' flag is also specified.

The precision specifies how many digits follow the decimal-point character for the '%f', '%e', and '%E' conversions. For these conversions, the default precision is 6. If the precision is explicitly 0, this suppresses the decimal point character entirely. For the '%g' and '%G' conversions, the precision specifies how many significant digits to print. Significant digits are the first digit before the decimal point, and all the digits after it. If the precision is 0 or not specified for '%g' or '%G', it is treated like a value of 1. If the value being printed

cannot be expressed precisely in the specified number of digits, the value is rounded to the nearest number that fits.

15.2.10 Other Output Conversions

This section describes miscellaneous conversions for `printf`.

The `'c'` conversion prints a single character. The `'-'` flag can be used to specify left-justification in the field, but no other flags are defined, and no precision or type modifier can be given. For example:

```
printf ("%c%c%c%c%c", "h", "e", "l", "l", "o");
```

prints `'hello'`.

The `'s'` conversion prints a string. The corresponding argument must be a string. A precision can be specified to indicate the maximum number of characters to write; otherwise characters in the string up to but not including the terminating null character are written to the output stream. The `'-'` flag can be used to specify left-justification in the field, but no other flags or type modifiers are defined for this conversion. For example:

```
printf ("%3s%-6s", "no", "where");
```

prints `' nowhere '` (note the leading and trailing spaces).

15.2.11 Formatted Input

Octave provides the `scanf`, `fscanf`, and `sscanf` functions to read formatted input. There are two forms of each of these functions. One can be used to extract vectors of data from a file, and the other is more `'C-like'`.

```
[val, count] = fscanf (fid, template, size) [Built-in Function]
[v1, v2, ..., count] = fscanf (fid, template, "C") [Built-in Function]
```

In the first form, read from *fid* according to *template*, returning the result in the matrix *val*.

The optional argument *size* specifies the amount of data to read and may be one of

- | | |
|------------------|--|
| Inf | Read as much as possible, returning a column vector. |
| nr | Read up to <i>nr</i> elements, returning a column vector. |
| [nr, Inf] | Read as much as possible, returning a matrix with <i>nr</i> rows. If the number of elements read is not an exact multiple of <i>nr</i> , the last column is padded with zeros. |
| [nr, nc] | Read up to <i>nr * nc</i> elements, returning a matrix with <i>nr</i> rows. If the number of elements read is not an exact multiple of <i>nr</i> , the last column is padded with zeros. |

If *size* is omitted, a value of **Inf** is assumed.

A string is returned if *template* specifies only character conversions.

The number of items successfully read is returned in *count*.

In the second form, read from *fid* according to *template*, with each conversion specifier in *template* corresponding to a single scalar return value. This form is more `'C-like'`, and also compatible with previous versions of Octave. The number of successful conversions is returned in *count*.

See also: `scanf`, `sscanf`, `fread`, `fprintf`.

```
[val, count] = sscanf (string, template, size)           [Built-in Function]
[v1, v2, ..., count] =  sscanf (string, template, "C")  [Built-in Function]
```

This is like `fscanf`, except that the characters are taken from the string *string* instead of from a stream. Reaching the end of the string is treated as an end-of-file condition.

See also: `fscanf`, `scanf`, `sprintf`.

Calls to `scanf` are superficially similar to calls to `printf` in that arbitrary arguments are read under the control of a template string. While the syntax of the conversion specifications in the template is very similar to that for `printf`, the interpretation of the template is oriented more towards free-format input and simple pattern matching, rather than fixed-field formatting. For example, most `scanf` conversions skip over any amount of “white space” (including spaces, tabs, and newlines) in the input file, and there is no concept of precision for the numeric input conversions as there is for the corresponding output conversions. Ordinarily, non-whitespace characters in the template are expected to match characters in the input stream exactly.

When a *matching failure* occurs, `scanf` returns immediately, leaving the first non-matching character as the next character to be read from the stream, and `scanf` returns all the items that were successfully converted.

The formatted input functions are not used as frequently as the formatted output functions. Partly, this is because it takes some care to use them properly. Another reason is that it is difficult to recover from a matching error.

15.2.12 Input Conversion Syntax

A `scanf` template string is a string that contains ordinary multibyte characters interspersed with conversion specifications that start with ‘%’.

Any whitespace character in the template causes any number of whitespace characters in the input stream to be read and discarded. The whitespace characters that are matched need not be exactly the same whitespace characters that appear in the template string. For example, write ‘ , ’ in the template to recognize a comma with optional whitespace before and after.

Other characters in the template string that are not part of conversion specifications must match characters in the input stream exactly; if this is not the case, a matching failure occurs.

The conversion specifications in a `scanf` template string have the general form:

% flags width type conversion

In more detail, an input conversion specification consists of an initial ‘%’ character followed in sequence by:

- An optional *flag character* ‘*’, which says to ignore the text read for this specification. When `scanf` finds a conversion specification that uses this flag, it reads input as directed by the rest of the conversion specification, but it discards this input, does not return any value, and does not increment the count of successful assignments.
- An optional decimal integer that specifies the *maximum field width*. Reading of characters from the input stream stops either when this maximum is reached or when a non-matching character is found, whichever happens first. Most conversions discard initial whitespace characters, and these discarded characters don’t count towards the

maximum field width. Conversions that do not discard initial whitespace are explicitly documented.

- An optional type modifier character. This character is ignored by Octave’s `scanf` function, but is recognized to provide compatibility with the C language `scanf`.
- A character that specifies the conversion to be applied.

The exact options that are permitted and how they are interpreted vary between the different conversion specifiers. See the descriptions of the individual conversions for information about the particular options that they allow.

15.2.13 Table of Input Conversions

Here is a table that summarizes the various conversion specifications:

<code>‘%d’</code>	Matches an optionally signed integer written in decimal. See Section 15.2.14 [Numeric Input Conversions] , page 172.
<code>‘%i’</code>	Matches an optionally signed integer in any of the formats that the C language defines for specifying an integer constant. See Section 15.2.14 [Numeric Input Conversions] , page 172.
<code>‘%o’</code>	Matches an unsigned integer written in octal radix. See Section 15.2.14 [Numeric Input Conversions] , page 172.
<code>‘%u’</code>	Matches an unsigned integer written in decimal radix. See Section 15.2.14 [Numeric Input Conversions] , page 172.
<code>‘%x’, ‘%X’</code>	Matches an unsigned integer written in hexadecimal radix. See Section 15.2.14 [Numeric Input Conversions] , page 172.
<code>‘%e’, ‘%f’, ‘%g’, ‘%E’, ‘%G’</code>	Matches an optionally signed floating-point number. See Section 15.2.14 [Numeric Input Conversions] , page 172.
<code>‘%s’</code>	Matches a string containing only non-whitespace characters. See Section 15.2.15 [String Input Conversions] , page 173.
<code>‘%c’</code>	Matches a string of one or more characters; the number of characters read is controlled by the maximum field width given for the conversion. See Section 15.2.15 [String Input Conversions] , page 173.
<code>‘%%’</code>	This matches a literal <code>‘%’</code> character in the input stream. No corresponding argument is used.

If the syntax of a conversion specification is invalid, the behavior is undefined. If there aren’t enough function arguments provided to supply addresses for all the conversion specifications in the template strings that perform assignments, or if the arguments are not of the correct types, the behavior is also undefined. On the other hand, extra arguments are simply ignored.

15.2.14 Numeric Input Conversions

This section describes the `scanf` conversions for reading numeric values.

The `‘%d’` conversion matches an optionally signed integer in decimal radix.

The ‘%i’ conversion matches an optionally signed integer in any of the formats that the C language defines for specifying an integer constant.

For example, any of the strings ‘10’, ‘0xa’, or ‘012’ could be read in as integers under the ‘%i’ conversion. Each of these specifies a number with decimal value 10.

The ‘%o’, ‘%u’, and ‘%x’ conversions match unsigned integers in octal, decimal, and hexadecimal radices, respectively.

The ‘%X’ conversion is identical to the ‘%x’ conversion. They both permit either uppercase or lowercase letters to be used as digits.

Unlike the C language `scanf`, Octave ignores the ‘h’, ‘l’, and ‘L’ modifiers.

15.2.15 String Input Conversions

This section describes the `scanf` input conversions for reading string and character values: ‘%s’ and ‘%c’.

The ‘%c’ conversion is the simplest: it matches a fixed number of characters, always. The maximum field width says how many characters to read; if you don’t specify the maximum, the default is 1. This conversion does not skip over initial whitespace characters. It reads precisely the next n characters, and fails if it cannot get that many.

The ‘%s’ conversion matches a string of non-whitespace characters. It skips and discards initial whitespace, but stops when it encounters more whitespace after having read something.

For example, reading the input:

```
hello, world
```

with the conversion ‘%10c’ produces "hello, wo", but reading the same input with the conversion ‘%10s’ produces "hello,".

15.2.16 Binary I/O

Octave can read and write binary data using the functions `fread` and `fwrite`, which are patterned after the standard C functions with the same names. They are able to automatically swap the byte order of integer data and convert among the supported floating point formats as the data are read.

`[val, count] = fread(fid, size, precision, skip, arch)` [Built-in Function]

Read binary data of type *precision* from the specified file ID *fid*.

The optional argument *size* specifies the amount of data to read and may be one of

Inf Read as much as possible, returning a column vector.

nr Read up to *nr* elements, returning a column vector.

[nr, Inf] Read as much as possible, returning a matrix with *nr* rows. If the number of elements read is not an exact multiple of *nr*, the last column is padded with zeros.

[nr, nc] Read up to $nr * nc$ elements, returning a matrix with *nr* rows. If the number of elements read is not an exact multiple of *nr*, the last column is padded with zeros.

If *size* is omitted, a value of `Inf` is assumed.

The optional argument *precision* is a string specifying the type of data to read and may be one of

```
"schar"
"signed char"
    Signed character.

"uchar"
"unsigned char"
    Unsigned character.

"int8"
"integer*1"
    8-bit signed integer.

"int16"
"integer*2"
    16-bit signed integer.

"int32"
"integer*4"
    32-bit signed integer.

"int64"
"integer*8"
    64-bit signed integer.

"uint8"   8-bit unsigned integer.
"uint16"  16-bit unsigned integer.
"uint32"  32-bit unsigned integer.
"uint64"  64-bit unsigned integer.

"single"
"float32"
"real*4"  32-bit floating point number.

"double"
"float64"
"real*8"  64-bit floating point number.

"char"
"char*1"  Single character.

"short"   Short integer (size is platform dependent).
"int"     Integer (size is platform dependent).
"long"    Long integer (size is platform dependent).

"ushort"
"unsigned short"
    Unsigned short integer (size is platform dependent).
```

"uint"

"unsigned int"

Unsigned integer (size is platform dependent).

"ulong"

"unsigned long"

Unsigned long integer (size is platform dependent).

"float" Single precision floating point number (size is platform dependent).

The default precision is "uchar".

The *precision* argument may also specify an optional repeat count. For example, '32*single' causes `fread` to read a block of 32 single precision floating point numbers. Reading in blocks is useful in combination with the *skip* argument.

The *precision* argument may also specify a type conversion. For example, 'int16=>int32' causes `fread` to read 16-bit integer values and return an array of 32-bit integer values. By default, `fread` returns a double precision array. The special form '*TYPE' is shorthand for 'TYPE=>TYPE'.

The conversion and repeat counts may be combined. For example, '32*single=>single' causes `fread` to read blocks of single precision floating point values and return an array of single precision values instead of the default array of double precision values.

The optional argument *skip* specifies the number of bytes to skip after each element (or block of elements) is read. If it is not specified, a value of 0 is assumed. If the final block read is not complete, the final skip is omitted. For example,

```
fread (f, 10, "3*single=>single", 8)
```

will omit the final 8-byte skip because the last read will not be a complete block of 3 values.

The optional argument *arch* is a string specifying the data format for the file. Valid values are

"native" The format of the current machine.

"ieee-be"

IEEE big endian.

"ieee-le"

IEEE little endian.

"vaxd" VAX D floating format.

"vaxg" VAX G floating format.

"cray" Cray floating format.

Conversions are currently only supported for "ieee-be" and "ieee-le" formats.

The data read from the file is returned in *val*, and the number of values read is returned in *count*

See also: `fwrite`, `fopen`, `fclose`.

`count = fwrite (fid, data, precision, skip, arch)` [Built-in Function]

Write data in binary form of type *precision* to the specified file ID *fid*, returning the number of values successfully written to the file.

The argument *data* is a matrix of values that are to be written to the file. The values are extracted in column-major order.

The remaining arguments *precision*, *skip*, and *arch* are optional, and are interpreted as described for `fread`.

The behavior of `fwrite` is undefined if the values in *data* are too large to fit in the specified precision.

See also: `fread`, `fopen`, `fclose`.

15.2.17 Temporary Files

Sometimes one needs to write data to a file that is only temporary. This is most commonly used when an external program launched from within Octave needs to access data. When Octave exits all temporary files will be deleted, so this step need not be executed manually.

`[fid, name, msg] = mkstemp (template, delete)` [Built-in Function]

Return the file ID corresponding to a new temporary file with a unique name created from *template*. The last six characters of *template* must be `XXXXXX` and these are replaced with a string that makes the filename unique. The file is then created with mode read/write and permissions that are system dependent (on GNU/Linux systems, the permissions will be 0600 for versions of glibc 2.0.7 and later). The file is opened with the `O_EXCL` flag.

If the optional argument *delete* is supplied and is true, the file will be deleted automatically when Octave exits, or when the function `purge_tmp_files` is called.

If successful, *fid* is a valid file ID, *name* is the name of the file, and *msg* is an empty string. Otherwise, *fid* is -1, *name* is empty, and *msg* contains a system-dependent error message.

See also: `tmpfile`, `tmpnam`, `P_tmpdir`.

`[fid, msg] = tmpfile ()` [Built-in Function]

Return the file ID corresponding to a new temporary file with a unique name. The file is opened in binary read/write ("`w+b`") mode. The file will be deleted automatically when it is closed or when Octave exits.

If successful, *fid* is a valid file ID and *msg* is an empty string. Otherwise, *fid* is -1 and *msg* contains a system-dependent error message.

See also: `tmpnam`, `mkstemp`, `P_tmpdir`.

`tmpnam (dir, prefix)` [Built-in Function]

Return a unique temporary file name as a string.

If *prefix* is omitted, a value of `"oct-"` is used. If *dir* is also omitted, the default directory for temporary files is used. If *dir* is provided, it must exist, otherwise the default directory for temporary files is used. Since the named file is not opened, by

`tmpnam`, it is possible (though relatively unlikely) that it will not be available by the time your program attempts to open it.

See also: `tmpfile`, `mkstemp`, `P_tmpdir`.

15.2.18 End of File and Errors

Once a file has been opened its status can be acquired. As an example the `feof` functions determines if the end of the file has been reached. This can be very useful when reading small parts of a file at a time. The following example shows how to read one line at a time from a file until the end has been reached.

```
filename = "myfile.txt";
fid = fopen (filename, "r");
while (! feof (fid) )
    text_line = fgetl (fid);
endwhile
fclose (fid);
```

Note that in some situations it is more efficient to read the entire contents of a file and then process it, than it is to read it line by line. This has the potential advantage of removing the loop in the above code.

feof (*fid*) [Built-in Function]

Return 1 if an end-of-file condition has been encountered for a given file and 0 otherwise. Note that it will only return 1 if the end of the file has already been encountered, not if the next read operation will result in an end-of-file condition.

See also: `fread`, `fopen`, `fclose`.

ferror (*fid*) [Built-in Function]

Return 1 if an error condition has been encountered for a given file and 0 otherwise. Note that it will only return 1 if an error has already been encountered, not if the next operation will result in an error condition.

freport () [Built-in Function]

Print a list of which files have been opened, and whether they are open for reading, writing, or both. For example,

```
freport ()

+  number  mode  name
+
+      0    r  stdin
+      1    w  stdout
+      2    w  stderr
+      3    r  myfile
```

15.2.19 File Positioning

Three functions are available for setting and determining the position of the file pointer for a given file.

ftell (*fid*) [Built-in Function]
Return the position of the file pointer as the number of characters from the beginning of the file *fid*.

See also: fseek, fopen, fclose.

fseek (*fid*, *offset*, *origin*) [Built-in Function]
Set the file pointer to any location within the file *fid*.
The pointer is positioned *offset* characters from the *origin*, which may be one of the predefined variables SEEK_CUR (current position), SEEK_SET (beginning), or SEEK_END (end of file) or strings "cof", "bof" or "eof". If *origin* is omitted, SEEK_SET is assumed. The offset must be zero, or a value returned by **ftell** (in which case *origin* must be SEEK_SET).

Return 0 on success and -1 on error.

See also: ftell, fopen, fclose.

SEEK_SET () [Built-in Function]
SEEK_CUR () [Built-in Function]
SEEK_END () [Built-in Function]

Return the value required to request that **fseek** perform one of the following actions:

SEEK_SET Position file relative to the beginning.

SEEK_CUR Position file relative to the current position.

SEEK_END Position file relative to the end.

frewind (*fid*) [Built-in Function]
Move the file pointer to the beginning of the file *fid*, returning 0 for success, and -1 if an error was encountered. It is equivalent to **fseek** (*fid*, 0, SEEK_SET).

The following example stores the current file position in the variable **marker**, moves the pointer to the beginning of the file, reads four characters, and then returns to the original position.

```
marker = ftell (myfile);  
frewind (myfile);  
fourch = fgets (myfile, 4);  
fseek (myfile, marker, SEEK_SET);
```

16 Plotting

16.1 Two-Dimensional Plotting

The basic plotting commands are:

<code>axes ()</code>	[Function File]
<code>axes (property, value, ...)</code>	[Function File]
<code>axes (h)</code>	[Function File]

Create an axes object and return a handle to it.

<code>plot (args)</code>	[Function File]
--------------------------	-----------------

This function produces two-dimensional plots. Many different combinations of arguments are possible. The simplest form is

`plot (y)`

where the argument is taken as the set of *y* coordinates and the *x* coordinates are taken to be the indices of the elements, starting with 1.

To save a plot, in one of several image formats such as PostScript or PNG, use the `print` command.

If more than one argument is given, they are interpreted as

`plot (x, y, fmt, ...)`

or as

`plot (x, y, property, value, ...)`

where *y*, *fmt*, *property* and *value* are optional, and any number of argument sets may appear. The *x* and *y* values are interpreted as follows:

- If a single data argument is supplied, it is taken as the set of *y* coordinates and the *x* coordinates are taken to be the indices of the elements, starting with 1.
- If the *x* is a vector and *y* is a matrix, then the columns (or rows) of *y* are plotted versus *x*. (using whichever combination matches, with columns tried first.)
- If the *x* is a matrix and *y* is a vector, *y* is plotted versus the columns (or rows) of *x*. (using whichever combination matches, with columns tried first.)
- If both arguments are vectors, the elements of *y* are plotted versus the elements of *x*.
- If both arguments are matrices, the columns of *y* are plotted versus the columns of *x*. In this case, both matrices must have the same number of rows and columns and no attempt is made to transpose the arguments to make the number of rows match.

If both arguments are scalars, a single point is plotted.

If the *fmt* argument is supplied, it is interpreted as follows. If *fmt* is missing, the default gnuplot line style is assumed.

<code>'-'</code>	Set lines plot style (default).
<code>'.'</code>	Set dots plot style.

'^'	Set impulses plot style.
'L'	Set steps plot style.
'n'	Interpreted as the plot color if <i>n</i> is an integer in the range 1 to 6.
'nm'	If <i>nm</i> is a two digit integer and <i>m</i> is an integer in the range 1 to 6, <i>m</i> is interpreted as the point style. This is only valid in combination with the @ or -@ specifiers.
'c'	If <i>c</i> is one of "k" (black), "r" (red), "g" (green), "b" (blue), "m" (magenta), "c" (cyan), or "w" (white), it is interpreted as the line plot color.
" ; title ; "	Here "title" is the label for the key.
'+'	
'*'	
'o'	
'x'	Used in combination with the points or linespoints styles, set the point style.

The color line styles have the following meanings on terminals that support color.

Number	Gnuplot colors	(lines)points style
1	red	*
2	green	+
3	blue	o
4	magenta	x
5	cyan	house
6	brown	there exists

The *fnt* argument can also be used to assign key titles. To do so, include the desired title between semi-colons after the formatting sequence described above, e.g. "+3;Key Title;" Note that the last semi-colon is required and will generate an error if it is left out.

If a *property* is given it must be followed by *value*. The property value pairs are applied to the lines drawn by `plot`.

Here are some plot examples:

```
plot (x, y, "@12", x, y2, x, y3, "4", x, y4, "+")
```

This command will plot *y* with points of type 2 (displayed as '+') and color 1 (red), *y2* with lines, *y3* with lines of color 4 (magenta) and *y4* with points displayed as '+'.

```
plot (b, "*", "markersize", 3)
```

This command will plot the data in the variable *b*, with points displayed as '*' with a marker size of 3.

```
t = 0:0.1:6.3;
plot (t, cos(t), "-;cos(t);", t, sin(t), "+3;sin(t);");
```

This will plot the cosine and sine functions and label them accordingly in the key.

See also: `semilogx`, `semilogy`, `loglog`, `polar`, `mesh`, `contour`, `--pltopt--` `bar`, `stairs`, `errorbar`, `xlabel`, `ylabel`, `title`, `print`.

`line ()` [Function File]
`line (x, y)` [Function File]
`line (x, y, z)` [Function File]
`line (x, y, z, opts)` [Function File]

Create line object from *x* and *y* and insert in current axes object. Return handle to line object.

`fplot (fn, limits)` [Function File]
`fplot (fn, limits, tol)` [Function File]
`fplot (fn, limits, n)` [Function File]
`fplot (... , LineSpec)` [Function File]

Plot a function *fn*, within the defined limits. *fn* can be either a string, a function handle or an inline function. The limits of the plot are given by *limits* of the form `[xlo, xhi]` or `[xlo, xhi, ylo, yhi]`. *tol* is the default tolerance to use for the plot, and if *tol* is an integer it is assumed that it defines the number points to use in the plot. The *LineSpec* is passed to the plot command.

```
fplot ("cos", [0, 2*pi])
fplot ("[cos(x), sin(x)]", [0, 2*pi])
```

See also: plot.

`drawnow ()` [Function File]
 Display the current graphics.

`shg` [Function File]
 Show the graph window. Currently, this is the same as executing `drawnow`.

See also: plot, semilogx, semilogy, loglog, polar, mesh, contour, bar, stairs, xlabel, ylabel.

`hold args` [Function File]
 Tell Octave to ‘hold’ the current data on the plot when executing subsequent plotting commands. This allows you to execute a series of plot commands and have all the lines end up on the same figure. The default is for each new plot command to clear the plot device first. For example, the command

```
hold on
```

turns the hold state on. An argument of `"off"` turns the hold state off, and `hold` with no arguments toggles the current hold state.

`ishold` [Function File]
 Return 1 if the next line will be added to the current plot, or 0 if the plot device will be cleared before drawing the next line.

`newplot ()` [Function File]

16.2 Specialized Two-Dimensional Plots

`h = bar (x, y, style)` [Function File]

`[xb, yb] = bar (...)` [Function File]

Given two vectors of x-y data, `bar` produces a bar graph.

If only one argument is given, it is taken as a vector of y-values and the x coordinates are taken to be the indices of the elements.

If y is a matrix, then each column of y is taken to be a separate bar graph plotted on the same graph. By default the columns are plotted side-by-side. This behavior can be changed by the *style* argument, which can take the values 'group' (the default), or 'stack'.

If two output arguments are specified, the data are generated but not plotted. For example,

```
bar (x, y);
```

and

```
[xb, yb] = bar (x, y);
plot (xb, yb);
```

are equivalent.

See also: `hbar`, `plot`, `semilogx`, `semilogy`, `loglog`, `polar`, `mesh`, `contour`, `stairs`, `xlabel`, `ylabel`, `title`.

`c = contour (z)` [Function File]

`c = contour (z, vn)` [Function File]

`c = contour (x, y, z)` [Function File]

`c = contour (x, y, z, vn)` [Function File]

Plot level curves (contour lines) of the matrix *z*, using the contour matrix *c* computed by `contourc` from the same arguments; see the latter for their interpretation. *c* is only returned if requested. For example:

```
x = 0:2;
y = x;
z = x' * y;
contour (x, y, z, 2:3)
```

See also: `contourc`, `line`, `plot`.

`[c, lev] = contourc (x, y, z, vn)` [Function File]

Compute isolines (countour lines) of the matrix *z*. Parameters *x*, *y* and *vn* are optional.

The return value *lev* is a vector of the contour levels. The return value *c* is a 2 by *n* matrix containing the contour lines in the following format

```
c = [lev1, x1, x2, ..., levn, x1, x2, ...
     len1, y1, y2, ..., lenn, y1, y2, ...]
```

in which contour line *n* has a level (height) of *levn* and length of *lenn*.

If x and y are omitted they are taken as the row/column index of z . vn is either a scalar denoting the number of lines to compute or a vector containing the values of the lines. If only one value is wanted, set $vn = [val, val]$; If vn is omitted it defaults to 10.

For example,

```
x = 0:2;
y = x;
z = x' * y;
contourc (x, y, z, 2:3)
⇒   2.0000   2.0000   1.0000   3.0000   1.5000   2.0000
     2.0000   1.0000   2.0000   2.0000   2.0000   1.5000
```

See also: `contour`.

hist ($y, x, norm$) [Function File]

Produce histogram counts or plots.

With one vector input argument, plot a histogram of the values with 10 bins. The range of the histogram bins is determined by the range of the data.

Given a second scalar argument, use that as the number of bins.

Given a second vector argument, use that as the centers of the bins, with the width of the bins determined from the adjacent values in the vector.

If third argument is provided, the histogram is normalised such that the sum of the bars is equal to *norm*.

Extreme values are lumped in the first and last bins.

With two output arguments, produce the values *nn* and *xx* such that `bar (xx, nn)` will plot the histogram.

See also: `bar`.

loglog ($args$) [Function File]

Make a two-dimensional plot using log scales for both axes. See the description of `plot` for a description of the arguments that `loglog` will accept.

See also: `plot`, `semilogy`, `loglog`, `polar`, `mesh`, `contour`, `bar`, `stairs`, `xlabel`, `ylabel`, `title`.

polar ($theta, rho, fmt$) [Function File]

Make a two-dimensional plot given polar the coordinates *theta* and *rho*.

The optional third argument specifies the line type.

See also: `plot`, `semilogx`, `semilogy`, `loglog`, `mesh`, `contour`, `bar`, `stairs`, `xlabel`, `ylabel`, `title`.

semilogx ($args$) [Function File]

Make a two-dimensional plot using a log scale for the x axis. See the description of `plot` for a description of the arguments that `semilogx` will accept.

See also: `plot`, `semilogy`, `loglog`, `polar`, `mesh`, `contour`, `bar`, `stairs`, `xlabel`, `ylabel`, `title`.

semilogy (*args*) [Function File]

Make a two-dimensional plot using a log scale for the *y* axis. See the description of **plot** for a description of the arguments that **semilogy** will accept.

See also: **plot**, **semilogx**, **loglog**, **polar**, **mesh**, **contour**, **bar**, **stairs**, **xlabel**, **ylabel**, **title**.

h = stem (*x*, *y*, *linespec*) [Function File]

Plot a stem graph and return the handles of the line and marker objects used to draw the stems. The default color is "r" (red). The default line style is "-" and the default marker is "o".

For example,

```
x = 1:10;
stem (x);
```

plots 10 stems with heights from 1 to 10;

```
x = 1:10;
y = ones (1, length (x))*2.*x;
stem (x, y);
```

plots 10 stems with heights from 2 to 20;

```
x = 1:10;
y = ones (size (x))*2.*x;
h = stem (x, y, "b");
```

plots 10 bars with heights from 2 to 20 (the color is blue, and *h* is a 2-by-10 array of handles in which the first row holds the line handles and the the second row holds the marker handles);

```
x = 1:10;
y = ones (size (x))*2.*x;
h = stem (x, y, "-.k");
```

plots 10 stems with heights from 2 to 20 (the color is black, line style is "-.", and *h* is a 2-by-10 array of handles in which the first row holds the line handles and the second rows holds the marker handles);

```
x = 1:10;
y = ones (size (x))*2.*x;
h = stem (x, y, "-.k.");
```

plots 10 stems with heights from 2 to 20 (the color is black, line style is "-." and the marker style is ".", and *h* is a 2-by-10 array of handles in which the first row holds the line handles and the second row holds the marker handles);

```
x = 1:10;
y = ones (size (x))*2.*x;
h = stem (x, y, "fill");
```

plots 10 stems with heights from 2 to 20 (the color is rgb-triple defined, the line style is "-", the marker style is "o", and *h* is a 2-by-10 array of handles in which the first row holds the line handles and the second row holds the marker handles).

Color definitions with rgb-triples are not valid!

See also: **bar**, **barh**, **plot**.

stairs (*x*, *y*) [Function File]

Given two vectors of x-y data, `stairs` produces a ‘stairstep’ plot.

If only one argument is given, it is taken as a vector of y-values and the x coordinates are taken to be the indices of the elements.

If two output arguments are specified, the data are generated but not plotted. For example,

```
stairs (x, y);
```

and

```
[xs, ys] = stairs (x, y);
plot (xs, ys);
```

are equivalent.

See also: `plot`, `semilogx`, `semilogy`, `loglog`, `polar`, `mesh`, `contour`, `bar`, `xlabel`, `ylabel`, `title`.

errorbar (*args*) [Function File]

This function produces two-dimensional plots with errorbars. Many different combinations of arguments are possible. The simplest form is

```
errorbar (y, ey)
```

where the first argument is taken as the set of *y* coordinates and the second argument *ey* is taken as the errors of the *y* values. *x* coordinates are taken to be the indices of the elements, starting with 1.

If more than two arguments are given, they are interpreted as

```
errorbar (x, y, ..., fmt, ...)
```

where after *x* and *y* there can be up to four error parameters such as *ey*, *ex*, *ly*, *uy* etc., depending on the plot type. Any number of argument sets may appear, as long as they are separated with a format string *fmt*.

If *y* is a matrix, *x* and error parameters must also be matrices having same dimensions. The columns of *y* are plotted versus the corresponding columns of *x* and errorbars are drawn from the corresponding columns of error parameters.

If *fmt* is missing, `yerrorbars` (“~”) plot style is assumed. If the *fmt* argument is supplied, it is interpreted as in normal plots (See `--pltopt--`). In addition the following plot styles are supported by `errorbar`:

‘~’	Set <code>yerrorbars</code> plot style (default).
‘>’	Set <code>xerrorbars</code> plot style.
‘~>’	Set <code>xyerrorbars</code> plot style.
‘#’	Set <code>boxes</code> plot style.
‘#~’	Set <code>boxerrorbars</code> plot style.
‘#~>’	Set <code>boxxyerrorbars</code> plot style.

Examples:

```
errorbar(x, y, ex, ">")
```

xerrorbar plot of y versus x with x errorbars drawn from $x-ex$ to $x+ex$.

```
errorbar(x, y1, ey, "~", x, y2, ly, uy)
```

Two yerrorbar plots with $y1$ and $y2$ versus x . Errorbars for $y1$ are drawn from $y1-ey$ to $y1+ey$, errorbars for $y2$ from $y2-ly$ to $y2+uy$.

```
errorbar(x, y, lx, ux, ly, uy, "~>")
```

xyerrorbar plot of y versus x where x errorbars are drawn from $x-lx$ to $x+ux$ and y errorbars from $y-ly$ to $y+uy$.

See also: semilogx, semilogy, loglog, polar, mesh, contour, --pltopt--, bar, stairs, xlabel, ylabel, title.

loglogerr (*args*) [Function File]

This function produces two-dimensional plots on double logarithm axis with errorbars. Many different combinations of arguments are possible. The most used form is

```
loglogerr (x, y, ey, fmt)
```

which produces a double logarithm plot of y versus x with errors in the y -scale defined by ey and the plot format defined by fmt . See errorbar for available formats and additional information.

See also: errorbar, semilogxerr, semilogyerr, polar, mesh, contour, --pltopt--, bar, stairs, xlabel, ylabel, title.

semilogxerr (*args*) [Function File]

This function produces two-dimensional plots on a semilogarithm axis with errorbars. Many different combinations of arguments are possible. The most used form is

```
semilogxerr (x, y, ey, fmt)
```

which produces a semi-logarithm plot of y versus x with errors in the y -scale defined by ey and the plot format defined by fmt . See errorbar for available formats and additional information.

See also: errorbar, loglogerr, semilogyerr, polar, mesh, contour, --pltopt--, bar, stairs, xlabel, ylabel, title.

semilogyerr (*args*) [Function File]

This function produces two-dimensional plots on a semilogarithm axis with errorbars. Many different combinations of arguments are possible. The most used form is

```
semilogyerr (x, y, ey, fmt)
```

which produces a semi-logarithm plot of y versus x with errors in the y -scale defined by ey and the plot format defined by fmt . See errorbar for available formats and additional information.

See also: errorbar, loglogerr, semilogxerr, polar, mesh, contour, --pltopt--, bar, stairs, xlabel, ylabel, title.

16.3 Three-Dimensional Plotting

plot3 (*args*) [Function File]

This function produces three-dimensional plots. Many different combinations of arguments are possible. The simplest form is

```
plot3 (x, y, z)
```

where the arguments are taken to be the vertices of the points to be plotted in three dimensions. If all arguments are vectors of the same length, then a single continuous line is drawn. If all arguments are matrices, then each column of the matrices is treated as a separate line. No attempt is made to transpose the arguments to make the number of rows match.

Additionally, only two arguments can be given as

```
plot3 (x, c)
```

where the real and imaginary parts of the second argument are used as the *y* and *z* coordinates, respectively.

If only one argument is given, as

```
plot3 (c)
```

the real and imaginary parts of the argument are used as the *y* and *z* values, and they are plotted versus their index.

To save a plot, in one of several image formats such as PostScript or PNG, use the **print** command.

See `--pltopt--` for a description of the optional format argument.

Arguments can also be given in groups of three as

```
plot3 (x1, y1, z1, x2, y2, z2, ...)
```

where each set of three arguments is treated as a separate line or set of lines in three dimensions.

To plot multiple one- or two-argument groups, separate each group with an empty format string, as

```
plot3 (x1, c1, '', c2, '', ...)
```

An example of the use of `plot3` is

```
z = [0:0.05:5];
plot3(cos(2*pi*z), sin(2*pi*z), z, ";helix;");
plot3(z, exp(2i*pi*z), ";complex sinusoid;");
```

See also: `plot`, `semilogx`, `semilogy`, `loglog`, `polar`, `mesh`, `contour`, `--pltopt--` `bar`, `stairs`, `errorbar`, `xlabel`, `ylabel`, `title`, `print`.

mesh (*x*, *y*, *z*) [Function File]

Plot a mesh given matrices *x*, and *y* from `meshgrid` and a matrix *z* corresponding to the *x* and *y* coordinates of the mesh. If *x* and *y* are vectors, then a typical vertex is (*x*(*j*), *y*(*i*), *z*(*i,j*)). Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

See also: `meshgrid`, `contour`.

meshc (*x*, *y*, *z*) [Function File]

Plot a mesh and contour given matrices *x*, and *y* from **meshgrid** and a matrix *z* corresponding to the *x* and *y* coordinates of the mesh. If *x* and *y* are vectors, then a typical vertex is (*x*(*j*), *y*(*i*), *z*(*i,j*)). Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

See also: **meshgrid**, **mesh**, **contour**.

[*xx*, *yy*, *zz*] = **meshgrid** (*x*, *y*, *z*) [Function File]

[*xx*, *yy*] = **meshgrid** (*x*, *y*) [Function File]

[*xx*, *yy*] = **meshgrid** (*x*) [Function File]

Given vectors of *x* and *y* and *z* coordinates, and returning 3 arguments, return three dimensional arrays corresponding to the *x*, *y*, and *z* coordinates of a mesh. When returning only 2 arguments, return matrices corresponding to the *x* and *y* coordinates of a mesh. The rows of *xx* are copies of *x*, and the columns of *yy* are copies of *y*. If *y* is omitted, then it is assumed to be the same as *x*, and *z* is assumed the same as *y*.

See also: **mesh**, **contour**.

[*y1*, *y2*, ..., *yn*] = **ndgrid** (*x1*, *x2*, ..., *xn*) [Function File]

[*y1*, *y2*, ..., *yn*] = **ndgrid** (*x*) [Function File]

Given *n* vectors *x1*, ... *xn*, **ndgrid** returns *n* arrays of dimension *n*. The elements of the *i*th output argument contains the elements of the vector *xi* repeated over all dimensions different from the *i*th dimension. Calling **ndgrid** with only one input argument *x* is equivalent of calling **ndgrid** with all *n* input arguments equal to *x*:

[*y1*, *y2*, ..., *yn*] = **ndgrid** (*x*, ..., *x*)

See also: **meshgrid**.

view (*azimuth*, *elevation*) [Function File]

view (*dims*) [Function File]

[*azimuth*, *elevation*] = **view** () [Function File]

Set or get the viewpoint for the current axes.

16.4 Manipulating Existing Plots

axis (*limits*) [Function File]

Set axis limits for plots.

The argument *limits* should be a 2, 4, or 6 element vector. The first and second elements specify the lower and upper limits for the *x* axis. The third and fourth specify the limits for the *y* axis, and the fifth and sixth specify the limits for the *z* axis.

Without any arguments, **axis** turns autoscaling on.

With one output argument, **x=axis** returns the current axes (this is not yet implemented for automatic axes).

The vector argument specifying limits is optional, and additional string arguments may be used to specify various axis properties. For example,


```
axis ([1, 2, 3, 4], "square");
```

forces a square aspect ratio, and

```
axis ("labely", "tic");
```

turns tic marks on for all axes and tic mark labels on for the y-axis only.

The following options control the aspect ratio of the axes.

"square" Force a square aspect ratio.

"equal" Force x distance to equal y-distance.

"normal" Restore the balance.

The following options control the way axis limits are interpreted.

"auto" Set the specified axes to have nice limits around the data or all if no axes are specified.

"manual" Fix the current axes limits.

"tight" Fix axes to the limits of the data (not implemented).

The option "image" is equivalent to "tight" and "equal".

The following options affect the appearance of tic marks.

"on" Turn tic marks and labels on for all axes.

"off" Turn tic marks off for all axes.

"tic[xyz]" Turn tic marks on for all axes, or turn them on for the specified axes and off for the remainder.

"label[xyz]" Turn tic labels on for all axes, or turn them on for the specified axes and off for the remainder.

"nolabel" Turn tic labels off for all axes.

Note, if there are no tic marks for an axis, there can be no labels.

The following options affect the direction of increasing values on the axes.

"ij" Reverse y-axis, so lower values are nearer the top.

"xy" Restore y-axis, so higher values are nearer the top.

gca () [Function File]
Return the current axis handle.

gcf () [Function File]
Return the current figure handle.

get (h, p) [Built-in Function]
Return the named property *p* from the graphics handle *h*. If *p* is omitted, return the complete property list for *h*. If *h* is a vector, return a cell array including the property values or lists respectively.

`set (h, p, v, ...)` [Built-in Function]
 Set the named property value or vector *p* to the value *v* in the graphics handle *h*.

`clf ()` [Function File]
 Clear the current figure.

`delete (file)` [Function File]

`delete (h)` [Function File]
 Delete the named file or figure handle.

`close` [Command]

`close (n)` [Command]

`close all` [Command]

`close all hidden` [Command]
 Close the plot window(s).

`closereq ()` [Function File]
 Close the current figure.

`isfigure (h)` [Function File]
 Return 1 if *h* is a graphics handle that contains a figure object and 0 otherwise.

16.5 Plot Annotations

`title (title)` [Function File]
 Create a title object and return a handle to it.

`legend (st1, st2, ...)` [Function File]

`legend (st1, st2, ..., pos)` [Function File]

`legend (matstr)` [Function File]

`legend (matstr, pos)` [Function File]

`legend (cell)` [Function File]

`legend (cell, pos)` [Function File]

`legend ('func')` [Function File]

Legend puts a legend on the current plot using the specified strings as labels. Use independent strings (*st1*, *st2*, ...), a matrix of strings (*matstr*), or a cell array of strings (*cell*) to specify legends. Legend works on line graphs, bar graphs, etc. Be sure to call `plot` before calling `legend`.

pos optionally places the legend in the specified location:

- 1 To the top right of the plot
- 0 Don't move the legend box (default)
- 1 Upper right-hand corner
- 2 Upper left-hand corner
- 3 Lower left-hand corner
- 4 Lower right-hand corner

Some specific functions are directly available using *func*:

`show` Show legends from the plot

`hide`

`off` Hide legends from the plot

`boxon` Draw a box around legends

`boxoff` Withdraw the box around legends

`left` Text is to the left of the keys

`right` Text is to the right of the keys

`h = text (x, y, label)` [Function File]

`h = text (x, y, z, label)` [Function File]

Create a text object with text *label* at position *x*, *y*, *z* on the current axes. The label may be followed by property-value pairs.

`xlabel (string)` [Function File]

`ylabel (string)` [Function File]

`zlabel (string)` [Function File]

Specify *x*, *y*, and *z* axis labels for the plot.

See also: `plot`, `semilogx`, `semilogy`, `loglog`, `polar`, `mesh`, `contour`, `bar`, `stairs`, `ylabel`, `title`.

`box (arg)` [Function File]

`grid (h, ...)` [Function File]

Control the display of a border around the plot. The argument may be either **"on"** or **"off"**. If it is omitted, the current box state is toggled.

See also: `grid`.

`grid (arg)` [Function File]

`grid ("minor", arg2)` [Function File]

Force the display of a grid on the plot. The argument may be either **"on"** or **"off"**. If it is omitted, the current grid state is toggled.

If *arg* is **"minor"** then the minor grid is toggled. When using a minor grid a second argument *arg2* is allowed, which can be either **"on"** or **"off"** to explicitly set the state of the minor grid.

See also: `plot`, `semilogx`, `semilogy`, `loglog`, `polar`, `mesh`, `contour`, `bar`, `stairs`, `xlabel`, `ylabel`, `title`.

16.6 Multiple Plots on One Page

`subplot (rows, cols, index)` [Function File]

`subplot (rcn)` [Function File]

Set up a plot grid with *cols* by *rows* subwindows and plot in location given by *index*.

If only one argument is supplied, then it must be a three digit value specifying the location in digits 1 (rows) and 2 (columns) and the plot index in digit 3.

The plot index runs row-wise. First all the columns in a row are filled and then the next row is filled.

For example, a plot with 2 by 3 grid will have plot indices running as follows:

1	2	3	4
5	6	7	8

See also: `plot`.

16.7 Multiple Plot Windows

`figure (n, p, v, ...)` [Function File]

Set the current plot window to plot window *n*. If no arguments are specified, the next available window number is chosen. Parameters *p* and settings *v* can be given in pairs if *n* is specified.

16.8 Printing Plots

`print (filename, options)` [Function File]

Print a graph, or save it to a file

filename defines the file name of the output file. If no filename is specified, output is sent to the printer.

options:

`-Pprinter`

Set the *printer* name to which the graph is sent if no *filename* is specified.

`-color`

`-mono` Monochrome or colour lines.

`-solid`

`-dashed` Solid or dashed lines.

`-portrait`

`-landscape`

Plot orientation, as returned by "orient".

`-ddevice` Output device, where *device* is one of:

`ps`

`ps2`

`psc`

`psc2` Postscript (level 1 and 2, mono and color)

`eps`

`eps2`

`epsc`

`epsc2` Encapsulated postscript (level 1 and 2, mono and color)

```

tex
epslatex
epslatexstandalone
pstex
pslatex  Generate a LaTeX (or TeX) file for labels, and eps/ps for
          graphics. The file produced by epslatexstandalone can be
          processed directly by LaTeX. The other formats are intended
          to be included in a LaTeX (or TeX) document. The tex
          device is the same as the epslatex device.

ill
aifm      Adobe Illustrator

cdr
corel     CorelDraw

hpgl      HP plotter language

fig       XFig

dxf       AutoCAD

mf        Metafont

png       Portable network graphics

pbm       PBMplus

emf       Microsoft Enhanced Metafile

```

Other devices are supported by "convert" from ImageMagick. Type `system("convert")` to see what formats are available.

If the device is omitted, it is inferred from the file extension, or if there is no filename it is sent to the printer as postscript.

```

-Ffontname
-Ffontname:size
-F:size  fontname set the postscript font (for use with postscript, aifm, corel and
          fig). By default, 'Helvetica' is set for PS/Aifm, and 'SwitzerlandLight' for
          Corel. It can also be 'Times-Roman'. size is given in points. fontname is
          ignored for the fig device.

```

The filename and options can be given in any order.

orient (*orientation*) [Function File]
 Set default print orientation. Valid values for *orientation* include "landscape" and "portrait". If called with no arguments, return the default print orientation.

16.9 Test Plotting Functions

```

peaks () [Function File]
peaks (n) [Function File]
peaks (x, y) [Function File]

```

`z = peaks (...)` [Function File]

`[x, y, z] = peaks (...)` [Function File]

Generate a function with lots of local maxima and minima. The function has the form

$$f(x, y) = 3(1 - x)^2 e^{(-x^2 - (y+1)^2)} - 10\left(\frac{x}{5} - x^3 - y^5\right) - \frac{1}{3}e^{-(x+1)^2 - y^2}$$

Called without a return argument, **peaks** plots the surface of the above function using **mesh**. If *n* is a scalar, the **peaks** returns the values of the above function on a *n*-by-*n* mesh over the range $[-3, 3]$. The default value for *n* is 49.

If *n* is a vector, then it represents the *x* and *y* values of the grid on which to calculate the above function. The *x* and *y* values can be specified separately.

See also: `mesh`.

16.10 Interaction with gnuplot

`val = gnuplot_binary ()` [Loadable Function]

`old_val = gnuplot_binary (new_val)` [Loadable Function]

Query or set the name of the program invoked by the plot command. The default value "gnuplot". See [Appendix E \[Installation\]](#), page 525.

`val = gnuplot_use_title_option ()` [Loadable Function]

`old_val = gnuplot_use_title_option (new_val)` [Loadable Function]

If enabled, append '-title "Figure NN"' to the gnuplot command. By default, this feature is enabled if the DISPLAY environment variable is set when Octave starts.

17 Matrix Manipulation

There are a number of functions available for checking to see if the elements of a matrix meet some condition, and for rearranging the elements of a matrix. For example, Octave can easily tell you if all the elements of a matrix are finite, or are less than some specified value. Octave can also rotate the elements, extract the upper- or lower-triangular parts, or sort the columns of a matrix.

17.1 Finding Elements and Checking Conditions

The functions **any** and **all** are useful for determining whether any or all of the elements of a matrix satisfy some condition. The **find** function is also useful in determining which elements of a matrix meet a specified condition.

any (*x*, *dim*) [Built-in Function]

For a vector argument, return 1 if any element of the vector is nonzero.

For a matrix argument, return a row vector of ones and zeros with each element indicating whether any of the elements of the corresponding column of the matrix are nonzero. For example,

```
any (eye (2, 4))
⇒ [ 1, 1, 0, 0 ]
```

If the optional argument *dim* is supplied, work along dimension *dim*. For example,

```
any (eye (2, 4), 2)
⇒ [ 1; 1 ]
```

all (*x*, *dim*) [Built-in Function]

The function **all** behaves like the function **any**, except that it returns true only if all the elements of a vector, or all the elements along dimension *dim* of a matrix, are nonzero.

Since the comparison operators (see [Section 9.4 \[Comparison Ops\]](#), page 93) return matrices of ones and zeros, it is easy to test a matrix for many things, not just whether the elements are nonzero. For example,

```
all (all (rand (5) < 0.9))
⇒ 0
```

tests a random 5 by 5 matrix to see if all of its elements are less than 0.9.

Note that in conditional contexts (like the test clause of **if** and **while** statements) Octave treats the test as if you had typed **all (all (condition))**.

xor (*x*, *y*) [Mapping Function]

Return the ‘exclusive or’ of the entries of *x* and *y*. For boolean expressions *x* and *y*, **xor** (*x*, *y*) is true if and only if *x* or *y* is true, but not if both *x* and *y* are true.

is_duplicate_entry (*x*) [Function File]

Return non-zero if any entries in *x* are duplicates of one another.

diff (*x*, *k*, *dim*) [Function File]

If *x* is a vector of length *n*, **diff** (*x*) is the vector of first differences $x_2 - x_1, \dots, x_n - x_{n-1}$.

If *x* is a matrix, **diff** (*x*) is the matrix of column differences along the first non-singleton dimension.

The second argument is optional. If supplied, **diff** (*x*, *k*), where *k* is a nonnegative integer, returns the *k*-th differences. It is possible that *k* is larger than the first non-singleton dimension of the matrix. In this case, **diff** continues to take the differences along the next non-singleton dimension.

The dimension along which to take the difference can be explicitly stated with the optional variable *dim*. In this case the *k*-th order differences are calculated along this dimension. In the case where *k* exceeds **size** (*x*, *dim*) then an empty matrix is returned.

isinf (*x*) [Mapping Function]

Return 1 for elements of *x* that are infinite and zero otherwise. For example,

```
isinf ([13, Inf, NA, NaN])
⇒ [ 0, 1, 0, 0 ]
```

isnan (*x*) [Mapping Function]

Return 1 for elements of *x* that are NaN values and zero otherwise. NA values are also considered NaN values. For example,

```
isnan ([13, Inf, NA, NaN])
⇒ [ 0, 0, 1, 1 ]
```

finite (*x*) [Mapping Function]

Return 1 for elements of *x* that are finite values and zero otherwise. For example,

```
finite ([13, Inf, NA, NaN])
⇒ [ 1, 0, 0, 0 ]
```

find (*x*) [Loadable Function]

find (*x*, *n*) [Loadable Function]

find (*x*, *n*, *direction*) [Loadable Function]

Return a vector of indices of nonzero elements of a matrix, as a row if *x* is a row or as a column otherwise. To obtain a single index for each matrix element, Octave pretends that the columns of a matrix form one long vector (like Fortran arrays are stored). For example,

```
find (eye (2))
⇒ [ 1; 4 ]
```

If two outputs are requested, **find** returns the row and column indices of nonzero elements of a matrix. For example,

```
[i, j] = find (2 * eye (2))
⇒ i = [ 1; 2 ]
⇒ j = [ 1; 2 ]
```


If three outputs are requested, `find` also returns a vector containing the nonzero values. For example,

```
[i, j, v] = find (3 * eye (2))
⇒ i = [ 1; 2 ]
⇒ j = [ 1; 2 ]
⇒ v = [ 3; 3 ]
```

If two inputs are given, *n* indicates the number of elements to find from the beginning of the matrix or vector.

If three inputs are given, *direction* should be one of "first" or "last" indicating that it should start counting found elements from the first or last element.

`[err, y1, ...] = common_size (x1, ...)` [Function File]

Determine if all input arguments are either scalar or of common size. If so, *err* is zero, and *y1* is a matrix of the common size with all entries equal to *xi* if this is a scalar or *xi* otherwise. If the inputs cannot be brought to a common size, errorcode is 1, and *y1* is *xi*. For example,

```
[errorcode, a, b] = common_size ([1 2; 3 4], 5)
⇒ errorcode = 0
⇒ a = [ 1, 2; 3, 4 ]
⇒ b = [ 5, 5; 5, 5 ]
```

This is useful for implementing functions where arguments can either be scalars or of common size.

17.2 Rearranging Matrices

`fliplr (x)` [Function File]

Return a copy of *x* with the order of the columns reversed. For example,

```
fliplr ([1, 2; 3, 4])
⇒ 2 1
   4 3
```

Note that `fliplr` only work with 2-D arrays. To flip N-d arrays use `flipdim` instead.

See also: `flipud`, `flipdim`, `rot90`, `rotdim`.

`flipud (x)` [Function File]

Return a copy of *x* with the order of the rows reversed. For example,

```
flipud ([1, 2; 3, 4])
⇒ 3 4
   1 2
```

Due to the difficulty of defining which axis about which to flip the matrix `flipud` only work with 2-d arrays. To flip N-d arrays use `flipdim` instead.

See also: `fliplr`, `flipdim`, `rot90`, `rotdim`.

flipdim (*x*, *dim*) [Function File]

Return a copy of *x* flipped about the dimension *dim*. For example

```
flipdim ([1, 2; 3, 4], 2)
⇒  2  1
    4  3
```

See also: `fliplr`, `flipud`, `rot90`, `rotdim`.

rot90 (*x*, *n*) [Function File]

Return a copy of *x* with the elements rotated counterclockwise in 90-degree increments. The second argument is optional, and specifies how many 90-degree rotations are to be applied (the default value is 1). Negative values of *n* rotate the matrix in a clockwise direction. For example,

```
rot90 ([1, 2; 3, 4], -1)
⇒  3  1
    4  2
```

rotates the given matrix clockwise by 90 degrees. The following are all equivalent statements:

```
rot90 ([1, 2; 3, 4], -1)
rot90 ([1, 2; 3, 4], 3)
rot90 ([1, 2; 3, 4], 7)
```

Due to the difficulty of defining an axis about which to rotate the matrix `rot90` only work with 2-D arrays. To rotate N-d arrays use `rotdim` instead.

See also: `rotdim`, `flipud`, `fliplr`, `flipdim`.

rotdim (*x*, *n*, *plane*) [Function File]

Return a copy of *x* with the elements rotated counterclockwise in 90-degree increments. The second argument is optional, and specifies how many 90-degree rotations are to be applied (the default value is 1). The third argument is also optional and defines the plane of the rotation. As such *plane* is a two element vector containing two different valid dimensions of the matrix. If *plane* is not given Then the first two non-singleton dimensions are used.

Negative values of *n* rotate the matrix in a clockwise direction. For example,

```
rotdim ([1, 2; 3, 4], -1, [1, 2])
⇒  3  1
    4  2
```

rotates the given matrix clockwise by 90 degrees. The following are all equivalent statements:

```
rotdim ([1, 2; 3, 4], -1, [1, 2])
rotdim ([1, 2; 3, 4], 3, [1, 2])
rotdim ([1, 2; 3, 4], 7, [1, 2])
```

See also: `rot90`, `flipud`, `fliplr`, `flipdim`.

cat (*dim*, *array1*, *array2*, ..., *arrayN*) [Built-in Function]

Return the concatenation of N-d array objects, *array1*, *array2*, ..., *arrayN* along dimension *dim*.

```
A = ones (2, 2);
B = zeros (2, 2);
cat (2, A, B)
⇒ ans =
```

```
1 1 0 0
1 1 0 0
```

Alternatively, we can concatenate *A* and *B* along the second dimension the following way:

```
[A, B].
```

dim can be larger than the dimensions of the N-d array objects and the result will thus have *dim* dimensions as the following example shows:

```
cat (4, ones(2, 2), zeros (2, 2))
⇒ ans =
```

```
ans(:,:,1,1) =
```

```
1 1
1 1
```

```
ans(:,:,1,2) =
```

```
0 0
0 0
```

See also: horzcat, vertcat.

horzcat (*array1*, *array2*, ..., *arrayN*) [Built-in Function]

Return the horizontal concatenation of N-d array objects, *array1*, *array2*, ..., *arrayN* along dimension 2.

See also: cat, vertcat.

vertcat (*array1*, *array2*, ..., *arrayN*) [Built-in Function]

Return the vertical concatenation of N-d array objects, *array1*, *array2*, ..., *arrayN* along dimension 1.

See also: cat, horzcat.

permute (*a*, *perm*) [Built-in Function]

Return the generalized transpose for an N-d array object *a*. The permutation vector *perm* must contain the elements 1:ndims(*a*) (in any order, but each element must appear just once).

See also: ipermute.

`ipermute (a, iperm)` [Built-in Function]

The inverse of the `permute` function. The expression

```
ipermute (permute (a, perm), perm)
```

returns the original array `a`.

See also: `permute`.

`reshape (a, m, n, ...)` [Built-in Function]

`reshape (a, siz)` [Built-in Function]

Return a matrix with the given dimensions whose elements are taken from the matrix `a`. The elements of the matrix are accessed in column-major order (like Fortran arrays are stored).

For example,

```
reshape ([1, 2, 3, 4], 2, 2)
⇒  1  3
   2  4
```

Note that the total number of elements in the original matrix must match the total number of elements in the new matrix.

A single dimension of the return matrix can be unknown and is flagged by an empty argument.

`y = circshift (x, n)` [Function File]

Circularly shifts the values of the array `x`. `n` must be a vector of integers no longer than the number of dimensions in `x`. The values of `n` can be either positive or negative, which determines the direction in which the values of `x` are shifted. If an element of `n` is zero, then the corresponding dimension of `x` will not be shifted. For example

```
x = [1, 2, 3; 4, 5, 6; 7, 8, 9];
circshift (x, 1)
⇒  7, 8, 9
   1, 2, 3
   4, 5, 6
circshift (x, -2)
⇒  7, 8, 9
   1, 2, 3
   4, 5, 6
circshift (x, [0,1])
⇒  3, 1, 2
   6, 4, 5
   9, 7, 8
```

See also: `permute`, `ipermute`, `shiftdim`.

`y = shiftdim (x, n)` [Function File]

`[y, ns] = shiftdim (x)` [Function File]

Shifts the dimension of `x` by `n`, where `n` must be an integer scalar. When `n` is positive, the dimensions of `x` are shifted to the left, with the leading dimensions circulated to

the end. If n is negative, then the dimensions of x are shifted to the right, with n leading singleton dimensions added.

Called with a single argument, `shiftdim`, removes the leading singleton dimensions, returning the number of dimensions removed in the second output argument ns .

For example

```
x = ones (1, 2, 3);
size (shiftdim (x, -1))
⇒ [1, 1, 2, 3]
size (shiftdim (x, 1))
⇒ [2, 3]
[b, ns] = shiftdim (x);
⇒ b = [1, 1, 1; 1, 1, 1]
⇒ ns = 1
```

See also: `reshape`, `permute`, `ipermute`, `circshift`, `squeeze`.

`shift (x, b)` [Function File]
`shift (x, b, dim)` [Function File]

If x is a vector, perform a circular shift of length b of the elements of x .

If x is a matrix, do the same for each column of x . If the optional dim argument is given, operate along this dimension

`[s, i] = sort (x)` [Loadable Function]
`[s, i] = sort (x, dim)` [Loadable Function]
`[s, i] = sort (x, mode)` [Loadable Function]
`[s, i] = sort (x, dim, mode)` [Loadable Function]

Return a copy of x with the elements arranged in increasing order. For matrices, `sort` orders the elements in each column.

For example,

```
sort ([1, 2; 2, 3; 3, 1])
⇒ 1 1
   2 2
   3 3
```

The `sort` function may also be used to produce a matrix containing the original row indices of the elements in the sorted matrix. For example,

```
[s, i] = sort ([1, 2; 2, 3; 3, 1])
⇒ s = 1 1
     2 2
     3 3
⇒ i = 1 3
     2 1
     3 2
```

If the optional argument dim is given, then the matrix is sorted along the dimension defined by dim . The optional argument `mode` defines the order in which the values will be sorted. Valid values of `mode` are ‘ascend’ or ‘descend’.

For equal elements, the indices are such that the equal elements are listed in the order that appeared in the original list.

The `sort` function may also be used to sort strings and cell arrays of strings, in which case the dictionary order of the strings is used.

The algorithm used in `sort` is optimized for the sorting of partially ordered lists.

`sortrows (a, c)` [Function File]

Sort the rows of the matrix `a` according to the order of the columns specified in `c`. If `c` is omitted, a lexicographical sort is used.

Since the `sort` function does not allow sort keys to be specified, it can't be used to order the rows of a matrix according to the values of the elements in various columns¹ in a single call. Using the second output, however, it is possible to sort all rows based on the values in a given column. Here's an example that sorts the rows of a matrix based on the values in the second column.

```
a = [1, 2; 2, 3; 3, 1];
[s, i] = sort (a (:, 2));
a (i, :)
⇒  3  1
   1  2
   2  3
```

`swap (inputs)` [Function File]

`[a1,b1] = swap(a,b)`
interchange `a` and `b`

`swapcols (inputs)` [Function File]

function `B = swapcols(A)`
permute columns of `A` into reverse order

`swaprows (inputs)` [Function File]

function `B = swaprows(A)`
permute rows of `A` into reverse order

`tril (a, k)` [Function File]

`triu (a, k)` [Function File]

Return a new matrix formed by extracting the lower (`tril`) or upper (`triu`) triangular part of the matrix `a`, and setting all other elements to zero. The second argument is optional, and specifies how many diagonals above or below the main diagonal should also be set to zero.

The default value of `k` is zero, so that `triu` and `tril` normally include the main diagonal as part of the result matrix.

If the value of `k` is negative, additional elements above (for `tril`) or below (for `triu`) the main diagonal are also selected.

¹ For example, to first sort based on the values in column 1, and then, for any values that are repeated in column 1, sort based on the values found in column 2, etc.

The absolute value of k must not be greater than the number of sub- or super-diagonals.

For example,

```
tril (ones (3), -1)
⇒  0  0  0
    1  0  0
    1  1  0
```

and

```
tril (ones (3), 1)
⇒  1  1  0
    1  1  1
    1  1  1
```

See also: triu, diag.

vec (x) [Function File]

Return the vector obtained by stacking the columns of the matrix x one above the other.

vech (x) [Function File]

Return the vector obtained by eliminating all supradiagonal elements of the square matrix x and stacking the result one column above the other.

prepad (x , l , c) [Function File]

postpad (x , l , c) [Function File]

postpad (x , l , c , dim) [Function File]

Prepends (appends) the scalar value c to the vector x until it is of length l . If the third argument is not supplied, a value of 0 is used.

If `length (x) > l` , elements from the beginning (end) of x are removed until a vector of length l is obtained.

If x is a matrix, elements are prepended or removed from each row.

If the optional *dim* argument is given, then operate along this dimension.

blkdiag (a , b , c , ...) [Function File]

Build a block diagonal matrix from a , b , c , All the arguments must be numeric and are two-dimensional matrices or scalars.

See also: diag, horzcat, vertcat.

17.3 Applying a Function to an Array

a = arrayfun (*name*, c) [Function File]

a = arrayfun (*func*, c) [Function File]

a = arrayfun (*func*, c , d) [Function File]

a = arrayfun (*func*, c , *options*) [Function File]

`[a, b, ...] = arrayfun (func, c, ...)` [Function File]

Execute a function on each element of an array. This is useful for functions that do not accept array arguments. If the function does accept array arguments it is better to call the function directly.

See `cellfun` for complete usage instructions.

See also: `cellfun`.

17.4 Special Utility Matrices

`eye (x)` [Built-in Function]

`eye (n, m)` [Built-in Function]

`eye (... , class)` [Built-in Function]

Return an identity matrix. If invoked with a single scalar argument, `eye` returns a square matrix with the dimension specified. If you supply two scalar arguments, `eye` takes them to be the number of rows and columns. If given a vector with two elements, `eye` uses the values of the elements as the number of rows and columns, respectively. For example,

```
eye (3)
⇒  1  0  0
    0  1  0
    0  0  1
```

The following expressions all produce the same result:

```
eye (2)
≡
eye (2, 2)
≡
eye (size ([1, 2; 3, 4]))
```

The optional argument `class`, allows `eye` to return an array of the specified type, like

```
val = zeros (n,m, "uint8")
```

Calling `eye` with no arguments is equivalent to calling it with an argument of 1. This odd definition is for compatibility with MATLAB.

`ones (x)` [Built-in Function]

`ones (n, m)` [Built-in Function]

`ones (n, m, k, ...)` [Built-in Function]

`ones (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all 1. The arguments are handled the same as the arguments for `eye`.

If you need to create a matrix whose values are all the same, you should use an expression like

```
val_matrix = val * ones (n, m)
```

The optional argument `class`, allows `ones` to return an array of the specified type, for example

```
val = ones (n,m, "uint8")
```


`zeros (x)` [Built-in Function]
`zeros (n, m)` [Built-in Function]
`zeros (n, m, k, ...)` [Built-in Function]
`zeros (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all 0. The arguments are handled the same as the arguments for `eye`.

The optional argument *class*, allows `zeros` to return an array of the specified type, for example

```
val = zeros (n,m, "uint8")
```

`repmat (A, m, n)` [Function File]
`repmat (A, [m n])` [Function File]

Form a block matrix of size *m* by *n*, with a copy of matrix *A* as each element. If *n* is not specified, form an *m* by *m* block matrix.

`rand (x)` [Loadable Function]
`rand (n, m)` [Loadable Function]
`rand ("state", x)` [Loadable Function]
`rand ("seed", x)` [Loadable Function]

Return a matrix with random elements uniformly distributed on the interval (0, 1). The arguments are handled the same as the arguments for `eye`.

You can query the state of the random number generator using the form

```
v = rand ("state")
```

This returns a column vector *v* of length 625. Later, you can restore the random number generator to the state *v* using the form

```
rand ("state", v)
```

You may also initialize the state vector from an arbitrary vector of length ≤ 625 for *v*. This new state will be a hash based on the value of *v*, not *v* itself.

By default, the generator is initialized from `/dev/urandom` if it is available, otherwise from cpu time, wall clock time and the current fraction of a second.

`rand` uses the Mersenne Twister with a period of $2^{19937}-1$ (See M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, Januray pp.3-30 1998, <http://www.math.keio.ac.jp/~matumoto/emt.html>). Do **not** use for cryptography without securely hashing several returned values together, otherwise the generator state can be learned after reading 624 consecutive values.

`rand` includes a second random number generator, that was the previous generator used in Octave. The new generator is used by default as it is significantly faster than the old generator, and produces random numbers with a significantly longer cycle time. However, in some circumstances it might be desirable to obtain the same random sequences as used by the old generators. To do this the keyword "seed" is used to specify that the old generators should be use, as in

```
rand ("seed", val)
```

which sets the seed of the generator to *val*. The seed of the generator can be queried with

```
s = rand ("seed")
```

However, it should be noted that querying the seed will not cause **rand** to use the old generators, only setting the seed will. To cause **rand** to once again use the new generators, the keyword "state" should be used to reset the state of the **rand**.

See also: randn, rande, randg, randp.

```
randn (x) [Loadable Function]
randn (n, m) [Loadable Function]
randn ("state", x) [Loadable Function]
randn ("seed", x) [Loadable Function]
```

Return a matrix with normally distributed random elements. The arguments are handled the same as the arguments for **rand**.

By default, **randn** uses a Marsaglia and Tsang Ziggurat technique to transform from a uniform to a normal distribution. (G. Marsaglia and W.W. Tsang, 'Ziggurat method for generating random variables', J. Statistical Software, vol 5, 2000, <http://www.jstatsoft.org/v05/i08/>)

See also: rand, rande, randg, randp.

```
rande (x) [Loadable Function]
rande (n, m) [Loadable Function]
rande ("state", x) [Loadable Function]
rande ("seed", x) [Loadable Function]
```

Return a matrix with exponentially distributed random elements. The arguments are handled the same as the arguments for **rand**.

By default, **randn** uses a Marsaglia and Tsang Ziggurat technique to transform from a uniform to a exponential distribution. (G. Marsaglia and W.W. Tsang, 'Ziggurat method for generating random variables', J. Statistical Software, vol 5, 2000, <http://www.jstatsoft.org/v05/i08/>)

See also: rand, randn, randg, randp.

```
randp (l, x) [Loadable Function]
randp (l, n, m) [Loadable Function]
randp ("state", x) [Loadable Function]
randp ("seed", x) [Loadable Function]
```

Return a matrix with Poisson distributed random elements. The arguments are handled the same as the arguments for **rand**, except for the argument *l*.

Five different algorithms are used depending on the range of *l* and whether or not *l* is a scalar or a matrix.

For scalar *l* ≤ 12, use direct method.

Press, et al., 'Numerical Recipes in C', Cambridge University Press, 1992.

For scalar $l > 12$, use rejection method.[1]

Press, et al., 'Numerical Recipes in C', Cambridge University Press, 1992.

For matrix $l \leq 10$, use inversion method.[2]

Stadlober E., et al., WinRand source code, available via FTP.

For matrix $l > 10$, use patchwork rejection method.

Stadlober E., et al., WinRand source code, available via FTP, or H. Zechner, 'Efficient sampling from continuous and discrete unimodal distributions', Doctoral Dissertaion, 156pp., Technical University Graz, Austria, 1994.

For $l > 1e8$, use normal approximation.

L. Montanet, et al., 'Review of Particle Properties', Physical Review D 50 p1284, 1994

See also: rand, randn, rande, randg.

<code>randg (a, x)</code>	[Loadable Function]
<code>randg (a, n, m)</code>	[Loadable Function]
<code>randg ("state", x)</code>	[Loadable Function]
<code>randg ("seed", x)</code>	[Loadable Function]

Return a matrix with `gamma(a,1)` distributed random elements. The arguments are handled the same as the arguments for `rand`, except for the argument `a`.

This can be used to generate many distributions:

`gamma (a, b)` for $a > -1, b > 0$

`r = b * randg (a)`

`beta (a, b)` for $a > -1, b > -1$

`r1 = randg (a, 1)`

`r = r1 / (r1 + randg (b, 1))`

`Erlang (a, n)`

`r = a * randg (n)`

`chisq (df)` for $df > 0$

`r = 2 * randg (df / 2)`

`t(df)` for $0 < df < \text{inf}$ (use `randn` if `df` is infinite)

`r = randn () / sqrt (2 * randg (df / 2) / df)`

`F (n1, n2)` for $0 < n1, 0 < n2$

`r1 = 2 * randg (n1 / 2) / n1` or 1 if `n1` is infinite

`r2 = 2 * randg (n2 / 2) / n2` or 1 if `n2` is infinite

`r = r1 / r2`

`negative binomial (n, p)` for $n > 0, 0 < p \leq 1$

`r = randp ((1 - p) / p * randg (n))`

`non-central chisq (df, L)`, for $df \geq 0$ and $L > 0$

(use `chisq` if $L = 0$)

```

r = randp (L / 2)
r(r > 0) = 2 * randg (r(r > 0))
r(df > 0) += 2 * randg (df(df > 0)/2)

Dirichlet (a1, ..., ak)
r = (randg (a1), ..., randg (ak))
r = r / sum (r)

```

See also: rand, randn, rande, randp.

The new random generators all use a common Mersenne Twister generator, and so the state of only one of the generators needs to be reset. The old generator function use separate generators. This ensures that

```

rand ("seed", 13);
randn ("seed", 13);
u = rand (100, 1);
n = randn (100, 1);

```

and

```

rand ("seed", 13);
randn ("seed", 13);
u = zeros (100, 1);
n = zeros (100, 1);
for i = 1:100
    u(i) = rand ();
    n(i) = randn ();
end

```

produce equivalent results.

Normally, **rand** and **randn** obtain their initial seeds from the system clock, so that the sequence of random numbers is not the same each time you run Octave. If you really do need for to reproduce a sequence of numbers exactly, you can set the seed to a specific value.

If it is invoked without arguments, **rand** and **randn** return a single element of a random sequence.

The **rand** and **randn** functions use Fortran code from RANLIB, a library of fortran routines for random number generation, compiled by Barry W. Brown and James Lovato of the Department of Biomathematics at The University of Texas, M.D. Anderson Cancer Center, Houston, TX 77030.

randperm (n) [Function File]

Return a row vector containing a random permutation of the integers from 1 to *n*.

diag (v, k) [Built-in Function]

Return a diagonal matrix with vector *v* on diagonal *k*. The second argument is optional. If it is positive, the vector is placed on the *k*-th super-diagonal. If it is negative, it is placed on the *-k*-th sub-diagonal. The default value of *k* is 0, and the vector is placed on the main diagonal. For example,

```
diag ([1, 2, 3], 1)
⇒  0  1  0  0
    0  0  2  0
    0  0  0  3
    0  0  0  0
```

Given a matrix argument, instead of a vector, `diag` extracts the k -th diagonal of the matrix.

The functions `linspace` and `logspace` make it very easy to create vectors with evenly or logarithmically spaced elements. See [Section 4.2 \[Ranges\]](#), page 39.

`linspace (base, limit, n)` [Built-in Function]

Return a row vector with n linearly spaced elements between *base* and *limit*. If the number of elements is greater than one, then the *base* and *limit* are always included in the range. If *base* is greater than *limit*, the elements are stored in decreasing order. If the number of points is not specified, a value of 100 is used.

The `linspace` function always returns a row vector.

For compatibility with MATLAB, return the second argument if fewer than two values are requested.

`logspace (base, limit, n)` [Function File]

Similar to `linspace` except that the values are logarithmically spaced from 10^{base} to 10^{limit} .

If *limit* is equal to π , the points are between 10^{base} and π , *not* 10^{base} and 10^π , in order to be compatible with the corresponding MATLAB function.

Also for compatibility, return the second argument if fewer than two values are requested.

See also: `linspace`.

17.5 Famous Matrices

The following functions return famous matrix forms.

`hadamard (n)` [Function File]

Construct a Hadamard matrix *Hn* of size n -by- n . The size n must be of the form $2^k \cdot p$ in which p is one of 1, 12, 20 or 28. The returned matrix is normalized, meaning $Hn(:, 1) == 1$ and $H(1, :) == 1$.

Some of the properties of Hadamard matrices are:

- `kron (Hm, Hn)` is a Hadamard matrix of size m -by- n .
- `Hn * Hn' == n * eye (n)`.
- The rows of *Hn* are orthogonal.
- `det (A) <= det (Hn)` for all *A* with `abs (A (i, j)) <= 1`.
- Multiply any row or column by -1 and still have a Hadamard matrix.

hankel (*c*, *r*) [Function File]

Return the Hankel matrix constructed given the first column *c*, and (optionally) the last row *r*. If the last element of *c* is not the same as the first element of *r*, the last element of *c* is used. If the second argument is omitted, it is assumed to be a vector of zeros with the same size as *c*.

A Hankel matrix formed from an *m*-vector *c*, and an *n*-vector *r*, has the elements

$$H(i, j) = \begin{cases} c_{i+j-1}, & i + j - 1 \leq m; \\ r_{i+j-m}, & \text{otherwise.} \end{cases}$$

See also: vander, sylvester_matrix, hilb, invhilb, toeplitz.

hilb (*n*) [Function File]

Return the Hilbert matrix of order *n*. The *i*, *j* element of a Hilbert matrix is defined as

$$H(i, j) = \frac{1}{(i + j - 1)}$$

See also: hankel, vander, sylvester_matrix, invhilb, toeplitz.

invhilb (*n*) [Function File]

Return the inverse of a Hilbert matrix of order *n*. This can be computed exactly using

$$\begin{aligned} A_{ij} &= -1^{i+j}(i+j-1) \binom{n+i-1}{n-j} \binom{n+j-1}{n-i} \binom{i+j-2}{i-2}^2 \\ &= \frac{p(i)p(j)}{(i+j-1)} \end{aligned}$$

where

$$p(k) = -1^k \binom{k+n-1}{k-1} \binom{n}{k}$$

The validity of this formula can easily be checked by expanding the binomial coefficients in both formulas as factorials. It can be derived more directly via the theory of Cauchy matrices: see J. W. Demmel, Applied Numerical Linear Algebra, page 92.

Compare this with the numerical calculation of `inverse (hilb (n))`, which suffers from the ill-conditioning of the Hilbert matrix, and the finite precision of your computer's floating point arithmetic.

See also: hankel, vander, sylvester_matrix, hilb, toeplitz.

magic (*n*) [Function File]

Create an *n*-by-*n* magic square. Note that `magic (2)` is undefined since there is no 2-by-2 magic square.

pascal (*n*, *t*) [Function File]

Return the Pascal matrix of order *n* if *t* = 0. *t* defaults to 0. Return lower triangular Cholesky factor of the Pascal matrix if *t* = 1. This matrix is its own inverse, that is `pascal (n, 1) ^ 2 == eye (n)`. If *t* = 2, return a transposed and permuted version of `pascal (n, 1)`, which is the cube-root of the identity matrix. That is `pascal (n, 2) ^ 3 == eye (n)`.

See also: `hankel`, `vander`, `sylvester_matrix`, `hilb`, `invhilb`, `toeplitz` `hadamard`, `wilkinson`, `compan`, `rosser`.

rosser () [Function File]

Returns the Rosser matrix. This is a difficult test case used to test eigenvalue algorithms.

See also: `hankel`, `vander`, `sylvester_matrix`, `hilb`, `invhilb`, `toeplitz` `hadamard`, `wilkinson`, `compan`, `pascal`.

sylvester_matrix (*k*) [Function File]

Return the Sylvester matrix of order $n = 2^k$.

See also: `hankel`, `vander`, `hilb`, `invhilb`, `toeplitz`.

toeplitz (*c*, *r*) [Function File]

Return the Toeplitz matrix constructed given the first column *c*, and (optionally) the first row *r*. If the first element of *c* is not the same as the first element of *r*, the first element of *c* is used. If the second argument is omitted, the first row is taken to be the same as the first column.

A square Toeplitz matrix has the form:

$$\begin{bmatrix} c_0 & r_1 & r_2 & \cdots & r_n \\ c_1 & c_0 & r_1 & \cdots & r_{n-1} \\ c_2 & c_1 & c_0 & \cdots & r_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_n & c_{n-1} & c_{n-2} & \cdots & c_0 \end{bmatrix}$$

See also: `hankel`, `vander`, `sylvester_matrix`, `hilb`, `invhilb`.

vander (*c*) [Function File]

Return the Vandermonde matrix whose next to last column is *c*.

A Vandermonde matrix has the form:

$$\begin{bmatrix} c_1^{n-1} & \cdots & c_1^2 & c_1 & 1 \\ c_2^{n-1} & \cdots & c_2^2 & c_2 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ c_n^{n-1} & \cdots & c_n^2 & c_n & 1 \end{bmatrix}$$

See also: `hankel`, `sylvester_matrix`, `hilb`, `invhilb`, `toeplitz`.

`wilkinson` (*n*)

[Function File]

Return the Wilkinson matrix of order *n*.

See also: `hankel`, `vander`, `sylvester_matrix`, `hilb`, `invhilb`, `toeplitz`, `hadamard`, `rosser`, `companion`, `pascal`.

18 Arithmetic

Unless otherwise noted, all of the functions described in this chapter will work for real and complex scalar or matrix arguments.

18.1 Utility Functions

The following functions are available for working with complex numbers. Each expects a single argument. They are called *mapping functions* because when given a matrix argument, they apply the given function to each element of the matrix.

ceil (*x*) [Mapping Function]
 Return the smallest integer not less than *x*. If *x* is complex, return **ceil** (**real** (*x*)) + **ceil** (**imag** (*x*)) * **I**.

cplxpair (*z*, *tol*, *dim*) [Function File]
 Sort the numbers *z* into complex conjugate pairs ordered by increasing real part. With identical real parts, order by increasing imaginary magnitude. Place the negative imaginary complex number first within each pair. Place all the real numbers after all the complex pairs (those with **abs** (**imag** (*z*) / *z*) < *tol*), where the default value of *tol* is 100 * **eps**.

By default the complex pairs are sorted along the first non-singleton dimension of *z*. If *dim* is specified, then the complex pairs are sorted along this dimension.

Signal an error if some complex numbers could not be paired. Requires all complex numbers to be exact conjugates within *tol*, or signals an error. Note that there are no guarantees on the order of the returned pairs with identical real parts but differing imaginary parts.

```
cplxpair (exp(2i*pi*[0:4]'/5)) == exp(2i*pi*[3; 2; 4; 1; 0]/5)
```

d = **del2** (*m*) [Function File]
d = **del2** (*m*, *h*) [Function File]
d = **del2** (*m*, *dx*, *dy*, ...) [Function File]

Calculates the discrete Laplace operator. If *m* is a matrix this is defined as

$$d = \frac{1}{4} \left(\frac{d^2}{dx^2} M(x, y) + \frac{d^2}{dy^2} M(x, y) \right)$$

The above is continued to N-dimensional arrays calculating the second derivative over the higher dimensions.

The spacing between evaluation points may be defined by *h*, which is a scalar defining the spacing in all dimensions. Or alternative, the spacing in each dimension may be defined separately by *dx*, *dy*, etc. Scalar spacing values give equidistant spacing, whereas vector spacing values can be used to specify variable spacing. The length of the vectors must match the respective dimension of *m*. The default spacing value is 1.

You need at least 3 data points for each dimension. Boundary points are calculated as the linear extrapolation of the interior points.

See also: **gradient**, **diff**.

`exp (x)` [Mapping Function]
 Compute the exponential of x . To compute the matrix exponential, see [Chapter 19](#) [[Linear Algebra](#)], page 231.

`p = factor (q)` [Function File]
`[p, n] = factor (q)` [Function File]
 Return prime factorization of q . That is `prod (p) == q`. If $q == 1$, returns 1.
 With two output arguments, returns the unique primes p and their multiplicities.
 That is `prod (p .^ n) == q`.

`factorial (n)` [Function File]
 Return the factorial of n . If n is scalar, this is equivalent to `prod (1:n)`. If n is an array, the factorial of the elements of the array are returned.

`fix (x)` [Mapping Function]
 Truncate x toward zero. If x is complex, return `fix (real (x)) + fix (imag (x)) * I`.

`floor (x)` [Mapping Function]
 Return the largest integer not greater than x . If x is complex, return `floor (real (x)) + floor (imag (x)) * I`.

`fmod (x, y)` [Mapping Function]
 Compute the floating point remainder of dividing x by y using the C library function `fmod`. The result has the same sign as x . If y is zero, the result implementation-defined.

`g = gcd (a1, ...)` [Loadable Function]
`[g, v1, ...] = gcd (a1, ...)` [Loadable Function]
 If a single argument is given then compute the greatest common divisor of the elements of this argument. Otherwise if more than one argument is given all arguments must be the same size or scalar. In this case the greatest common divisor is calculated for element individually. All elements must be integers. For example,

```
gcd ([15, 20])
⇒ 5
```

and

```
gcd ([15, 9], [20 18])
⇒ 5 9
```

Optional return arguments $v1$, etc, contain integer vectors such that,

$$g = v_1 a_1 + v_2 a_2 + \dots$$

For backward compatibility with previous versions of this function, when all arguments are scalar, a single return argument $v1$ containing all of the values of $v1, \dots$ is acceptable.

See also: `lcm`, `min`, `max`, `ceil`, `floor`.

```

x = gradient (M) [Function File]
[x, y, ...] = gradient (M) [Function File]
[...] = gradient (M, s) [Function File]
[...] = gradient (M, dx, dy, ...) [Function File]

```

Calculates the gradient. `x = gradient (M)` calculates the one dimensional gradient if `M` is a vector. If `M` is a matrix the gradient is calculated for each row.

`[x, y] = gradient (M)` calculates the one dimensional gradient for each direction if `M` if `M` is a matrix. Additional return arguments can be use for multi-dimensional matrices.

Spacing values between two points can be provided by the `dx`, `dy` or `h` parameters. If `h` is supplied it is assumed to be the spacing in all directions. Otherwise, seperate values of the spacing can be supplied by the `dx`, etc variables. A scalar value specifies an equidistant spacing, while a vector value can be used to specify a variable spacing. The length must match their respective dimension of `M`.

At boundary points a linear extrapolation is applied. Interior points are calculated with the first approximation of the numerical gradient

$$y'(i) = 1/(x(i+1)-x(i-1)) * (y(i-1)-y(i+1)).$$

```
lcm (x, ...) [Mapping Function]
```

Compute the least common multiple of the elements elements of `x`, or the list of all the arguments. For example,

```
lcm (a1, ..., ak)
```

is the same as

```
lcm ([a1, ..., ak]).
```

All elements must be the same size or scalar.

See also: gcd, min, max, ceil, floor.

```
log (x) [Mapping Function]
```

Compute the natural logarithm for each element of `x`. To compute the matrix logarithm, see [Chapter 19 \[Linear Algebra\]](#), page 231.

See also: log2, log10, logspace, exp.

```
log10 (x) [Mapping Function]
```

Compute the base-10 logarithm for each element of `x`.

See also: log, log2, logspace, exp.

```
log2 (x) [Mapping Function]
```

```
[f, e] = log2 (x) [Mapping Function]
```

Compute the base-2 logarithm of `x`. With two outputs, returns `f` and `e` such that $1/2 \leq |f| < 1$ and $x = f \cdot 2^e$.

See also: log, log10, logspace, exp.

`max (x, y, dim)` [Mapping Function]
`[w, iw] = max (x)` [Mapping Function]

For a vector argument, return the maximum value. For a matrix argument, return the maximum value from each column, as a row vector, or over the dimension *dim* if defined. For two matrices (or a matrix and scalar), return the pair-wise maximum. Thus,

```
max (max (x))
```

returns the largest element of *x*, and

```
max (2:5, pi)
⇒ 3.1416 3.1416 4.0000 5.0000
```

compares each element of the range 2:5 with *pi*, and returns a row vector of the maximum values.

For complex arguments, the magnitude of the elements are used for comparison.

If called with one input and two output arguments, `max` also returns the first index of the maximum value(s). Thus,

```
[x, ix] = max ([1, 3, 5, 2, 5])
⇒ x = 5
   ix = 3
```

`min (x, y, dim)` [Mapping Function]
`[w, iw] = min (x)` [Mapping Function]

For a vector argument, return the minimum value. For a matrix argument, return the minimum value from each column, as a row vector, or over the dimension *dim* if defined. For two matrices (or a matrix and scalar), return the pair-wise minimum. Thus,

```
min (min (x))
```

returns the smallest element of *x*, and

```
min (2:5, pi)
⇒ 2.0000 3.0000 3.1416 3.1416
```

compares each element of the range 2:5 with *pi*, and returns a row vector of the minimum values.

For complex arguments, the magnitude of the elements are used for comparison.

If called with one input and two output arguments, `min` also returns the first index of the minimum value(s). Thus,

```
[x, ix] = min ([1, 3, 0, 2, 5])
⇒ x = 0
   ix = 3
```

`mod (x, y)` [Mapping Function]

Compute modulo function, using

```
x - y .* floor (x ./ y)
```

Note that this handles negative numbers correctly: `mod (-1, 3)` is 2, not -1 as `rem (-1, 3)` returns. Also, `mod (x, 0)` returns *x*.

An error message is printed if the dimensions of the arguments do not agree, or if either of the arguments is complex.

See also: `rem`, `round`.

`nextpow2 (x)` [Function File]

If x is a scalar, returns the first integer n such that $2^n \geq |x|$.

If x is a vector, return `nextpow2 (length (x))`.

See also: `pow2`.

`nthroot (x, n)` [Function File]

Compute the n th root of x , returning real results for real components of x . For example

```

nthroot (-1, 3)
⇒ -1
(-1) ^ (1 / 3)
⇒ 0.50000 - 0.86603i

```

`pow2 (x)` [Mapping Function]

`pow2 (f, e)` [Mapping Function]

With one argument, computes 2^x for each element of x . With two arguments, returns $f \cdot 2^e$.

See also: `nextpow2`.

`primes (n)` [Function File]

Return all primes up to n .

Note that if you need a specific number of primes, you can use the fact the distance from one prime to the next is on average proportional to the logarithm of the prime. Integrating, you find that there are about k primes less than $k \log(5k)$.

The algorithm used is called the Sieve of Erasthenes.

`rem (x, y)` [Mapping Function]

Return the remainder of x / y , computed using the expression

```

x - y .* fix (x ./ y)

```

An error message is printed if the dimensions of the arguments do not agree, or if either of the arguments is complex.

See also: `mod`, `round`.

`round (x)` [Mapping Function]

Return the integer nearest to x . If x is complex, return `round (real (x)) + round (imag (x)) * I`.

See also: `rem`.

sign (x) [Mapping Function]
 Compute the *signum* function, which is defined as

$$\text{sign}(x) = \begin{cases} 1, & x > 0; \\ 0, & x = 0; \\ -1, & x < 0. \end{cases}$$

For complex arguments, **sign** returns `x ./ abs (x)`.

sqrt (x) [Mapping Function]
 Compute the square root of x. If x is negative, a complex result is returned. To compute the matrix square root, see [Chapter 19 \[Linear Algebra\]](#), page 231.

18.2 Complex Arithmetic

The following functions are available for working with complex numbers. Each expects a single argument. Given a matrix they work on an element by element basis. In the descriptions of the following functions, z is the complex number $x + iy$, where i is defined as $\sqrt{-1}$.

abs (z) [Mapping Function]
 Compute the magnitude of z , defined as $|z| = \sqrt{x^2 + y^2}$.

For example,

$$\begin{aligned} \text{abs} (3 + 4i) \\ \Rightarrow 5 \end{aligned}$$

arg (z) [Mapping Function]
angle (z) [Mapping Function]

Compute the argument of z , defined as $\theta = \tan^{-1}(y/x)$. in radians.

For example,

$$\begin{aligned} \text{arg} (3 + 4i) \\ \Rightarrow 0.92730 \end{aligned}$$

conj (z) [Mapping Function]
 Return the complex conjugate of z , defined as $\bar{z} = x - iy$.

See also: `real`, `imag`.

imag (z) [Mapping Function]
 Return the imaginary part of z as a real number.

See also: `real`, `conj`.

real (z) [Mapping Function]
 Return the real part of z .

See also: `imag`, `conj`.

18.3 Trigonometry

Octave provides the following trigonometric functions. Angles are specified in radians. To convert from degrees to radians multiply by $\pi/180$ (e.g. `sin (30 * pi/180)` returns the sine of 30 degrees).

sin (x) [Mapping Function]
Compute the sine of each element of x.

cos (x) [Mapping Function]
Compute the cosine of each element of x.

tan (z) [Mapping Function]
Compute tangent of each element of x.

sec (x) [Mapping Function]
Compute the secant of each element of x.

csc (x) [Mapping Function]
Compute the cosecant of each element of x.

cot (x) [Mapping Function]
Compute the cotangent of each element of x.

asin (x) [Mapping Function]
Compute the inverse sine of each element of x.

acos (x) [Mapping Function]
Compute the inverse cosine of each element of x.

atan (x) [Mapping Function]
Compute the inverse tangent of each element of x.

asec (x) [Mapping Function]
Compute the inverse secant of each element of x.

acsc (x) [Mapping Function]
Compute the inverse cosecant of each element of x.

acot (x) [Mapping Function]
Compute the inverse cotangent of each element of x.

sinh (x) [Mapping Function]
Compute the hyperbolic sine of each element of x.

cosh (x) [Mapping Function]
Compute the hyperbolic cosine of each element of x.

tanh (*x*) [Mapping Function]
 Compute hyperbolic tangent of each element of *x*.

sech (*x*) [Mapping Function]
 Compute the hyperbolic secant of each element of *x*.

csch (*x*) [Mapping Function]
 Compute the hyperbolic cosecant of each element of *x*.

coth (*x*) [Mapping Function]
 Compute the hyperbolic cotangent of each element of *x*.

asinh (*x*) [Mapping Function]
 Compute the inverse hyperbolic sine of each element of *x*.

acosh (*x*) [Mapping Function]
 Compute the inverse hyperbolic cosine of each element of *x*.

atanh (*x*) [Mapping Function]
 Compute the inverse hyperbolic tangent of each element of *x*.

asech (*x*) [Mapping Function]
 Compute the inverse hyperbolic secant of each element of *x*.

acsch (*x*) [Mapping Function]
 Compute the inverse hyperbolic cosecant of each element of *x*.

acoth (*x*) [Mapping Function]
 Compute the inverse hyperbolic cotangent of each element of *x*.

Each of these functions expect a single argument. For matrix arguments, they work on an element by element basis. For example,

```
sin ([1, 2; 3, 4])
⇒  0.84147  0.90930
    0.14112 -0.75680
```

atan2 (*y*, *x*) [Mapping Function]
 Compute $\text{atan}(y / x)$ for corresponding elements of *y* and *x*. The result is in range $-\pi$ to π .

In addition to the trigonometric functions that work with radians, Octave also provides the following functions which work with degrees.

sind (*x*) [Function File]
 Compute the sine of each element of *x*. Returns zero in elements for which *x*/180 is an integer.

See also: sin, cosd, tand, acosd, asind, atand.

cosd (x) [Function File]
Compute the cosine of an angle in degrees. Returns zero in elements for which $(x-90)/180$ is an integer.

See also: cos, sind, tand, acosd, asind, atand.

tand (x) [Function File]
Compute the tangent of an angle in degrees. Returns zero for elements of for which $x/180$ is an integer and **Inf** for elements where $(x-90)/180$ is an integer.

See also: tan, cosd, sind, acosd, asind, atand.

secd (x) [Function File]
Compute the secant of an angle in degrees.

See also: sec, cscd, sind, cosd.

cscd (x) [Function File]
Compute the cosecant of an angle in degrees.

See also: csc, secd, sind, cosd.

cotd (x) [Function File]
Compute the cotangent of an angle in degrees.

See also: cot, tand.

asind (x) [Function File]
Compute the inverse sine of an angle in degrees.

See also: asin, sind, acosd.

acosd (x) [Function File]
Compute the inverse cosine of an angle in degrees.

See also: acos, cosd, asecd.

atand (x) [Function File]
Compute the inverse tangent of an angle in degrees.

See also: acot, tand.

asecd (x) [Function File]
Compute inverse secant in degrees.

See also: asec, secd, acscd.

acscd (x) [Function File]
Compute the inverse cosecant of an angle in degrees.

See also: acsc, cscd, asecd.

acotd (x) [Function File]
Compute the inverse cotangent of an angle in degrees.

See also: atan, tand.

18.4 Sums and Products

sum (*x*, *dim*) [Built-in Function]

Sum of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise sum).

As a special case, if *x* is a vector and *dim* is omitted, return the sum of the elements.

prod (*x*, *dim*) [Built-in Function]

Product of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise products).

As a special case, if *x* is a vector and *dim* is omitted, return the product of the elements.

cumsum (*x*, *dim*) [Built-in Function]

Cumulative sum of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise cumulative sums).

As a special case, if *x* is a vector and *dim* is omitted, return the cumulative sum of the elements as a vector with the same orientation as *x*.

cumprod (*x*, *dim*) [Built-in Function]

Cumulative product of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise cumulative products).

As a special case, if *x* is a vector and *dim* is omitted, return the cumulative product of the elements as a vector with the same orientation as *x*.

sumsq (*x*, *dim*) [Built-in Function]

Sum of squares of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise sum of squares).

As a special case, if *x* is a vector and *dim* is omitted, return the sum of squares of the elements.

This function is conceptually equivalent to computing

```
sum (x .* conj (x), dim)
```

but it uses less memory and avoids calling `conj` if *x* is real.

accumarray (*subs*, *vals*, *sz*, *fun*, *fillval*, *issparse*) [Function File]

accumarray (*csubs*, *vals*, ...) [Function File]

Creates an array by accumulating the elements of a vector into the positions of defined by their subscripts. The subscripts are defined by the rows of the matrix *subs* and the values by *vals*. Each row of *subs* corresponds to one of the values in *vals*.

The size of the matrix will be determined by the subscripts themselves. However, if *sz* is defined it determines the matrix size. The length of *sz* must correspond to the number of columns in *subs*.

The default action of **accumarray** is to sum the elements with the same subscripts. This behavior can be modified by defining the *fun* function. This should be a function

or function handle that accepts a column vector and returns a scalar. The result of the function should not depend on the order of the subscripts.

The elements of the returned array that have no subscripts associated with them are set to zero. Defining *fillval* to some other value allows these values to be defined.

By default `accumarray` returns a full matrix. If *issparse* is logically true, then a sparse matrix is returned instead.

An example of the use of `accumarray` is:

```
accumarray ([1,1,1;2,1,2;2,3,2;2,1,2;2,3,2],101:105)
⇒ ans(:,1) = [101, 0, 0; 0, 0, 0]
   ans(:,2) = [0, 0, 0; 206, 0, 208]
```

18.5 Special Functions

<code>[j, ierr] = besselj (alpha, x, opt)</code>	[Loadable Function]
<code>[y, ierr] = bessely (alpha, x, opt)</code>	[Loadable Function]
<code>[i, ierr] = besseli (alpha, x, opt)</code>	[Loadable Function]
<code>[k, ierr] = besselk (alpha, x, opt)</code>	[Loadable Function]
<code>[h, ierr] = besselh (alpha, k, x, opt)</code>	[Loadable Function]

Compute Bessel or Hankel functions of various kinds:

<code>besselj</code>	Bessel functions of the first kind.
<code>bessely</code>	Bessel functions of the second kind.
<code>besseli</code>	Modified Bessel functions of the first kind.
<code>besselk</code>	Modified Bessel functions of the second kind.
<code>besselh</code>	Compute Hankel functions of the first ($k = 1$) or second ($k = 2$) kind.

If the argument *opt* is supplied, the result is scaled by the `exp (-I*x)` for $k = 1$ or `exp (I*x)` for $k = 2$.

If *alpha* is a scalar, the result is the same size as *x*. If *x* is a scalar, the result is the same size as *alpha*. If *alpha* is a row vector and *x* is a column vector, the result is a matrix with `length (x)` rows and `length (alpha)` columns. Otherwise, *alpha* and *x* must conform and the result will be the same size.

The value of *alpha* must be real. The value of *x* may be complex.

If requested, *ierr* contains the following status information and is the same size as the result.

0. Normal return.
1. Input error, return `NaN`.
2. Overflow, return `Inf`.
3. Loss of significance by argument reduction results in less than half of machine accuracy.
4. Complete loss of significance by argument reduction, return `NaN`.
5. Error—no computation, algorithm termination condition not met, return `NaN`.

`[a, ierr] = airy (k, z, opt)` [Loadable Function]

Compute Airy functions of the first and second kind, and their derivatives.

K	Function	Scale factor (if a third argument is supplied)
---	-----	-----
0	Ai (Z)	exp ((2/3) * Z * sqrt (Z))
1	dAi(Z)/dZ	exp ((2/3) * Z * sqrt (Z))
2	Bi (Z)	exp (-abs (real ((2/3) * Z *sqrt (Z))))
3	dBi(Z)/dZ	exp (-abs (real ((2/3) * Z *sqrt (Z))))

The function call `airy (z)` is equivalent to `airy (0, z)`.

The result is the same size as `z`.

If requested, `ierr` contains the following status information and is the same size as the result.

0. Normal return.
1. Input error, return `NaN`.
2. Overflow, return `Inf`.
3. Loss of significance by argument reduction results in less than half of machine accuracy.
4. Complete loss of significance by argument reduction, return `NaN`.
5. Error—no computation, algorithm termination condition not met, return `NaN`.

`beta (a, b)` [Mapping Function]

Return the Beta function,

$$B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

`betainc (x, a, b)` [Mapping Function]

Return the incomplete Beta function,

$$\beta(x, a, b) = B(a, b)^{-1} \int_0^x t^{(a-1)}(1-t)^{(b-1)} dt.$$

If `x` has more than one component, both `a` and `b` must be scalars. If `x` is a scalar, `a` and `b` must be of compatible dimensions.

`betaln (a, b)` [Mapping Function]

Return the log of the Beta function,

$$B(a, b) = \log \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

See also: `beta`, `betai`, `gammaln`.

bincoeff (*n*, *k*)

[Mapping Function]

Return the binomial coefficient of *n* and *k*, defined as

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}$$

For example,

```
bincoeff (5, 2)
⇒ 10
```

erf (*z*)

[Mapping Function]

Computes the error function,

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

See also: erfc, erfinv.**erfc** (*z*)

[Mapping Function]

Computes the complementary error function, $1 - \operatorname{erf}(z)$.**See also:** erf, erfinv.**erfinv** (*z*)

[Mapping Function]

Computes the inverse of the error function.

See also: erf, erfc.**gamma** (*z*)

[Mapping Function]

Computes the Gamma function,

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt.$$

See also: gammai, lgamma.**gammainc** (*x*, *a*)

[Mapping Function]

Compute the normalized incomplete gamma function,

$$\gamma(x, a) = \frac{\int_0^x e^{-t} t^{a-1} dt}{\Gamma(a)}$$

with the limiting value of 1 as *x* approaches infinity. The standard notation is $P(a, x)$, e.g. Abramowitz and Stegun (6.5.1).

If *a* is scalar, then **gammainc** (*x*, *a*) is returned for each element of *x* and vice versa.

If neither *x* nor *a* is scalar, the sizes of *x* and *a* must agree, and **gammainc** is applied element-by-element.

See also: gamma, lgamma.

`L = legendre (n, X)` [Function File]

Legendre Function of degree n and order m where all values for $m = 0..n$ are returned. n must be a scalar in the range $[0..255]$. The return value has one dimension more than x .

The Legendre Function of degree n and order m

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x)$$

with:

Legendre Polynom of degree n

$$P_n(x) = \frac{1}{2^n n!} \left[\frac{d^n}{dx^n} (x^2 - 1)^n \right]$$

`legendre(3, [-1.0 -0.9 -0.8])` returns the matrix

x	-1.0	-0.9	-0.8
m=0	-1.00000	-0.47250	-0.08000
m=1	0.00000	-1.99420	-1.98000
m=2	0.00000	-2.56500	-4.32000
m=3	0.00000	-1.24229	-3.24000

`lgamma (x)` [Mapping Function]

`gammaln (x)` [Mapping Function]

Return the natural logarithm of the gamma function.

See also: `gamma`, `gammai`.

`cross (x, y, dim)` [Function File]

Computes the vector cross product of the two 3-dimensional vectors x and y .

`cross ([1,1,0], [0,1,1])`
 $\Rightarrow [1; -1; 1]$

If x and y are matrices, the cross product is applied along the first dimension with 3 elements. The optional argument *dim* is used to force the cross product to be calculated along the dimension defined by *dim*.

`commutation_matrix (m, n)` [Function File]

Return the commutation matrix $K_{m,n}$ which is the unique $mn \times mn$ matrix such that $K_{m,n} \cdot \text{vec}(A) = \text{vec}(A^T)$ for all $m \times n$ matrices A .

If only one argument m is given, $K_{m,m}$ is returned.

See Magnus and Neudecker (1988), Matrix differential calculus with applications in statistics and econometrics.

`duplication_matrix (n)` [Function File]

Return the duplication matrix D_n which is the unique $n^2 \times n(n+1)/2$ matrix such that $D_n * \text{vech}(A) = \text{vec}(A)$ for all symmetric $n \times n$ matrices A .

See Magnus and Neudecker (1988), Matrix differential calculus with applications in statistics and econometrics.

18.6 Coordinate Transformations

`[theta, r] = cart2pol (x, y)` [Function File]

`[theta, r, z] = cart2pol (x, y, z)` [Function File]

Transform cartesian to polar or cylindrical coordinates. x , y (and z) must be of same shape. θ describes the angle relative to the x - axis. r is the distance to the z - axis $(0, 0, z)$.

See also: `pol2cart`, `cart2sph`, `sph2cart`.

`[x, y] = pol2cart (theta, r)` [Function File]

`[x, y, z] = pol2cart (theta, r, z)` [Function File]

Transform polar or cylindrical to cartesian coordinates. θ , r (and z) must be of same shape. θ describes the angle relative to the x - axis. r is the distance to the z - axis $(0, 0, z)$.

See also: `cart2pol`, `cart2sph`, `sph2cart`.

`[theta, phi, r] = cart2sph (x, y, z)` [Function File]

Transform cartesian to spherical coordinates. x , y and z must be of same shape. θ describes the angle relative to the x - axis. ϕ is the angle relative to the xy - plane. r is the distance to the origin $(0, 0, 0)$.

See also: `pol2cart`, `cart2pol`, `sph2cart`.

`[x, y, z] = sph2cart (theta, phi, r)` [Function File]

Transform spherical to cartesian coordinates. x , y and z must be of same shape. θ describes the angle relative to the x -axis. ϕ is the angle relative to the xy -plane. r is the distance to the origin $(0, 0, 0)$.

See also: `pol2cart`, `cart2pol`, `cart2sph`.

18.7 Mathematical Constants

`I (x)` [Built-in Function]

`I (n, m)` [Built-in Function]

`I (n, m, k, ...)` [Built-in Function]

`I (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all equal to the pure imaginary unit, defined as $\sqrt{-1}$. Since `I` (also `i`, `J`, and `J`) is a function, you can use the name(s) for other purposes.

`Inf (x)` [Built-in Function]
`Inf (n, m)` [Built-in Function]
`Inf (n, m, k, ...)` [Built-in Function]
`Inf (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all Infinity. The arguments are handled the same as the arguments for `eye`. The optional argument *class* may be either `"single"` or `"double"`. The default is `"double"`.

`NaN (x)` [Built-in Function]
`NaN (n, m)` [Built-in Function]
`NaN (n, m, k, ...)` [Built-in Function]
`NaN (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all NaN (Not a Number). The value NaN is the result of an operation like $0/0$, or $\infty - \infty$, or any operation with a NaN.

Note that NaN always compares not equal to NaN. This behavior is specified by the IEEE standard for floating point arithmetic. To find NaN values, you must use the `isnan` function.

The arguments are handled the same as the arguments for `eye`. The optional argument *class* may be either `"single"` or `"double"`. The default is `"double"`.

`pi (x)` [Built-in Function]
`pi (n, m)` [Built-in Function]
`pi (n, m, k, ...)` [Built-in Function]
`pi (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all equal to the ratio of the circumference of a circle to its diameter. Internally, `pi` is computed as `'4.0 * atan (1.0)'`.

`e (x)` [Built-in Function]
`e (n, m)` [Built-in Function]
`e (n, m, k, ...)` [Built-in Function]
`e (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all equal to the base of natural logarithms. The constant e satisfies the equation $\log(e) = 1$.

`eps (x)` [Built-in Function]
`eps (n, m)` [Built-in Function]
`eps (n, m, k, ...)` [Built-in Function]
`eps (... , class)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all `eps`, the machine precision. More precisely, `eps` is the largest relative spacing between any two adjacent numbers in the machine's floating point system. This number is obviously system-dependent. On machines that support 64 bit IEEE floating point arithmetic, `eps` is approximately 2.2204×10^{-16} .

<code>realmax (x)</code>	[Built-in Function]
<code>realmax (n, m)</code>	[Built-in Function]
<code>realmax (n, m, k, ...)</code>	[Built-in Function]
<code>realmax (... , class)</code>	[Built-in Function]

Return a matrix or N-dimensional array whose elements are all equal to the largest floating point number that is representable. The actual value is system-dependent. On machines that support 64-bit IEEE floating point arithmetic, **realmax** is approximately 1.7977×10^{308} .

See also: `realmin`.

<code>realmin (x)</code>	[Built-in Function]
<code>realmin (n, m)</code>	[Built-in Function]
<code>realmin (n, m, k, ...)</code>	[Built-in Function]
<code>realmin (... , class)</code>	[Built-in Function]

Return a matrix or N-dimensional array whose elements are all equal to the smallest normalized floating point number that is representable. The actual value is system-dependent. On machines that support 64-bit IEEE floating point arithmetic, **realmin** is approximately 2.2251×10^{-308} .

See also: `realmax`.

19 Linear Algebra

This chapter documents the linear algebra functions of Octave. Reference material for many of these functions may be found in Golub and Van Loan, *Matrix Computations, 2nd Ed.*, Johns Hopkins, 1989, and in LAPACK *Users' Guide*, SIAM, 1992.

19.1 Techniques used for Linear Algebra

Octave includes a poly-morphic solver, that selects an appropriate matrix factorization depending on the properties of the matrix itself. Generally, the cost of determining the matrix type is small relative to the cost of factorizing the matrix itself, but in any case the matrix type is cached once it is calculated, so that it is not re-determined each time it is used in a linear equation.

The selection tree for how the linear equation is solve or a matrix inverse is form is given by

1. If the matrix is upper or lower triangular sparse a forward or backward substitution using the LAPACK xTRTRS function, and goto 4.
2. If the matrix is square, hermitian with a real positive diagonal, attempt Cholesky factorization using the LAPACK xPOTRF function.
3. If the Cholesky factorization failed or the matrix is not hermitian with a real positive diagonal, and the matrix is square, factorize using the LAPACK xGETRF function.
4. If the matrix is not square, or any of the previous solvers flags a singular or near singular matrix, find a least squares solution using the LAPACK xGELSS function.

The user can force the type of the matrix with the `matrix_type` function. This overcomes the cost of discovering the type of the matrix. However, it should be noted incorrectly identifying the type of the matrix will lead to unpredictable results, and so `matrix_type` should be used with care.

It should be noted that the test for whether a matrix is a candidate for Cholesky factorization, performed above and by the `matrix_type` function, does not give a certainty that the matrix is Hermitian. However, the attempt to factorize the matrix will quickly flag a non-Hermitian matrix.

19.2 Basic Matrix Functions

```
aa = balance (a, opt) [Loadable Function]
[dd, aa] = balance (a, opt) [Loadable Function]
[cc, dd, aa, bb] = balance (a, b, opt) [Loadable Function]
```

[dd, aa] = balance (a) returns aa = dd \ a * dd. aa is a matrix whose row and column norms are roughly equal in magnitude, and dd = p * d, where p is a permutation matrix and d is a diagonal matrix of powers of two. This allows the equilibration to be computed without roundoff. Results of eigenvalue calculation are typically improved by balancing first.

[cc, dd, aa, bb] = balance (a, b) returns aa = cc*a*dd and bb = cc*b*dd), where aa and bb have non-zero elements of approximately the same magnitude and cc and dd are permuted diagonal matrices as in dd for the algebraic eigenvalue problem.

The eigenvalue balancing option `opt` is selected as follows:

- "N", "n" No balancing; arguments copied, transformation(s) set to identity.
- "P", "p" Permute argument(s) to isolate eigenvalues where possible.
- "S", "s" Scale to improve accuracy of computed eigenvalues.
- "B", "b" Permute and scale, in that order. Rows/columns of `a` (and `b`) that are isolated by permutation are not scaled. This is the default behavior.

Algebraic eigenvalue balancing uses standard LAPACK routines.

Generalized eigenvalue problem balancing uses Ward's algorithm (SIAM Journal on Scientific and Statistical Computing, 1981).

`cond (a)` [Function File]

Compute the (two-norm) condition number of a matrix. `cond (a)` is defined as `norm (a) * norm (inv (a))`, and is computed via a singular value decomposition.

See also: `norm`, `svd`, `rank`.

`[d, rcond] = det (a)` [Loadable Function]

Compute the determinant of `a` using LAPACK. Return an estimate of the reciprocal condition number if requested.

`dmult (a, b)` [Function File]

If `a` is a vector of length `rows (b)`, return `diag (a) * b` (but computed much more efficiently).

`dot (x, y, dim)` [Function File]

Computes the dot product of two vectors. If `x` and `y` are matrices, calculate the dot-product along the first non-singleton dimension. If the optional argument `dim` is given, calculate the dot-product along this dimension.

`lambda = eig (a)` [Loadable Function]

`[v, lambda] = eig (a)` [Loadable Function]

The eigenvalues (and eigenvectors) of a matrix are computed in a several step process which begins with a Hessenberg decomposition, followed by a Schur decomposition, from which the eigenvalues are apparent. The eigenvectors, when desired, are computed by further manipulations of the Schur decomposition.

The eigenvalues returned by `eig` are not ordered.

`g = givens (x, y)` [Loadable Function]

`[c, s] = givens (x, y)` [Loadable Function]

Return a 2×2 orthogonal matrix

$$G = \begin{bmatrix} c & s \\ -s' & c \end{bmatrix}$$

such that

$$G \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}$$

with x and y scalars.

For example,

```
givens (1, 1)
⇒      0.70711  0.70711
      -0.70711  0.70711
```

```
[x, rcond] = inv (a) [Loadable Function]
```

```
[x, rcond] = inverse (a) [Loadable Function]
```

Compute the inverse of the square matrix a . Return an estimate of the reciprocal condition number if requested, otherwise warn of an ill-conditioned matrix if the reciprocal condition number is small.

```
type = matrix_type (a) [Loadable Function]
```

```
a = matrix_type (a, type) [Loadable Function]
```

```
a = matrix_type (a, 'upper', perm) [Loadable Function]
```

```
a = matrix_type (a, 'lower', perm) [Loadable Function]
```

```
a = matrix_type (a, 'banded', nl, nu) [Loadable Function]
```

Identify the matrix type or mark a matrix as a particular type. This allows rapid for solutions of linear equations involving a to be performed. Called with a single argument, `matrix_type` returns the type of the matrix and caches it for future use. Called with more than one argument, `matrix_type` allows the type of the matrix to be defined.

The possible matrix types depend on whether the matrix is full or sparse, and can be one of the following

'unknown' Remove any previously cached matrix type, and mark type as unknown

'full' Mark the matrix as full.

'positive definite'
Full positive definite matrix.

'diagonal' Diagonal Matrix. (Sparse matrices only)

'permuted diagonal'
Permuted Diagonal matrix. The permutation does not need to be specifically indicated, as the structure of the matrix explicitly gives this. (Sparse matrices only)

'upper' Upper triangular. If the optional third argument *perm* is given, the matrix is assumed to be a permuted upper triangular with the permutations defined by the vector *perm*.

'lower' Lower triangular. If the optional third argument *perm* is given, the matrix is assumed to be a permuted lower triangular with the permutations defined by the vector *perm*.

'banded'
 'banded positive definite'
 Banded matrix with the band size of *nl* below the diagonal and *nu* above it. If *nl* and *nu* are 1, then the matrix is tridiagonal and treated with specialized code. In addition the matrix can be marked as positive definite (Sparse matrices only)
 'singular' The matrix is assumed to be singular and will be treated with a minimum norm solution

Note that the matrix type will be discovered automatically on the first attempt to solve a linear equation involving *a*. Therefore `matrix_type` is only useful to give Octave hints of the matrix type. Incorrectly defining the matrix type will result in incorrect results from solutions of linear equations, and so it is entirely the responsibility of the user to correctly indentify the matrix type.

norm (*a*, *p*) [Function File]
 Compute the p-norm of the matrix *a*. If the second argument is missing, *p* = 2 is assumed.
 If *a* is a matrix:
 p = 1 1-norm, the largest column sum of the absolute values of *a*.
 p = 2 Largest singular value of *a*.
 p = Inf Infinity norm, the largest row sum of the absolute values of *a*.
 p = "fro" Frobenius norm of *a*, `sqrt (sum (diag (a' * a)))`.
 If *a* is a vector or a scalar:
 p = Inf `max (abs (a))`.
 p = -Inf `min (abs (a))`.
 other *p*-norm of *a*, `(sum (abs (a) .^ p)) ^ (1/p)`.
See also: `cond`, `svd`.

null (*a*, *tol*) [Function File]
 Return an orthonormal basis of the null space of *a*.
 The dimension of the null space is taken as the number of singular values of *a* not greater than *tol*. If the argument *tol* is missing, it is computed as
 `max (size (a)) * max (svd (a)) * eps`

orth (*a*, *tol*) [Function File]
 Return an orthonormal basis of the range space of *a*.
 The dimension of the range space is taken as the number of singular values of *a* greater than *tol*. If the argument *tol* is missing, it is computed as
 `max (size (a)) * max (svd (a)) * eps`

`pinv (x, tol)` [Loadable Function]

Return the pseudoinverse of x . Singular values less than tol are ignored.

If the second argument is omitted, it is assumed that

`tol = max (size (x)) * sigma_max (x) * eps,`

where `sigma_max (x)` is the maximal singular value of x .

`rank (a, tol)` [Function File]

Compute the rank of a , using the singular value decomposition. The rank is taken to be the number of singular values of a that are greater than the specified tolerance tol . If the second argument is omitted, it is taken to be

`tol = max (size (a)) * sigma(1) * eps;`

where `eps` is machine precision and `sigma(1)` is the largest singular value of a .

`trace (a)` [Function File]

Compute the trace of a , `sum (diag (a))`.

`[r, k] = rref (a, tol)` [Function File]

Returns the reduced row echelon form of a . tol defaults to `eps * max (size (a)) * norm (a, inf)`.

Called with two return arguments, k returns the vector of "bound variables", which are those columns on which elimination has been performed.

19.3 Matrix Factorizations

`chol (a)` [Loadable Function]

Compute the Cholesky factor, r , of the symmetric positive definite matrix a , where $R^T R = A$.

See also: `cholinv`, `chol2inv`.

`cholinv (a)` [Loadable Function]

Use the Cholesky factorization to compute the inverse of the symmetric positive definite matrix a .

See also: `chol`, `chol2inv`.

`chol2inv (u)` [Loadable Function]

Invert a symmetric, positive definite square matrix from its Cholesky decomposition, u . Note that u should be an upper-triangular matrix with positive diagonal elements. `chol2inv (u)` provides `inv (u'*u)` but it is much faster than using `inv`.

See also: `chol`, `cholinv`.

`h = hess (a)` [Loadable Function]

`[p, h] = hess (a)` [Loadable Function]

Compute the Hessenberg decomposition of the matrix a .

The Hessenberg decomposition is usually used as the first step in an eigenvalue computation, but has other applications as well (see Golub, Nash, and Van Loan, IEEE Transactions on Automatic Control, 1979). The Hessenberg decomposition is

$$A = PHP^T$$

where P is a square unitary matrix ($P^H P = I$), and H is upper Hessenberg ($H_{i,j} = 0, \forall i \geq j + 1$).

`[l, u, p] = lu (a)` [Loadable Function]

Compute the LU decomposition of a , using subroutines from LAPACK. The result is returned in a permuted form, according to the optional return value p . For example, given the matrix $a = [1, 2; 3, 4]$,

`[l, u, p] = lu (a)`

returns

`l =`

```
1.00000  0.00000
0.33333  1.00000
```

`u =`

```
3.00000  4.00000
0.00000  0.66667
```

`p =`

```
0  1
1  0
```

The matrix is not required to be square.

`[q, r, p] = qr (a)` [Loadable Function]

Compute the QR factorization of a , using standard LAPACK subroutines. For example, given the matrix $a = [1, 2; 3, 4]$,

`[q, r] = qr (a)`

returns

`q =`

```
-0.31623  -0.94868
-0.94868   0.31623
```

`r =`

```
-3.16228  -4.42719
0.00000  -0.63246
```


The **qr** factorization has applications in the solution of least squares problems

$$\min_x \|Ax - b\|_2$$

for overdetermined systems of equations (i.e., A is a tall, thin matrix). The QR factorization is $QR = A$ where Q is an orthogonal matrix and R is upper triangular. The permuted QR factorization $[q, r, p] = \text{qr}(a)$ forms the QR factorization such that the diagonal entries of r are decreasing in magnitude order. For example, given the matrix $a = [1, 2; 3, 4]$,

$$[q, r, p] = \text{qr}(a)$$

returns

$q =$

$$\begin{bmatrix} -0.44721 & -0.89443 \\ -0.89443 & 0.44721 \end{bmatrix}$$

$r =$

$$\begin{bmatrix} -4.47214 & -3.13050 \\ 0.00000 & 0.44721 \end{bmatrix}$$

$p =$

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

The permuted **qr** factorization $[q, r, p] = \text{qr}(a)$ factorization allows the construction of an orthogonal basis of $\text{span}(a)$.

lambda = qz(a, b) [Loadable Function]

Generalized eigenvalue problem $Ax = sBx$, QZ decomposition. There are three ways to call this function:

1. **lambda = qz(A, B)**

Computes the generalized eigenvalues λ of $(A - sB)$.

2. **[AA, BB, Q, Z, V, W, lambda] = qz(A, B)**

Computes qz decomposition, generalized eigenvectors, and generalized eigenvalues of $(A - sB)$

$$AV = BV \text{diag}(\lambda)$$

$$W^T A = \text{diag}(\lambda) W^T B$$

$$AA = Q^T AZ, BB = Q^T BZ$$

with Q and Z orthogonal (unitary) = I

3. **[AA, BB, Z{, lambda}] = qz(A, B, opt)**

As in form [2], but allows ordering of generalized eigenpairs for (e.g.) solution of discrete time algebraic Riccati equations. Form 3 is not available for complex matrices, and does not compute the generalized eigenvectors V , W , nor the orthogonal matrix Q .

opt for ordering eigenvalues of the GEP pencil. The leading block of the revised pencil contains all eigenvalues that satisfy:

"N"	= unordered (default)
"S"	= small: leading block has all $ \lambda \leq 1$
"B"	= big: leading block has all $ \lambda \geq 1$
"-"	= negative real part: leading block has all eigenvalues in the open left half-plane
"+"	= nonnegative real part: leading block has all eigenvalues in the closed right half-plane

Note: qz performs permutation balancing, but not scaling (see `balance`). Order of output arguments was selected for compatibility with MATLAB

See also: `balance`, `dare`, `eig`, `schur`.

`[aa, bb, q, z] = qzhess (a, b)` [Function File]

Compute the Hessenberg-triangular decomposition of the matrix pencil (a, b) , returning $aa = q * a * z$, $bb = q * b * z$, with q and z orthogonal. For example,

```
[aa, bb, q, z] = qzhess ([1, 2; 3, 4], [5, 6; 7, 8])
⇒ aa = [ -3.02244, -4.41741;  0.92998,  0.69749 ]
⇒ bb = [ -8.60233, -9.99730;  0.00000, -0.23250 ]
⇒  q = [ -0.58124, -0.81373; -0.81373,  0.58124 ]
⇒  z = [ 1, 0; 0, 1 ]
```

The Hessenberg-triangular decomposition is the first step in Moler and Stewart's QZ decomposition algorithm.

Algorithm taken from Golub and Van Loan, *Matrix Computations*, 2nd edition.

`s = schur (a)` [Loadable Function]

`[u, s] = schur (a, opt)` [Loadable Function]

The Schur decomposition is used to compute eigenvalues of a square matrix, and has applications in the solution of algebraic Riccati equations in control (see `are` and `dare`). `schur` always returns $S = U^T A U$ where U is a unitary matrix ($U^T U$ is identity) and S is upper triangular. The eigenvalues of A (and S) are the diagonal elements of S . If the matrix A is real, then the real Schur decomposition is computed, in which the matrix U is orthogonal and S is block upper triangular with blocks of size at most 2×2 along the diagonal. The diagonal elements of S (or the eigenvalues of the 2×2 blocks, when appropriate) are the eigenvalues of A and S .

The eigenvalues are optionally ordered along the diagonal according to the value of `opt`. `opt = "a"` indicates that all eigenvalues with negative real parts should be moved to the leading block of S (used in `are`), `opt = "d"` indicates that all eigenvalues with magnitude less than one should be moved to the leading block of S (used in `dare`), and `opt = "u"`, the default, indicates that no ordering of eigenvalues should occur. The leading k columns of U always span the A -invariant subspace corresponding to the k leading eigenvalues of S .

`s = svd (a)` [Loadable Function]
`[u, s, v] = svd (a)` [Loadable Function]
 Compute the singular value decomposition of a

$$A = USV^H$$

The function `svd` normally returns the vector of singular values. If asked for three return values, it computes U , S , and V . For example,

```
svd (hilb (3))
returns
ans =

    1.4083189
    0.1223271
    0.0026873

and
[u, s, v] = svd (hilb (3))
returns
u =

   -0.82704    0.54745    0.12766
   -0.45986   -0.52829   -0.71375
   -0.32330   -0.64901    0.68867

s =

    1.40832    0.00000    0.00000
    0.00000    0.12233    0.00000
    0.00000    0.00000    0.00269

v =

   -0.82704    0.54745    0.12766
   -0.45986   -0.52829   -0.71375
   -0.32330   -0.64901    0.68867
```

If given a second argument, `svd` returns an economy-sized decomposition, eliminating the unnecessary rows or columns of u or v .

`[housv, beta, zer] = housh (x, j, z)` [Function File]

Computes householder reflection vector `housv` to reflect x to be j th column of identity, i.e., $(I - \beta \text{housv} \cdot \text{housv}')x = e(j)$ inputs x : vector j : index into vector z : threshold for zero (usually should be the number 0) outputs: (see Golub and Van Loan) β : If $\beta = 0$, then no reflection need be applied (zer set to 0) `housv`: householder vector

`[u, h, nu] = krylov (a, v, k, eps1, pflg)` [Function File]

Construct an orthogonal basis u of block Krylov subspace

`[v a*v a^2*v ... a^(k+1)*v]`

Using Householder reflections to guard against loss of orthogonality.

If v is a vector, then h contains the Hessenberg matrix such that $a*u == u*h$. Otherwise, h is meaningless.

The value of nu is the dimension of the span of the krylov subspace (based on $eps1$).

If b is a vector and k is greater than $m-1$, then h contains the Hessenberg decomposition of a .

The optional parameter $eps1$ is the threshold for zero. The default value is $1e-12$.

If the optional parameter $pflg$ is nonzero, row pivoting is used to improve numerical behavior. The default value is 0.

Reference: Hodel and Misra, "Partial Pivoting in the Computation of Krylov Subspaces", to be submitted to Linear Algebra and its Applications

19.4 Functions of a Matrix

`expm (a)` [Loadable Function]

Return the exponential of a matrix, defined as the infinite Taylor series

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots$$

The Taylor series is *not* the way to compute the matrix exponential; see Moler and Van Loan, *Nineteen Dubious Ways to Compute the Exponential of a Matrix*, SIAM Review, 1978. This routine uses Ward's diagonal Padé approximation method with three step preconditioning (SIAM Journal on Numerical Analysis, 1977). Diagonal Padé approximations are rational polynomials of matrices $D_q(a)^{-1}N_q(a)$ whose Taylor series matches the first $2q + 1$ terms of the Taylor series above; direct evaluation of the Taylor series (with the same preconditioning steps) may be desirable in lieu of the Padé approximation when $D_q(a)$ is ill-conditioned.

`logm (a)` [Function File]

Compute the matrix logarithm of the square matrix a . Note that this is currently implemented in terms of an eigenvalue expansion and needs to be improved to be more robust.

`[result, error_estimate] = sqrtm (a)` [Loadable Function]

Compute the matrix square root of the square matrix a .

Ref: Nicholas J. Higham. A new sqrtm for MATLAB. Numerical Analysis Report No. 336, Manchester Centre for Computational Mathematics, Manchester, England, January 1999.

See also: expm, logm, funm.

kron (a, b) [Loadable Function]

Form the kronecker product of two matrices, defined block by block as

$$x = [a(i, j) \ b]$$

For example,

$$\begin{aligned} &\text{kron} (1:4, \text{ones} (3, 1)) \\ \Rightarrow &\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} \end{aligned}$$

x = syl (a, b, c) [Loadable Function]

Solve the Sylvester equation

$$AX + XB + C = 0$$

using standard LAPACK subroutines. For example,

$$\begin{aligned} &\text{syl} ([1, 2; 3, 4], [5, 6; 7, 8], [9, 10; 11, 12]) \\ \Rightarrow &[-0.50000, -0.66667; -0.66667, -0.50000] \end{aligned}$$

20 Nonlinear Equations

Octave can solve sets of nonlinear equations of the form

$$f(x) = 0$$

using the function `fsolve`, which is based on the MINPACK subroutine `hybrd`.

`[x, info, msg] = fsolve (fcn, x0)` [Loadable Function]

Given `fcn`, the name of a function of the form `f (x)` and an initial starting point `x0`, `fsolve` solves the set of equations such that `f(x) == 0`.

If `fcn` is a two-element string array, or a two element cell array containing either the function name or inline or function handle. The first element names the function `f` described above, and the second element names a function of the form `j (x)` to compute the Jacobian matrix with elements

$$J = \frac{\partial f_i}{\partial x_j}$$

You can use the function `fsolve_options` to set optional parameters for `fsolve`.

`fsolve_options (opt, val)` [Loadable Function]

When called with two arguments, this function allows you set options parameters for the function `fsolve`. Given one argument, `fsolve_options` returns the value of the corresponding option. If no arguments are supplied, the names of all the available options and their current values are displayed.

Options include

"tolerance"

Nonnegative relative tolerance.

Here is a complete example. To solve the set of equations

$$\begin{aligned} -2x^2 + 3xy + 4\sin(y) - 6 &= 0 \\ 3x^2 - 2xy^2 + 3\cos(x) + 4 &= 0 \end{aligned}$$

you first need to write a function to compute the value of the given function. For example:

```
function y = f (x)
    y(1) = -2*x(1)^2 + 3*x(1)*x(2) + 4*sin(x(2)) - 6;
    y(2) = 3*x(1)^2 - 2*x(1)*x(2)^2 + 3*cos(x(1)) + 4;
endfunction
```

Then, call `fsolve` with a specified initial condition to find the roots of the system of equations. For example, given the function `f` defined above,

```
[x, info] = fsolve ("f", [1; 2])
```

results in the solution

```
x =
```

```
0.57983
```

```
2.54621
```

```
info = 1
```

A value of `info = 1` indicates that the solution has converged.

The function `perror` may be used to print English messages corresponding to the numeric error codes. For example,

```
perror ("fsolve", 1)
```

```
  + solution converged to requested tolerance
```


21 Sparse Matrices

21.1 The Creation and Manipulation of Sparse Matrices

The size of mathematical problems that can be treated at any particular time is generally limited by the available computing resources. Both, the speed of the computer and its available memory place limitation on the problem size.

There are many classes of mathematical problems which give rise to matrices, where a large number of the elements are zero. In this case it makes sense to have a special matrix type to handle this class of problems where only the non-zero elements of the matrix are stored. Not only does this reduce the amount of memory to store the matrix, but it also means that operations on this type of matrix can take advantage of the a-priori knowledge of the positions of the non-zero elements to accelerate their calculations.

A matrix type that stores only the non-zero elements is generally called sparse. It is the purpose of this document to discuss the basics of the storage and creation of sparse matrices and the fundamental operations on them.

21.1.1 Storage of Sparse Matrices

It is not strictly speaking necessary for the user to understand how sparse matrices are stored. However, such an understanding will help to get an understanding of the size of sparse matrices. Understanding the storage technique is also necessary for those users wishing to create their own oct-files.

There are many different means of storing sparse matrix data. What all of the methods have in common is that they attempt to reduce the complexity and storage given a-priori knowledge of the particular class of problems that will be solved. A good summary of the available techniques for storing sparse matrix is given by Saad¹. With full matrices, knowledge of the point of an element of the matrix within the matrix is implied by its position in the computers memory. However, this is not the case for sparse matrices, and so the positions of the non-zero elements of the matrix must equally be stored.

An obvious way to do this is by storing the elements of the matrix as triplets, with two elements being their position in the array (rows and column) and the third being the data itself. This is conceptually easy to grasp, but requires more storage than is strictly needed.

The storage technique used within Octave is the compressed column format. In this format the position of each element in a row and the data are stored as previously. However, if we assume that all elements in the same column are stored adjacent in the computers memory, then we only need to store information on the number of non-zero elements in each column, rather than their positions. Thus assuming that the matrix has more non-zero elements than there are columns in the matrix, we win in terms of the amount of memory used.

In fact, the column index contains one more element than the number of columns, with the first element always being zero. The advantage of this is a simplification in the code, in that there is no special case for the first or last columns. A short example, demonstrating this in C is.

¹ Youcef Saad "SPARSKIT: A basic toolkit for sparse matrix computation", 1994, <http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps>

```

for (j = 0; j < nc; j++)
  for (i = cidx (j); i < cidx(j+1); i++)
    printf ("non-zero element (%i,%i) is %d\n",
           ridx(i), j, data(i));

```

A clear understanding might be had by considering an example of how the above applies to an example matrix. Consider the matrix

```

1  2  0  0
0  0  0  3
0  0  0  4

```

The non-zero elements of this matrix are

```

(1, 1)  ⇒ 1
(1, 2)  ⇒ 2
(2, 4)  ⇒ 3
(3, 4)  ⇒ 4

```

This will be stored as three vectors *cidx*, *ridx* and *data*, representing the column indexing, row indexing and data respectively. The contents of these three vectors for the above matrix will be

```

cidx = [0, 1, 2, 2, 4]
ridx = [0, 0, 1, 2]
data = [1, 2, 3, 4]

```

Note that this is the representation of these elements with the first row and column assumed to start at zero, while in Octave itself the row and column indexing starts at one. Thus the number of elements in the *i*-th column is given by *cidx* (*i* + 1) - *cidx* (*i*).

Although Octave uses a compressed column format, it should be noted that compressed row formats are equally possible. However, in the context of mixed operations between mixed sparse and dense matrices, it makes sense that the elements of the sparse matrices are in the same order as the dense matrices. Octave stores dense matrices in column major ordering, and so sparse matrices are equally stored in this manner.

A further constraint on the sparse matrix storage used by Octave is that all elements in the rows are stored in increasing order of their row index, which makes certain operations faster. However, it imposes the need to sort the elements on the creation of sparse matrices. Having dis-ordered elements is potentially an advantage in that it makes operations such as concatenating two sparse matrices together easier and faster, however it adds complexity and speed problems elsewhere.

21.1.2 Creating Sparse Matrices

There are several means to create sparse matrix.

Returned from a function

There are many functions that directly return sparse matrices. These include *speye*, *sprand*, *spdiag*, etc.

Constructed from matrices or vectors

The function *sparse* allows a sparse matrix to be constructed from three vectors representing the row, column and data. Alternatively, the function *spconvert*

uses a three column matrix format to allow easy importation of data from elsewhere.

Created and then filled

The function *sparse* or *spalloc* can be used to create an empty matrix that is then filled by the user

From a user binary program

The user can directly create the sparse matrix within an oct-file.

There are several basic functions to return specific sparse matrices. For example the sparse identity matrix, is a matrix that is often needed. It therefore has its own function to create it as *speye* (*n*) or *speye* (*r*, *c*), which creates an *n*-by-*n* or *r*-by-*c* sparse identity matrix.

Another typical sparse matrix that is often needed is a random distribution of random elements. The functions *sprand* and *sprandn* perform this for uniform and normal random distributions of elements. They have exactly the same calling convention, where *sprand* (*r*, *c*, *d*), creates an *r*-by-*c* sparse matrix with a density of filled elements of *d*.

Other functions of interest that directly creates a sparse matrices, are *spdiag* or its generalization *spdiags*, that can take the definition of the diagonals of the matrix and create the sparse matrix that corresponds to this. For example

```
s = spdiag (sparse(randn(1,n)), -1);
```

creates a sparse (*n*+1)-by-(*n*+1) sparse matrix with a single diagonal defined.

spatan2 (*y*, *x*) [Loadable Function]

Compute atan (*Y* / *X*) for corresponding sparse matrix elements of *Y* and *X*. The result is in range -pi to pi.

y = spcumprod (*x*,*dim*) [Loadable Function]

Cumulative product of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise cumulative products).

See also: spcumsum.

y = spcumsum (*x*,*dim*) [Loadable Function]

Cumulative sum of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise cumulative sums).

See also: spcumprod.

spdiag (*v*, *k*) [Loadable Function]

Return a diagonal matrix with the sparse vector *v* on diagonal *k*. The second argument is optional. If it is positive, the vector is placed on the *k*-th super-diagonal. If it is negative, it is placed on the -*k*-th sub-diagonal. The default value of *k* is 0, and the vector is placed on the main diagonal. For example,

```

spdiag ([1, 2, 3], 1)
ans =

Compressed Column Sparse (rows=4, cols=4, nnz=3)
(1 , 2) -> 1
(2 , 3) -> 2
(3 , 4) -> 3

```

Given a matrix argument, instead of a vector, **spdiag** extracts the k -th diagonal of the sparse matrix.

See also: **diag**.

```

[b, c] = spdiags (a) [function File]
b = spdiags (a, c) [function File]
b = spdiags (v, c, a) [function File]
b = spdiags (v, c, m, n) [function File]

```

A generalization of the function **spdiag**. Called with a single input argument, the non-zero diagonals c of A are extracted. With two arguments the diagonals to extract are given by the vector c .

The other two forms of **spdiags** modify the input matrix by replacing the diagonals. They use the columns of v to replace the columns represented by the vector c . If the sparse matrix a is defined then the diagonals of this matrix are replaced. Otherwise a matrix of m by n is created with the diagonals given by v .

Negative values of c representative diagonals below the main diagonal, and positive values of c diagonals above the main diagonal.

For example

```

spdiags (reshape (1:12, 4, 3), [-1 0 1], 5, 4)
⇒
    5 10  0  0
    1  6 11  0
    0  2  7 12
    0  0  3  8
    0  0  0  4

```

```

y = speye (m) [Function File]
y = speye (m, n) [Function File]
y = speye (sz) [Function File]

```

Returns a sparse identity matrix. This is significantly more efficient than **sparse** (**eye** (m))) as the full matrix is not constructed.

Called with a single argument a square matrix of size m by m is created. Otherwise a matrix of m by n is created. If called with a single vector argument, this argument is taken to be the size of the matrix to create.

```

y = spfun (f,x) [Function File]

```

Compute $f(x)$ for the non-zero values of x . This results in a sparse matrix with the same structure as x . The function f can be passed as a string, a function handle or an inline function.

`spmax (x, y, dim)` [Mapping Function]
`[w, iw] = spmax (x)` [Mapping Function]

For a vector argument, return the maximum value. For a matrix argument, return the maximum value from each column, as a row vector, or over the dimension *dim* if defined. For two matrices (or a matrix and scalar), return the pair-wise maximum. Thus,

```
max (max (x))
```

returns the largest element of *x*, and

```
max (2:5, pi)
⇒ 3.1416 3.1416 4.0000 5.0000
```

compares each element of the range 2:5 with *pi*, and returns a row vector of the maximum values.

For complex arguments, the magnitude of the elements are used for comparison.

If called with one input and two output arguments, `max` also returns the first index of the maximum value(s). Thus,

```
[x, ix] = max ([1, 3, 5, 2, 5])
⇒ x = 5
   ix = 3
```

`spmin (x, y, dim)` [Mapping Function]
`[w, iw] = spmin (x)` [Mapping Function]

For a vector argument, return the minimum value. For a matrix argument, return the minimum value from each column, as a row vector, or over the dimension *dim* if defined. For two matrices (or a matrix and scalar), return the pair-wise minimum. Thus,

```
min (min (x))
```

returns the smallest element of *x*, and

```
min (2:5, pi)
⇒ 2.0000 3.0000 3.1416 3.1416
```

compares each element of the range 2:5 with *pi*, and returns a row vector of the minimum values.

For complex arguments, the magnitude of the elements are used for comparison.

If called with one input and two output arguments, `min` also returns the first index of the minimum value(s). Thus,

```
[x, ix] = min ([1, 3, 0, 2, 5])
⇒ x = 0
   ix = 3
```

`y = spones (x)` [Function File]

Replace the non-zero entries of *x* with ones. This creates a sparse matrix with the same structure as *x*.

`y = spprod (x,dim)` [Loadable Function]
 Product of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise products).

See also: spsum, spsumsq.

`sprand (m, n, d)` [Function File]

`sprand (s)` [Function File]

Generate a random sparse matrix. The size of the matrix will be *m* by *n*, with a density of values given by *d*. *d* should be between 0 and 1. Values will be uniformly distributed between 0 and 1.

Note: sometimes the actual density may be a bit smaller than *d*. This is unlikely to happen for large really sparse matrices.

If called with a single matrix argument, a random sparse matrix is generated wherever the matrix *S* is non-zero.

See also: sprandn.

`sprandn (m, n, d)` [Function File]

`sprandn (s)` [Function File]

Generate a random sparse matrix. The size of the matrix will be *m* by *n*, with a density of values given by *d*. *d* should be between 0 and 1. Values will be normally distributed with mean of zero and variance 1.

Note: sometimes the actual density may be a bit smaller than *d*. This is unlikely to happen for large really sparse matrices.

If called with a single matrix argument, a random sparse matrix is generated wherever the matrix *S* is non-zero.

See also: sprand.

`sprandsym (n, d)` [Function File]

`sprandsym (s)` [Function File]

Generate a symmetric random sparse matrix. The size of the matrix will be *n* by *n*, with a density of values given by *d*. *d* should be between 0 and 1. Values will be normally distributed with mean of zero and variance 1.

Note: sometimes the actual density may be a bit smaller than *d*. This is unlikely to happen for large really sparse matrices.

If called with a single matrix argument, a random sparse matrix is generated wherever the matrix *S* is non-zero in its lower triangular part.

See also: sprand, sprandn.

`y = spsum (x,dim)` [Loadable Function]

Sum of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise sum).

See also: spprod, spsumsq.

`y = spsumsq (x,dim)` [Loadable Function]
 Sum of squares of elements along dimension *dim*. If *dim* is omitted, it defaults to 1 (column-wise sum of squares). This function is equivalent to computing

```
spsum (x .* spconj (x), dim)
```

but it uses less memory and avoids calling `spconj` if *x* is real.

See also: `spprod`, `spsum`.

The recommended way for the user to create a sparse matrix, is to create two vectors containing the row and column index of the data and a third vector of the same size containing the data to be stored. For example

```
ri = ci = d = [];  
for j = 1:c  
    ri = [ri; randperm(r)(1:n)'];  
    ci = [ci; j*ones(n,1)];  
    d = [d; rand(n,1)];  
endfor  
s = sparse (ri, ci, d, r, c);
```

creates an *r*-by-*c* sparse matrix with a random distribution of *n* ($< r$) elements per column. The elements of the vectors do not need to be sorted in any particular order as Octave will sort them prior to storing the data. However, pre-sorting the data will make the creation of the sparse matrix faster.

The function `spconvert` takes a three or four column real matrix. The first two columns represent the row and column index respectively and the third and four columns, the real and imaginary parts of the sparse matrix. The matrix can contain zero elements and the elements can be sorted in any order. Adding zero elements is a convenient way to define the size of the sparse matrix. For example

```
s = spconvert ([1 2 3 4; 1 3 4 4; 1 2 3 0]')  
⇒ Compressed Column Sparse (rows=4, cols=4, nnz=3)  
    (1 , 1) -> 1  
    (2 , 3) -> 2  
    (3 , 4) -> 3
```

An example of creating and filling a matrix might be

```
k = 5;  
nz = r * k;  
s = spalloc (r, c, nz)  
for j = 1:c  
    idx = randperm (r);  
    s (:, j) = [zeros(r - k, 1); ...  
               rand(k, 1)] (idx);  
endfor
```

It should be noted, that due to the way that the Octave assignment functions are written that the assignment will reallocate the memory used by the sparse matrix at each iteration of the above loop. Therefore the `spalloc` function ignores the *nz* argument and does not preassign the memory for the matrix. Therefore, it is vitally important that code using

to above structure should be vectorized as much as possible to minimize the number of assignments and reduce the number of memory allocations.

FM = **full** (*SM*) [Loadable Function]

returns a full storage matrix from a sparse one

See also: sparse.

s = **spalloc** (*r*, *c*, *nz*) [Function File]

Returns an empty sparse matrix of size *r*-by-*c*. As Octave resizes sparse matrices at the first opportunity, so that no additional space is needed, the argument *nz* is ignored. This function is provided only for compatiability reasons.

It should be noted that this means that code like

```
k = 5;
nz = r * k;
s = spalloc (r, c, nz)
for j = 1:c
    idx = randperm (r);
    s (:, j) = [zeros(r - k, 1); rand(k, 1)] (idx);
endfor
```

will reallocate memory at each step. It is therefore vitally important that code like this is vectorized as much as possible.

See also: sparse, nzmax.

s = **sparse** (*a*) [Loadable Function]

Create a sparse matrix from the full matrix *a*. *a* is forced back to a full matrix if resulting matrix is sparse

s = **sparse** (*a*, 1) [Loadable Function]

Create a sparse matrix and convert it back to a full matrix. *a* is forced back to a full matrix if resulting matrix is sparse

s = **sparse** (*i*, *j*, *sv*, *m*, *n*, *nzmax*) [Loadable Function]

Create a sparse matrix given integer index vectors *i* and *j*, a 1-by-*nnz* vector of real or complex values *sv*, overall dimensions *m* and *n* of the sparse matrix. The argument *nzmax* is ignored but accepted for compatability with MATLAB.

Note: if multiple values are specified with the same *i*, *j* indices, the corresponding values in *s* will be added.

The following are all equivalent:

```
s = sparse (i, j, s, m, n)
s = sparse (i, j, s, m, n, "summation")
s = sparse (i, j, s, m, n, "sum")
```

s = **sparse** (*i*, *j*, *s*, *m*, *n*, "unique") [Loadable Function]

Same as above, except that if more than two values are specified for the same *i*, *j* indices, the last specified value will be used.

s = **sparse** (*i*, *j*, *sv*) [Loadable Function]

Uses *m* = max (*i*), *n* = max (*j*)

`s = sparse (m, n)` [Loadable Function]

Equivalent to `sparse ([], [], [], m, n, 0)`

If any of `sv`, `i` or `j` are scalars, they are expanded to have a common size.

See also: `full`.

`x = spconvert (m)` [Function File]

This function converts for a simple sparse matrix format easily produced by other programs into Octave's internal sparse format. The input `x` is either a 3 or 4 column real matrix, containing the row, column, real and imaginary parts of the elements of the sparse matrix. An element with a zero real and imaginary part can be used to force a particular matrix size.

`spfind (x)` [Loadable Function]

`spfind (x, n)` [Loadable Function]

`spfind (x, n, direction)` [Loadable Function]

`[i, j, v spfind (...)]` [Loadable Function]

A sparse version of the `find` function. Please see the `find` for details of its use.

Note that this function is particularly useful for sparse matrices, as it extracts the non-zero elements as vectors, which can then be used to create the original matrix.

For example,

```
sz = size(a);
[i, j, v] = spfind (a);
b = sparse(i, j, v, sz(1), sz(2));
```

See also: `sparse`.

The above problem can be avoided in oct-files. However, the construction of a sparse matrix from an oct-file is more complex than can be discussed in this brief introduction, and you are referred to chapter [Appendix A \[Dynamically Linked Functions\]](#), page 461, to have a full description of the techniques involved.

21.1.3 Finding out Information about Sparse Matrices

There are a number of functions that allow information concerning sparse matrices to be obtained. The most basic of these is `issparse` that identifies whether a particular Octave object is in fact a sparse matrix.

Another very basic function is `nnz` that returns the number of non-zero entries there are in a sparse matrix, while the function `nzmax` returns the amount of storage allocated to the sparse matrix. Note that Octave tends to crop unused memory at the first opportunity for sparse objects. There are some cases of user created sparse objects where the value returned by `nzmax` will not be the same as `nnz`, but in general they will give the same result. The function `spstats` returns some basic statistics on the columns of a sparse matrix including the number of elements, the mean and the variance of each column.

`issparse (expr)` [Loadable Function]

Return 1 if the value of the expression `expr` is a sparse matrix.

`scalar = nnz (a)` [Built-in Function]

Returns the number of non zero elements in *a*.

See also: `sparse`.

`nonzeros (s)` [Function File]

Returns a vector of the non-zero values of the sparse matrix *s*.

`scalar = nzmax (SM)` [Built-in Function]

Return the amount of storage allocated to the sparse matrix *SM*. Note that Octave tends to crop unused memory at the first opportunity for sparse objects. There are some cases of user created sparse objects where the value returned by *nzmax* will not be the same as *nnz*, but in general they will give the same result.

See also: `sparse`, `spalloc`.

`[count, mean, var] = spstats (s)` [Function File]

`[count, mean, var] = spstats (s, j)` [Function File]

Return the stats for the non-zero elements of the sparse matrix *s*. *count* is the number of non-zeros in each column, *mean* is the mean of the non-zeros in each column, and *var* is the variance of the non-zeros in each column.

Called with two input arguments, if *s* is the data and *j* is the bin number for the data, compute the stats for each bin. In this case, bins can contain data values of zero, whereas with `spstats (s)` the zeros may disappear.

When solving linear equations involving sparse matrices Octave determines the means to solve the equation based on the type of the matrix as discussed in [Section 21.2 \[Sparse Linear Algebra\]](#), [page 265](#). Octave probes the matrix type when the `div (/)` or `ldiv (\)` operator is first used with the matrix and then caches the type. However the *matrix_type* function can be used to determine the type of the sparse matrix prior to use of the `div` or `ldiv` operators. For example

```
a = tril (sprandn(1024, 1024, 0.02), -1) ...
    + speye(1024);
matrix_type (a);
ans = Lower
```

show that Octave correctly determines the matrix type for lower triangular matrices. *matrix_type* can also be used to force the type of a matrix to be a particular type. For example

```
a = matrix_type (tril (sprandn (1024, ...
    1024, 0.02), -1) + speye(1024), 'Lower');
```

This allows the cost of determining the matrix type to be avoided. However, incorrectly defining the matrix type will result in incorrect results from solutions of linear equations, and so it is entirely the responsibility of the user to correctly identify the matrix type

There are several graphical means of finding out information about sparse matrices. The first is the *spy* command, which displays the structure of the non-zero elements of the matrix. See [Figure 21.1](#), for an example of the use of *spy*. More advanced graphical information can be obtained with the *treeplot*, *etreeplot* and *gplot* commands.

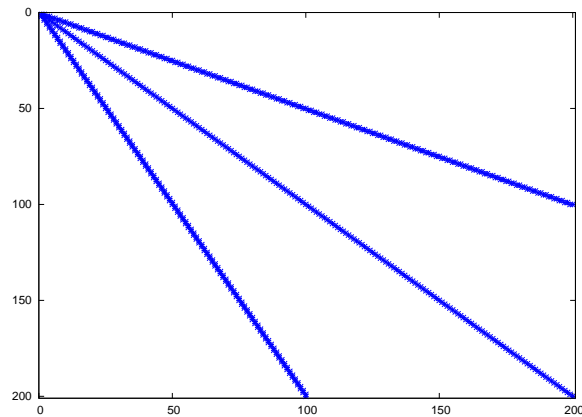


Figure 21.1: Structure of simple sparse matrix.

One use of sparse matrices is in graph theory, where the interconnections between nodes is represented as an adjacency matrix. That is, if the i -th node in a graph is connected to the j -th node. Then the ij -th node (and in the case of undirected graphs the ji -th node) of the sparse adjacency matrix is non-zero. If each node is then associated with a set of co-ordinates, then the *gplot* command can be used to graphically display the interconnections between nodes.

As a trivial example of the use of *gplot*, consider the example

```
A = sparse([2,6,1,3,2,4,3,5,4,6,1,5],
           [1,1,2,2,3,3,4,4,5,5,6,6],1,6,6);
xy = [0,4,8,6,4,2;5,0,5,7,5,7]';
gplot(A,xy)
```

which creates an adjacency matrix **A** where node 1 is connected to nodes 2 and 6, node 2 with nodes 1 and 3, etc. The co-ordinates of the nodes are given in the n -by-2 matrix **xy**. See [Figure 21.2](#).

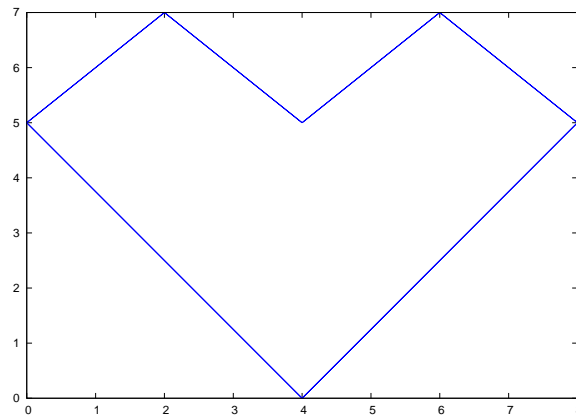


Figure 21.2: Simple use of the *gplot* command.

The dependencies between the nodes of a Cholesky factorization can be calculated in linear time without explicitly needing to calculate the Cholesky factorization by the `etree` command. This command returns the elimination tree of the matrix and can be displayed graphically by the command `treeplot(etree(A))` if **A** is symmetric or `treeplot(etree(A+A'))` otherwise.

`spy (x)` [Function File]
`spy (... , markersize)` [Function File]
`spy (... , LineStyle)` [Function File]

Plot the sparsity pattern of the sparse matrix *x*. If the argument *markersize* is given as an scalar value, it is used to determine the point size in the plot. If the string *LineStyle* is given it is passed to `plot` and determines the appearance of the plot.

See also: `plot`.

`p = etree (s)` [Loadable Function]
`p = etree (s, typ)` [Loadable Function]
`[p, q] = etree (s, typ)` [Loadable Function]

Returns the elimination tree for the matrix *s*. By default *s* is assumed to be symmetric and the symmetric elimination tree is returned. The argument *typ* controls whether a symmetric or column elimination tree is returned. Valid values of *typ* are 'sym' or 'col', for symmetric or column elimination tree respectively

Called with a second argument, *etree* also returns the postorder permutations on the tree.

`etreeplot (tree)` [Function File]
`etreeplot (tree, node_style, edge_style)` [Function File]

Plot the elimination tree of the matrix *s* or *s+s'* if *s* is non-symmetric. The optional parameters *line_style* and *edge_style* define the output style.

See also: `treeplot`, `gplot`.

`gplot (a, xy)` [Function File]
`gplot (a, xy, line_style)` [Function File]
`[x, y] = gplot (a, xy)` [Function File]

Plot a graph defined by *A* and *xy* in the graph theory sense. *A* is the adjacency matrix of the array to be plotted and *xy* is an *n*-by-2 matrix containing the coordinates of the nodes of the graph.

The optional parameter *line_style* defines the output style for the plot. Called with no output arguments the graph is plotted directly. Otherwise, return the coordinates of the plot in *x* and *y*.

See also: `treeplot`, `etreeplot`, `spy`.

`treeplot (Tree)` [Function File]
`treeplot (Tree, LineStyle, EdgeStyle)` [Function File]

Produces a graph of tree or forest. The first argument is vector of predecessors, optional parameters *LineStyle* and *EdgeStyle* define the output style. The complexity of the algorithm is $O(n)$ in terms of its time and memory requirements.

See also: `etreeplot`, `gplot`.

21.1.4 Basic Operators and Functions on Sparse Matrices

21.1.4.1 Sparse Functions

An important consideration in the use of the sparse functions of Octave is that many of the internal functions of Octave, such as *diag*, can not accept sparse matrices as an input. The sparse implementation in Octave therefore uses the *dispatch* function to overload the normal Octave functions with equivalent functions that work with sparse matrices. However, at any time the sparse matrix specific version of the function can be used by explicitly calling its function name.

The table below lists all of the sparse functions of Octave. Note that in this specific sparse forms of the functions are typically the same as the general versions with a *sp* prefix. In the table below, and the rest of this article the specific sparse versions of the functions are used.

Generate sparse matrices:

spalloc, *spdiags*, *speye*, *sprand*, *sprandn*, *sprandsym*

Sparse matrix conversion:

full, *sparse*, *spconvert*, *spfind*

Manipulate sparse matrices

issparse, *nnz*, *nonzeros*, *nzmax*, *spfun*, *spones*, *spy*

Graph Theory:

etree, *etreeplot*, *gplot*, *treeplot*

Sparse matrix reordering:

ccolamd, *colamd*, *colperm*, *csymamd*, *dmperm*, *symamd*, *randperm*, *symrcm*

Linear algebra:

matrix_type, *spchol*, *cpcholinv*, *spchol2inv*, *spdet*, *spinv*, *spkron*, *splchol*, *splu*, *spqr*, *normest*, *sprank*

Iterative techniques:

luinc, *pcg*, *pcr*

Miscellaneous:

spparms, *symbfact*, *spstats*, *spprod*, *spcumsum*, *spsum*, *spsumsq*, *spmin*, *spmax*, *spatan2*, *spdiag*

In addition all of the standard Octave mapper functions (ie. basic math functions that take a single argument) such as *abs*, etc can accept sparse matrices. The reader is referred to the documentation supplied with these functions within Octave itself for further details.

21.1.4.2 The Return Types of Operators and Functions

The two basic reasons to use sparse matrices are to reduce the memory usage and to not have to do calculations on zero elements. The two are closely related in that the computation time on a sparse matrix operator or function is roughly linear with the number of non-zero elements.

Therefore, there is a certain density of non-zero elements of a matrix where it no longer makes sense to store it as a sparse matrix, but rather as a full matrix. For this reason operators and functions that have a high probability of returning a full matrix will always return one. For example adding a scalar constant to a sparse matrix will almost always make it a full matrix, and so the example

```
speye(3) + 0
⇒  1  0  0
   0  1  0
   0  0  1
```

returns a full matrix as can be seen. Additionally all sparse functions test the amount of memory occupied by the sparse matrix to see if the amount of storage used is larger than the amount used by the full equivalent. Therefore `speye(2) * 1` will return a full matrix as the memory used is smaller for the full version than the sparse version.

As all of the mixed operators and functions between full and sparse matrices exist, in general this does not cause any problems. However, one area where it does cause a problem is where a sparse matrix is promoted to a full matrix, where subsequent operations would resparsify the matrix. Such cases are rare, but can be artificially created, for example `(fliplr(speye(3)) + speye(3)) - speye(3)` gives a full matrix when it should give a sparse one. In general, where such cases occur, they impose only a small memory penalty.

There is however one known case where this behavior of Octave's sparse matrices will cause a problem. That is in the handling of the *diag* function. Whether *diag* returns a sparse or full matrix depending on the type of its input arguments. So

```
a = diag (sparse([1,2,3]), -1);
```

should return a sparse matrix. To ensure this actually happens, the *sparse* function, and other functions based on it like *speye*, always returns a sparse matrix, even if the memory used will be larger than its full representation.

21.1.4.3 Mathematical Considerations

The attempt has been made to make sparse matrices behave in exactly the same manner as their full counterparts. However, there are certain differences and especially differences with other products sparse implementations.

Firstly, the `./` and `.^` operators must be used with care. Consider what the examples

```
s = speye (4);
a1 = s .^ 2;
a2 = s .^ s;
a3 = s .^ -2;
a4 = s ./ 2;
a5 = 2 ./ s;
a6 = s ./ s;
```

will give. The first example of s raised to the power of 2 causes no problems. However s raised element-wise to itself involves a large number of terms $0.^0$ which is 1. There $s.^s$ is a full matrix.

Likewise $s.^{-2}$ involves terms like $0.^{-2}$ which is infinity, and so $s.^{-2}$ is equally a full matrix.

For the `./` operator $s ./ 2$ has no problems, but $2 ./ s$ involves a large number of infinity terms as well and is equally a full matrix. The case of $s ./ s$ involves terms like $0 ./ 0$ which is a NaN and so this is equally a full matrix with the zero elements of s filled with NaN values.

The above behavior is consistent with full matrices, but is not consistent with sparse implementations in other products.

A particular problem of sparse matrices comes about due to the fact that as the zeros are not stored, the sign-bit of these zeros is equally not stored. In certain cases the sign-bit of zero is important. For example

```
a = 0 ./ [-1, 1; 1, -1];
b = 1 ./ a
⇒ -Inf      Inf
   Inf      -Inf
c = 1 ./ sparse (a)
⇒  Inf      Inf
   Inf      Inf
```

To correct this behavior would mean that zero elements with a negative sign-bit would need to be stored in the matrix to ensure that their sign-bit was respected. This is not done at this time, for reasons of efficiency, and so the user is warned that calculations where the sign-bit of zero is important must not be done using sparse matrices.

In general any function or operator used on a sparse matrix will result in a sparse matrix with the same or a larger number of non-zero elements than the original matrix. This is particularly true for the important case of sparse matrix factorizations. The usual way to address this is to reorder the matrix, such that its factorization is sparser than the factorization of the original matrix. That is the factorization of $L * U = P * S * Q$ has sparser terms L and U than the equivalent factorization $L * U = S$.

Several functions are available to reorder depending on the type of the matrix to be factorized. If the matrix is symmetric positive-definite, then *symamd* or *csymamd* should be used. Otherwise *colamd* or *ccolamd* should be used. For completeness the reordering functions *colperm* and *randperm* are also available.

See [Figure 21.3](#), for an example of the structure of a simple positive definite matrix.

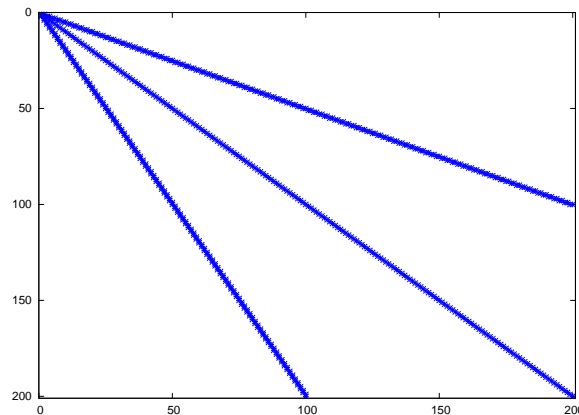


Figure 21.3: Structure of simple sparse matrix.

The standard Cholesky factorization of this matrix, can be obtained by the same command that would be used for a full matrix. This can be visualized with the command `r = chol(A); spy(r);`. The original matrix had 598 non-zero terms, while this Cholesky factorization has 10200, with only half of the symmetric matrix being stored. This is a significant level of fill in, and although not an issue for such a small test case, can represents a large overhead in working with other sparse matrices.

The appropriate sparsity preserving permutation of the original matrix is given by *symamd* and the factorization using this reordering can be visualized using the command `q = symamd(A); r = chol(A(q,q)); spy(r)`. This gives 399 non-zero terms which is a significant improvement.

The Cholesky factorization itself can be used to determine the appropriate sparsity preserving reordering of the matrix during the factorization, In that case this might be obtained with three return arguments as `r[r, p, q] = chol(A); spy(r)`.

In the case of an asymmetric matrix, the appropriate sparsity preserving permutation is *colamd* and the factorization using this reordering can be visualized using the command `q = colamd(A); [l, u, p] = lu(A(:,q)); spy(l+u)`.

Finally, Octave implicitly reorders the matrix when using the *div* (/) and *ldiv* (\) operators, and so no the user does not need to explicitly reorder the matrix to maximize performance.

<code>p = ccolamd (s)</code>	[Loadable Function]
<code>p = ccolamd (s, knobs)</code>	[Loadable Function]
<code>p = ccolamd (s, knobs, cmember)</code>	[Loadable Function]
<code>[p, stats] = ccolamd (...)</code>	[Loadable Function]

Constrained column approximate minimum degree permutation. `p = ccolamd (s)` returns the column approximate minimum degree permutation vector for the sparse matrix *s*. For a non-symmetric matrix *s*, `s(:,p)` tends to have sparser LU factors than *s*. `chol (s(:,p)'*s(:,p))` also tends to be sparser than `chol (s'*s)`. `p = ccolamd (s,1)` optimizes the ordering for `lu (s(:,p))`. The ordering is followed by a column elimination tree post-ordering.

knobs is an optional one- to five-element input vector, with a default value of [0 10 10 1 0] if not present or empty. Entries not present are set to their defaults.

knobs(1) if nonzero, the ordering is optimized for `lu(S(:,p))`. It will be a poor ordering for `chol(s(:,p)'*s(:,p))`. This is the most important knob for `ccolamd`.

knob(2) if *s* is *m*-by-*n*, rows with more than `max(16, knobs(2)*sqrt(n))` entries are ignored.

knob(3) columns with more than `max(16, knobs(3)*sqrt(min(m,n)))` entries are ignored and ordered last in the output permutation (subject to the *cmember* constraints).

knob(4) if nonzero, aggressive absorption is performed.

knob(5) if nonzero, statistics and knobs are printed.

cmember is an optional vector of length *n*. It defines the constraints on the column ordering. If *cmember*(*j*) = *c*, then column *j* is in constraint set *c* (*c* must be in the range 1 to *n*). In the output permutation *p*, all columns in set 1 appear first, followed by all columns in set 2, and so on. *cmember* = `ones(1,n)` if not present or empty. `ccolamd(s, [], 1:n)` returns `1:n`

`p = ccolamd(s)` is about the same as `p = colamd(s)`. *knobs* and its default values differ. `colamd` always does aggressive absorption, and it finds an ordering suitable for both `lu(s(:,p))` and `chol(S(:,p)'*s(:,p))`; it cannot optimize its ordering for `lu(s(:,p))` to the extent that `ccolamd(s,1)` can.

stats is an optional 20-element output vector that provides data about the ordering and the validity of the input matrix *s*. Ordering statistics are in *stats*(1:3). *stats*(1) and *stats*(2) are the number of dense or empty rows and columns ignored by CCOLAMD and *stats*(3) is the number of garbage collections performed on the internal data structure used by CCOLAMD (roughly of size `2.2 * nnz(s) + 4 * m + 7 * n` integers).

stats(4:7) provide information if CCOLAMD was able to continue. The matrix is OK if *stats*(4) is zero, or 1 if invalid. *stats*(5) is the rightmost column index that is unsorted or contains duplicate entries, or zero if no such column exists. *stats*(6) is the last seen duplicate or out-of-order row index in the column index given by *stats*(5), or zero if no such row index exists. *stats*(7) is the number of duplicate or out-of-order row indices. *stats*(8:20) is always zero in the current version of CCOLAMD (reserved for future use).

The authors of the code itself are S. Larimore, T. Davis (Uni of Florida) and S. Rajamanickam in collaboration with J. Bilbert and E. Ng. Supported by the National Science Foundation (DMS-9504974, DMS-9803599, CCR-0203270), and a grant from Sandia National Lab. See <http://www.cise.ufl.edu/research/sparse> for `ccolamd`, `csymamd`, `amd`, `colamd`, `symamd`, and other related orderings.

See also: `colamd`, `csymamd`.

<code>p = colamd(s)</code>	[Loadable Function]
<code>p = colamd(s, knobs)</code>	[Loadable Function]

```
[p, stats] = colamd (s) [Loadable Function]
[p, stats] = colamd (s, knobs) [Loadable Function]
```

Column approximate minimum degree permutation. `p = colamd (s)` returns the column approximate minimum degree permutation vector for the sparse matrix `s`. For a non-symmetric matrix `s`, `s (:,p)` tends to have sparser LU factors than `s`. The Cholesky factorization of `s (:,p)' * s (:,p)` also tends to be sparser than that of `s' * s`.

`knobs` is an optional one- to three-element input vector. If `s` is `m`-by-`n`, then rows with more than `max(16, knobs(1)*sqrt(n))` entries are ignored. Columns with more than `max(16, knobs(2)*sqrt(min(m,n)))` entries are removed prior to ordering, and ordered last in the output permutation `p`. Only completely dense rows or columns are removed if `knobs (1)` and `knobs (2)` are `< 0`, respectively. If `knobs (3)` is nonzero, `stats` and `knobs` are printed. The default is `knobs = [10 10 0]`. Note that `knobs` differs from earlier versions of `colamd`.

`stats` is an optional 20-element output vector that provides data about the ordering and the validity of the input matrix `s`. Ordering statistics are in `stats (1:3)`. `stats (1)` and `stats (2)` are the number of dense or empty rows and columns ignored by COLAMD and `stats (3)` is the number of garbage collections performed on the internal data structure used by COLAMD (roughly of size `2.2 * nnz(s) + 4 * m + 7 * n` integers).

Octave built-in functions are intended to generate valid sparse matrices, with no duplicate entries, with ascending row indices of the nonzeros in each column, with a non-negative number of entries in each column (!) and so on. If a matrix is invalid, then COLAMD may or may not be able to continue. If there are duplicate entries (a row index appears two or more times in the same column) or if the row indices in a column are out of order, then COLAMD can correct these errors by ignoring the duplicate entries and sorting each column of its internal copy of the matrix `s` (the input matrix `s` is not repaired, however). If a matrix is invalid in other ways then COLAMD cannot continue, an error message is printed, and no output arguments (`p` or `stats`) are returned. COLAMD is thus a simple way to check a sparse matrix to see if it's valid.

`stats (4:7)` provide information if COLAMD was able to continue. The matrix is OK if `stats (4)` is zero, or 1 if invalid. `stats (5)` is the rightmost column index that is unsorted or contains duplicate entries, or zero if no such column exists. `stats (6)` is the last seen duplicate or out-of-order row index in the column index given by `stats (5)`, or zero if no such row index exists. `stats (7)` is the number of duplicate or out-of-order row indices. `stats (8:20)` is always zero in the current version of COLAMD (reserved for future use).

The ordering is followed by a column elimination tree post-ordering.

The authors of the code itself are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. (see <http://www.cise.ufl.edu/research/sparse/colamd>)

See also: `colperm`, `symamd`.

`p = colperm (s)` [Function File]

Returns the column permutations such that the columns of $s(:, p)$ are ordered in terms of increase number of non-zero elements. If s is symmetric, then p is chosen such that $s(p, p)$ orders the rows and columns with increasing number of non zeros elements.

`p = csymamd (s)` [Loadable Function]

`p = csymamd (s, knobs)` [Loadable Function]

`p = csymamd (s, knobs, cmember)` [Loadable Function]

`[p, stats] = csymamd (...)` [Loadable Function]

For a symmetric positive definite matrix s , returns the permutation vector p such that $s(p,p)$ tends to have a sparser Cholesky factor than s . Sometimes `csymamd` works well for symmetric indefinite matrices too. The matrix s is assumed to be symmetric; only the strictly lower triangular part is referenced. s must be square. The ordering is followed by an elimination tree post-ordering.

`knobs` is an optional one- to three-element input vector, with a default value of `[10 1 0]` if present or empty. Entries not present are set to their defaults.

`knobs(1)` If s is n -by- n , then rows and columns with more than $\max(16, \text{knobs}(1)*\sqrt{n})$ entries are ignored, and ordered last in the output permutation (subject to the `cmember` constraints).

`knobs(2)` If nonzero, aggressive absorption is performed.

`knobs(3)` If nonzero, statistics and knobs are printed.

`cmember` is an optional vector of length n . It defines the constraints on the ordering. If `cmember(j) = s`, then row/column j is in constraint set c (c must be in the range 1 to n). In the output permutation p , rows/columns in set 1 appear first, followed by all rows/columns in set 2, and so on. `cmember = ones(1,n)` if not present or empty. `csymamd(s, [], 1:n)` returns `1:n`.

`p = csymamd(s)` is about the same as `p = symamd(s)`. `knobs` and its default values differ.

`stats(4:7)` provide information if CCOLAMD was able to continue. The matrix is OK if `stats(4)` is zero, or 1 if invalid. `stats(5)` is the rightmost column index that is unsorted or contains duplicate entries, or zero if no such column exists. `stats(6)` is the last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or zero if no such row index exists. `stats(7)` is the number of duplicate or out-of-order row indices. `stats(8:20)` is always zero in the current version of CCOLAMD (reserved for future use).

The authors of the code itself are S. Larimore, T. Davis (Uni of Florida) and S. Rajamanickam in collaboration with J. Bilbert and E. Ng. Supported by the National Science Foundation (DMS-9504974, DMS-9803599, CCR-0203270), and a grant from Sandia National Lab. See <http://www.cise.ufl.edu/research/sparse> for `ccolamd`, `csymamd`, `amd`, `colamd`, `symamd`, and other related orderings.

See also: `symamd`, `ccolamd`.

`p = dmperm (s)` [Loadable Function]
`[p, q, r, s] = dmperm (s)` [Loadable Function]

Perform a Dulmage-Mendelsohn permutation on the sparse matrix `s`. With a single output argument `dmperm` performs the row permutations `p` such that `s (p,:)` has no zero elements on the diagonal.

Called with two or more output arguments, returns the row and column permutations, such that `s (p, q)` is in block triangular form. The values of `r` and `s` define the boundaries of the blocks. If `s` is square then `r == s`.

The method used is described in: A. Pothén & C.-J. Fan. Computing the block triangular form of a sparse matrix. ACM Trans. Math. Software, 16(4):303-324, 1990.

See also: `colamd`, `ccolamd`.

`p = symamd (s)` [Loadable Function]
`p = symamd (s, knobs)` [Loadable Function]
`[p, stats] = symamd (s)` [Loadable Function]
`[p, stats] = symamd (s, knobs)` [Loadable Function]

For a symmetric positive definite matrix `s`, returns the permutation vector `p` such that `s (p, p)` tends to have a sparser Cholesky factor than `s`. Sometimes SYMAMD works well for symmetric indefinite matrices too. The matrix `s` is assumed to be symmetric; only the strictly lower triangular part is referenced. `s` must be square.

`knobs` is an optional one- to two-element input vector. If `s` is `n`-by-`n`, then rows and columns with more than `max(16, knobs(1)*sqrt(n))` entries are removed prior to ordering, and ordered last in the output permutation `p`. No rows/columns are removed if `knobs(1) < 0`. If `knobs(2)` is nonzero, `stats` and `knobs` are printed. The default is `knobs = [10 0]`. Note that `knobs` differs from earlier versions of `symamd`.

`stats` is an optional 20-element output vector that provides data about the ordering and the validity of the input matrix `s`. Ordering statistics are in `stats (1:3)`. `stats (1) = stats (2)` is the number of dense or empty rows and columns ignored by SYMAMD and `stats (3)` is the number of garbage collections performed on the internal data structure used by SYMAMD (roughly of size `8.4 * nnz (tril (s, -1)) + 9 * n` integers).

Octave built-in functions are intended to generate valid sparse matrices, with no duplicate entries, with ascending row indices of the nonzeros in each column, with a non-negative number of entries in each column (!) and so on. If a matrix is invalid, then SYMAMD may or may not be able to continue. If there are duplicate entries (a row index appears two or more times in the same column) or if the row indices in a column are out of order, then SYMAMD can correct these errors by ignoring the duplicate entries and sorting each column of its internal copy of the matrix `S` (the input matrix `S` is not repaired, however). If a matrix is invalid in other ways then SYMAMD cannot continue, an error message is printed, and no output arguments (`p` or `stats`) are returned. SYMAMD is thus a simple way to check a sparse matrix to see if it's valid.

`stats (4:7)` provide information if SYMAMD was able to continue. The matrix is OK if `stats (4)` is zero, or 1 if invalid. `stats (5)` is the rightmost column index

that is unsorted or contains duplicate entries, or zero if no such column exists. `stats` (6) is the last seen duplicate or out-of-order row index in the column index given by `stats` (5), or zero if no such row index exists. `stats` (7) is the number of duplicate or out-of-order row indices. `stats` (8:20) is always zero in the current version of SYMAMD (reserved for future use).

The ordering is followed by a column elimination tree post-ordering.

The authors of the code itself are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. (see <http://www.cise.ufl.edu/research/sparse/colamd>)

See also: colperm, colamd.

`p = symrcm (S)` [Loadable Function]

Symmetric reverse Cuthill-McKee permutation of S . Return a permutation vector p such that $S(p, p)$ tends to have its diagonal elements closer to the diagonal than S . This is a good preordering for LU or Cholesky factorization of matrices that come from 'long, skinny' problems. It works for both symmetric and asymmetric S .

The algorithm represents a heuristic approach to the NP-complete bandwidth minimization problem. The implementation is based in the descriptions found in

E. Cuthill, J. McKee: Reducing the Bandwidth of Sparse Symmetric Matrices. Proceedings of the 24th ACM National Conference, 157-172 1969, Brandon Press, New Jersey.

Alan George, Joseph W. H. Liu: Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall Series in Computational Mathematics, ISBN 0-13-165274-5, 1981.

See also: colperm, colamd, symamd.

21.2 Linear Algebra on Sparse Matrices

Octave includes a poly-morphic solver for sparse matrices, where the exact solver used to factorize the matrix, depends on the properties of the sparse matrix itself. Generally, the cost of determining the matrix type is small relative to the cost of factorizing the matrix itself, but in any case the matrix type is cached once it is calculated, so that it is not re-determined each time it is used in a linear equation.

The selection tree for how the linear equation is solve is

1. If the matrix is diagonal, solve directly and goto 8
2. If the matrix is a permuted diagonal, solve directly taking into account the permutations. Goto 8
3. If the matrix is square, banded and if the band density is less than that given by `spparms` ("bandden") continue, else goto 4.
 - a. If the matrix is tridiagonal and the right-hand side is not sparse continue, else goto 3b.
 1. If the matrix is hermitian, with a positive real diagonal, attempt Cholesky factorization using LAPACK xPTSV.

2. If the above failed or the matrix is not hermitian with a positive real diagonal use Gaussian elimination with pivoting using LAPACK xGTSV, and goto 8.
- b. If the matrix is hermitian with a positive real diagonal, attempt Cholesky factorization using LAPACK xPBTRF.
- c. if the above failed or the matrix is not hermitian with a positive real diagonal use Gaussian elimination with pivoting using LAPACK xGBTRF, and goto 8.
4. If the matrix is upper or lower triangular perform a sparse forward or backward substitution, and goto 8
5. If the matrix is a upper triangular matrix with column permutations or lower triangular matrix with row permutations, perform a sparse forward or backward substitution, and goto 8
6. If the matrix is square, hermitian with a real positive diagonal, attempt sparse Cholesky factorization using CHOLMOD.
7. If the sparse Cholesky factorization failed or the matrix is not hermitian with a real positive diagonal, and the matrix is square, factorize using UMFPACK.
8. If the matrix is not square, or any of the previous solvers flags a singular or near singular matrix, find a minimum norm solution using CXSPARSE².

The band density is defined as the number of non-zero values in the matrix divided by the number of non-zero values in the matrix. The banded matrix solvers can be entirely disabled by using *spparms* to set **bandden** to 1 (i.e. **spparms ("bandden", 1)**).

The QR solver factorizes the problem with a Dulmage-Mendhelsohn, to seperate the problem into blocks that can be treated as over-determined, multiple well determined blocks, and a final over-determined block. For matrices with blocks of strongly connected nodes this is a big win as LU decomposition can be used for many blocks. It also significantly improves the chance of finding a solution to over-determined problems rather than just returning a vector of NaN's.

All of the solvers above, can calculate an estimate of the condition number. This can be used to detect numerical stability problems in the solution and force a minimum norm solution to be used. However, for narrow banded, triangular or diagonal matrices, the cost of calculating the condition number is significant, and can in fact exceed the cost of factoring the matrix. Therefore the condition number is not calculated in these case, and Octave relies on simpler techniques to detect singular matrices or the underlying LAPACK code in the case of banded matrices.

The user can force the type of the matrix with the **matrix_type** function. This overcomes the cost of discovering the type of the matrix. However, it should be noted incorrectly identifying the type of the matrix will lead to unpredictable results, and so **matrix_type** should be used with care.

[n, c] = normest (a, tol) [Function File]

Estimate the 2-norm of the matrix **a** using a power series analysis. This is typically used for large matrices, where the cost of calculating the **norm (a)** is prohibitive and an approximation to the 2-norm is acceptable.

² CHOLMOD, UMFPACK and CXSPARSE are written by Tim Davis and are available at <http://www.cise.ufl.edu/research/sparse/>

tol is the tolerance to which the 2-norm is calculated. By default *tol* is 1e-6. *c* returns the number of iterations needed for **normest** to converge.

`r = spchol (a)` [Loadable Function]
`[r, p] = spchol (a)` [Loadable Function]
`[r, p, q] = spchol (a)` [Loadable Function]

Compute the Cholesky factor, *r*, of the symmetric positive definite sparse matrix *a*, where $R^T R = A$.

If called with 2 or more outputs *p* is the 0 when *r* is positive definite and *p* is a positive integer otherwise.

If called with 3 outputs then a sparsity preserving row/column permutation is applied to *a* prior to the factorization. That is *r* is the factorization of $a(q,q)$ such that $R^T R = Q A Q^T$.

Note that **splchol** factorizations is faster and use less memory.

See also: **spcholinv**, **spchol2inv**, **splchol**.

`spcholinv (a)` [Loadable Function]

Use the Cholesky factorization to compute the inverse of the sparse symmetric positive definite matrix *a*.

See also: **spchol**, **spchol2inv**.

`spchol2inv (u)` [Loadable Function]

Invert a sparse symmetric, positive definite square matrix from its Cholesky decomposition, *u*. Note that *u* should be an upper-triangular matrix with positive diagonal elements. **chol2inv** (*u*) provides **inv** (*u'***u*) but it is much faster than using **inv**.

See also: **spchol**, **spcholinv**.

`[d, rcond] = spdet (a)` [Loadable Function]

Compute the determinant of sparse matrix *a* using UMFPACK. Return an estimate of the reciprocal condition number if requested.

`[x, rcond] = spinv (a, Q)` [Loadable Function]

Compute the inverse of the sparse square matrix *a*. Return an estimate of the reciprocal condition number if requested, otherwise warn of an ill-conditioned matrix if the reciprocal condition number is small. This function takes advantage of the sparsity of the matrix to accelerate the calculation of the inverse.

In general *x* will be a full matrix, and so if possible forming the inverse of a sparse matrix should be avoided. It is significantly more accurate and faster to do $y = a \setminus b$, rather than $y = \text{spinv}(a) * b$.

`spkron (a, b)` [Loadable Function]

Form the kronecker product of two sparse matrices. This is defined block by block as

$$x = [a(i, j) \ b]$$

For example,

```
kron(speye(3),spdiag([1,2,3]))
⇒
Compressed Column Sparse (rows = 9, cols = 9, nnz = 9)
```

```
(1, 1) -> 1
(2, 2) -> 2
(3, 3) -> 3
(4, 4) -> 1
(5, 5) -> 2
(6, 6) -> 3
(7, 7) -> 1
(8, 8) -> 2
(9, 9) -> 3
```

```
l = splchol (a) [Loadable Function]
[l, p] = splchol (a) [Loadable Function]
[l, p, q] = splchol (a) [Loadable Function]
```

Compute the Cholesky factor, l , of the symmetric positive definite sparse matrix a , where $LL^T = A$.

If called with 2 or more outputs p is the 0 when l is positive definite and l is a positive integer otherwise.

If called with 3 outputs that a sparsity preserving row/column permutation is applied to a prior to the factorization. That is l is the factorization of $a(q,q)$ such that $LR^T = A(Q,Q)$.

Note that `splchol` factorizations is faster and use less memory than `spchol`. `splchol(a)` is equivalent to `spchol(a)'`.

See also: `spcholinv`, `spchol2inv`, `splchol`.

```
[l, u] = splu (a) [Loadable Function]
[l, u, P] = splu (a) [Loadable Function]
[l, u, P, Q] = splu (a) [Loadable Function]
[l, u, P, Q] = splu (... , thres) [Loadable Function]
[l, u, P] = splu (... , Q) [Loadable Function]
```

Compute the LU decomposition of the sparse matrix a , using subroutines from UMF-PACK. The result is returned in a permuted form, according to the optional return values P and Q .

Called with two or three output arguments and a single input argument, `splu` is a replacement for `lu`, and therefore the sparsity preserving column permutations Q are not performed. Called with a fourth output argument, the sparsity preserving column transformation Q is returned, such that $P * a * Q = l * u$.

An additional input argument `thres`, that defines the pivoting threshold can be given. Alternatively, the desired sparsity preserving column permutations Q can be passed. Note that Q is assumed to be fixed if there are fewer than four output arguments. Otherwise, the updated column permutations are returned as the fourth argument.

With two output arguments, returns the permuted forms of the upper and lower triangular matrices, such that $\mathbf{a} = \mathbf{l} * \mathbf{u}$. With two or three output arguments, if a user-defined \mathbf{Q} is given, then $\mathbf{u} * \mathbf{Q}'$ is returned. The matrix is not required to be square.

See also: `sparse`, `spinv`, `colamd`, `symamd`.

<code>spparms ()</code>	[Loadable Function]
<code>vals = spparms ()</code>	[Loadable Function]
<code>[keys, vals] = spparms ()</code>	[Loadable Function]
<code>val = spparms (key)</code>	[Loadable Function]
<code>spparms (vals)</code>	[Loadable Function]
<code>spparms ('defaults')</code>	[Loadable Function]
<code>spparms ('tight')</code>	[Loadable Function]
<code>spparms (key, val)</code>	[Loadable Function]

Sets or displays the parameters used by the sparse solvers and factorization functions. The first four calls above get information about the current settings, while the others change the current settings. The parameters are stored as pairs of keys and values, where the values are all floats and the keys are one of the strings

- `spumoni` Printing level of debugging information of the solvers (default 0)
- `ths_rel` Included for compatiability. Bot used. (default 1)
- `ths_abs` Included for compatiability. Bot used. (default 1)
- `exact_d` Included for compatiability. Bot used. (default 0)
- `supernd` Included for compatiability. Not used. (default 3)
- `rreduce` Included for compatiability. Not used. (default 3)
- `wh_frac` Inluded for compatiability. Not used. (default 0.5)
- `autommd` Flag whether the LU/QR and the '\ ' and '/' operators will automatically use the sparsity preserving mmd functions (default 1)
- `autoamd` Flag whether the LU and the '\ ' and '/' operators will automatically use the sparsity preserving amd functions (default 1)
- `piv_tol` The pivot tolerance of the UMFPACK solvers (default 0.1)
- `bandden` ?? (default 0.5)
- `umfpack` Flag whether the UMFPACK or mmd solvers are used for the LU, '\ ' and '/' operations (default 1)

The value of individual keys can be set with `spparms (key, val)`. The default values can be restored with the special keyword 'defaults'. The special keyword 'tight' can be used to set the mmd solvers to attempt for a sparser solution at the potetial cost of longer running time.

<code>r = spqr (a)</code>	[Loadable Function]
<code>r = spqr (a,0)</code>	[Loadable Function]
<code>[c, r] = spqr (a,b)</code>	[Loadable Function]
<code>[c, r] = spqr (a,b,0)</code>	[Loadable Function]

Compute the sparse QR factorization of \mathbf{a} , using CSPARSE. As the matrix \mathbf{Q} is in general a full matrix, this function returns the \mathbf{Q} -less factorization \mathbf{r} of \mathbf{a} , such that $\mathbf{r} = \mathbf{chol}(\mathbf{a}' * \mathbf{a})$.

If the final argument is the scalar 0 and the number of rows is larger than the number of columns, then an economy factorization is returned. That is r will have only `size(a,1)` rows.

If an additional matrix b is supplied, then `spqr` returns c , where $c = q' * b$. This allows the least squares approximation of $a \setminus b$ to be calculated as

$$\begin{aligned} [c, r] &= \text{spqr}(a, b) \\ x &= r \setminus c \end{aligned}$$

See also: `spchol`, `qr`.

`p = sprank(s)` [Loadable Function]
 Calculates the structural rank of a sparse matrix s . Note that only the structure of the matrix is used in this calculation based on a Dulmage-Mendelsohn to block triangular form. As such the numerical rank of the matrix s is bounded by `sprank(s) >= rank(s)`. Ignoring floating point errors `sprank(s) == rank(s)`.

See also: `dmperm`.

`[count, h, parent, post, r] = symbfact(s, typ, mode)` [Loadable Function]
 Performs a symbolic factorization analysis on the sparse matrix s . Where

s s is a complex or real sparse matrix.

typ Is the type of the factorization and can be one of

`sym` Factorize s . This is the default.

`col` Factorize $s' * s$.

`row` Factorize $s * s'$.

`lo` Factorize s'

$mode$ The default is to return the Cholesky factorization for r , and if $mode$ is 'L', the conjugate transpose of the Cholesky factorization is returned. The conjugate transpose version is faster and uses less memory, but returns the same values for $count$, h , $parent$ and $post$ outputs.

The output variables are

$count$ The row counts of the Cholesky factorization as determined by typ .

h The height of the elimination tree.

$parent$ The elimination tree itself.

$post$ A sparse boolean matrix whose structure is that of the Cholesky factorization as determined by typ .

21.3 Iterative Techniques applied to sparse matrices

The left division \backslash and right division $/$ operators, discussed in the previous section, use direct solvers to resolve a linear equation of the form $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$ or $\mathbf{x} = \mathbf{b} / \mathbf{A}$. Octave equally includes a number of functions to solve sparse linear equations using iterative techniques.

`x = pcg (a, b, tol, maxit, m1, m2, x0, ...)` [Function File]

`[x, flag, relres, iter, resvec, eigest] = pcg (...)` [Function File]

Solves the linear system of equations $\mathbf{a} * \mathbf{x} = \mathbf{b}$ by means of the Preconditioned Conjugate Gradient iterative method. The input arguments are

- \mathbf{a} can be either a square (preferably sparse) matrix or a function handle, inline function or string containing the name of a function which computes $\mathbf{a} * \mathbf{x}$. In principle \mathbf{a} should be symmetric and positive definite; if `pcg` finds \mathbf{a} to not be positive definite, you will get a warning message and the `flag` output parameter will be set.
- \mathbf{b} is the right hand side vector.
- `tol` is the required relative tolerance for the residual error, $\mathbf{b} - \mathbf{a} * \mathbf{x}$. The iteration stops if `norm (b - a * x) <= tol * norm (b - a * x0)`. If `tol` is empty or is omitted, the function sets `tol = 1e-6` by default.
- `maxit` is the maximum allowable number of iterations; if `[]` is supplied for `maxit`, or `pcg` has less arguments, a default value equal to 20 is used.
- $\mathbf{m} = \mathbf{m1} * \mathbf{m2}$ is the (left) preconditioning matrix, so that the iteration is (theoretically) equivalent to solving by `pcg P * x = m \ b`, with $\mathbf{P} = \mathbf{m} \backslash \mathbf{a}$. Note that a proper choice of the preconditioner may dramatically improve the overall performance of the method. Instead of matrices `m1` and `m2`, the user may pass two functions which return the results of applying the inverse of `m1` and `m2` to a vector (usually this is the preferred way of using the preconditioner). If `[]` is supplied for `m1`, or `m1` is omitted, no preconditioning is applied. If `m2` is omitted, $\mathbf{m} = \mathbf{m1}$ will be used as preconditioner.
- `x0` is the initial guess. If `x0` is empty or omitted, the function sets `x0` to a zero vector by default.

The arguments which follow `x0` are treated as parameters, and passed in a proper way to any of the functions (\mathbf{a} or \mathbf{m}) which are passed to `pcg`. See the examples below for further details. The output arguments are

- \mathbf{x} is the computed approximation to the solution of $\mathbf{a} * \mathbf{x} = \mathbf{b}$.
- `flag` reports on the convergence. `flag = 0` means the solution converged and the tolerance criterion given by `tol` is satisfied. `flag = 1` means that the `maxit` limit for the iteration count was reached. `flag = 3` reports that the (preconditioned) matrix was found not positive definite.
- `relres` is the ratio of the final residual to its initial value, measured in the Euclidean norm.
- `iter` is the actual number of iterations performed.
- `resvec` describes the convergence history of the method. `resvec (i,1)` is the Euclidean norm of the residual, and `resvec (i,2)` is the preconditioned residual norm, after the $(i-1)$ -th iteration, $i = 1, 2, \dots, \text{iter}+1$. The preconditioned

residual norm is defined as $\text{norm}(r)^2 = r' * (m \setminus r)$ where $r = b - a * x$, see also the description of m . If *eigest* is not required, only *resvec*(:,1) is returned.

- *eigest* returns the estimate for the smallest *eigest* (1) and largest *eigest* (2) eigenvalues of the preconditioned matrix $P = m \setminus a$. In particular, if no preconditioning is used, the estimates for the extreme eigenvalues of a are returned. *eigest* (1) is an overestimate and *eigest* (2) is an underestimate, so that *eigest* (2) / *eigest* (1) is a lower bound for $\text{cond}(P, 2)$, which nevertheless in the limit should theoretically be equal to the actual value of the condition number. The method which computes *eigest* works only for symmetric positive definite a and m , and the user is responsible for verifying this assumption.

Let us consider a trivial problem with a diagonal matrix (we exploit the sparsity of A)

```
N = 10;
A = spdiag ([1:N]);
b = rand (N, 1);
[L, U, P, Q] = luinc (A,1.e-3);
```

EXAMPLE 1: Simplest use of *pcg*

```
x = pcg(A,b)
```

EXAMPLE 2: *pcg* with a function which computes $a * x$

```
function y = applyA (x)
  y = [1:N]' .* x;
endfunction
```

```
x = pcg ("applyA", b)
```

EXAMPLE 3: *pcg* with a preconditioner: $l * u$

```
x=pcg(A,b,1.e-6,500,L*U);
```

EXAMPLE 4: *pcg* with a preconditioner: $l * u$. Faster than EXAMPLE 3 since lower and upper triangular matrices are easier to invert

```
x=pcg(A,b,1.e-6,500,L,U);
```

EXAMPLE 5: Preconditioned iteration, with full diagnostics. The preconditioner (quite strange, because even the original matrix a is trivial) is defined as a function

```
function y = applyM(x)
  K = floor (length (x) - 2);
  y = x;
  y(1:K) = x(1:K) ./ [1:K]';
endfunction
```

```
[x, flag, relres, iter, resvec, eigest] = pcg (A, b, [], [], "applyM");
semilogy (1:iter+1, resvec);
```

EXAMPLE 6: Finally, a preconditioner which depends on a parameter k .

```
function y = applyM (x, varargin)
K = varargin{1};
y = x;
y(1:K) = x(1:K)./[1:K]';
endfunction

[x, flag, relres, iter, resvec, eigest] = ...
    pcg (A, b, [], [], "applyM", [], [], 3)
```

REFERENCES

- [1] C.T.Kelley, 'Iterative methods for linear and nonlinear equations', SIAM, 1995 (the base PCG algorithm)
- [2] Y.Saad, 'Iterative methods for sparse linear systems', PWS 1996 (condition number estimate from PCG) Revised version of this book is available online at <http://www-users.cs.umn.edu/~saad/books.html>

See also: sparse, pcr.

`x = pcr (a, b, tol, maxit, m, x0, ...)` [Function File]
`[x, flag, relres, iter, resvec] = pcr (...)` [Function File]

Solves the linear system of equations $\mathbf{a} * \mathbf{x} = \mathbf{b}$ by means of the Preconditioned Conjugate Residuals iterative method. The input arguments are

- \mathbf{a} can be either a square (preferably sparse) matrix or a function handle, inline function or string containing the name of a function which computes $\mathbf{a} * \mathbf{x}$. In principle \mathbf{a} should be symmetric and non-singular; if `pcr` finds \mathbf{a} to be numerically singular, you will get a warning message and the `flag` output parameter will be set.
- \mathbf{b} is the right hand side vector.
- `tol` is the required relative tolerance for the residual error, $\mathbf{b} - \mathbf{a} * \mathbf{x}$. The iteration stops if $\text{norm}(\mathbf{b} - \mathbf{a} * \mathbf{x}) \leq \text{tol} * \text{norm}(\mathbf{b} - \mathbf{a} * \mathbf{x0})$. If `tol` is empty or is omitted, the function sets `tol` = 1e-6 by default.
- `maxit` is the maximum allowable number of iterations; if [] is supplied for `maxit`, or `pcr` has less arguments, a default value equal to 20 is used.
- \mathbf{m} is the (left) preconditioning matrix, so that the iteration is (theoretically) equivalent to solving by `pcr` $\mathbf{P} * \mathbf{x} = \mathbf{m} \setminus \mathbf{b}$, with $\mathbf{P} = \mathbf{m} \setminus \mathbf{a}$. Note that a proper choice of the preconditioner may dramatically improve the overall performance of the method. Instead of matrix \mathbf{m} , the user may pass a function which returns the results of applying the inverse of \mathbf{m} to a vector (usually this is the preferred way of using the preconditioner). If [] is supplied for \mathbf{m} , or \mathbf{m} is omitted, no preconditioning is applied.
- $\mathbf{x0}$ is the initial guess. If $\mathbf{x0}$ is empty or omitted, the function sets $\mathbf{x0}$ to a zero vector by default.

The arguments which follow $\mathbf{x0}$ are treated as parameters, and passed in a proper way to any of the functions (\mathbf{a} or \mathbf{m}) which are passed to `pcr`. See the examples below for further details. The output arguments are

- \mathbf{x} is the computed approximation to the solution of $\mathbf{a} * \mathbf{x} = \mathbf{b}$.

- *flag* reports on the convergence. *flag* = 0 means the solution converged and the tolerance criterion given by *tol* is satisfied. *flag* = 1 means that the *maxit* limit for the iteration count was reached. *flag* = 3 reports the *pcr* breakdown, see [1] for details.
- *relres* is the ratio of the final residual to its initial value, measured in the Euclidean norm.
- *iter* is the actual number of iterations performed.
- *resvec* describes the convergence history of the method, so that *resvec* (*i*) contains the Euclidean norms of the residual after the (*i*-1)-th iteration, *i* = 1, 2, ..., *iter*+1.

Let us consider a trivial problem with a diagonal matrix (we exploit the sparsity of A)

```
N = 10;
A = diag([1:N]); A = sparse(A);
b = rand(N,1);
```

EXAMPLE 1: Simplest use of *pcr*

```
x = pcr(A, b)
```

EXAMPLE 2: *pcr* with a function which computes *a * x*.

```
function y = applyA(x)
    y = [1:10]' .* x;
endfunction
```

```
x = pcr('applyA', b)
```

EXAMPLE 3: Preconditioned iteration, with full diagnostics. The preconditioner (quite strange, because even the original matrix *a* is trivial) is defined as a function

```
function y = applyM(x)
    K = floor(length(x)-2);
    y = x;
    y(1:K) = x(1:K) ./ [1:K]';
endfunction
```

```
[x, flag, relres, iter, resvec] = pcr(A, b, [], [], 'applyM')
semilogy([1:iter+1], resvec);
```

EXAMPLE 4: Finally, a preconditioner which depends on a parameter *k*.

```
function y = applyM(x, varargin)
    K = varargin{1};
    y = x; y(1:K) = x(1:K) ./ [1:K]';
endfunction
```

```
[x, flag, relres, iter, resvec] = pcr(A, b, [], [], 'applyM', [], 3)
```

REFERENCES

[1] W. Hackbusch, "Iterative Solution of Large Sparse Systems of Equations", section 9.5.4; Springer, 1994

See also: *sparse*, *pcg*.

The speed with which an iterative solver converges to a solution can be accelerated with the use of a pre-conditioning matrix M . In this case the linear equation $M^{-1} * x = M^{-1} * A \setminus b$ is solved instead. Typical pre-conditioning matrices are partial factorizations of the original matrix.

```
[l, u, p, q] = luinc (a, '0') [Loadable Function]
[l, u, p, q] = luinc (a, droptol) [Loadable Function]
[l, u, p, q] = luinc (a, opts) [Loadable Function]
```

Produce the incomplete LU factorization of the sparse matrix a . Two types of incomplete factorization are possible, and the type is determined by the second argument to *luinc*.

Called with a second argument of '0', the zero-level incomplete LU factorization is produced. This creates a factorization of a where the position of the non-zero arguments correspond to the same positions as in the matrix a .

Alternatively, the fill-in of the incomplete LU factorization can be controlled through the variable *droptol* or the structure *opts*. The UMFPACK multifrontal factorization code by Tim A. Davis is used for the incomplete LU factorization, (availability <http://www.cise.ufl.edu/research/sparse/umfpack/>)

droptol determines the values below which the values in the LU factorization are dropped and replaced by zero. It must be a positive scalar, and any values in the factorization whose absolute value are less than this value are dropped, expect if leaving them increase the sparsity of the matrix. Setting *droptol* to zero results in a complete LU factorization which is the default.

opts is a structure containing one or more of the fields

droptol	The drop tolerance as above. If <i>opts</i> only contains droptol then this is equivalent to using the variable <i>droptol</i> .
milu	A logical variable flagging whether to use the modified incomplete LU factorization. In the case that milu is true, the dropped values are subtract from the diagonal of the matrix U of the factorization. The default is false .
udiag	A logical variable that flags whether zero elements on the diagonal of U should be replaced with <i>droptol</i> to attempt to avoid singular factors. The default is false .
thresh	Defines the pivot threshold in the interval [0,1]. Values outside that range are ignored.

All other fields in *opts* are ignored. The outputs from *luinc* are the same as for *lu*.

See also: *sparse*, *lu*, *cholinc*.

21.4 Real Life Example of the use of Sparse Matrices

A common application for sparse matrices is in the solution of Finite Element Models. Finite element models allow numerical solution of partial differential equations that do not have closed form solutions, typically because of the complex shape of the domain.

In order to motivate this application, we consider the boundary value Laplace equation. This system can model scalar potential fields, such as heat or electrical potential. Given a medium Ω with boundary $\partial\Omega$. At all points on the $\partial\Omega$ the boundary conditions are known, and we wish to calculate the potential in Ω . Boundary conditions may specify the potential (Dirichlet boundary condition), its normal derivative across the boundary (Neumann boundary condition), or a weighted sum of the potential and its derivative (Cauchy boundary condition).

In a thermal model, we want to calculate the temperature in Ω and know the boundary temperature (Dirichlet condition) or heat flux (from which we can calculate the Neumann condition by dividing by the thermal conductivity at the boundary). Similarly, in an electrical model, we want to calculate the voltage in Ω and know the boundary voltage (Dirichlet) or current (Neumann condition after dividing by the electrical conductivity). In an electrical model, it is common for much of the boundary to be electrically isolated; this is a Neumann boundary condition with the current equal to zero.

The simplest finite element models will divide Ω into simplexes (triangles in 2D, pyramids in 3D). We take as an 3D example a cylindrical liquid filled tank with a small non-conductive ball from the EIDORS project³. This model is designed to reflect an application of electrical impedance tomography, where current patterns are applied to such a tank in order to image the internal conductivity distribution. In order to describe the FEM geometry, we have a matrix of vertices `nodes` and simplices `elems`.

The following example creates a simple rectangular 2D electrically conductive medium with 10 V and 20 V imposed on opposite sides (Dirichlet boundary conditions). All other edges are electrically isolated.

```
node_y= [1;1.2;1.5;1.8;2]*ones(1,11);
node_x= ones(5,1)*[1,1.05,1.1,1.2, ...
    1.3,1.5,1.7,1.8,1.9,1.95,2];
nodes= [node_x(:), node_y(:)];

[h,w]= size(node_x);
elems= [];
for idx= 1:w-1
    widx= (idx-1)*h;
    elems= [elems; ...
        widx+[(1:h-1);(2:h);h+(1:h-1)]'; ...
        widx+[(2:h);h+(2:h);h+(1:h-1)]' ];
endfor

E= size(elems,1); # No. of simplices
N= size(nodes,1); # No. of vertices
D= size(elems,2); # dimensions+1
```

This creates a N-by-2 matrix `nodes` and a E-by-3 matrix `elems` with values, which define finite element triangles:

```
nodes(1:7,:)'
```

³ EIDORS - Electrical Impedance Tomography and Diffuse optical Tomography Reconstruction Software
<http://eidors3d.sourceforge.net>


```

1.00 1.00 1.00 1.00 1.00 1.05 1.05 ...
1.00 1.20 1.50 1.80 2.00 1.00 1.20 ...

elems(1:7,:)
 1   2   3   4   2   3   4 ...
 2   3   4   5   7   8   9 ...
 6   7   8   9   6   7   8 ...

```

Using a first order FEM, we approximate the electrical conductivity distribution in Ω as constant on each simplex (represented by the vector `conductivity`). Based on the finite element geometry, we first calculate a system (or stiffness) matrix for each simplex (represented as 3-by-3 elements on the diagonal of the element-wise system matrix `SE`). Based on `SE` and a N-by-DE connectivity matrix `C`, representing the connections between simplices and vertices, the global connectivity matrix `S` is calculated.

```

# Element conductivity
conductivity= [1*ones(1,16), ...
               2*ones(1,48), 1*ones(1,16)];

# Connectivity matrix
C = sparse ((1:D*E), reshape (elems', ...
                              D*E, 1), 1, D*E, N);

# Calculate system matrix
Siidx = floor ([0:D*E-1]'/D) * D * ...
          ones(1,D) + ones(D*E,1)*(1:D) ;
Sjidx = [1:D*E]'*ones(1,D);
Sdata = zeros(D*E,D);
dfact = factorial(D-1);
for j=1:E
    a = inv([ones(D,1), ...
            nodes(elems(j,:), :)]);
    const = conductivity(j) * 2 / ...
            dfact / abs(det(a));
    Sdata(D*(j-1)+(1:D),:) = const * ...
            a(2:D,:)' * a(2:D,:);
endfor
# Element-wise system matrix
SE= sparse(Siidx,Sjidx,Sdata);
# Global system matrix
S= C'* SE *C;

```

The system matrix acts like the conductivity S in Ohm's law $SV = I$. Based on the Dirichlet and Neumann boundary conditions, we are able to solve for the voltages at each vertex V .

```

# Dirichlet boundary conditions
D_nodes=[1:5, 51:55];
D_value=[10*ones(1,5), 20*ones(1,5)];

```

```

V= zeros(N,1);
V(D_nodes) = D_value;
idx = 1:N; # vertices without Dirichlet
           # boundary condns
idx(D_nodes) = [];

# Neumann boundary conditions. Note that
# N_value must be normalized by the
# boundary length and element conductivity
N_nodes=[];
N_value=[];

Q = zeros(N,1);
Q(N_nodes) = N_value;

V(idx) = S(idx,idx) \ ( Q(idx) - ...
                       S(idx,D_nodes) * V(D_nodes));

```

Finally, in order to display the solution, we show each solved voltage value in the z-axis for each simplex vertex.

```

elemx = elems(:, [1,2,3,1])';
xelems = reshape (nodes(elemx, 1), 4, E);
yelems = reshape (nodes(elemx, 2), 4, E);
velems = reshape (V(elemx), 4, E);
plot3 (xelems,yelems,velems,'k');
print ('grid.eps');

```

22 Numerical Integration

Octave comes with several built-in functions for computing the integral of a function numerically. These functions all solve 1-dimensional integration problems.

22.1 Functions of One Variable

Octave supports three different algorithms for computing the integral

$$\int_a^b f(x)dx$$

of a function f over the interval from a to b . These are

quad Numerical integration based on Gaussian quadrature.
quadl Numerical integration using an adaptive Lobatto rule.
trapz Numerical integration using the trapezodial method.

Besides these functions Octave also allows you to perform cumulative numerical integration using the trapezodial method through the **cumtrapz** function.

[v, ier, nfun, err] = quad (f, a, b, tol, sing) [Loadable Function]

Integrate a nonlinear function of one variable using Quadpack. The first argument is the name of the function, the function handle or the inline function to call to compute the value of the integrand. It must have the form

$$y = f(x)$$

where y and x are scalars.

The second and third arguments are limits of integration. Either or both may be infinite.

The optional argument *tol* is a vector that specifies the desired accuracy of the result. The first element of the vector is the desired absolute tolerance, and the second element is the desired relative tolerance. To choose a relative test only, set the absolute tolerance to zero. To choose an absolute test only, set the relative tolerance to zero.

The optional argument *sing* is a vector of values at which the integrand is known to be singular.

The result of the integration is returned in *v* and *ier* contains an integer error code (0 indicates a successful integration). The value of *nfun* indicates how many function evaluations were required, and *err* contains an estimate of the error in the solution.

You can use the function **quad_options** to set optional parameters for **quad**.

It should be noted that since **quad** is written in Fortran it cannot be called recursively.

quad_options (opt, val) [Loadable Function]

When called with two arguments, this function allows you set options parameters for the function **quad**. Given one argument, **quad_options** returns the value of the corresponding option. If no arguments are supplied, the names of all the available options and their current values are displayed.

Options include

"absolute tolerance"

Absolute tolerance; may be zero for pure relative error test.

"relative tolerance"

Nonnegative relative tolerance. If the absolute tolerance is zero, the relative tolerance must be greater than or equal to `max (50*eps, 0.5e-28)`.

Here is an example of using `quad` to integrate the function

$$f(x) = x \sin(1/x) \sqrt{|1-x|}$$

from $x = 0$ to $x = 3$.

This is a fairly difficult integration (plot the function over the range of integration to see why).

The first step is to define the function:

```
function y = f (x)
  y = x .* sin (1 ./ x) .* sqrt (abs (1 - x));
endfunction
```

Note the use of the 'dot' forms of the operators. This is not necessary for the call to `quad`, but it makes it much easier to generate a set of points for plotting (because it makes it possible to call the function with a vector argument to produce a vector result).

Then we simply call `quad`:

```
[v, ier, nfun, err] = quad ("f", 0, 3)
⇒ 1.9819
⇒ 1
⇒ 5061
⇒ 1.1522e-07
```

Although `quad` returns a nonzero value for `ier`, the result is reasonably accurate (to see why, examine what happens to the result if you move the lower bound to 0.1, then 0.01, then 0.001, etc.).

```
q = quadl (f, a, b) [Function File]
q = quadl (f, a, b, tol) [Function File]
q = quadl (f, a, b, tol, trace) [Function File]
q = quadl (f, a, b, tol, trace, p1, p2, ...) [Function File]
```

Numerically evaluate integral using adaptive Lobatto rule. `quadl (f, a, b)` approximates the integral of $f(x)$ to machine precision. f is either a function handle, inline function or string containing the name of the function to evaluate. The function f must return a vector of output values if given a vector of input values.

If defined, `tol` defines the relative tolerance to which to which to integrate $f(x)$. While if `trace` is defined, displays the left end point of the current interval, the interval length, and the partial integral.

Additional arguments `p1`, etc, are passed directly to f . To use default values for `tol` and `trace`, one may pass empty matrices.

Reference: W. Gander and W. Gautschi, 'Adaptive Quadrature - Revisited', BIT Vol. 40, No. 1, March 2000, pp. 84–101. <http://www.inf.ethz.ch/personal/gander/>

```

z = trapz (y) [Function File]
z = trapz (x, y) [Function File]
z = trapz (... , dim) [Function File]

```

Numerical intergration using trapezodial method. `trapz (y)` computes the integral of the `y` along the first non singleton dimension. If the argument `x` is omitted a equally spaced vector is assumed. `trapz (x, y)` evaluates the integral with respect to `x`.

See also: `cumtrapz`.

```

z = cumtrapz (y) [Function File]
z = cumtrapz (x, y) [Function File]
z = cumtrapz (... , dim) [Function File]

```

Cumulative numerical intergration using trapezodial method. `cumtrapz (y)` computes the cummulative integral of the `y` along the first non singleton dimension. If the argument `x` is omitted a equally spaced vector is assumed. `cumtrapz (x, y)` evaluates the cummulative integral with respect to `x`.

See also: `trapz`, `cumsum`.

22.2 Orthogonal Collocation

```

[r, amat, bmat, q] = colloc (n, "left", "right") [Loadable Function]

```

Compute derivative and integral weight matrices for orthogonal collocation using the subroutines given in J. Villadsen and M. L. Michelsen, *Solution of Differential Equation Models by Polynomial Approximation*.

Here is an example of using `colloc` to generate weight matrices for solving the second order differential equation $u' - \alpha u'' = 0$ with the boundary conditions $u(0) = 0$ and $u(1) = 1$.

First, we can generate the weight matrices for n points (including the endpoints of the interval), and incorporate the boundary conditions in the right hand side (for a specific value of α).

```

n = 7;
alpha = 0.1;
[r, a, b] = colloc (n-2, "left", "right");
at = a(2:n-1,2:n-1);
bt = b(2:n-1,2:n-1);
rhs = alpha * b(2:n-1,n) - a(2:n-1,n);

```

Then the solution at the roots `r` is

```

u = [ 0; (at - alpha * bt) \ rhs; 1]
    => [ 0.00; 0.004; 0.01 0.00; 0.12; 0.62; 1.00 ]

```

22.3 Functions of Multiple Variables

Octave does not have built-in functions for computing the integral of functions of multiple variables. It is however possible to compute the integral of a function of multiple variables using the functions for one-dimensional integrals.

To illustrate how the integration can be performed, we will integrate the function

$$f(x, y) = \sin(\pi xy)\sqrt{xy}$$

for x and y between 0 and 1.

The first approach creates a function that integrates f with respect to x , and then integrates that function with respect to y . Since `quad` is written in Fortran it cannot be called recursively. This means that `quad` cannot integrate a function that calls `quad`, and hence cannot be used to perform the double integration. It is however possible with `quadl`, which is what the following code does.

```
function I = g(y)
  I = ones(1, length(y));
  for i = 1:length(y)
    f = @(x) sin(pi.*x.*y(i)).*sqrt(x.*y(i));
    I(i) = quadl(f, 0, 1);
  endfor
endfunction

I = quadl("g", 0, 1)
    ⇒ 0.30022
```

The above mentioned approach works but is fairly slow, and that problem increases exponentially with the dimensionality the problem. Another possible solution is to use Orthogonal Collocation as described in the previous section. The integral of a function $f(x, y)$ for x and y between 0 and 1 can be approximated using n points by

$$\int_0^1 \int_0^1 f(x, y) dx dy \approx \sum_{i=1}^n \sum_{j=1}^n q_i q_j f(r_i, r_j),$$

where q and r is as returned by `colloc(n)`. The generalisation to more than two variables is straight forward. The following code computes the studied integral using $n = 7$ points.

```
f = @(x,y) sin(pi*x*y').*sqrt(x*y');
n = 7;
[t, A, B, q] = colloc(n);
I = q'*f(t,t)*q;
    ⇒ 0.30022
```

It should be noted that the number of points determines the quality of the approximation. If the integration needs to be performed between a and b instead of 0 and 1, a change of variables is needed.

23 Differential Equations

Octave has built-in functions for solving ordinary differential equations, and differential-algebraic equations. All solvers are based on reliable ODE solvers written in Fortran.

23.1 Ordinary Differential Equations

The function `lsode` can be used to solve ODEs of the form

$$\frac{dx}{dt} = f(x, t)$$

using Hindmarsh's ODE solver LSODE.

`[x, istate, msg] = lsode (fcn, x_0, t, t_crit)` [Loadable Function]
Solve the set of differential equations

$$\frac{dx}{dt} = f(x, t)$$

with

$$x(t_0) = x_0$$

The solution is returned in the matrix `x`, with each row corresponding to an element of the vector `t`. The first element of `t` should be t_0 and should correspond to the initial state of the system `x_0`, so that the first row of the output is `x_0`.

The first argument, `fcn`, is a string, or cell array of strings, inline or function handles, that names the function to call to compute the vector of right hand sides for the set of equations. The function must have the form

$$\mathbf{xdot} = \mathbf{f}(\mathbf{x}, t)$$

in which `xdot` and `x` are vectors and `t` is a scalar.

If `fcn` is a two-element string array, the first element names the function f described above, and the second element names a function to compute the Jacobian of f . The Jacobian function must have the form

$$\mathbf{jac} = \mathbf{j}(\mathbf{x}, t)$$

in which `jac` is the matrix of partial derivatives

$$J = \frac{\partial f_i}{\partial x_j} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \cdots & \frac{\partial f_3}{\partial x_N} \end{bmatrix}$$

The second and third arguments specify the initial state of the system, x_0 , and the initial value of the independent variable t_0 .

The fourth argument is optional, and may be used to specify a set of times that the ODE solver should not integrate past. It is useful for avoiding difficulties with singularities and points where there is a discontinuity in the derivative.

After a successful computation, the value of *istate* will be 2 (consistent with the Fortran version of LSODE).

If the computation is not successful, *istate* will be something other than 2 and *msg* will contain additional information.

You can use the function `lsode_options` to set optional parameters for `lsode`.

See also: `daspk`, `dassl`, `dasrt`.

`lsode_options (opt, val)` [Loadable Function]

When called with two arguments, this function allows you set options parameters for the function `lsode`. Given one argument, `lsode_options` returns the value of the corresponding option. If no arguments are supplied, the names of all the available options and their current values are displayed.

Options include

"absolute tolerance"

Absolute tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector.

"relative tolerance"

Relative tolerance parameter. Unlike the absolute tolerance, this parameter may only be a scalar.

The local error test applied at each integration step is

$$\text{abs}(\text{local error in } x(i)) \leq \text{rtol} * \text{abs}(y(i)) + \text{atol}(i)$$

"integration method"

A string specifying the method of integration to use to solve the ODE system. Valid values are

"adams"

"non-stiff"

No Jacobian used (even if it is available).

"bdf"

"stiff" Use stiff backward differentiation formula (BDF) method. If a function to compute the Jacobian is not supplied, `lsode` will compute a finite difference approximation of the Jacobian matrix.

"initial step size"

The step size to be attempted on the first step (default is determined automatically).

"maximum order"

Restrict the maximum order of the solution method. If using the Adams method, this option must be between 1 and 12. Otherwise, it must be between 1 and 5, inclusive.

"maximum step size"

Setting the maximum stepsize will avoid passing over very large regions (default is not specified).

"minimum step size"

The minimum absolute step size allowed (default is 0).

"step limit"

Maximum number of steps allowed (default is 100000).

Here is an example of solving a set of three differential equations using `lsode`. Given the function

```
function xdot = f (x, t)

    xdot = zeros (3,1);

    xdot(1) = 77.27 * (x(2) - x(1)*x(2) + x(1) \
        - 8.375e-06*x(1)^2);
    xdot(2) = (x(3) - x(1)*x(2) - x(2)) / 77.27;
    xdot(3) = 0.161*(x(1) - x(3));

endfunction
```

and the initial condition `x0 = [4; 1.1; 4]`, the set of equations can be integrated using the command

```
t = linspace (0, 500, 1000);

y = lsode ("f", x0, t);
```

If you try this, you will see that the value of the result changes dramatically between $t = 0$ and 5, and again around $t = 305$. A more efficient set of output points might be

```
t = [0, logspace (-1, log10(303), 150), \
    logspace (log10(304), log10(500), 150)];
```

See Alan C. Hindmarsh, *ODEPACK, A Systematized Collection of ODE Solvers*, in Scientific Computing, R. S. Stepleman, editor, (1983) for more information about the inner workings of `lsode`.

23.2 Differential-Algebraic Equations

The function `daspk` can be used to solve DAEs of the form

$$0 = f(\dot{x}, x, t), \quad x(t = 0) = x_0, \dot{x}(t = 0) = \dot{x}_0$$

where $\dot{x} = \frac{dx}{dt}$ is the derivative of x . The equation is solved using Petzold's DAE solver DASPK.

```
[x, xdot, istate, msg] = daspk (fcn, x_0, xdot_0, t, [Loadable Function]
    t_crit)
```

Solve the set of differential-algebraic equations

$$0 = f(x, \dot{x}, t)$$

with

$$x(t_0) = x_0, \dot{x}(t_0) = \dot{x}_0$$

The solution is returned in the matrices `x` and `xdot`, with each row in the result matrices corresponding to one of the elements in the vector `t`. The first element of `t` should be t_0 and correspond to the initial state of the system `x_0` and its derivative `xdot_0`, so that the first row of the output `x` is `x_0` and the first row of the output `xdot` is `xdot_0`.

The first argument, `fcn`, is a string or a two element cell array of strings, inline or function handle, that names the function, to call to compute the vector of residuals for the set of equations. It must have the form

```
res = f (x, xdot, t)
```

in which `x`, `xdot`, and `res` are vectors, and `t` is a scalar.

If `fcn` is a two-element string array, the first element names the function f described above, and the second element names a function to compute the modified Jacobian

$$J = \frac{\partial f}{\partial x} + c \frac{\partial f}{\partial \dot{x}}$$

The modified Jacobian function must have the form

```
jac = j (x, xdot, t, c)
```

The second and third arguments to `daspk` specify the initial condition of the states and their derivatives, and the fourth argument specifies a vector of output times at which the solution is desired, including the time corresponding to the initial condition.

The set of initial states and derivatives are not strictly required to be consistent. If they are not consistent, you must use the `daspk_options` function to provide additional information so that `daspk` can compute a consistent starting point.

The fifth argument is optional, and may be used to specify a set of times that the DAE solver should not integrate past. It is useful for avoiding difficulties with singularities and points where there is a discontinuity in the derivative.

After a successful computation, the value of `istate` will be greater than zero (consistent with the Fortran version of DASPCK).

If the computation is not successful, the value of `istate` will be less than zero and `msg` will contain additional information.

You can use the function `daspk_options` to set optional parameters for `daspk`.

See also: `dassl`.

`daspk_options (opt, val)`

[Loadable Function]

When called with two arguments, this function allows you set options parameters for the function `daspk`. Given one argument, `daspk_options` returns the value of the corresponding option. If no arguments are supplied, the names of all the available options and their current values are displayed.

Options include

"absolute tolerance"

Absolute tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the relative tolerance must also be a vector of the same length.

"relative tolerance"

Relative tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the absolute tolerance must also be a vector of the same length.

The local error test applied at each integration step is

$$\begin{aligned} &\text{abs (local error in } x(i)) \\ &\leq \text{rtol}(i) * \text{abs } (Y(i)) + \text{atol}(i) \end{aligned}$$

"compute consistent initial condition"

Denoting the differential variables in the state vector by 'Y_d' and the algebraic variables by 'Y_a', **ddaspk** can solve one of two initialization problems:

1. Given Y_d, calculate Y_a and Y'_d
2. Given Y', calculate Y.

In either case, initial values for the given components are input, and initial guesses for the unknown components must also be provided as input. Set this option to 1 to solve the first problem, or 2 to solve the second (the default is 0, so you must provide a set of initial conditions that are consistent).

If this option is set to a nonzero value, you must also set the **"algebraic variables"** option to declare which variables in the problem are algebraic.

"use initial condition heuristics"

Set to a nonzero value to use the initial condition heuristics options described below.

"initial condition heuristics"

A vector of the following parameters that can be used to control the initial condition calculation.

MXNIT	Maximum number of Newton iterations (default is 5).
MXNJ	Maximum number of Jacobian evaluations (default is 6).
MXNH	Maximum number of values of the artificial stepsize parameter to be tried if the "compute consistent initial condition" option has been set to 1 (default is 5). Note that the maximum number of Newton iterations allowed in all is MXNIT*MXNJ*MXNH if the "compute consistent initial condition" option has been set to 1 and MXNIT*MXNJ if it is set to 2.
LSOFF	Set to a nonzero value to disable the linesearch algorithm (default is 0).

STPTOL Minimum scaled step in linesearch algorithm (default is $\text{eps}^{(2/3)}$).

EPINIT Swing factor in the Newton iteration convergence test. The test is applied to the residual vector, premultiplied by the approximate Jacobian. For convergence, the weighted RMS norm of this vector (scaled by the error weights) must be less than **EPINIT*EPCON**, where **EPCON** = 0.33 is the analogous test constant used in the time steps. The default is **EPINIT** = 0.01.

"print initial condition info"

Set this option to a nonzero value to display detailed information about the initial condition calculation (default is 0).

"exclude algebraic variables from error test"

Set to a nonzero value to exclude algebraic variables from the error test. You must also set the **"algebraic variables"** option to declare which variables in the problem are algebraic (default is 0).

"algebraic variables"

A vector of the same length as the state vector. A nonzero element indicates that the corresponding element of the state vector is an algebraic variable (i.e., its derivative does not appear explicitly in the equation set. This option is required by the **compute consistent initial condition** and **"exclude algebraic variables from error test"** options.

"enforce inequality constraints"

Set to one of the following values to enforce the inequality constraints specified by the **"inequality constraint types"** option (default is 0).

1. To have constraint checking only in the initial condition calculation.
2. To enforce constraint checking during the integration.
3. To enforce both options 1 and 2.

"inequality constraint types"

A vector of the same length as the state specifying the type of inequality constraint. Each element of the vector corresponds to an element of the state and should be assigned one of the following codes

- | | |
|----|--------------------------------|
| -2 | Less than zero. |
| -1 | Less than or equal to zero. |
| 0 | Not constrained. |
| 1 | Greater than or equal to zero. |
| 2 | Greater than zero. |

This option only has an effect if the **"enforce inequality constraints"** option is nonzero.

"initial step size"

Differential-algebraic problems may occasionally suffer from severe scaling difficulties on the first step. If you know a great deal about the scaling of your problem, you can help to alleviate this problem by specifying an initial stepsize (default is computed automatically).

"maximum order"

Restrict the maximum order of the solution method. This option must be between 1 and 5, inclusive (default is 5).

"maximum step size"

Setting the maximum stepsize will avoid passing over very large regions (default is not specified).

Octave also includes DASSL, an earlier version of *Dasgk*, and *dasrt*, which can be used to solve DAEs with constraints (stopping conditions).

```
[x, xdot, t_out, istat, msg] = dasrt (fcn [, g], x_0,      [Loadable Function]
    xdot_0, t [, t_crit])
Solve the set of differential-algebraic equations
```

$$0 = f(x, \dot{x}, t)$$

with

$$x(t_0) = x_0, \dot{x}(t_0) = \dot{x}_0$$

with functional stopping criteria (root solving).

The solution is returned in the matrices *x* and *xdot*, with each row in the result matrices corresponding to one of the elements in the vector *t_out*. The first element of *t* should be *t*₀ and correspond to the initial state of the system *x_0* and its derivative *xdot_0*, so that the first row of the output *x* is *x_0* and the first row of the output *xdot* is *xdot_0*.

The vector *t* provides an upper limit on the length of the integration. If the stopping condition is met, the vector *t_out* will be shorter than *t*, and the final element of *t_out* will be the point at which the stopping condition was met, and may not correspond to any element of the vector *t*.

The first argument, *fcn*, is a string, or cell array of strings or inline or function handles, that names the function to call to compute the vector of residuals for the set of equations. It must have the form

$$res = f(x, xdot, t)$$

in which *x*, *xdot*, and *res* are vectors, and *t* is a scalar.

If *fcn* is a two-element string array, or two element cell array, the first element names the function *f* described above, and the second element names a function to compute the modified Jacobian

$$J = \frac{\partial f}{\partial x} + c \frac{\partial f}{\partial \dot{x}}$$

The modified Jacobian function must have the form

```
jac = j (x, xdot, t, c)
```

The optional second argument names a function that defines the constraint functions whose roots are desired during the integration. This function must have the form

```
g_out = g (x, t)
```

and return a vector of the constraint function values. If the value of any of the constraint functions changes sign, DASRT will attempt to stop the integration at the point of the sign change.

If the name of the constraint function is omitted, **dasrt** solves the same problem as **daspk** or **dassl**.

Note that because of numerical errors in the constraint functions due to roundoff and integration error, DASRT may return false roots, or return the same root at two or more nearly equal values of T . If such false roots are suspected, the user should consider smaller error tolerances or higher precision in the evaluation of the constraint functions.

If a root of some constraint function defines the end of the problem, the input to DASRT should nevertheless allow integration to a point slightly past that root, so that DASRT can locate the root by interpolation.

The third and fourth arguments to **dasrt** specify the initial condition of the states and their derivatives, and the fourth argument specifies a vector of output times at which the solution is desired, including the time corresponding to the initial condition.

The set of initial states and derivatives are not strictly required to be consistent. In practice, however, DASSL is not very good at determining a consistent set for you, so it is best if you ensure that the initial values result in the function evaluating to zero.

The sixth argument is optional, and may be used to specify a set of times that the DAE solver should not integrate past. It is useful for avoiding difficulties with singularities and points where there is a discontinuity in the derivative.

After a successful computation, the value of *istate* will be greater than zero (consistent with the Fortran version of DASSL).

If the computation is not successful, the value of *istate* will be less than zero and *msg* will contain additional information.

You can use the function **dasrt_options** to set optional parameters for **dasrt**.

See also: **daspk**, **dasrt**, **lsode**.

dasrt_options (*opt*, *val*)

[Loadable Function]

When called with two arguments, this function allows you set options parameters for the function **dasrt**. Given one argument, **dasrt_options** returns the value of the corresponding option. If no arguments are supplied, the names of all the available options and their current values are displayed.

Options include

"absolute tolerance"

Absolute tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the relative tolerance must also be a vector of the same length.

"relative tolerance"

Relative tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the absolute tolerance must also be a vector of the same length.

The local error test applied at each integration step is

$$\text{abs}(\text{local error in } x(i)) \leq \text{rtol}(i) * \text{abs}(Y(i)) + \text{atol}(i)$$
"initial step size"

Differential-algebraic problems may occasionally suffer from severe scaling difficulties on the first step. If you know a great deal about the scaling of your problem, you can help to alleviate this problem by specifying an initial stepsize.

"maximum order"

Restrict the maximum order of the solution method. This option must be between 1 and 5, inclusive.

"maximum step size"

Setting the maximum stepsize will avoid passing over very large regions.

"step limit"

Maximum number of integration steps to attempt on a single call to the underlying Fortran code.

See K. E. Brenan, et al., *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, North-Holland (1989) for more information about the implementation of DASSL.

24 Optimization

Octave comes with support for solving various kinds of optimization problems. Specifically Octave can solve problems in Linear Programming, Quadratic Programming, Nonlinear Programming, and Linear Least Squares Minimization.

24.1 Linear Programming

Octave can solve Linear Programming problems using the `glpk` function. That is, Octave can solve

$$\min_x c^T x$$

subject to the linear constraints $Ax = b$ where $x \geq 0$.

The `glpk` function also supports variations of this problem.

`[xopt, fmin, status, extra] = glpk (c, a, b, lb, ub, ctype, [Function File]
vartype, sense, param)`

Solve a linear program using the GNU GLPK library. Given three arguments, `glpk` solves the following standard LP:

$$\min_x C^T x$$

subject to

$$Ax = b \quad x \geq 0$$

but may also solve problems of the form

$$[\min_x | \max_x] C^T x$$

subject to

$$Ax [= | \leq | \geq] b \quad LB \leq x \leq UB$$

Input arguments:

- | | |
|-----------|--|
| <i>c</i> | A column array containing the objective function coefficients. |
| <i>a</i> | A matrix containing the constraints coefficients. |
| <i>b</i> | A column array containing the right-hand side value for each constraint in the constraint matrix. |
| <i>lb</i> | An array containing the lower bound on each of the variables. If <i>lb</i> is not supplied, the default lower bound for the variables is zero. |
| <i>ub</i> | An array containing the upper bound on each of the variables. If <i>ub</i> is not supplied, the default upper bound is assumed to be infinite. |

ctype An array of characters containing the sense of each constraint in the constraint matrix. Each element of the array may be one of the following values

"F" Free (unbounded) variable (the constraint is ignored).

"U" Variable with upper bound ($A(i,:) * x \leq b(i)$).

"S" Fixed Variable ($A(i,:) * x = b(i)$).

"L" Variable with lower bound ($A(i,:) * x \geq b(i)$).

"D" Double-bounded variable ($A(i,:) * x \geq -b(i)$ and $A(i,:) * x \leq b(i)$).

vartype A column array containing the types of the variables.

"F" "C" Continuous variable. "I" Integer variable

sense If *sense* is 1, the problem is a minimization. If *sense* is -1, the problem is a maximization. The default value is 1.

param A structure containing the following parameters used to define the behavior of solver. Missing elements in the structure take on default values, so you only need to set the elements that you wish to change from the default.

Integer parameters:

`msglev (LPX_K_MSGLEV, default: 1)`
 Level of messages output by solver routines:

0 No output.

1 Error messages only.

2 Normal output .

3 Full output (includes informational messages).

`scale (LPX_K_SCALE, default: 1)`
 Scaling option:

0 No scaling .

1 Equilibration scaling .

2 Geometric mean scaling, then equilibration scaling.

`dual (LPX_K_DUAL, default: 0)`
 Dual simplex option:

0 Do not use the dual simplex.

1 If initial basic solution is dual feasible, use the dual simplex.

`price (LPX_K_PRICE, default: 1)`
 Pricing option (for both primal and dual simplex):

0	Textbook pricing.
1	Steepest edge pricing.

round (LPX_K_ROUND, default: 0)
Solution rounding option:

0	Report all primal and dual values "as is".
1	Replace tiny primal and dual values by exact zero.

itlim (LPX_K_ITLIM, default: -1)
Simplex iterations limit. If this value is positive, it is decreased by one each time when one simplex iteration has been performed, and reaching zero value signals the solver to stop the search. Negative value means no iterations limit.

itcnt (LPX_K_OUTFRQ, default: 200)
Output frequency, in iterations. This parameter specifies how frequently the solver sends information about the solution to the standard output.

branch (LPX_K_BRANCH, default: 2)
Branching heuristic option (for MIP only):

0	Branch on the first variable.
1	Branch on the last variable.
2	Branch using a heuristic by Driebeck and Tomlin.

btrack (LPX_K_BTRACK, default: 2)
Backtracking heuristic option (for MIP only):

0	Depth first search.
1	Breadth first search.
2	Backtrack using the best projection heuristic.

presol (LPX_K_PRESOL, default: 1)
If this flag is set, the routine `lpx_simplex` solves the problem using the built-in LP presolver. Otherwise the LP presolver is not used.

lpsolver (default: 1)
Select which solver to use. If the problem is a MIP problem this flag will be ignored.

1	Revised simplex method.
2	Interior point method.

save (default: 0)
If this parameter is nonzero, save a copy of the problem problem in CPLEX LP format to the file `"outpb.lp"`. There is currently no way to change the name of the output file.

Real parameters:

`relax (LPX_K_RELAX, default: 0.07)`

Relaxation parameter used in the ratio test. If it is zero, the textbook ratio test is used. If it is non-zero (should be positive), Harris' two-pass ratio test is used. In the latter case on the first pass of the ratio test basic variables (in the case of primal simplex) or reduced costs of non-basic variables (in the case of dual simplex) are allowed to slightly violate their bounds, but not more than `relax*tolbnd` or `relax*toldj` (thus, `relax` is a percentage of `tolbnd` or `toldj`).

`tolbnd (LPX_K_TOLBND, default: 10e-7)`

Relative tolerance used to check if the current basic solution is primal feasible. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

`toldj (LPX_K_TOLDJ, default: 10e-7)`

Absolute tolerance used to check if the current basic solution is dual feasible. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

`tolpiv (LPX_K_TOLPIV, default: 10e-9)`

Relative tolerance used to choose eligible pivotal elements of the simplex table. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

`objll (LPX_K_OBJLL, default: -DBL_MAX)`

Lower limit of the objective function. If on the phase II the objective function reaches this limit and continues decreasing, the solver stops the search. This parameter is used in the dual simplex method only.

`objul (LPX_K_OBJUL, default: +DBL_MAX)`

Upper limit of the objective function. If on the phase II the objective function reaches this limit and continues increasing, the solver stops the search. This parameter is used in the dual simplex only.

`tmlim (LPX_K_TMLIM, default: -1.0)`

Searching time limit, in seconds. If this value is positive, it is decreased each time when one simplex iteration has been performed by the amount of time spent for the iteration, and reaching zero value signals the solver to stop the search. Negative value means no time limit.

`outdly (LPX_K_OUTDLY, default: 0.0)`

Output delay, in seconds. This parameter specifies how long the solver should delay sending information about the solu-

tion to the standard output. Non-positive value means no delay.

`tolint (LPX_K_TOLINT, default: 10e-5)`

Relative tolerance used to check if the current basic solution is integer feasible. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

`tolobj (LPX_K_TOLOBJ, default: 10e-7)`

Relative tolerance used to check if the value of the objective function is not better than in the best known integer feasible solution. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

Output values:

xopt The optimizer (the value of the decision variables at the optimum).

fopt The optimum value of the objective function.

status Status of the optimization.

Simplex Method:

180 (LPX_OPT)

Solution is optimal.

181 (LPX_FEAS)

Solution is feasible.

182 (LPX_INFEAS)

Solution is infeasible.

183 (LPX_NOFEAS)

Problem has no feasible solution.

184 (LPX_UNBND)

Problem has no unbounded solution.

185 (LPX_UNDEF)

Solution status is undefined.

Interior Point Method:

150 (LPX_T_UNDEF)

The interior point method is undefined.

151 (LPX_T_OPT)

The interior point method is optimal.

Mixed Integer Method:

170 (LPX_I_UNDEF)

The status is undefined.

171 (LPX_I_OPT)

The solution is integer optimal.

172 (LPX_I_FEAS)
 Solution integer feasible but its optimality has not been
 proven

173 (LPX_I_NOFEAS)
 No integer feasible solution.

If an error occurs, *status* will contain one of the following codes:

204 (LPX_E_FAULT)
 Unable to start the search.

205 (LPX_E_OBJLL)
 Objective function lower limit reached.

206 (LPX_E_OBJUL)
 Objective function upper limit reached.

207 (LPX_E_ITLIM)
 Iterations limit exhausted.

208 (LPX_E_TMLIM)
 Time limit exhausted.

209 (LPX_E_NOFEAS)
 No feasible solution.

210 (LPX_E_INSTAB)
 Numerical instability.

211 (LPX_E_SING)
 Problems with basis matrix.

212 (LPX_E_NOCONV)
 No convergence (interior).

213 (LPX_E_NOPFS)
 No primal feasible solution (LP presolver).

214 (LPX_E_NODFS)
 No dual feasible solution (LP presolver).

extra A data structure containing the following fields:

lambda Dual variables.

redcosts Reduced Costs.

time Time (in seconds) used for solving LP/MIP problem.

mem Memory (in bytes) used for solving LP/MIP problem (this is
 not available if the version of GLPK is 4.15 or later).

Example:

```

c = [10, 6, 4]';
a = [ 1, 1, 1;
     10, 4, 5;
      2, 2, 6];
b = [100, 600, 300]';
lb = [0, 0, 0]';
ub = [];
ctype = "UUU";
vartype = "CCC";
s = -1;

param.msglev = 1;
param.itlim = 100;

[xmin, fmin, status, extra] = ...
    glpk (c, a, b, lb, ub, ctype, vartype, s, param);

```

24.2 Quadratic Programming

Octave can also solve Quadratic Programming problems, this is

$$\min_x \frac{1}{2} x^T H x + x^T q$$

subject to

$$Ax = b \quad lb \leq x \leq ub \quad A_{lb} \leq A_{in} \leq A_{ub}$$

```
[x, obj, info, lambda] = qp (x0, H, q, A, b, lb, ub, A_lb,      [Function File]
                             A_in, A_ub)
```

Solve the quadratic program

$$\min_x \frac{1}{2} x^T H x + x^T q$$

subject to

$$Ax = b \quad lb \leq x \leq ub \quad A_{lb} \leq A_{in} \leq A_{ub}$$

using a null-space active-set method.

Any bound (A , b , lb , ub , A_{lb} , A_{ub}) may be set to the empty matrix ($[]$) if not present. If the initial guess is feasible the algorithm is faster.

The value *info* is a structure with the following fields:

solveiter

The number of iterations required to find the solution.

info

An integer indicating the status of the solution, as follows:

- | | |
|---|--|
| 0 | The problem is feasible and convex. Global solution found. |
| 1 | The problem is not convex. Local solution found. |
| 2 | The problem is not convex and unbounded. |
| 3 | Maximum number of iterations reached. |
| 6 | The problem is infeasible. |

24.3 Nonlinear Programming

Octave can also perform general nonlinear minimization using a successive quadratic programming solver.

`[x, obj, info, iter, nf, lambda] = sqp (x, phi, g, h)` [Function File]
Solve the nonlinear program

$$\min_x \phi(x)$$

subject to

$$g(x) = 0 \quad h(x) \geq 0$$

using a successive quadratic programming method.

The first argument is the initial guess for the vector x .

The second argument is a function handle pointing to the objective function. The objective function must be of the form

$$y = \text{phi} (x)$$

in which x is a vector and y is a scalar.

The second argument may also be a 2- or 3-element cell array of function handles. The first element should point to the objective function, the second should point to a function that computes the gradient of the objective function, and the third should point to a function to compute the hessian of the objective function. If the gradient function is not supplied, the gradient is computed by finite differences. If the hessian function is not supplied, a BFGS update formula is used to approximate the hessian.

If supplied, the gradient function must be of the form

$$g = \text{gradient} (x)$$

in which x is a vector and g is a vector.

If supplied, the hessian function must be of the form

$$h = \text{hessian} (x)$$

in which x is a vector and h is a matrix.

The third and fourth arguments are function handles pointing to functions that compute the equality constraints and the inequality constraints, respectively.

If your problem does not have equality (or inequality) constraints, you may pass an empty matrix for *cef* (or *cif*).

If supplied, the equality and inequality constraint functions must be of the form

$$r = f (x)$$

in which x is a vector and r is a vector.

The third and fourth arguments may also be 2-element cell arrays of function handles. The first element should point to the constraint function and the second should point to a function that computes the gradient of the constraint function:

$$\left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \dots, \frac{\partial f(x)}{\partial x_N} \right)^T$$

Here is an example of calling `sqp`:


```

function r = g (x)
  r = [ sumsq(x)-10; x(2)*x(3)-5*x(4)*x(5); x(1)^3+x(2)^3+1];
endfunction

function obj = phi (x)
  obj = exp(prod(x)) - 0.5*(x(1)^3+x(2)^3+1)^2;
endfunction

x0 = [-1.8; 1.7; 1.9; -0.8; -0.8];

[x, obj, info, iter, nf, lambda] = sqp (x0, @phi, @g, [])

x =

    -1.71714
     1.59571
     1.82725
    -0.76364
    -0.76364

obj = 0.053950
info = 101
iter = 8
nf = 10
lambda =

    -0.0401627
     0.0379578
    -0.0052227

```

The value returned in *info* may be one of the following:

- 101 The algorithm terminated because the norm of the last step was less than `tol * norm (x)` (the value of `tol` is currently fixed at `sqrt (eps)`—edit ‘`sqp.m`’ to modify this value).
- 102 The BFGS update failed.
- 103 The maximum number of iterations was reached (the maximum number of allowed iterations is currently fixed at 100—edit ‘`sqp.m`’ to increase this value).

See also: `qp`.

24.4 Linear Least Squares

Octave also supports linear least squares minimization. That is, Octave can find the parameter b such the the model $y = xb$ fits data (x, y) as good as possible, assuming zero-mean Gaussian noise. If the noise is assumed to be isotropic the problem can be solved using the ‘\’ or ‘/’ operators, or the `ols` function. In the general case where the noise is assumed to be anisotropic the `gls` is needed.

`[beta, sigma, r] = ols (y, x)` [Function File]

Ordinary least squares estimation for the multivariate model $y = xb + e$ with $\bar{e} = 0$, and $\text{cov}(\text{vec}(e)) = \text{kron}(s, I)$ where y is a $t \times p$ matrix, x is a $t \times k$ matrix, b is a $k \times p$ matrix, and e is a $t \times p$ matrix.

Each row of y and x is an observation and each column a variable.

The return values $beta$, $sigma$, and r are defined as follows.

$beta$ The OLS estimator for b , $beta = \text{pinv}(x) * y$, where $\text{pinv}(x)$ denotes the pseudoinverse of x .

$sigma$ The OLS estimator for the matrix s ,

$$sigma = (y - x * beta)' \\ * (y - x * beta) \\ / (t - \text{rank}(x))$$

r The matrix of OLS residuals, $r = y - x * beta$.

`[beta, v, r] = gls (y, x, o)` [Function File]

Generalized least squares estimation for the multivariate model $y = xb + e$ with $\bar{e} = 0$ and $\text{cov}(\text{vec}(e)) = (s^2)o$, where y is a $t \times p$ matrix, x is a $t \times k$ matrix, b is a $k \times p$ matrix, e is a $t \times p$ matrix, and o is a $tp \times tp$ matrix.

Each row of y and x is an observation and each column a variable. The return values $beta$, v , and r are defined as follows.

$beta$ The GLS estimator for b .

v The GLS estimator for s^2 .

r The matrix of GLS residuals, $r = y - xbeta$.

25 Statistics

Octave has support for various statistical methods. This includes basic descriptive statistics, statistical tests, random number generation, and much more.

The functions that analyze data all assume that multidimensional data is arranged in a matrix where each row is an observation, and each column is a variable. So, the matrix defined by

```
a = [ 0.9, 0.7;
      0.1, 0.1;
      0.5, 0.4 ];
```

contains three observations from a two-dimensional distribution. While this is the default data arrangement, most functions support different arrangements.

It should be noted that the statistics functions doesn't handle data containing NaN, NA, or Inf. Such values needs to be handled explicitly.

25.1 Descriptive Statistics

Octave can compute various statistics such as the moments of a data set.

mean (*x*, *dim*, *opt*) [Function File]

If *x* is a vector, compute the mean of the elements of *x*

$$\text{mean}(x) = \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

If *x* is a matrix, compute the mean for each column and return them in a row vector.

With the optional argument *opt*, the kind of mean computed can be selected. The following options are recognized:

"a"	Compute the (ordinary) arithmetic mean. This is the default.
"g"	Computer the geometric mean.
"h"	Compute the harmonic mean.

If the optional argument *dim* is supplied, work along dimension *dim*.

Both *dim* and *opt* are optional. If both are supplied, either may appear first.

median (*x*, *dim*) [Function File]

If *x* is a vector, compute the median value of the elements of *x*. If the elements of *x* are sorted, the median is defined as

$$\text{median}(x) = \begin{cases} x(\lceil N/2 \rceil), & N \text{ odd;} \\ (x(N/2) + x(N/2 + 1))/2, & N \text{ even.} \end{cases}$$

If *x* is a matrix, compute the median value for each column and return them in a row vector. If the optional *dim* argument is given, operate along this dimension.

See also: std, mean.

meansq (*x*) [Function File]
meansq (*x*, *dim*) [Function File]

For vector arguments, return the mean square of the values. For matrix arguments, return a row vector containing the mean square of each column. With the optional *dim* argument, returns the mean squared of the values along this dimension.

std (*x*) [Function File]
std (*x*, *opt*) [Function File]
std (*x*, *opt*, *dim*) [Function File]

If *x* is a vector, compute the standard deviation of the elements of *x*.

$$\text{std}(x) = \sigma(x) = \sqrt{\frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N - 1}}$$

where \bar{x} is the mean value of *x*. If *x* is a matrix, compute the standard deviation for each column and return them in a row vector.

The argument *opt* determines the type of normalization to use. Valid values are

- 0: normalizes with $N - 1$, provides the square root of best unbiased estimator of the variance [default]
- 1: normalizes with N , this provides the square root of the second moment around the mean

The third argument *dim* determines the dimension along which the standard deviation is calculated.

See also: mean, median.

var (*x*) [Function File]

For vector arguments, return the (real) variance of the values. For matrix arguments, return a row vector containing the variance for each column.

The argument *opt* determines the type of normalization to use. Valid values are

- 0: Normalizes with $N - 1$, provides the best unbiased estimator of the variance [default].
- 1: Normalizes with N , this provides the second moment around the mean.

The third argument *dim* determines the dimension along which the variance is calculated.

cov (*x*, *y*) [Function File]

Compute covariance.

If each row of *x* and *y* is an observation and each column is a variable, the (*i*, *j*)-th entry of **cov** (*x*, *y*) is the covariance between the *i*-th variable in *x* and the *j*-th variable in *y*.

$$\sigma_{ij} = \frac{1}{N - 1} \sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y})$$

where \bar{x} and \bar{y} are the mean values of *x* and *y*. If called with one argument, compute **cov** (*x*, *x*).

cor (**x**, **y**) [Function File]

Compute correlation.

The (i, j) -th entry of **cor** (**x**, **y**) is the correlation between the i -th variable in **x** and the j -th variable in **y**.

$$\text{corrcoef}(x, y) = \frac{\text{cov}(x, y)}{\text{std}(x)\text{std}(y)}$$

For matrices, each row is an observation and each column a variable; vectors are always observations and may be row or column vectors.

cor (**x**) is equivalent to **cor** (**x**, **x**).

Note that the **corrcoef** function does the same as **cor**.

corrcoef (**x**, **y**) [Function File]

Compute correlation.

If each row of **x** and **y** is an observation and each column is a variable, the (i, j) -th entry of **corrcoef** (**x**, **y**) is the correlation between the i -th variable in **x** and the j -th variable in **y**.

$$\text{corrcoef}(x, y) = \frac{\text{cov}(x, y)}{\text{std}(x)\text{std}(y)}$$

If called with one argument, compute **corrcoef** (**x**, **x**).

kurtosis (**x**, **dim**) [Function File]

If **x** is a vector of length N , return the kurtosis

$$\text{kurtosis}(x) = \frac{1}{N\sigma(x)^4} \sum_{i=1}^N (x_i - \bar{x})^4 - 3$$

where \bar{x} is the mean value of x .

of **x**. If **x** is a matrix, return the kurtosis over the first non-singleton dimension. The optional argument **dim** can be given to force the kurtosis to be given over that dimension.

skewness (**x**, **dim**) [Function File]

If **x** is a vector of length n , return the skewness

$$\text{skewness}(x) = \frac{1}{N\sigma(x)^3} \sum_{i=1}^N (x_i - \bar{x})^3$$

where \bar{x} is the mean value of x .

of **x**. If **x** is a matrix, return the skewness along the first non-singleton dimension of the matrix. If the optional **dim** argument is given, operate along this dimension.

statistics (*x*) [Function File]

If *x* is a matrix, return a matrix with the minimum, first quartile, median, third quartile, maximum, mean, standard deviation, skewness and kurtosis of the columns of *x* as its rows.

If *x* is a vector, treat it as a column vector.

moment (*x*, *p*, *opt*, *dim*) [Function File]

If *x* is a vector, compute the *p*-th moment of *x*.

If *x* is a matrix, return the row vector containing the *p*-th moment of each column.

With the optional string *opt*, the kind of moment to be computed can be specified. If *opt* contains "c" or "a", central and/or absolute moments are returned. For example,

```
moment (x, 3, "ac")
```

computes the third central absolute moment of *x*.

If the optional argument *dim* is supplied, work along dimension *dim*.

25.2 Basic Statistical Functions

Octave also supports various helpful statistical functions.

mahalanobis (*x*, *y*) [Function File]

Return the Mahalanobis' D-square distance between the multivariate samples *x* and *y*, which must have the same number of components (columns), but may have a different number of observations (rows).

center (*x*) [Function File]

center (*x*, *dim*) [Function File]

If *x* is a vector, subtract its mean. If *x* is a matrix, do the above for each column. If the optional argument *dim* is given, perform the above operation along this dimension

studentize (*x*, *dim*) [Function File]

If *x* is a vector, subtract its mean and divide by its standard deviation.

If *x* is a matrix, do the above along the first non-singleton dimension. If the optional argument *dim* is given then operate along this dimension.

c = **nchoosek** (*n*, *k*) [Function File]

Compute the binomial coefficient or all combinations of *n*. If *n* is a scalar then, calculate the binomial coefficient of *n* and *k*, defined as

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$$

If *n* is a vector generate all combinations of the elements of *n*, taken *k* at a time, one row per combination. The resulting *c* has size **[nchoosek (length (*n*), *k*), *k*]**.

See also: **bincoeff**.

perms (*v*) [Function File]

Generate all permutations of *v*, one row per permutation. The result has size **factorial** (*n*) * *n*, where *n* is the length of *v*.

As an example, **perms**([1, 2, 3]) returns the matrix

1	2	3
2	1	3
1	3	2
2	3	1
3	1	2
3	2	1

values (*x*) [Function File]

Return the different values in a column vector, arranged in ascending order.

As an example, **values**([1, 2, 3, 1]) returns the vector [1, 2, 3].

[t, l_x] = table (*x*) [Function File]

[t, l_x, l_y] = table (*x*, *y*) [Function File]

Create a contingency table *t* from data vectors. The *l* vectors are the corresponding levels.

Currently, only 1- and 2-dimensional tables are supported.

spearman (*x*, *y*) [Function File]

Compute Spearman's rank correlation coefficient *rho* for each of the variables specified by the input arguments.

For matrices, each row is an observation and each column a variable; vectors are always observations and may be row or column vectors.

spearman (*x*) is equivalent to **spearman** (*x*, *x*).

For two data vectors *x* and *y*, Spearman's *rho* is the correlation of the ranks of *x* and *y*.

If *x* and *y* are drawn from independent distributions, *rho* has zero mean and variance $1 / (n - 1)$, and is asymptotically normally distributed.

run_count (*x*, *n*) [Function File]

Count the upward runs along the first non-singleton dimension of *x* of length 1, 2, ..., *n*-1 and greater than or equal to *n*. If the optional argument *dim* is given operate along this dimension

ranks (*x*, *dim*) [Function File]

If *x* is a vector, return the (column) vector of ranks of *x* adjusted for ties.

If *x* is a matrix, do the above for along the first non-singleton dimension. If the optional argument *dim* is given, operate along this dimension.

range (*x*) [Function File]

range (*x*, *dim*) [Function File]

If *x* is a vector, return the range, i.e., the difference between the maximum and the minimum, of the input data.

If *x* is a matrix, do the above for each column of *x*.

If the optional argument *dim* is supplied, work along dimension *dim*.

probit (*p*) [Function File]

For each component of *p*, return the probit (the quantile of the standard normal distribution) of *p*.

logit (*p*) [Function File]

For each component of *p*, return the logit of *p* defined as

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

cloglog (*x*) [Function File]

Return the complementary log-log function of *x*, defined as

$$\text{cloglog}(x) = -\log(-\log(x))$$

kendall (*x*, *y*) [Function File]

Compute Kendall's *tau* for each of the variables specified by the input arguments.

For matrices, each row is an observation and each column a variable; vectors are always observations and may be row or column vectors.

kendall (*x*) is equivalent to **kendall** (*x*, *x*).

For two data vectors *x*, *y* of common length *n*, Kendall's *tau* is the correlation of the signs of all rank differences of *x* and *y*; i.e., if both *x* and *y* have distinct entries, then

$$\tau = \frac{1}{n(n-1)} \sum_{i,j} \text{sign}(q_i - q_j) \text{sign}(r_i - r_j)$$

in which the q_i and r_i are the ranks of *x* and *y*, respectively.

If *x* and *y* are drawn from independent distributions, Kendall's *tau* is asymptotically normal with mean 0 and variance $\frac{2(2n+5)}{9n(n-1)}$.

iqr (*x*, *dim*) [Function File]

If *x* is a vector, return the interquartile range, i.e., the difference between the upper and lower quartile, of the input data.

If *x* is a matrix, do the above for first non singleton dimension of *x*. If the option *dim* argument is given, then operate along this dimension.

`cut (x, breaks)` [Function File]

Create categorical data out of numerical or continuous data by cutting into intervals.

If *breaks* is a scalar, the data is cut into that many equal-width intervals. If *breaks* is a vector of break points, the category has `length (breaks) - 1` groups.

The returned value is a vector of the same size as *x* telling which group each point in *x* belongs to. Groups are labelled from 1 to the number of groups; points outside the range of *breaks* are labelled by `NaN`.

25.3 Statistical Plots

Octave can create Quantile Plots (QQ-Plots), and Probability Plots (PP-Plots). These are simple graphical tests for determining if a data set comes from a certain distribution.

It is worth noticing that Octave can also show histograms of data using the `hist` function as described in [Section 16.2 \[Specialized Two-Dimensional Plots\]](#), page 182.

`[q, s] = qqplot (x, dist, params)` [Function File]

Perform a QQ-plot (quantile plot).

If *F* is the CDF of the distribution *dist* with parameters *params* and *G* its inverse, and *x* a sample vector of length *n*, the QQ-plot graphs ordinate $s(i) = i$ -th largest element of *x* versus abscissa $q(i) = G((i - 0.5)/n)$.

If the sample comes from *F* except for a transformation of location and scale, the pairs will approximately follow a straight line.

The default for *dist* is the standard normal distribution. The optional argument *params* contains a list of parameters of *dist*. For example, for a quantile plot of the uniform distribution on [2,4] and *x*, use

```
qqplot (x, "uniform", 2, 4)
```

dist can be any string for which a function *dist_inv* that calculates the inverse CDF of distribution *dist* exists.

If no output arguments are given, the data are plotted directly.

`[p, y] = ppplot (x, dist, params)` [Function File]

Perform a PP-plot (probability plot).

If *F* is the CDF of the distribution *dist* with parameters *params* and *x* a sample vector of length *n*, the PP-plot graphs ordinate $y(i) = F$ (*i*-th largest element of *x*) versus abscissa $p(i) = (i - 0.5)/n$. If the sample comes from *F*, the pairs will approximately follow a straight line.

The default for *dist* is the standard normal distribution. The optional argument *params* contains a list of parameters of *dist*. For example, for a probability plot of the uniform distribution on [2,4] and *x*, use

```
ppplot (x, "uniform", 2, 4)
```

dist can be any string for which a function *dist_cdf* that calculates the CDF of distribution *dist* exists.

If no output arguments are given, the data are plotted directly.

25.4 Tests

Octave can perform several different statistical tests. The following table summarizes the available tests.

Hypothesis	Test Functions
Equal mean values	<code>anova</code> , <code>hotelling_test2</code> , <code>t_test_2</code> , <code>welch_test</code> , <code>wilcoxon_test</code> , <code>z_test_2</code>
Equal medians	<code>kruskal_wallis_test</code> , <code>sign_test</code>
Equal variances	<code>bartlett_test</code> , <code>manova</code> , <code>var_test</code>
Equal distributions	<code>chisquare_test_homogeneity</code> , <code>kolmogorov_smirnov_test_2</code> , <code>u_test</code>
Equal marginal frequencies	<code>mcnemar_test</code>
Equal success probabilities	<code>prop_test_2</code>
Independent observations	<code>chisquare_test_independence</code> , <code>run_test</code>
Uncorrelated observations	<code>cor_test</code>
Given mean value	<code>hotelling_test</code> , <code>t_test</code> , <code>z_test</code>
Observations from given distribution	<code>kolmogorov_smirnov_test</code>
Regression	<code>f_test_regression</code> , <code>t_test_regression</code>

The tests return a p-value that describes the outcome of the test. Assuming that the test hypothesis is true, the p-value is the probability of obtaining a worse result than the observed one. So large p-values corresponds to a successful test. Usually a test hypothesis is accepted if the p-value exceeds 0.05.

`[pval, f, df_b, df_w] = anova (y, g)` [Function File]

Perform a one-way analysis of variance (ANOVA). The goal is to test whether the population means of data taken from k different groups are all equal.

Data may be given in a single vector y with groups specified by a corresponding vector of group labels g (e.g., numbers from 1 to k). This is the general form which does not impose any restriction on the number of data in each group or the group labels.

If y is a matrix and g is omitted, each column of y is treated as a group. This form is only appropriate for balanced ANOVA in which the numbers of samples from each group are all equal.

Under the null of constant means, the statistic f follows an F distribution with df_b and df_w degrees of freedom.

The p-value (1 minus the CDF of this distribution at f) is returned in $pval$.

If no output argument is given, the standard one-way ANOVA table is printed.

`[pval, chisq, df] = bartlett_test (x1, ...)` [Function File]

Perform a Bartlett test for the homogeneity of variances in the data vectors x_1 , x_2 , ..., x_k , where $k > 1$.

Under the null of equal variances, the test statistic $chisq$ approximately follows a chi-square distribution with df degrees of freedom.

The p-value (1 minus the CDF of this distribution at $chisq$) is returned in $pval$.

If no output argument is given, the p-value is displayed.

`[pval, chisq, df] = chisquare_test_homogeneity (x, y, c)` [Function File]

Given two samples *x* and *y*, perform a chisquare test for homogeneity of the null hypothesis that *x* and *y* come from the same distribution, based on the partition induced by the (strictly increasing) entries of *c*.

For large samples, the test statistic *chisq* approximately follows a chisquare distribution with $df = \text{length}(c)$ degrees of freedom.

The p-value (1 minus the CDF of this distribution at *chisq*) is returned in *pval*.

If no output argument is given, the p-value is displayed.

`[pval, chisq, df] = chisquare_test_independence (x)` [Function File]

Perform a chi-square test for indepenence based on the contingency table *x*. Under the null hypothesis of independence, *chisq* approximately has a chi-square distribution with *df* degrees of freedom.

The p-value (1 minus the CDF of this distribution at *chisq*) of the test is returned in *pval*.

If no output argument is given, the p-value is displayed.

`cor_test (x, y, alt, method)` [Function File]

Test whether two samples *x* and *y* come from uncorrelated populations.

The optional argument string *alt* describes the alternative hypothesis, and can be "*!=*" or "<>" (non-zero), ">" (greater than 0), or "<" (less than 0). The default is the two-sided case.

The optional argument string *method* specifies on which correlation coefficient the test should be based. If *method* is "**pearson**" (default), the (usual) Pearson's product moment correlation coefficient is used. In this case, the data should come from a bivariate normal distribution. Otherwise, the other two methods offer nonparametric alternatives. If *method* is "**kendall**", then Kendall's rank correlation tau is used. If *method* is "**spearman**", then Spearman's rank correlation rho is used. Only the first character is necessary.

The output is a structure with the following elements:

<i>pval</i>	The p-value of the test.
<i>stat</i>	The value of the test statistic.
<i>dist</i>	The distribution of the test statistic.
<i>params</i>	The parameters of the null distribution of the test statistic.
<i>alternative</i>	The alternative hypothesis.
<i>method</i>	The method used for testing.

If no output argument is given, the p-value is displayed.

`[pval, f, df_num, df_den] = f_test_regression (y, x, rr, r)` [Function File]

Perform an F test for the null hypothesis $rr * b = r$ in a classical normal regression model $y = X * b + e$.

Under the null, the test statistic f follows an F distribution with df_num and df_den degrees of freedom.

The p-value (1 minus the CDF of this distribution at f) is returned in $pval$.

If not given explicitly, $r = 0$.

If no output argument is given, the p-value is displayed.

`[pval, tsq] = hotelling_test (x, m)` [Function File]

For a sample x from a multivariate normal distribution with unknown mean and covariance matrix, test the null hypothesis that $\text{mean}(x) == m$.

Hotelling's T^2 is returned in tsq . Under the null, $(n - p)T^2 / (p(n - 1))$ has an F distribution with p and $n - p$ degrees of freedom, where n and p are the numbers of samples and variables, respectively.

The p-value of the test is returned in $pval$.

If no output argument is given, the p-value of the test is displayed.

`[pval, tsq] = hotelling_test_2 (x, y)` [Function File]

For two samples x from multivariate normal distributions with the same number of variables (columns), unknown means and unknown equal covariance matrices, test the null hypothesis $\text{mean}(x) == \text{mean}(y)$.

Hotelling's two-sample T^2 is returned in tsq . Under the null,

$$\frac{(n_x + n_y - p - 1)T^2}{p(n_x + n_y - 2)}$$

has an F distribution with p and $n_x + n_y - p - 1$ degrees of freedom, where n_x and n_y are the sample sizes and p is the number of variables.

The p-value of the test is returned in $pval$.

If no output argument is given, the p-value of the test is displayed.

`[pval, ks] = kolmogorov_smirnov_test (x, dist, params, alt)` [Function File]

Perform a Kolmogorov-Smirnov test of the null hypothesis that the sample x comes from the (continuous) distribution $dist$. I.e., if F and G are the CDFs corresponding to the sample and $dist$, respectively, then the null is that $F == G$.

The optional argument $params$ contains a list of parameters of $dist$. For example, to test whether a sample x comes from a uniform distribution on $[2,4]$, use

```
kolmogorov_smirnov_test(x, "uniform", 2, 4)
```

$dist$ can be any string for which a function $dist_cdf$ that calculates the CDF of distribution $dist$ exists.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is "!=" or "<>", the null is tested against the two-sided alternative $F \neq G$. In this case, the test statistic *ks* follows a two-sided Kolmogorov-Smirnov distribution. If *alt* is ">", the one-sided alternative $F > G$ is considered. Similarly for "<", the one-sided alternative $F < G$ is considered. In this case, the test statistic *ks* has a one-sided Kolmogorov-Smirnov distribution. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value is displayed.

`[pval, ks, d] = kolmogorov_smirnov_test_2(x, y, alt)` [Function File]

Perform a 2-sample Kolmogorov-Smirnov test of the null hypothesis that the samples *x* and *y* come from the same (continuous) distribution. I.e., if *F* and *G* are the CDFs corresponding to the *x* and *y* samples, respectively, then the null is that $F = G$.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is "!=" or "<>", the null is tested against the two-sided alternative $F \neq G$. In this case, the test statistic *ks* follows a two-sided Kolmogorov-Smirnov distribution. If *alt* is ">", the one-sided alternative $F > G$ is considered. Similarly for "<", the one-sided alternative $F < G$ is considered. In this case, the test statistic *ks* has a one-sided Kolmogorov-Smirnov distribution. The default is the two-sided case.

The p-value of the test is returned in *pval*.

The third returned value, *d*, is the test statistic, the maximum vertical distance between the two cumulative distribution functions.

If no output argument is given, the p-value is displayed.

`[pval, k, df] = kruskal_wallis_test(x1, ...)` [Function File]

Perform a Kruskal-Wallis one-factor "analysis of variance".

Suppose a variable is observed for $k > 1$ different groups, and let *x1*, ..., *xk* be the corresponding data vectors.

Under the null hypothesis that the ranks in the pooled sample are not affected by the group memberships, the test statistic *k* is approximately chi-square with $df = k - 1$ degrees of freedom.

The p-value (1 minus the CDF of this distribution at *k*) is returned in *pval*.

If no output argument is given, the p-value is displayed.

`manova(y, g)` [Function File]

Perform a one-way multivariate analysis of variance (MANOVA). The goal is to test whether the *p*-dimensional population means of data taken from *k* different groups are all equal. All data are assumed drawn independently from *p*-dimensional normal distributions with the same covariance matrix.

The data matrix is given by *y*. As usual, rows are observations and columns are variables. The vector *g* specifies the corresponding group labels (e.g., numbers from 1 to *k*).

The LR test statistic (Wilks' Lambda) and approximate p-values are computed and displayed.

`[pval, chisq, df] = mcnemar_test (x)` [Function File]

For a square contingency table *x* of data cross-classified on the row and column variables, McNemar's test can be used for testing the null hypothesis of symmetry of the classification probabilities.

Under the null, *chisq* is approximately distributed as chisquare with *df* degrees of freedom.

The p-value (1 minus the CDF of this distribution at *chisq*) is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, z] = prop_test_2 (x1, n1, x2, n2, alt)` [Function File]

If *x1* and *n1* are the counts of successes and trials in one sample, and *x2* and *n2* those in a second one, test the null hypothesis that the success probabilities *p1* and *p2* are the same. Under the null, the test statistic *z* approximately follows a standard normal distribution.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is "*!=*" or "*<>*", the null is tested against the two-sided alternative *p1* *!=* *p2*. If *alt* is "*>*", the one-sided alternative *p1* *>* *p2* is used. Similarly for "*<*", the one-sided alternative *p1* *<* *p2* is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, chisq] = run_test (x)` [Function File]

Perform a chi-square test with 6 degrees of freedom based on the upward runs in the columns of *x*. Can be used to test whether *x* contains independent data.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value is displayed.

`[pval, b, n] = sign_test (x, y, alt)` [Function File]

For two matched-pair samples *x* and *y*, perform a sign test of the null hypothesis $\text{PROB}(x > y) == \text{PROB}(x < y) == 1/2$. Under the null, the test statistic *b* roughly follows a binomial distribution with parameters $n = \text{sum}(x \neq y)$ and $p = 1/2$.

With the optional argument *alt*, the alternative of interest can be selected. If *alt* is "*!=*" or "*<>*", the null hypothesis is tested against the two-sided alternative $\text{PROB}(x < y) \neq 1/2$. If *alt* is "*>*", the one-sided alternative $\text{PROB}(x > y) > 1/2$ ("x is stochastically greater than y") is considered. Similarly for "*<*", the one-sided alternative $\text{PROB}(x > y) < 1/2$ ("x is stochastically less than y") is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, t, df] = t_test (x, m, alt)` [Function File]

For a sample *x* from a normal distribution with unknown mean and variance, perform a t-test of the null hypothesis $\text{mean}(x) == m$. Under the null, the test statistic *t* follows a Student distribution with $df = \text{length}(x) - 1$ degrees of freedom.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is " \neq " or " $<>$ ", the null is tested against the two-sided alternative $\text{mean}(x) \neq m$. If *alt* is " $>$ ", the one-sided alternative $\text{mean}(x) > m$ is considered. Similarly for " $<$ ", the one-sided alternative $\text{mean}(x) < m$ is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, t, df] = t_test_2(x, y, alt)` [Function File]

For two samples *x* and *y* from normal distributions with unknown means and unknown equal variances, perform a two-sample t-test of the null hypothesis of equal means. Under the null, the test statistic *t* follows a Student distribution with *df* degrees of freedom.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is " \neq " or " $<>$ ", the null is tested against the two-sided alternative $\text{mean}(x) \neq \text{mean}(y)$. If *alt* is " $>$ ", the one-sided alternative $\text{mean}(x) > \text{mean}(y)$ is used. Similarly for " $<$ ", the one-sided alternative $\text{mean}(x) < \text{mean}(y)$ is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, t, df] = t_test_regression(y, x, rr, r, alt)` [Function File]

Perform an t test for the null hypothesis $rr * b = r$ in a classical normal regression model $y = x * b + e$. Under the null, the test statistic *t* follows a *t* distribution with *df* degrees of freedom.

If *r* is omitted, a value of 0 is assumed.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is " \neq " or " $<>$ ", the null is tested against the two-sided alternative $rr * b \neq r$. If *alt* is " $>$ ", the one-sided alternative $rr * b > r$ is used. Similarly for " $<$ ", the one-sided alternative $rr * b < r$ is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, z] = u_test(x, y, alt)` [Function File]

For two samples *x* and *y*, perform a Mann-Whitney U-test of the null hypothesis $\text{PROB}(x > y) == 1/2 == \text{PROB}(x < y)$. Under the null, the test statistic *z* approximately follows a standard normal distribution. Note that this test is equivalent to the Wilcoxon rank-sum test.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is " \neq " or " $<>$ ", the null is tested against the two-sided alternative $\text{PROB}(x > y) \neq 1/2$. If *alt* is " $>$ ", the one-sided alternative $\text{PROB}(x > y) > 1/2$ is considered. Similarly for " $<$ ", the one-sided alternative $\text{PROB}(x > y) < 1/2$ is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, f, df_num, df_den] = var_test (x, y, alt)` [Function File]

For two samples x and y from normal distributions with unknown means and unknown variances, perform an F-test of the null hypothesis of equal variances. Under the null, the test statistic f follows an F-distribution with df_num and df_den degrees of freedom.

With the optional argument string alt , the alternative of interest can be selected. If alt is `"!="` or `"<>"`, the null is tested against the two-sided alternative `var (x) != var (y)`. If alt is `">"`, the one-sided alternative `var (x) > var (y)` is used. Similarly for `"<"`, the one-sided alternative `var (x) < var (y)` is used. The default is the two-sided case.

The p-value of the test is returned in $pval$.

If no output argument is given, the p-value of the test is displayed.

`[pval, t, df] = welch_test (x, y, alt)` [Function File]

For two samples x and y from normal distributions with unknown means and unknown and not necessarily equal variances, perform a Welch test of the null hypothesis of equal means. Under the null, the test statistic t approximately follows a Student distribution with df degrees of freedom.

With the optional argument string alt , the alternative of interest can be selected. If alt is `"!="` or `"<>"`, the null is tested against the two-sided alternative `mean (x) != m`. If alt is `">"`, the one-sided alternative `mean(x) > m` is considered. Similarly for `"<"`, the one-sided alternative `mean(x) < m` is considered. The default is the two-sided case.

The p-value of the test is returned in $pval$.

If no output argument is given, the p-value of the test is displayed.

`[pval, z] = wilcoxon_test (x, y, alt)` [Function File]

For two matched-pair sample vectors x and y , perform a Wilcoxon signed-rank test of the null hypothesis `PROB (x > y) == 1/2`. Under the null, the test statistic z approximately follows a standard normal distribution when $n > 25$.

Warning: This function assumes a normal distribution for z and thus is invalid for $n \leq 25$.

With the optional argument string alt , the alternative of interest can be selected. If alt is `"!="` or `"<>"`, the null is tested against the two-sided alternative `PROB (x > y) != 1/2`. If alt is `">"`, the one-sided alternative `PROB (x > y) > 1/2` is considered. Similarly for `"<"`, the one-sided alternative `PROB (x > y) < 1/2` is considered. The default is the two-sided case.

The p-value of the test is returned in $pval$.

If no output argument is given, the p-value of the test is displayed.

`[pval, z] = z_test (x, m, v, alt)` [Function File]

Perform a Z-test of the null hypothesis `mean (x) == m` for a sample x from a normal distribution with unknown mean and known variance v . Under the null, the test statistic z follows a standard normal distribution.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is "!=" or "<>", the null is tested against the two-sided alternative $\text{mean}(x) \neq m$. If *alt* is ">", the one-sided alternative $\text{mean}(x) > m$ is considered. Similarly for "<", the one-sided alternative $\text{mean}(x) < m$ is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed along with some information.

`[pval, z] = z_test_2(x, y, v_x, v_y, alt)` [Function File]

For two samples *x* and *y* from normal distributions with unknown means and known variances *v_x* and *v_y*, perform a Z-test of the hypothesis of equal means. Under the null, the test statistic *z* follows a standard normal distribution.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is "!=" or "<>", the null is tested against the two-sided alternative $\text{mean}(x) \neq \text{mean}(y)$. If *alt* is ">", the one-sided alternative $\text{mean}(x) > \text{mean}(y)$ is used. Similarly for "<", the one-sided alternative $\text{mean}(x) < \text{mean}(y)$ is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed along with some information.

25.5 Models

`[theta, beta, dev, dl, d2l, p] = logistic_regression(y, x, print, theta, beta)` [Function File]

Perform ordinal logistic regression.

Suppose *y* takes values in *k* ordered categories, and let $\text{gamma}_i(x)$ be the cumulative probability that *y* falls in one of the first *i* categories given the covariate *x*. Then

$$[\text{theta}, \text{beta}] = \text{logistic_regression}(y, x)$$

fits the model

$$\text{logit}(\text{gamma}_i(x)) = \text{theta}_i - \text{beta}' * x, \quad i = 1, \dots, k-1$$

The number of ordinal categories, *k*, is taken to be the number of distinct values of `round(y)`. If *k* equals 2, *y* is binary and the model is ordinary logistic regression. The matrix *x* is assumed to have full column rank.

Given *y* only, `theta = logistic_regression(y)` fits the model with baseline logit odds only.

The full form is

$$[\text{theta}, \text{beta}, \text{dev}, \text{dl}, \text{d2l}, \text{gamma}] = \text{logistic_regression}(y, x, \text{print}, \text{theta}, \text{beta})$$

in which all output arguments and all input arguments except *y* are optional.

Setting *print* to 1 requests summary information about the fitted model to be displayed. Setting *print* to 2 requests information about convergence at each iteration.

Other values request no information to be displayed. The input arguments *theta* and *beta* give initial estimates for *theta* and *beta*.

The returned value *dev* holds minus twice the log-likelihood.

The returned values *dl* and *d2l* are the vector of first and the matrix of second derivatives of the log-likelihood with respect to *theta* and *beta*.

p holds estimates for the conditional distribution of *y* given *x*.

25.6 Distributions

Octave has functions for computing the Probability Density Function (PDF), the Cumulative Distribution function (CDF), and the quantile (the inverse of the CDF) of a large number of distributions.

The following table summarizes the supported distributions (in alphabetical order).

Distribution	PDF	CDF	Quantile
Beta Distribution	betapdf	betacdf	betainv
Binomial Distribution	binopdf	binocdf	binoinv
Cauchy Distribution	cauchy_pdf	cauchy_cdf	cauchy_inv
Chi-Square Distribution	chi2pdf	chi2cdf	chi2inv
Univariate Discrete Distribution	discrete_pdf	discrete_cdf	discrete_inv
Empirical Distribution	empirical_ pdf	empirical_ cdf	empirical_ inv
Exponential Distribution	exppdf	expcdf	expinv
F Distribution	fpdf	fcdf	finv
Gamma Distribution	gampdf	gamcdf	gaminv
Geometric Distribution	geopdf	geocdf	geoinv
Hypergeometric Distribution	hygepdf	hygecdf	hygeinv
Kolmogorov Smirnov Distribution	<i>Not Available</i>	kolmogorov_ smirnov_cdf	<i>Not Available</i>
Laplace Distribution	laplace_pdf	laplace_cdf	laplace_inv
Logistic Distribution	logistic_pdf	logistic_cdf	logistic_inv
Log-Normal Distribution	lognpdf	logncdf	logninv
Pascal Distribution	nbinpdl	nbincdf	nbinin
Univariate Normal Distribution	normpdf	normcdf	norminv
Poisson Distribution	poisspdf	poisscdf	poissinv
t (Student) Distribution	tpdf	tcdf	tin
Univariate Discrete Distribution	unidpdf	unidcdf	unidinv
Uniform Distribution	unifpdf	unifcdf	unifinv
Weibull Distribution	wblpdf	wblcdf	wblinv

betacdf (*x*, *a*, *b*) [Function File]

For each element of *x*, returns the CDF at *x* of the beta distribution with parameters *a* and *b*, i.e., $\text{PROB}(\text{beta}(a, b) \leq x)$.

betainv (*x*, *a*, *b*) [Function File]

For each component of *x*, compute the quantile (the inverse of the CDF) at *x* of the Beta distribution with parameters *a* and *b*.

betapdf (*x*, *a*, *b*) [Function File]
For each element of *x*, returns the PDF at *x* of the beta distribution with parameters *a* and *b*.

binocdf (*x*, *n*, *p*) [Function File]
For each element of *x*, compute the CDF at *x* of the binomial distribution with parameters *n* and *p*.

binoinv (*x*, *n*, *p*) [Function File]
For each element of *x*, compute the quantile at *x* of the binomial distribution with parameters *n* and *p*.

binopdf (*x*, *n*, *p*) [Function File]
For each element of *x*, compute the probability density function (PDF) at *x* of the binomial distribution with parameters *n* and *p*.

cauchy_cdf (*x*, *lambda*, *sigma*) [Function File]
For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Cauchy distribution with location parameter *lambda* and scale parameter *sigma*. Default values are *lambda* = 0, *sigma* = 1.

cauchy_inv (*x*, *lambda*, *sigma*) [Function File]
For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the Cauchy distribution with location parameter *lambda* and scale parameter *sigma*. Default values are *lambda* = 0, *sigma* = 1.

cauchy_pdf (*x*, *lambda*, *sigma*) [Function File]
For each element of *x*, compute the probability density function (PDF) at *x* of the Cauchy distribution with location parameter *lambda* and scale parameter *sigma* > 0. Default values are *lambda* = 0, *sigma* = 1.

chi2cdf (*x*, *n*) [Function File]
For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the chisquare distribution with *n* degrees of freedom.

chi2inv (*x*, *n*) [Function File]
For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the chisquare distribution with *n* degrees of freedom.

chisquare_pdf (*x*, *n*) [Function File]
For each element of *x*, compute the probability density function (PDF) at *x* of the chisquare distribution with *n* degrees of freedom.

discrete_cdf (*x*, *v*, *p*) [Function File]
For each element of *x*, compute the cumulative distribution function (CDF) at *x* of a univariate discrete distribution which assumes the values in *v* with probabilities *p*.

`discrete_inv (x, v, p)` [Function File]
For each component of x , compute the quantile (the inverse of the CDF) at x of the univariate distribution which assumes the values in v with probabilities p .

`discrete_pdf (x, v, p)` [Function File]
For each element of x , compute the probability density function (PDF) at x of a univariate discrete distribution which assumes the values in v with probabilities p .

`empirical_cdf (x, data)` [Function File]
For each element of x , compute the cumulative distribution function (CDF) at x of the empirical distribution obtained from the univariate sample $data$.

`empirical_inv (x, data)` [Function File]
For each element of x , compute the quantile (the inverse of the CDF) at x of the empirical distribution obtained from the univariate sample $data$.

`empirical_pdf (x, data)` [Function File]
For each element of x , compute the probability density function (PDF) at x of the empirical distribution obtained from the univariate sample $data$.

`expcdf (x, lambda)` [Function File]
For each element of x , compute the cumulative distribution function (CDF) at x of the exponential distribution with parameter $lambda$.
The arguments can be of common size or scalar.

`expinv (x, lambda)` [Function File]
For each element of x , compute the quantile (the inverse of the CDF) at x of the exponential distribution with parameter $lambda$.

`exppdf (x, lambda)` [Function File]
For each element of x , compute the probability density function (PDF) of the exponential distribution with parameter $lambda$.

`fcdf (x, m, n)` [Function File]
For each element of x , compute the CDF at x of the F distribution with m and n degrees of freedom, i.e., $\text{PROB}(F(m, n) \leq x)$.

`finv (x, m, n)` [Function File]
For each component of x , compute the quantile (the inverse of the CDF) at x of the F distribution with parameters m and n .

`fpdf (x, m, n)` [Function File]
For each element of x , compute the probability density function (PDF) at x of the F distribution with m and n degrees of freedom.

gamcdf (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Gamma distribution with parameters *a* and *b*.

See also: gamma, gammaln, gammainc, gampdf, gaminv, gamrnd.

gaminv (*x*, *a*, *b*) [Function File]

For each component of *x*, compute the quantile (the inverse of the CDF) at *x* of the Gamma distribution with parameters *a* and *b*.

See also: gamma, gammaln, gammainc, gampdf, gamcdf, gamrnd.

gampdf (*x*, *a*, *b*) [Function File]

For each element of *x*, return the probability density function (PDF) at *x* of the Gamma distribution with parameters *a* and *b*.

See also: gamma, gammaln, gammainc, gamcdf, gaminv, gamrnd.

geocdf (*x*, *p*) [Function File]

For each element of *x*, compute the CDF at *x* of the geometric distribution with parameter *p*.

geoinv (*x*, *p*) [Function File]

For each element of *x*, compute the quantile at *x* of the geometric distribution with parameter *p*.

geopdf (*x*, *p*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the geometric distribution with parameter *p*.

hygecdf (*x*, *m*, *t*, *n*) [Function File]

Compute the cumulative distribution function (CDF) at *x* of the hypergeometric distribution with parameters *m*, *t*, and *n*. This is the probability of obtaining not more than *x* marked items when randomly drawing a sample of size *n* without replacement from a population of total size *t* containing *m* marked items.

The parameters *m*, *t*, and *n* must positive integers with *m* and *n* not greater than *t*.

hygeinv (*x*, *m*, *t*, *n*) [Function File]

For each element of *x*, compute the quantile at *x* of the hypergeometric distribution with parameters *m*, *t*, and *n*.

The parameters *m*, *t*, and *n* must positive integers with *m* and *n* not greater than *t*.

hygepdf (*x*, *m*, *t*, *n*) [Function File]

Compute the probability density function (PDF) at *x* of the hypergeometric distribution with parameters *m*, *t*, and *n*. This is the probability of obtaining *x* marked items when randomly drawing a sample of size *n* without replacement from a population of total size *t* containing *m* marked items.

The arguments must be of common size or scalar.

kolmogorov_smirnov_cdf (*x*, *tol*) [Function File]
 Return the CDF at *x* of the Kolmogorov-Smirnov distribution,

$$Q(x) = \sum_{k=-\infty}^{\infty} (-1)^k \exp(-2k^2 x^2)$$

for $x > 0$.

The optional parameter *tol* specifies the precision up to which the series should be evaluated; the default is *tol* = **eps**.

laplace_cdf (*x*) [Function File]
 For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Laplace distribution.

laplace_inv (*x*) [Function File]
 For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the Laplace distribution.

laplace_pdf (*x*) [Function File]
 For each element of *x*, compute the probability density function (PDF) at *x* of the Laplace distribution.

logistic_cdf (*x*) [Function File]
 For each component of *x*, compute the CDF at *x* of the logistic distribution.

logistic_inv (*x*) [Function File]
 For each component of *x*, compute the quantile (the inverse of the CDF) at *x* of the logistic distribution.

logistic_pdf (*x*) [Function File]
 For each component of *x*, compute the PDF at *x* of the logistic distribution.

logncdf (*x*, *mu*, *sigma*) [Function File]
 For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the lognormal distribution with parameters *mu* and *sigma*. If a random variable follows this distribution, its logarithm is normally distributed with mean *mu* and standard deviation *sigma*.

Default values are *mu* = 1, *sigma* = 1.

logninv (*x*, *mu*, *sigma*) [Function File]
 For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the lognormal distribution with parameters *mu* and *sigma*. If a random variable follows this distribution, its logarithm is normally distributed with mean **log** (*mu*) and variance *sigma*.

Default values are *mu* = 1, *sigma* = 1.

lognpdf (*x*, *mu*, *sigma*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the lognormal distribution with parameters *mu* and *sigma*. If a random variable follows this distribution, its logarithm is normally distributed with mean *mu* and standard deviation *sigma*.

Default values are *mu* = 1, *sigma* = 1.

nbincdf (*x*, *n*, *p*) [Function File]

For each element of *x*, compute the CDF at *x* of the Pascal (negative binomial) distribution with parameters *n* and *p*.

The number of failures in a Bernoulli experiment with success probability *p* before the *n*-th success follows this distribution.

nbinin (*x*, *n*, *p*) [Function File]

For each element of *x*, compute the quantile at *x* of the Pascal (negative binomial) distribution with parameters *n* and *p*.

The number of failures in a Bernoulli experiment with success probability *p* before the *n*-th success follows this distribution.

nbinp (*x*, *n*, *p*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the Pascal (negative binomial) distribution with parameters *n* and *p*.

The number of failures in a Bernoulli experiment with success probability *p* before the *n*-th success follows this distribution.

normcdf (*x*, *m*, *s*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the normal distribution with mean *m* and standard deviation *s*.

Default values are *m* = 0, *s* = 1.

norminv (*x*, *m*, *s*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the normal distribution with mean *m* and standard deviation *s*.

Default values are *m* = 0, *s* = 1.

normpdf (*x*, *m*, *s*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the normal distribution with mean *m* and standard deviation *s*.

Default values are *m* = 0, *s* = 1.

poisscdf (*x*, *lambda*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Poisson distribution with parameter *lambda*.

poissinv (*x*, *lambda*) [Function File]
 For each component of *x*, compute the quantile (the inverse of the CDF) at *x* of the Poisson distribution with parameter *lambda*.

poisspdf (*x*, *lambda*) [Function File]
 For each element of *x*, compute the probability density function (PDF) at *x* of the poisson distribution with parameter *lambda*.

tcdf (*x*, *n*) [Function File]
 For each element of *x*, compute the CDF at *x* of the t (Student) distribution with *n* degrees of freedom, i.e., $\text{PROB}(t(n) \leq x)$.

tinvs (*x*, *n*) [Function File]
 For each component of *x*, compute the quantile (the inverse of the CDF) at *x* of the t (Student) distribution with parameter *n*.

tpdf (*x*, *n*) [Function File]
 For each element of *x*, compute the probability density function (PDF) at *x* of the *t* (Student) distribution with *n* degrees of freedom.

unifcdf (*x*, *a*, *b*) [Function File]
 Return the CDF at *x* of the uniform distribution on $[a, b]$, i.e., $\text{PROB}(\text{uniform}(a, b) \leq x)$.
 Default values are $a = 0$, $b = 1$.

unifinv (*x*, *a*, *b*) [Function File]
 For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the uniform distribution on $[a, b]$.
 Default values are $a = 0$, $b = 1$.

unifpdf (*x*, *a*, *b*) [Function File]
 For each element of *x*, compute the PDF at *x* of the uniform distribution on $[a, b]$.
 Default values are $a = 0$, $b = 1$.

wblcdf (*x*, *scale*, *shape*) [Function File]
 Compute the cumulative distribution function (CDF) at *x* of the Weibull distribution with shape parameter *scale* and scale parameter *shape*, which is

$$1 - \exp(-(x/\text{shape})^{\text{scale}})$$

for $x \geq 0$.

wblinv (*x*, *scale*, *shape*) [Function File]
 Compute the quantile (the inverse of the CDF) at *x* of the Weibull distribution with shape parameter *scale* and scale parameter *shape*.

wblpdf (*x*, *scale*, *shape*) [Function File]
 Compute the probability density function (PDF) at *x* of the Weibull distribution with shape parameter *scale* and scale parameter *shape* which is given by

$$scale \cdot shape^{-scale} x^{scale-1} \exp(-(x/shape)^{scale})$$

for $x > 0$.

25.7 Random Number Generation

Octave can generate random numbers from a large number of distributions. The random number generators are based on the random number generators described in [Section 17.4 \[Special Utility Matrices\]](#), page 204.

The following table summarizes the available random number generators (in alphabetical order).

Distribution	Function
Beta Distribution	betarnd
Binomial Distribution	binornd
Cauchy Distribution	cauchy_rnd
Chi-Square Distribution	chi2rnd
Univariate Discrete Distribution	discrete_rnd
Empirical Distribution	empirical_rnd
Exponential Distribution	exprnd
F Distribution	frnd
Gamma Distribution	gamrnd
Geometric Distribution	geornd
Hypergeometric Distribution	hygernd
Laplace Distribution	laplace_rnd
Logistic Distribution	logistic_rnd
Log-Normal Distribution	lognrnd
Pascal Distribution	nbinrnd
Univariate Normal Distribution	normrnd
Poisson Distribution	poissrnd
t (Student) Distribution	trnd
Univariate Discrete Distribution	unidrnd
Uniform Distribution	unifrnd
Weibull Distribution	wblrnd
Wiener Process	wienrnd

betarnd (*a*, *b*, *r*, *c*) [Function File]

betarnd (*a*, *b*, *sz*) [Function File]

Return an *r* by *c* or size (*sz*) matrix of random samples from the Beta distribution with parameters *a* and *b*. Both *a* and *b* must be scalar or of size *r* by *c*.

If *r* and *c* are omitted, the size of the result matrix is the common size of *a* and *b*.

binornd (*n*, *p*, *r*, *c*) [Function File]

`binornd` (*n*, *p*, *sz*) [Function File]

Return an *r* by *c* or a **size** (*sz*) matrix of random samples from the binomial distribution with parameters *n* and *p*. Both *n* and *p* must be scalar or of size *r* by *c*.

If *r* and *c* are omitted, the size of the result matrix is the common size of *n* and *p*.

`cauchy_rnd` (*lambda*, *sigma*, *r*, *c*) [Function File]

`cauchy_rnd` (*lambda*, *sigma*, *sz*) [Function File]

Return an *r* by *c* or a **size** (*sz*) matrix of random samples from the Cauchy distribution with parameters *lambda* and *sigma* which must both be scalar or of size *r* by *c*.

If *r* and *c* are omitted, the size of the result matrix is the common size of *lambda* and *sigma*.

`chi2rnd` (*n*, *r*, *c*) [Function File]

`chi2rnd` (*n*, *sz*) [Function File]

Return an *r* by *c* or a **size** (*sz*) matrix of random samples from the chisquare distribution with *n* degrees of freedom. *n* must be a scalar or of size *r* by *c*.

If *r* and *c* are omitted, the size of the result matrix is the size of *n*.

`discrete_rnd` (*n*, *v*, *p*) [Function File]

`discrete_rnd` (*v*, *p*, *r*, *c*) [Function File]

`discrete_rnd` (*v*, *p*, *sz*) [Function File]

Generate a row vector containing a random sample of size *n* from the univariate distribution which assumes the values in *v* with probabilities *p*. *n* must be a scalar.

If *r* and *c* are given create a matrix with *r* rows and *c* columns. Or if *sz* is a vector, create a matrix of size *sz*.

`empirical_rnd` (*n*, *data*) [Function File]

`empirical_rnd` (*data*, *r*, *c*) [Function File]

`empirical_rnd` (*data*, *sz*) [Function File]

Generate a bootstrap sample of size *n* from the empirical distribution obtained from the univariate sample *data*.

If *r* and *c* are given create a matrix with *r* rows and *c* columns. Or if *sz* is a vector, create a matrix of size *sz*.

`exprnd` (*lambda*, *r*, *c*) [Function File]

`exprnd` (*lambda*, *sz*) [Function File]

Return an *r* by *c* matrix of random samples from the exponential distribution with parameter *lambda*, which must be a scalar or of size *r* by *c*. Or if *sz* is a vector, create a matrix of size *sz*.

If *r* and *c* are omitted, the size of the result matrix is the size of *lambda*.

`frnd (m, n, r, c)` [Function File]

`frnd (m, n, sz)` [Function File]

Return an r by c matrix of random samples from the F distribution with m and n degrees of freedom. Both m and n must be scalar or of size r by c . If sz is a vector the random samples are in a matrix of size sz .

If r and c are omitted, the size of the result matrix is the common size of m and n .

`gamrnd (a, b, r, c)` [Function File]

`gamrnd (a, b, sz)` [Function File]

Return an r by c or a **size** (sz) matrix of random samples from the Gamma distribution with parameters a and b . Both a and b must be scalar or of size r by c .

If r and c are omitted, the size of the result matrix is the common size of a and b .

See also: `gamma`, `gammaln`, `gammainc`, `gampdf`, `gamcdf`, `gaminv`.

`geornd (p, r, c)` [Function File]

`geornd (p, sz)` [Function File]

Return an r by c matrix of random samples from the geometric distribution with parameter p , which must be a scalar or of size r by c .

If r and c are given create a matrix with r rows and c columns. Or if sz is a vector, create a matrix of size sz .

`hygernd (m, t, n, r, c)` [Function File]

`hygernd (m, t, n, sz)` [Function File]

Return an r by c matrix of random samples from the hypergeometric distribution with parameters m , t , and n .

The parameters m , t , and n must positive integers with m and n not greater than t .

`laplace_rnd (r, c)` [Function File]

`laplace_rnd (sz);` [Function File]

Return an r by c matrix of random numbers from the Laplace distribution. Or if sz is a vector, create a matrix of sz .

`logistic_rnd (r, c)` [Function File]

`logistic_rnd (sz)` [Function File]

Return an r by c matrix of random numbers from the logistic distribution. Or if sz is a vector, create a matrix of sz .

`lognrnd (mu, sigma, r, c)` [Function File]

`lognrnd (mu, sigma, sz)` [Function File]

Return an r by c matrix of random samples from the lognormal distribution with parameters mu and $sigma$. Both mu and $sigma$ must be scalar or of size r by c . Or if sz is a vector, create a matrix of size sz .

If r and c are omitted, the size of the result matrix is the common size of mu and $sigma$.

`nbinrnd (n, p, r, c)` [Function File]

`nbinrnd (n, p, sz)` [Function File]

Return an r by c matrix of random samples from the Pascal (negative binomial) distribution with parameters n and p . Both n and p must be scalar or of size r by c .

If r and c are omitted, the size of the result matrix is the common size of n and p .

Or if sz is a vector, create a matrix of size sz .

`normrnd (m, s, r, c)` [Function File]

`normrnd (m, s, sz)` [Function File]

Return an r by c or **size** (sz) matrix of random samples from the normal distribution with parameters mean m and standard deviation s . Both m and s must be scalar or of size r by c .

If r and c are omitted, the size of the result matrix is the common size of m and s .

`poissrnd (lambda, r, c)` [Function File]

Return an r by c matrix of random samples from the Poisson distribution with parameter $lambda$, which must be a scalar or of size r by c .

If r and c are omitted, the size of the result matrix is the size of $lambda$.

`trnd (n, r, c)` [Function File]

`trnd (n, sz)` [Function File]

Return an r by c matrix of random samples from the t (Student) distribution with n degrees of freedom. n must be a scalar or of size r by c . Or if sz is a vector create a matrix of size sz .

If r and c are omitted, the size of the result matrix is the size of n .

`unidrnd (mx);` [Function File]

`unidrnd (mx, v);` [Function File]

`unidrnd (mx, m, n, ...);` [Function File]

Return random values from discrete uniform distribution, with maximum value(s) given by the integer mx , which may be a scalar or multidimensional array.

If mx is a scalar, the size of the result is specified by the vector v , or by the optional arguments m, n, \dots . Otherwise, the size of the result is the same as the size of mx .

`unifrnd (a, b, r, c)` [Function File]

`unifrnd (a, b, sz)` [Function File]

Return an r by c or a **size** (sz) matrix of random samples from the uniform distribution on $[a, b]$. Both a and b must be scalar or of size r by c .

If r and c are omitted, the size of the result matrix is the common size of a and b .

`wblrnd (scale, shape, r, c)` [Function File]

`wblrnd (scale, shape, sz)` [Function File]

Return an r by c matrix of random samples from the Weibull distribution with parameters $scale$ and $shape$ which must be scalar or of size r by c . Or if sz is a vector return a matrix of size sz .

If r and c are omitted, the size of the result matrix is the common size of $alpha$ and $sigma$.

wienrnd (*t*, *d*, *n*) [Function File]

Return a simulated realization of the d -dimensional Wiener Process on the interval $[0, t]$. If d is omitted, $d = 1$ is used. The first column of the return matrix contains time, the remaining columns contain the Wiener process.

The optional parameter n gives the number of summands used for simulating the process over an interval of length 1. If n is omitted, $n = 1000$ is used.

26 Financial Functions

f_v (*r*, *n*, *p*, *l*, *method*) [Function File]

Return the future value at the end of period *n* of an investment which consists of *n* payments of *p* in each period, assuming an interest rate *r*.

The optional argument *l* may be used to specify an additional lump-sum payment.

The optional argument *method* may be used to specify whether the payments are made at the end ("e", default) or at the beginning ("b") of each period.

Note that the rate *r* is specified as a fraction (i.e., 0.05, not 5 percent).

f_{vl} (*r*, *n*, *l*) [Function File]

Return the future value at the end of *n* periods of an initial lump sum investment *l*, given a per-period interest rate *r*.

Note that the rate *r* is specified as a fraction (i.e., 0.05, not 5 percent).

irr (*p*, *i*) [Function File]

Return the internal rate of return of a series of payments *p* from an initial investment *i* (i.e., the solution of **npv** (*r*, *p*) = *i*). If the second argument is omitted, a value of 0 is used.

See also: npv, pv, rate.

nper (*r*, *p*, *a*, *l*, *method*) [Function File]

Return the number of regular payments of *p* necessary to amortize a loan of amount *a* and interest *r*.

The optional argument *l* may be used to specify an additional lump-sum payment of *l* made at the end of the amortization time.

The optional argument *method* may be used to specify whether payments are made at the end ("e", default) or at the beginning ("b") of each period.

Note that the rate *r* is specified as a fraction (i.e., 0.05, not 5 percent).

See also: pv, pmt, rate, npv.

npv (*r*, *p*, *i*) [Function File]

Returns the net present value of a series of irregular (i.e., not necessarily identical) payments *p* which occur at the ends of *n* consecutive periods. *r* specifies the one-period interest rates and can either be a scalar (constant rates) or a vector of the same length as *p*.

The optional argument *i* may be used to specify an initial investment.

Note that the rate *r* is specified as a fraction (i.e., 0.05, not 5 percent).

See also: irr, pv.

pmt (*r*, *n*, *a*, *l*, *method*) [Function File]

Return the amount of periodic payment necessary to amortize a loan of amount *a* with interest rate *r* in *n* periods.

The optional argument *l* may be used to specify a terminal lump-sum payment.

The optional argument *method* may be used to specify whether payments are made at the end ("e", default) or at the beginning ("b") of each period.

See also: pv, nper, rate.

pv (*r*, *n*, *p*, *l*, *method*) [Function File]

Returns the present value of an investment that will pay off *p* for *n* consecutive periods, assuming an interest *r*.

The optional argument *l* may be used to specify an additional lump-sum payment made at the end of *n* periods.

The optional argument *method* may be used to specify whether payments are made at the end ("e", default) or at the beginning ("b") of each period.

Note that the rate *r* is specified as a fraction (i.e., 0.05, not 5 percent).

See also: pmt, nper, rate, npv.

pvl (*r*, *n*, *p*) [Function File]

Return the present value of an investment that will pay off *p* in one lump sum at the end of *n* periods, given the interest rate *r*.

Note that the rate *r* is specified as a fraction (i.e., 0.05, not 5 percent).

rate (*n*, *p*, *v*, *l*, *method*) [Function File]

Return the rate of return on an investment of present value *v* which pays *p* in *n* consecutive periods.

The optional argument *l* may be used to specify an additional lump-sum payment made at the end of *n* periods.

The optional string argument *method* may be used to specify whether payments are made at the end ("e", default) or at the beginning ("b") of each period.

See also: pv, pmt, nper, npv.

vol (*x*, *m*, *n*) [Function File]

Return the volatility of each column of the input matrix *x*. The number of data sets per period is given by *m* (e.g. the number of data per year if you want to compute the volatility per year). The optional parameter *n* gives the number of past periods used for computation, if it is omitted, a value of 1 is used. If *t* is the number of rows of *x*, **vol** returns the volatility from *n***m* to *t*.

27 Sets

Octave has a limited set of functions for managing sets of data, where a set is defined as a collection unique elements. In Octave a set is represented as a vector of numbers.

create_set (x) [Function File]

Return a row vector containing the unique values in *x*, sorted in ascending order. For example,

```
create_set ([ 1, 2; 3, 4; 4, 2 ])
⇒ [ 1, 2, 3, 4 ]
```

See also: union, intersection, complement.

unique (x) [Function File]

Return the unique elements of *x*, sorted in ascending order. If *x* is a row vector, return a row vector, but if *x* is a column vector or a matrix return a column vector.

unique (A, 'rows') [Function File]

Return the unique rows of *A*, sorted in ascending order.

[y, i, j] = unique (x) [Function File]

Return index vectors *i* and *j* such that *x*(*i*)==*y* and *y*(*j*)==*x*.

See also: union, intersect, setdiff, setxor, ismember.

27.1 Set Operations

Octave supports the basic set operations. That is, Octave can compute the union, intersection, complement, and difference of two sets. Octave can also supports the *Exclusive Or* set operation, and membership determination. The functions for set operations all work in pretty much the same way. As an example, assume that *x* and *y* contains two sets, then

```
union(x, y)
```

computes the union of the two sets.

ismember (A, S) [Function File]

Return a matrix the same shape as *A* which has 1 if *A*(*i*,*j*) is in *S* or 0 if it isn't.

See also: unique, union, intersection, setxor, setdiff.

union (x, y) [Function File]

Return the set of elements that are in either of the sets *x* and *y*. For example,

```
union ([ 1, 2, 4 ], [ 2, 3, 5 ])
⇒ [ 1, 2, 3, 4, 5 ]
```

See also: create_set, intersection, complement.

intersect (a, b) [Function File]

[c, ia, ib] = intersect (a, b) [Function File]

Return the elements in both *a* and *b*, sorted in ascending order. If *a* and *b* are both column vectors return a column vector, otherwise return a row vector.

Return index vectors *ia* and *ib* such that *a*(*ia*)==*c* and *b*(*ib*)==*c*.

See also: `unique`, `union`, `setxor`, `setdiff`, `ismember`.

`complement (x, y)` [Function File]

Return the elements of set *y* that are not in set *x*. For example,

```
complement ([ 1, 2, 3 ], [ 2, 3, 5 ])
⇒ 5
```

See also: `create_set`, `union`, `intersection`.

`setdiff (a, b)` [Function File]

`setdiff (a, b, "rows")` [Function File]

Return the elements in *a* that are not in *b*, sorted in ascending order. If *a* and *b* are both column vectors return a column vector, otherwise return a row vector.

Given the optional third argument ‘**rows**’, return the rows in *a* that are not in *b*, sorted in ascending order by rows.

See also: `unique`, `union`, `intersect`, `setxor`, `ismember`.

`setxor (a, b)` [Function File]

Return the elements exclusive to *a* or *b*, sorted in ascending order. If *a* and *b* are both column vectors return a column vector, otherwise return a row vector.

See also: `unique`, `union`, `intersect`, `setdiff`, `ismember`.

28 Polynomial Manipulations

In Octave, a polynomial is represented by its coefficients (arranged in descending order). For example, a vector of length $N + 1$ corresponds to the following polynomial of order N

$$p(x) = c_1x^N + \dots + c_Nx + c_{N+1}.$$

companion (c) [Function File]

Compute the companion matrix corresponding to polynomial coefficient vector c .

The companion matrix is

$$A = \begin{bmatrix} -c_2/c_1 & -c_3/c_1 & \cdots & -c_N/c_1 & -c_{N+1}/c_1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

The eigenvalues of the companion matrix are equal to the roots of the polynomial.

See also: poly, roots, residue, conv, deconv, polyval, polyderiv, polyinteg.

conv (a, b) [Function File]

Convolve two vectors.

$y = \text{conv} (a, b)$ returns a vector of length equal to $\text{length} (a) + \text{length} (b) - 1$. If a and b are polynomial coefficient vectors, **conv** returns the coefficients of the product polynomial.

See also: deconv, poly, roots, residue, polyval, polyderiv, polyinteg.

deconv (y, a) [Function File]

Deconvolve two vectors.

$[b, r] = \text{deconv} (y, a)$ solves for b and r such that $y = \text{conv} (a, b) + r$.

If y and a are polynomial coefficient vectors, b will contain the coefficients of the polynomial quotient and r will be a remainder polynomial of lowest order.

See also: conv, poly, roots, residue, polyval, polyderiv, polyinteg.

$y = \text{conv2} (a, b, \text{shape})$ [Loadable Function]

$y = \text{conv2} (v1, v2, M, \text{shape})$ [Loadable Function]

Returns 2D convolution of a and b where the size of c is given by

$\text{shape} = \text{'full'}$

returns full 2-D convolution

$\text{shape} = \text{'same'}$

same size as a . 'central' part of convolution

$\text{shape} = \text{'valid'}$

only parts which do not include zero-padded edges

By default shape is 'full'. When the third argument is a matrix returns the convolution of the matrix M by the vector $v1$ in the column direction and by vector $v2$ in the row direction

`poly (a)` [Function File]

If a is a square N -by- N matrix, `poly (a)` is the row vector of the coefficients of `det (z * eye (N) - a)`, the characteristic polynomial of a . If x is a vector, `poly (x)` is a vector of coefficients of the polynomial whose roots are the elements of x .

`polyderiv (c)` [Function File]

`[q] = polyder (b, a)` [Function File]

`[q, r] = polyder (b, a)` [Function File]

Return the coefficients of the derivative of the polynomial whose coefficients are given by vector c . If a pair of polynomials is given b and a , the derivative of the product is returned in q , or the quotient numerator in q and the quotient denominator in r .

See also: `poly`, `polyinteg`, `polyreduce`, `roots`, `conv`, `deconv`, `residue`, `filter`, `polygcd`, `polyval`, `polyvalm`.

`polyder (c)` [Function File]

`[q] = polyder (b, a)` [Function File]

`[q, r] = polyder (b, a)` [Function File]

See `polyderiv`.

`[p, s] = polyfit (x, y, n)` [Function File]

Return the coefficients of a polynomial $p(x)$ of degree n that minimizes

$$\sum_{i=1}^N (p(x_i) - y_i)^2$$

to best fit the data in the least squares sense.

The polynomial coefficients are returned in a row vector.

If two output arguments are requested, the second is a structure containing the following fields:

R The Cholesky factor of the Vandermonde matrix used to compute the polynomial coefficients.

x The Vandermonde matrix used to compute the polynomial coefficients.

df The degrees of freedom.

normr The norm of the residuals.

yf The values of the polynomial for each value of x .

`q = polygcd (b, a, tol)` [Function File]

Find greatest common divisor of two polynomials. This is equivalent to the polynomial found by multiplying together all the common roots. Together with `deconv`, you can reduce a ratio of two polynomials. Tolerance defaults to

`sqrt(eps)`.

Note that this is an unstable algorithm, so don't try it on large polynomials.

Example

```
polygcd (poly(1:8), poly(3:12)) - poly(3:8)
    deconv (poly(1:8), polygcd (poly(1:8), poly(3:12))) - poly(1:2)■
```

See also: poly, polyinteg, polyderiv, polyreduce, roots, conv, deconv, residue, filter, polyval, and polyvalm.

polyinteg (*c*) [Function File]

Return the coefficients of the integral of the polynomial whose coefficients are represented by the vector *c*.

The constant of integration is set to zero.

See also: poly, polyderiv, polyreduce, roots, conv, deconv, residue, filter, polyval, and polyvalm.

polyreduce (*c*) [Function File]

Reduces a polynomial coefficient vector to a minimum number of terms by stripping off any leading zeros.

See also: poly, roots, conv, deconv, residue, filter, polyval, polyvalm, polyderiv, polyinteg.

polyval (*c*, *x*) [Function File]

Evaluate a polynomial.

polyval (*c*, *x*) will evaluate the polynomial at the specified value of *x*.

If *x* is a vector or matrix, the polynomial is evaluated at each of the elements of *x*.

See also: polyvalm, poly, roots, conv, deconv, residue, filter, polyderiv, polyinteg.

polyvalm (*c*, *x*) [Function File]

Evaluate a polynomial in the matrix sense.

polyvalm (*c*, *x*) will evaluate the polynomial in the matrix sense, i.e. matrix multiplication is used instead of element by element multiplication as is used in polyval.

The argument *x* must be a square matrix.

See also: polyval, poly, roots, conv, deconv, residue, filter, polyderiv, and polyinteg.

residue (*b*, *a*, *tol*) [Function File]

If *b* and *a* are vectors of polynomial coefficients, then residue calculates the partial fraction expansion corresponding to the ratio of the two polynomials.

The function **residue** returns *r*, *p*, *k*, and *e*, where the vector *r* contains the residue terms, *p* contains the pole values, *k* contains the coefficients of a direct polynomial term (if it exists) and *e* is a vector containing the powers of the denominators in the partial fraction terms.

Assuming *b* and *a* represent polynomials $P(s)$ and $Q(s)$ we have:

$$\frac{P(s)}{Q(s)} = \sum_{m=1}^M \frac{r_m}{(s - p_m)_m^e} + \sum_{i=1}^N k_i s^{N-i}.$$

where M is the number of poles (the length of the r , p , and e vectors) and N is the length of the k vector.

The argument *tol* is optional, and if not specified, a default value of 0.001 is assumed. The tolerance value is used to determine whether poles with small imaginary components are declared real. It is also used to determine if two poles are distinct. If the ratio of the imaginary part of a pole to the real part is less than *tol*, the imaginary part is discarded. If two poles are farther apart than *tol* they are distinct. For example,

```
b = [1, 1, 1];
a = [1, -5, 8, -4];
[r, p, k, e] = residue (b, a);
⇒ r = [-2, 7, 3]
⇒ p = [2, 2, 1]
⇒ k = [] (0x0)
⇒ e = [1, 2, 1]
```

which implies the following partial fraction expansion

$$\frac{s^2 + s + 1}{s^3 - 5s^2 + 8s - 4} = \frac{-2}{s - 2} + \frac{7}{(s - 2)^2} + \frac{3}{s - 1}$$

See also: poly, roots, conv, deconv, polyval, polyderiv, and polyinteg.

roots (*v*) [Function File]

For a vector v with N components, return the roots of the polynomial

$$v_1 z^{N-1} + \dots + v_{N-1} z + v_N.$$

polyout (*c*, *x*) [Function File]

Write formatted polynomial

$$c(x) = c_1 x^n + \dots + c_n x + c_{n+1}$$

and return it as a string or write it to the screen (if *nargout* is zero). x defaults to the string "s".

See also: polyval, polyvalm, poly, roots, conv, deconv, residue, filter, polyderiv, and polyinteg.

yi = **ppval** (*pp*, *xi*) [Function File]

Evaluate piece-wise polynomial *pp* at the points *xi*. If *pp.d* is a scalar greater than 1, or an array, then the returned value *yi* will be an array that is **d1**, **d1**, ..., **dk**, **length (xi)**].

See also: mkpp, unmkpp, spline.

`pp = mkpp (x, p)` [Function File]

`pp = mkpp (x, p, d)` [Function File]

Construct a piece-wise polynomial structure from sample points `x` and coefficients `p`. The i th row of `p`, `p (i, :)`, contains the coefficients for the polynomial over the i -th interval, ordered from highest to lowest. There must be one row for each interval in `x`, so `rows (p) == length (x) - 1`.

You can concatenate multiple polynomials of the same order over the same set of intervals using `p = [p1; p2; ...; pd]`. In this case, `rows (p) == d * (length (x) - 1)`.

`d` specifies the shape of the matrix `p` for all except the last dimension. If `d` is not specified it will be computed as `round (rows (p) / (length (x) - 1))` instead.

See also: `unmkpp`, `ppval`, `spline`.

`[x, p, n, k, d] = unmkpp (pp)` [Function File]

Extract the components of a piece-wise polynomial structure `pp`. These are as follows:

`x` Samples points.

`p` Polynomial coefficients for points in sample interval. `p (i, :)` contains the coefficients for the polynomial over interval i ordered from highest to lowest. If $d > 1$, `p (r, i, :)` contains the coefficients for the r -th polynomial defined on interval i . However, this is stored as a 2-D array such that `c = reshape (p (:, j), n, d)` gives `c (i, r)` is the j -th coefficient of the r -th polynomial over the i -th interval.

`n` Number of polynomial pieces.

`k` Order of the polynomial plus 1.

`d` Number of polynomials defined for each interval.

See also: `mkpp`, `ppval`, `spline`.

29 Interpolation

29.1 One-dimensional Interpolation

```

yi = interp1 (x, y, xi)           [Function File]
yi = interp1 (... , method)      [Function File]
yi = interp1 (... , extrapol)    [Function File]
pp = interp1 (... , 'pp')        [Function File]

```

One-dimensional interpolation. Interpolate y , defined at the points x , at the points x_i . The sample points x must be strictly monotonic. If y is an array, treat the columns of y separately.

Method is one of:

'nearest'	Return the nearest neighbour.
'linear'	Linear interpolation from nearest neighbours
'pchip'	Piece-wise cubic hermite interpolating polynomial
'cubic'	Cubic interpolation from four nearest neighbours
'spline'	Cubic spline interpolation—smooth first and second derivatives throughout the curve

Appending `''` to the start of the above method forces `interp1` to assume that x is uniformly spaced, and only $x(1)$ and $x(2)$ are referenced. This is usually faster, and is never slower. The default method is `'linear'`.

If `extrapol` is the string `'extrapol'`, then extrapolate values beyond the endpoints. If `extrapol` is a number, replace values beyond the endpoints with that number. If `extrapol` is missing, assume NA.

If the string argument `'pp'` is specified, then x_i should not be supplied and `interp1` returns the piece-wise polynomial that can later be used with `ppval` to evaluate the interpolation. There is an equivalence, such that `ppval(interp1(x, y, method, 'pp'), xi) == interp1(x, y, xi, method, 'extrapol')`.

An example of the use of `interp1` is

```

xf=[0:0.05:10]; yf = sin(2*pi*xf/5);
xp=[0:10];      yp = sin(2*pi*xp/5);
lin=interp1(xp,yp,xf);
spl=interp1(xp,yp,xf,'spline');
cub=interp1(xp,yp,xf,'cubic');
near=interp1(xp,yp,xf,'nearest');
plot(xf,yf,"r",xf,lin,"g",xf,spl,"b", ...
      xf,cub,"c",xf,near,"m",xp,yp,"r*");
legend ("original","linear","spline","cubic","nearest")

```

See also: `interpft`.

There are some important differences between the various interpolation methods. The `'spline'` method enforces that both the first and second derivatives of the interpolated values

have a continuous derivative, whereas the other methods do not. This means that the results of the 'spline' method are generally smoother. If the function to be interpolated is in fact smooth, then 'spline' will give excellent results. However, if the function to be evaluated is in some manner discontinuous, then 'pchip' interpolation might give better results.

This can be demonstrated by the code

```
t = -2:2;
dt = 1;
ti = -2:0.025:2;
dti = 0.025;
y = sign(t);
ys = interp1(t,y,ti,'spline');
yp = interp1(t,y,ti,'pchip');
ddys = diff(diff(ys)./dti)./dti;
ddyp = diff(diff(yp)./dti)./dti;
figure(1);
plot (ti, ys,'r-', ti, yp,'g-');
legend('spline','pchip',4);
figure(2);
plot (ti, ddys,'r+', ti, ddyp,'g*');
legend('spline','pchip');
```

The result of which can be seen in [Figure 29.1](#) and [Figure 29.2](#).

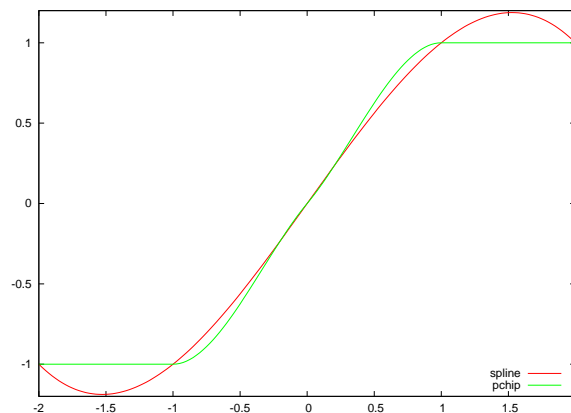


Figure 29.1: Comparison of 'pchip' and 'spline' interpolation methods for a step function

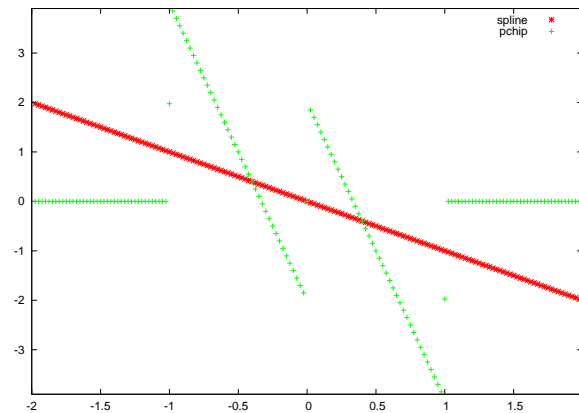


Figure 29.2: Comparison of the second derivate of the 'phcip' and 'spline' interpolation methods for a step function

Fourier interpolation, is a resampling technique where a signal is converted to the frequency domain, padded with zeros and then reconverted to the time domain.

`interpft (x, n)` [Function File]
`interpft (x, n, dim)` [Function File]

Fourier interpolation. If x is a vector, then x is resampled with n points. The data in x is assumed to be equispaced. If x is an array, then operate along each column of the array separately. If dim is specified, then interpolate along the dimension dim .

`interpft` assumes that the interpolated function is periodic, and so assumption are made about the end points of the inetpolation.

See also: `interp1`.

There are two significant limitations on Fourier interpolation. Firstly, the function signal is assumed to be periodic, and so no periodic signals will be poorly represented at the edges. Secondly, both the signal and its interpolation are required to be sampled at equispaced points. An example of the use of `interpft` is

```
t = 0 : 0.3 : pi; dt = t(2)-t(1);
n = length (t); k = 100;
ti = t(1) + [0 : k-1]*dt*n/k;
y = sin (4*t + 0.3) .* cos (3*t - 0.1);
yp = sin (4*ti + 0.3) .* cos (3*ti - 0.1);
plot (ti, yp, 'g', ti, interp1(t, y, ti, 'spline'), 'b', ...
      ti, interpft (y, k), 'c', t, y, 'r+');
legend ('sin(4t+0.3)cos(3t-0.1)', 'spline', 'interpft', 'data');
```

which demonstrates the poor behavior of Fourier interpolation for non periodic functions, as can be seen in [Figure 29.3](#).

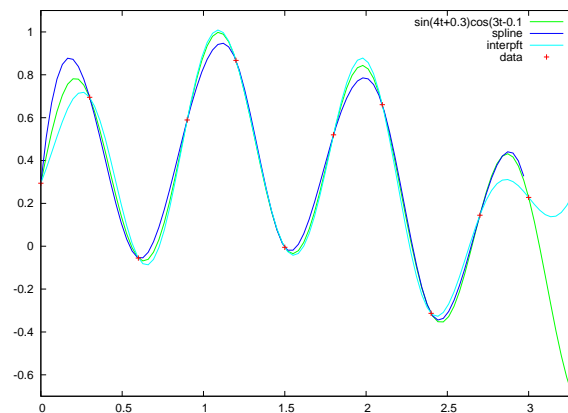


Figure 29.3: Comparison of `interp1` and `interpft` for non periodic data

In addition the support function `spline` and `lookup` that underlie the `interp1` function can be called directly.

```
pp = spline (x, y) [Function File]
yi = spline (x, y, xi) [Function File]
```

Returns the cubic spline interpolation of y at the point x . Called with two arguments the piece-wise polynomial pp that may later be used with `ppval` to evaluate the polynomial at specific points.

The variable x must be a vector of length n , and y can be either a vector or array. In the case where y is a vector, it can have a length of either n or $n + 2$. If the length of y is n , then the 'not-a-knot' end condition is used. If the length of y is $n + 2$, then the first and last values of the vector y are the first derivative of the cubic spline at the end-points.

If y is an array, then the size of y must have the form

$$[s_1, s_2, \dots, s_k, n]$$

or

$$[s_1, s_2, \dots, s_k, n + 2]$$

. The array is then reshaped internally to a matrix where the leading dimension is given by

$$s_1 s_2 \dots s_k$$

and each row of this matrix is then treated separately. Note that this is exactly the opposite treatment than `interp1` and is done for compatibility.

Called with a third input argument, `spline` evaluates the piece-wise spline at the points xi . There is an equivalence between `ppval (spline (x, y), xi)` and `spline (x, y, xi)`.

See also: `ppval`, `mkpp`, `unmkpp`.

The `lookup` is used by other interpolation function to identify the points of the original data that are closest to the current point of interest.

`idx = lookup (table, y)` [Function File]

Lookup values in a sorted table. Usually used as a prelude to interpolation.

If table is strictly increasing and `idx = lookup (table, y)`, then `table(idx(i)) <= y(i) < table(idx(i+1))` for all `y(i)` within the table. If `y(i)` is before the table, then `idx(i)` is 0. If `y(i)` is after the table then `idx(i)` is `table(n)`.

If the table is strictly decreasing, then the tests are reversed. There are no guarantees for tables which are non-monotonic or are not strictly monotonic.

To get an index value which lies within an interval of the table, use:

```
idx = lookup (table(2:length(table)-1), y) + 1
```

This expression puts values before the table into the first interval, and values after the table into the last interval.

29.2 Multi-dimensional Interpolation

There are three multi-dimensional interpolation function in Octave, with similar capabilities.

`zi = interp2 (x, y, z, xi, yi)` [Function File]

`zi = interp2 (Z, xi, yi)` [Function File]

`zi = interp2 (Z, n)` [Function File]

`zi = interp2 (... , method)` [Function File]

`zi = interp2 (... , method, extrapval)` [Function File]

Two-dimensional interpolation. `x`, `y` and `z` describe a surface function. If `x` and `y` are vectors their length must correspondent to the size of `z`. `x` and `Yy` must be monotonic. If they are matrices they must have the `meshgrid` format.

`interp2 (x, y, Z, xi, yi, ...)`

Returns a matrix corresponding to the points described by the matrices `XI`, `YI`.

If the last argument is a string, the interpolation method can be specified. The method can be 'linear', 'nearest' or 'cubic'. If it is omitted 'linear' interpolation is assumed.

`interp2 (z, xi, yi)`

Assumes `x = 1:rows (z)` and `y = 1:columns (z)`

`interp2 (z, n)`

Interleaves the Matrix `z` `n`-times. If `n` is ommited a value of `n = 1` is assumed.

The variable `method` defines the method to use for the interpolation. It can take one of the values

'nearest' Return the nearest neighbor.

'linear' Linear interpolation from nearest neighbors.

'pchip' Piece-wise cubic hermite interpolating polynomial (not implemented yet).

'cubic' Cubic interpolation from four nearest neighbors.

'spline' Cubic spline interpolation—smooth first and second derivatives throughout the curve.

If a scalar value *extrapval* is defined as the final value, then values outside the mesh are set to this value. Note that in this case *method* must be defined as well. If *extrapval* is not defined then NA is assumed.

See also: `interp1`.

```

vi = interp3 (x, y, z, v, xi, yi, zi)           [Function File]
vi = interp3 (v, xi, yi, zi)                   [Function File]
vi = interp3 (v, m)                           [Function File]
vi = interp3 (v)                               [Function File]
vi = interp3 (... , method)                   [Function File]
vi = interp3 (... , method, extrapval)         [Function File]

```

Perform 3-dimensional interpolation. Each element of then 3-dimensional array *v* represents a value at a location given by the parameters *x*, *y*, and *z*. The parameters *x*, *y*, and *z* are either 3-dimensional arrays of the same size as the array *v* in the 'meshgrid' format or vectors. The parameters *xi*, etc respect a similar format to *x*, etc, and they represent the points at which the array *vi* is interpolated.

If *x*, *y*, *z* are omitted, they are assumed to be `x = 1 : size (v, 2)`, `y = 1 : size (v, 1)` and `z = 1 : size (v, 3)`. If *m* is specified, then the interpolation adds a point half way between each of the interpolation points. This process is performed *m* times. If only *v* is specified, then *m* is assumed to be 1.

Method is one of:

'nearest' Return the nearest neighbour.

'linear' Linear interpolation from nearest neighbours.

'cubic' Cubic interpolation from four nearest neighbours (not implemented yet).

'spline' Cubic spline interpolation—smooth first and second derivatives throughout the curve.

The default method is 'linear'.

If *extrap* is the string 'extrap', then extrapolate values beyond the endpoints. If *extrap* is a number, replace values beyond the endpoints with that number. If *extrap* is missing, assume NA.

See also: `interp1`, `interp2`, `spline`, `meshgrid`.

```

vi = interpn (x1, x2, ..., v, y1, y2, ...)     [Function File]
vi = interpn (v, y1, y2, ...)                 [Function File]
vi = interpn (v, m)                           [Function File]
vi = interpn (v)                               [Function File]
vi = interpn (... , method)                   [Function File]
vi = interpn (... , method, extrapval)         [Function File]

```

Perform *n*-dimensional interpolation, where *n* is at least two. Each element of then *n*-dimensional array *v* represents a value at a location given by the parameters *x1*,

x_2, \dots, x_n . The parameters x_1, x_2, \dots, x_n are either n -dimensional arrays of the same size as the array v in the 'ndgrid' format or vectors. The parameters y_1 , etc respect a similar format to x_1 , etc, and they represent the points at which the array v_i is interpolated.

If x_1, \dots, x_n are omitted, they are assumed to be $x_1 = 1 : \text{size}(v, 1)$, etc. If m is specified, then the interpolation adds a point half way between each of the interpolation points. This process is performed m times. If only v is specified, then m is assumed to be 1.

Method is one of:

- 'nearest' Return the nearest neighbour.
- 'linear' Linear interpolation from nearest neighbours.
- 'cubic' Cubic interpolation from four nearest neighbours (not implemented yet).
- 'spline' Cubic spline interpolation—smooth first and second derivatives throughout the curve.

The default method is 'linear'.

If *extrap* is the string 'extrap', then extrapolate values beyond the endpoints. If *extrap* is a number, replace values beyond the endpoints with that number. If *extrap* is missing, assume NA.

See also: interp1, interp2, spline, ndgrid.

A significant difference between `interpn` and the other two multidimensional interpolation function is the fashion in which the dimensions are treated. For `interp2` and `interp3`, the 'y' axis is considered to be the columns of the matrix, whereas the 'x' axis corresponds to the rows of the array. As Octave indexes arrays in column major order, the first dimension of any array is the columns, and so `interpn` effectively reverses the 'x' and 'y' dimensions. Consider the example

```
x = y = z = -1:1;
f = @(x,y,z) x.^2 - y - z.^2;
[xx, yy, zz] = meshgrid (x, y, z);
v = f (xx,yy,zz);
xi = yi = zi = -1:0.1:1;
[xxi, yyi, zzi] = meshgrid (xi, yi, zi);
vi = interp3(x, y, z, v, xxi, yyi, zzi, 'spline');
[xxi, yyi, zzi] = ndgrid (xi, yi, zi);
vi2 = interpn(x, y, z, v, xxi, yyi, zzi, 'spline');
mesh (zi, yi, squeeze (vi2(1,:,:)));
```

where `vi` and `vi2` are identical. The reversal of the dimensions is treated in the `meshgrid` and `ndgrid` functions respectively. The result of this code can be seen in [Figure 29.4](#).

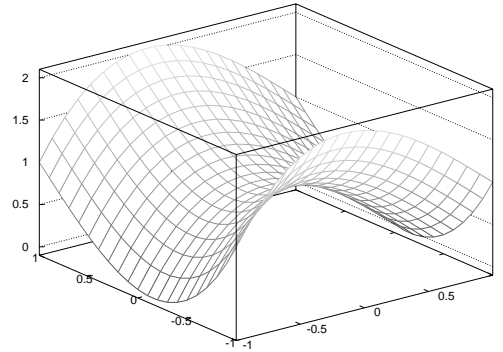


Figure 29.4: Demonstration of the use of `interp2n`

In addition the support function `bicubic` that underlies the cubic interpolation of `interp2` function can be called directly.

`zi = bicubic (x, y, z, xi, yi, extrapolval)` [Function File]

Return a matrix `zi` corresponding to the bicubic interpolations at `xi` and `yi` of the data supplied as `x`, `y` and `z`. Points outside the grid are set to `extrapolval`

See <http://wiki.woodpecker.org.cn/moin/Octave/Bicubic> for further information.

See also: `interp2`.

30 Geometry

`polyarea (x, y)` [Function File]

`polyarea (x, y, dim)` [Function File]

Determines area of a polygon by triangle method. The variables `x` and `y` define the vertex pairs, and must therefore have the same shape. They might be either vectors or arrays. If they are arrays then the columns of `x` and `y` are treated separately and an area returned for each.

If the optional `dim` argument is given, then `polyarea` works along this dimension of the arrays `x` and `y`.

31 Control Theory

The Octave Control Systems Toolbox (OCST) was initially developed by Dr. A. Scottedward Hodel a.s.hodel@eng.auburn.edu with the assistance of his students

- R. Bruce Tenison btenison@dibbs.net,
- David C. Clem,
- John E. Ingram John.Ingram@sea.siemans.com, and
- Kristi McGowan.

This development was supported in part by NASA's Marshall Space Flight Center as part of an in-house CACSD environment. Additional important contributions were made by Dr. Kai Mueller mueller@ifr.ing.tu-bs.de and Jose Daniel Munoz Frias (place.m).

An on-line menu-driven tutorial is available via `DEMOcontrol`; beginning OCST users should start with this program.

`DEMOcontrol` [Function File]

Octave Control Systems Toolbox demo/tutorial program. The demo allows the user to select among several categories of OCST function:

```
octave:1> DEMOcontrol
O C T A V E      C O N T R O L   S Y S T E M S   T O O L B O X
Octave Controls System Toolbox Demo

[ 1] System representation
[ 2] Block diagram manipulations
[ 3] Frequency response functions
[ 4] State space analysis functions
[ 5] Root locus functions
[ 6] LQG/H2/Hinfinity functions
[ 7] End
```

Command examples are interactively run for users to observe the use of OCST functions.

See also: Demo Programs: `bddemo.m`, `frdemo.m`, `analdemo.m`, `moddmeo.m`, `rldemo.m`.

31.1 System Data Structure

The OCST stores all dynamic systems in a single data structure format that can represent continuous systems, discrete-systems, and mixed (hybrid) systems in state-space form, and can also represent purely continuous/discrete systems in either transfer function or pole-zero form. In order to provide more flexibility in treatment of discrete/hybrid systems, the OCST also keeps a record of which system outputs are sampled.

Octave structures are accessed with a syntax much like that used by the C programming language. For consistency in use of the data structure used in the OCST, it is recommended that the system structure access m-files be used (see [Section 31.2 \[sysinterface\]](#), page 353). Some elements of the data structure are absent depending on the internal system representation(s) used. More than one system representation can be used for SISO systems; the OCST m-files ensure that all representations used are consistent with one another.

sysrepdemo

[Function File]

Tutorial for the use of the system data structure functions.

31.1.1 Variables common to all OCSST system formats

The data structure elements (and variable types) common to all system representations are listed below; examples of the initialization and use of the system data structures are given in subsequent sections and in the online demo `DEMOcontrol`.

<i>n</i>	
<i>nz</i>	The respective number of continuous and discrete states in the system (scalar)
<i>inname</i>	
<i>outname</i>	list of name(s) of the system input, output signal(s). (list of strings)
<i>sys</i>	System status vector. (vector) This vector indicates both what representation was used to initialize the system data structure (called the primary system type) and which other representations are currently up-to-date with the primary system type (see Section 31.2.5 [structaccess] , page 362). The value of the first element of the vector indicates the primary system type. 0 for tf form (initialized with <code>tf2sys</code> or <code>fir2sys</code>) 1 for zp form (initialized with <code>zp2sys</code>) 2 for ss form (initialized with <code>ss2sys</code>) The next three elements are boolean flags that indicate whether tf, zp, or ss, respectively, are “up to date” (whether it is safe to use the variables associated with these representations). These flags are changed when calls are made to the <code>sysupdate</code> command.
<i>tsam</i>	Discrete time sampling period (nonnegative scalar). <i>tsam</i> is set to 0 for continuous time systems.
<i>yd</i>	Discrete-time output list (vector) indicates which outputs are discrete time (i.e., produced by D/A converters) and which are continuous time. <i>yd</i> (ii) = 0 if output ii is continuous, = 1 if discrete.

The remaining variables of the system data structure are only present if the corresponding entry of the `sys` vector is true (=1).

31.1.2 tf format variables

<i>num</i>	numerator coefficients (vector)
<i>den</i>	denominator coefficients (vector)

31.1.3 zp format variables

<i>zer</i>	system zeros (vector)
<i>pol</i>	system poles (vector)
<i>k</i>	leading coefficient (scalar)

31.1.4 ss format variables

<i>a</i>	
<i>b</i>	
<i>c</i>	
<i>d</i>	The usual state-space matrices. If a system has both continuous and discrete states, they are sorted so that continuous states come first, then discrete states
	Note some functions (e.g., <code>bode</code> , <code>hinfsv</code>) will not accept systems with both discrete and continuous states/outputs
<i>stname</i>	names of system states (list of strings)

31.2 System Construction and Interface Functions

Construction and manipulations of the OCSST system data structure (see [Section 31.1 \[sysstruct\]](#), page 351) requires attention to many details in order to ensure that data structure contents remain consistent. Users are strongly encouraged to use the system interface functions in this section. Functions for the formatted display in of system data structures are given in [Section 31.3 \[sysdisp\]](#), page 367.

31.2.1 Finite impulse response system interface functions

`fir2sys (num, tsam, inname, outname)` [Function File]

construct a system data structure from FIR description

Inputs

num vector of coefficients $[c_0, c_1, \dots, c_n]$ of the SISO FIR transfer function

$$C(z) = c_0 + c_1 z^{-1} + c_2 z^{-2} + \dots + c_n z^{-n}$$

tsam sampling time (default: 1)

inname name of input signal; may be a string or a list with a single entry.

outname name of output signal; may be a string or a list with a single entry.

Output

sys system data structure

Example

```
octave:1> sys = fir2sys([1 -1 2 4],0.342,\
> "A/D input","filter output");
octave:2> sysout(sys)
Input(s)
    1: A/D input
```

```
Output(s):
    1: filter output (discrete)
```

```
Sampling interval: 0.342
```

```

transfer function form:
1*z^3 - 1*z^2 + 2*z^1 + 4
-----
1*z^3 + 0*z^2 + 0*z^1 + 0

```

`[c, tsam, input, output] = sys2fir (sys)` [Function File]

Extract FIR data from system data structure; see `fir2sys` for parameter descriptions.

See also: `fir2sys`.

31.2.2 State space system interface functions

`outsys = ss (a, b, c, d, tsam, n, nz, stname, inname, outname, outlist)` [Function File]

Create system structure from state-space data. May be continous, discrete, or mixed (sampled data)

Inputs

a

b

c

d

usual state space matrices.

default: *d* = zero matrix

tsam

sampling rate. Default: *tsam* = 0 (continuous system)

n

nz

number of continuous, discrete states in the system

If *tsam* is 0, *n* = `rows(a)`, *nz* = 0.

If *tsam* is greater than zero, *n* = 0, *nz* = `rows(a)`

see below for system partitioning

stname

cell array of strings of state signal names

default (*stname*=[] on input): *x_n* for continuous states, *xd_n* for discrete states

inname

cell array of strings of input signal names

default (*inname* = [] on input): *u_n*

outname

cell array of strings of output signal names

default (*outname* = [] on input): *y_n*

outlist

list of indices of outputs *y* that are sampled

If *tsam* is 0, *outlist* = [].

If *tsam* is greater than 0, *outlist* = 1 : `rows(c)`.

Unlike states, discrete/continous outputs may appear in any order.

`sys2ss` returns a vector *yd* where *yd(outlist)* = 1; all other entries of *yd* are 0.

Output

outsys system data structure

System partitioning

Suppose for simplicity that *outlist* specified that the first several outputs were continuous and the remaining outputs were discrete. Then the system is partitioned as

$$\begin{aligned} \mathbf{x} &= \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_d \end{bmatrix} \quad (\mathbf{n} \times 1) \\ &\quad (\mathbf{n}_z \times 1 \text{ discrete states}) \\ \mathbf{a} &= \begin{bmatrix} a_{cc} & a_{cd} \\ a_{dc} & a_{dd} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_c \\ b_d \end{bmatrix} \\ \mathbf{c} &= \begin{bmatrix} c_{cc} & c_{cd} \\ c_{dc} & c_{dd} \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} d_c \\ d_d \end{bmatrix} \end{aligned}$$

$$(cdc = c(\text{outlist}, 1:n), \text{ etc.})$$

with dynamic equations:

$$\begin{aligned} \frac{d}{dt}x_c(t) &= a_{cc}x_c(t) + a_{cd}x_d(k * t_{sam}) + b_c * u(t) \\ x_d((k+1) * t_{sam}) &= a_{dc}x_c(kt_{sam}) + a_{dd}x_d(kt_{sam}) + b_d u(kt_{sam}) \\ y_c(t) &= c_{cc}x_c(t) + c_{cd}x_d(kt_{sam}) + d_c u(t) \\ y_d(kt_{sam}) &= c_{dc}x_c(kt_{sam}) + c_{dd}x_d(kt_{sam}) + d_d u(kt_{sam}) \end{aligned}$$

Signal partitions

	continuous	discrete	
states	<code>stname(1:n,:)</code>	<code>stname((n+1):(n+nz),:)</code>	
outputs	<code>outname(cout,:)</code>	<code>outname(outlist,:)</code>	

where *cout* is the list of in `1:rows(p)` that are not contained in *outlist*. (Discrete/continuous outputs may be entered in any order desired by the user.)

Example

```
octave:1> a = [1 2 3; 4 5 6; 7 8 10];
octave:2> b = [0 0 ; 0 1 ; 1 0];
octave:3> c = eye (3);
octave:4> sys = ss (a, b, c, [], 0, 3, 0, {"volts", "amps", "joules"});
octave:5> sysout(sys);
Input(s)
    1: u_1
    2: u_2

Output(s):
    1: y_1
    2: y_2
    3: y_3
```

```

state-space form:
3 continuous states, 0 discrete states
State(s):
    1: volts
    2: amps
    3: joules

A matrix: 3 x 3
    1    2    3
    4    5    6
    7    8   10

B matrix: 3 x 2
    0    0
    0    1
    1    0

C matrix: 3 x 3
    1    0    0
    0    1    0
    0    0    1

D matrix: 3 x 3
    0    0
    0    0
    0    0

```

Notice that the D matrix is constructed by default to the correct dimensions. Default input and output signals names were assigned since none were given.

ss (*a*, *b*, *c*, *d*, *tsam*, *n*, *nz*, *stname*, *inname*, *outname*, *outlist*) [Function File]

Create system structure from state-space data. May be continous, discrete, or mixed (sampled data)

Inputs

<i>a</i>	
<i>b</i>	
<i>c</i>	
<i>d</i>	usual state space matrices. default: $d = \text{zero matrix}$
<i>tsam</i>	sampling rate. Default: $tsam = 0$ (continuous system)
<i>n</i>	
<i>nz</i>	number of continuous, discrete states in the system If $tsam$ is 0, $n = \text{rows}(a)$, $nz = 0$. If $tsam$ is greater than zero, $n = 0$, $nz = \text{rows}(a)$ see below for system partitioning
<i>stname</i>	cell array of strings of state signal names default ($stname = []$ on input): $\mathbf{x_n}$ for continuous states, $\mathbf{xd_n}$ for discrete states

inname cell array of strings of input signal names
 default (*inname* = [] on input): **u_n**

outname cell array of strings of input signal names
 default (*outname* = [] on input): **y_n**

outlist
 list of indices of outputs y that are sampled
 If *tsam* is 0, *outlist* = [].
 If *tsam* is greater than 0, *outlist* = 1 : rows(*c*).

Unlike states, discrete/continuous outputs may appear in any order.

sys2ss returns a vector *yd* where *yd(outlist)* = 1; all other entries of *yd* are 0.

Outputs *outsys* = system data structure

System partitioning

Suppose for simplicity that *outlist* specified that the first several outputs were continuous and the remaining outputs were discrete. Then the system is partitioned as

```
x = [ xc ]   (n x 1)
     [ xd ]   (nz x 1 discrete states)
a = [ acc acd ]   b = [ bc ]
     [ adc add ]   [ bd ]
c = [ ccc ccd ]   d = [ dc ]
     [ cdc cdd ]   [ dd ]

(cdc = c(outlist,1:n), etc.)
```

with dynamic equations:

$$\begin{aligned}\frac{d}{dt}x_c(t) &= a_{cc}x_c(t) + a_{cd}x_d(k * t_{sam}) + bc * u(t) \\ x_d((k+1) * t_{sam}) &= a_{dc}x_c(kt_{sam}) + a_{dd}x_d(kt_{sam}) + b_d u(kt_{sam}) \\ y_c(t) &= c_{cc}x_c(t) + c_{cd}x_d(kt_{sam}) + d_c u(t) \\ y_d(kt_{sam}) &= c_{dc}x_c(kt_{sam}) + c_{dd}x_d(kt_{sam}) + d_d u(kt_{sam})\end{aligned}$$

Signal partitions

	continuous	discrete	
states	stname(1:n,:)	stname((n+1):(n+nz),:)	
outputs	outname(cout,:)	outname(outlist,:)	

where *cout* is the list of in 1:rows(*p*) that are not contained in *outlist*. (Discrete/continuous outputs may be entered in any order desired by the user.)

Example

```

octave:1> a = [1 2 3; 4 5 6; 7 8 10];
octave:2> b = [0 0 ; 0 1 ; 1 0];
octave:3> c = eye (3);
octave:4> sys = ss (a, b, c, [], 0, 3, 0, {"volts", "amps", "joules"});
octave:5> sysout(sys);
Input(s)
      1: u_1
      2: u_2

Output(s):
      1: y_1
      2: y_2
      3: y_3

state-space form:
3 continuous states, 0 discrete states
State(s):
      1: volts
      2: amps
      3: joules

A matrix: 3 x 3
      1   2   3
      4   5   6
      7   8  10
B matrix: 3 x 2
      0   0
      0   1
      1   0
C matrix: 3 x 3
      1   0   0
      0   1   0
      0   0   1
D matrix: 3 x 3
      0   0
      0   0
      0   0

```

Notice that the D matrix is constructed by default to the correct dimensions. Default input and output signals names were assigned since none were given.

```
[a, b, c, d, tsam, n, nz, stname, inname, outname, yd] [Function File]
= sys2ss (sys)
```

Extract state space representation from system data structure.

Input

sys System data structure.

Outputs

a
b
c
d State space matrices for *sys*.
tsam Sampling time of *sys* (0 if continuous).
n
nz Number of continuous, discrete states (discrete states come last in state vector *x*).

stname
iname
outname Signal names (lists of strings); names of states, inputs, and outputs, respectively.

yd Binary vector; *yd(ii)* is 1 if output *y(ii)* is discrete (sampled); otherwise *yd(ii)* is 0.

A warning message is printed if the system is a mixed continuous and discrete system.

Example

```

octave:1> sys=tf2sys([1 2],[3 4 5]);
octave:2> [a,b,c,d] = sys2ss(sys)
a =
    0.00000    1.00000
   -1.66667   -1.33333
b =
     0
     1
c = 0.66667    0.33333
d = 0
  
```

31.2.3 Transfer function system interface functions

tf (*num*, *den*, *tsam*, *iname*, *outname*) [Function File]

build system data structure from transfer function format data

Inputs

num
den coefficients of numerator/denominator polynomials
tsam sampling interval. default: 0 (continuous time)
iname
outname input/output signal names; may be a string or cell array with a single string entry.

Outputs *sys* = system data structure

Example

```

octave:1> sys=tf([2 1],[1 2 1],0.1);
octave:2> sysout(sys)
  
```

```

Input(s)
    1: u_1
Output(s):
    1: y_1 (discrete)
Sampling interval: 0.1
transfer function form:
2*z^1 + 1
-----
1*z^2 + 2*z^1 + 1

```

tf2sys (*num*, *den*, *tsam*, *inname*, *outname*)

[Function File]

Build system data structure from transfer function format data.

Inputs

num
den Coefficients of numerator/denominator polynomials.
tsam Sampling interval; default: 0 (continuous time).
inname
outname Input/output signal names; may be a string or cell array with a single string entry.

Output

sys System data structure.

Example

```

octave:1> sys=tf2sys([2 1],[1 2 1],0.1);
octave:2> sysout(sys)
Input(s)
    1: u_1
Output(s):
    1: y_1 (discrete)
Sampling interval: 0.1
transfer function form:
2*z^1 + 1
-----
1*z^2 + 2*z^1 + 1

```

[num, den, tsam, inname, outname] = sys2tf (*sys*)

[Function File]

Extract transfer function data from a system data structure.

See **tf** for parameter descriptions.

Example

```

octave:1> sys=ss([1 -2; -1.1,-2.1],[0;1],[1 1]);
octave:2> [num,den] = sys2tf(sys)
num = 1.0000 -3.0000
den = 1.0000 1.1000 -4.3000

```

31.2.4 Zero-pole system interface functions

zp (*zer, pol, k, tsam, inname, outname*) [Function File]

Create system data structure from zero-pole data.

Inputs

zer vector of system zeros
pol vector of system poles
k scalar leading coefficient
tsam sampling period. default: 0 (continuous system)
inname
outname input/output signal names (lists of strings)

Outputs *sys*: system data structure

Example

```
octave:1> sys=zp([1 -1],[-2 -2 0],1);
octave:2> sysout(sys)
Input(s)
      1: u_1
Output(s):
      1: y_1
zero-pole form:
1 (s - 1) (s + 1)
-----
s (s + 2) (s + 2)
```

zp2sys (*zer, pol, k, tsam, inname, outname*) [Function File]

Create system data structure from zero-pole data.

Inputs

zer Vector of system zeros.
pol Vector of system poles.
k Scalar leading coefficient.
tsam Sampling period; default: 0 (continuous system).
inname
outname Input/output signal names (lists of strings).

Output

sys System data structure.

Example

```
octave:1> sys=zp2sys([1 -1],[-2 -2 0],1);
octave:2> sysout(sys)
Input(s)
      1: u_1
```

```

Output(s):
      1: y_1
zero-pole form:
1 (s - 1) (s + 1)
-----
s (s + 2) (s + 2)

```

`[zer, pol, k, tsam, inname, outname] = sys2zp (sys)` [Function File]

Extract zero/pole/leading coefficient information from a system data structure.

See `zp` for parameter descriptions.

Example

```

octave:1> sys=ss([1 -2; -1.1,-2.1],[0;1],[1 1]);
octave:2> [zer,pol,k] = sys2zp(sys)
zer = 3.0000
pol =
    -2.6953
     1.5953
k = 1

```

31.2.5 Data structure access functions

`syschnames (sys, opt, list, names)` [Function File]

Superseded by `syssetsignals`.

`syschtsam (sys, tsam)` [Function File]

This function changes the sampling time (`tsam`) of the system. Exits with an error if `sys` is purely continuous time.

`[n, nz, m, p, yd] = sysdimensions (sys, opt)` [Function File]

return the number of states, inputs, and/or outputs in the system `sys`.

Inputs

<code>sys</code>	system data structure
<code>opt</code>	String indicating which dimensions are desired. Values:
"all"	(default) return all parameters as specified under Outputs below.
"cst"	return n = number of continuous states
"dst"	return n = number of discrete states
"in"	return n = number of inputs
"out"	return n = number of outputs

Outputs

n	number of continuous states (or individual requested dimension as specified by <code>opt</code>).
-----	--

<i>nz</i>	number of discrete states
<i>m</i>	number of system inputs
<i>p</i>	number of system outputs
<i>yd</i>	binary vector; $yd(ii)$ is nonzero if output ii is discrete. $yd(ii) = 0$ if output ii is continuous

See also: `sysgetsignals`, `sysgettsam`.

<code>[stname, inname, outname, yd] = sysgetsignals (sys)</code>	[Function File]
<code>siglist = sysgetsignals (sys, sigid)</code>	[Function File]
<code>signame = sysgetsignals (sys, sigid, signum, strflg)</code>	[Function File]

Get signal names from a system

Inputs

<i>sys</i>	system data structure for the state space system								
<i>sigid</i>	signal id. String. Must be one of <table border="0"> <tr> <td>"in"</td><td>input signals</td></tr> <tr> <td>"out"</td><td>output signals</td></tr> <tr> <td>"st"</td><td>stage signals</td></tr> <tr> <td>"yd"</td><td>value of logical vector <i>yd</i></td></tr> </table>	"in"	input signals	"out"	output signals	"st"	stage signals	"yd"	value of logical vector <i>yd</i>
"in"	input signals								
"out"	output signals								
"st"	stage signals								
"yd"	value of logical vector <i>yd</i>								
<i>signum</i>	index(indices) or name(s) or signals; see <code>sysidx</code>								
<i>strflg</i>	flag to return a string instead of a cell array; Values: <table border="0"> <tr> <td>0</td><td>(default) return a cell array (even if <i>signum</i> specifies an individual signal)</td></tr> <tr> <td>1</td><td>return a string. Exits with an error if <i>signum</i> does not specify an individual signal.</td></tr> </table>	0	(default) return a cell array (even if <i>signum</i> specifies an individual signal)	1	return a string. Exits with an error if <i>signum</i> does not specify an individual signal.				
0	(default) return a cell array (even if <i>signum</i> specifies an individual signal)								
1	return a string. Exits with an error if <i>signum</i> does not specify an individual signal.								

Outputs

- If *sigid* is not specified:

<i>stname</i>	
<i>inname</i>	
<i>outname</i>	signal names (cell array of strings); names of states, inputs, and outputs, respectively.
<i>yd</i>	binary vector; $yd(ii)$ is nonzero if output ii is discrete.

- If *sigid* is specified but *signum* is not specified:

<code>sigid="in"</code>	<i>siglist</i> is set to the cell array of input names.
<code>sigid="out"</code>	<i>siglist</i> is set to the cell array of output names.

```
sigid="st"
```

siglist is set to the cell array of state names.

stage signals

```
sigid="yd"
```

siglist is set to logical vector indicating discrete outputs;
siglist(ii) = 0 indicates that output *ii* is continuous (unsampled), otherwise it is discrete.

- If the first three input arguments are specified:

signame is a cell array of the specified signal names (*sigid* is "in", "out", or "st"), or else the logical flag indicating whether output(s) *signame* is(are) discrete (*sigval*=1) or continuous (*sigval*=0).

Examples (From sysrepdemo)

```
octave> sys=ss(rand(4),rand(4,2),rand(3,4));
octave># get all signal names
octave> [Ast,Ain,Aout,Ayd] = sysgetsignals(sys)
Ast =
(
    [1] = x_1
    [2] = x_2
    [3] = x_3
    [4] = x_4
)
Ain =
(
    [1] = u_1
    [2] = u_2
)
Aout =
(
    [1] = y_1
    [2] = y_2
    [3] = y_3
)
Ayd =

    0    0    0
octave> # get only input signal names:
octave> Ain = sysgetsignals(sys,"in")
Ain =
(
    [1] = u_1
    [2] = u_2
)
octave> # get name of output 2 (in cell array):
octave> Aout = sysgetsignals(sys,"out",2)
```



```

Aout =
(
    [1] = y_2
)
octave> # get name of output 2 (as string):
octave> Aout = sysgetsignals(sys,"out",2,1)
Aout = y_2

```

sysgettype (*sys*) [Function File]
 return the initial system type of the system

Input

sys System data structure.

Output

systype String indicating how the structure was initially constructed. Values:
 "ss", "zp", or "tf".

FIR initialized systems return *systype*="tf".

syssetsignals (*sys, opt, names, sig_idx*) [Function File]
 change the names of selected inputs, outputs and states.

Inputs

sys System data structure.

opt Change default name (output).

 "out" Change selected output names.

 "in" Change selected input names.

 "st" Change selected state names.

 "yd" Change selected outputs from discrete to continuous or from
 continuous to discrete.

names

opt = "out", "in", "st"

 string or string array containing desired signal names or values.

opt = "yd"

 To desired output continuous/discrete flag. Set name to 0 for continuous, or 1 for discrete.

sig_idx indices or names of outputs, yd, inputs, or states whose respective
 names/values should be changed.

 Default: replace entire cell array of names/entire yd vector.

Outputs

retsys *sys* with appropriate signal names changed (or *yd* values, where appropriate).

Example

```
octave:1> sys=ss([1 2; 3 4],[5;6],[7 8]);
octave:2> sys = syssetsignals(sys,"st",str2mat("Posx","Velx"));
octave:3> sysout(sys)
Input(s)
      1: u_1
Output(s):
      1: y_1
state-space form:
2 continuous states, 0 discrete states
State(s):
      1: Posx
      2: Velx
A matrix: 2 x 2
      1  2
      3  4
B matrix: 2 x 1
      5
      6
C matrix: 1 x 2
      7  8
D matrix: 1 x 1
      0
```

sysupdate (*sys*, *opt*) [Function File]

Update the internal representation of a system.

Inputs

sys: system data structure

opt string:

- "tf" update transfer function form
- "zp" update zero-pole form
- "ss" update state space form
- "all" all of the above

Outputs

retsys Contains union of data in *sys* and requested data. If requested data in *sys* is already up to date then *retsys*=*sys*.

Conversion to **tf** or **zp** exits with an error if the system is mixed continuous/digital.

See also: **tf**, **ss**, **zp**, **sysout**, **sys2ss**, **sys2tf**, **sys2zp**.

`[systype, nout, nin, ncstates, ndstates] = minfo` [Function File]
 (*inmat*)

Determines the type of system matrix. *inmat* can be a varying, a system, a constant, and an empty matrix.

Outputs

systype Can be one of: varying, system, constant, and empty.

nout The number of outputs of the system.

nin The number of inputs of the system.

ncstates The number of continuous states of the system.

ndstates The number of discrete states of the system.

`sysgettsam (sys)` [Function File]

Return the sampling time of the system *sys*.

31.3 System display functions

`sysout (sys, opt)` [Function File]

print out a system data structure in desired format

sys system data structure

opt Display option

`[]` primary system form (default)

`"ss"` state space form

`"tf"` transfer function form

`"zp"` zero-pole form

`"all"` all of the above

`tfout (num, denom, x)` [Function File]

Print formatted transfer function $n(s)/d(s)$ to the screen. *x* defaults to the string `"s"`

See also: polyval, polyvalm, poly, roots, conv, deconv, residue, filter, polyderiv, polyinteg, polyout.

`zpout (zer, pol, k, x)` [Function File]

print formatted zero-pole form to the screen. *x* defaults to the string `"s"`

See also: polyval, polyvalm, poly, roots, conv, deconv, residue, filter, polyderiv, polyinteg, polyout.

31.4 Block Diagram Manipulations

See [Section 31.7 \[systime\]](#), page 385.

Unless otherwise noted, all parameters (input,output) are system data structures.

bddemo (*inputs*) [Function File]

Octave Controls toolbox demo: Block Diagram Manipulations demo.

buildssic (*clst, ulst, olst, ilst, s1, s2, s3, s4, s5, s6, s7, s8*) [Function File]

Form an arbitrary complex (open or closed loop) system in state-space form from several systems. **buildssic** can easily (despite its cryptic syntax) integrate transfer functions from a complex block diagram into a single system with one call. This function is especially useful for building open loop interconnections for \mathcal{H}_∞ and \mathcal{H}_2 designs or for closing loops with these controllers.

Although this function is general purpose, the use of **sysgroup** **sysmult**, **sysconnect** and the like is recommended for standard operations since they can handle mixed discrete and continuous systems and also the names of inputs, outputs, and states.

The parameters consist of 4 lists that describe the connections outputs and inputs and up to 8 systems *s1*–*s8*. Format of the lists:

clst connection list, describes the input signal of each system. The maximum number of rows of *Clst* is equal to the sum of all inputs of *s1*–*s8*.

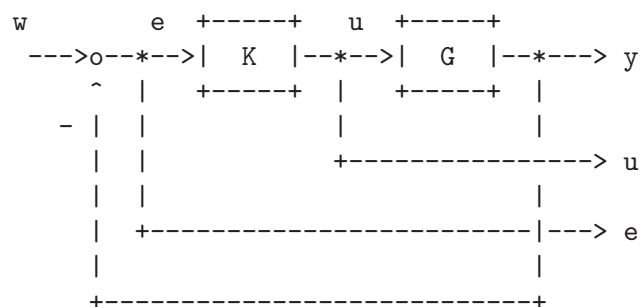
Example: `[1 2 -1; 2 1 0]` means that: new input 1 is old input 1 + output 2 - output 1, and new input 2 is old input 2 + output 1. The order of rows is arbitrary.

ulst if not empty the old inputs in vector *ulst* will be appended to the outputs. You need this if you want to “pull out” the input of a system. Elements are input numbers of *s1*–*s8*.

olst output list, specify the outputs of the resulting systems. Elements are output numbers of *s1*–*s8*. The numbers are allowed to be negative and may appear in any order. An empty matrix means all outputs.

ilst input list, specify the inputs of the resulting systems. Elements are input numbers of *s1*–*s8*. The numbers are allowed to be negative and may appear in any order. An empty matrix means all inputs.

Example: Very simple closed loop system.

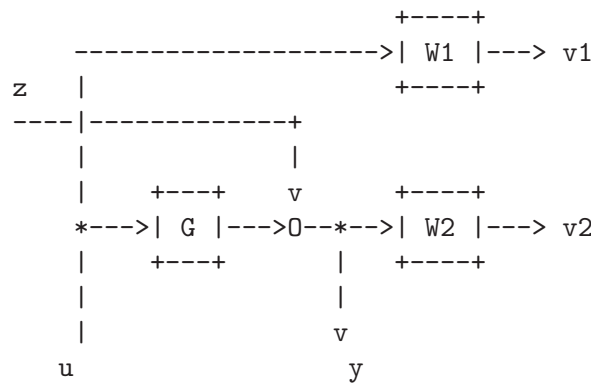


The closed loop system *GW* can be obtained by

```
GW = buildssic([1 2; 2 -1], 2, [1 2 3], 2, G, K);
```

clst 1st row: connect input 1 (G) with output 2 (K).
 2nd row: connect input 2 (K) with negative output 1 (G).
ulst Append input of 2 (K) to the number of outputs.
olst Outputs are output of 1 (G), 2 (K) and appended output 3 (from *ulst*).
ilst The only input is 2 (K).

Here is a real example:



$$\min \|GW_{vz}\|_{\infty}$$

The closed loop system GW from $[z, u]^T$ to $[v_1, v_2, y]^T$ can be obtained by (all SISO systems):

```
GW = buildssic([1, 4; 2, 4; 3, 1], 3, [2, 3, 5],
               [3, 4], G, W1, W2, One);
```

where “One” is a unity gain (auxillary) function with order 0. (e.g. `One = ugain(1);`)

`sys = jet707 ()` [Function File]

Creates a linearized state-space model of a Boeing 707-321 aircraft at $v=80$ m/s ($M = 0.26$, $G_{a0} = -3^\circ$, $\alpha_0 = 4^\circ$, $\kappa = 50^\circ$).

System inputs: (1) thrust and (2) elevator angle.

System outputs: (1) airspeed and (2) pitch angle.

Reference: R. Brockhaus: *Flugregelung* (Flight Control), Springer, 1994.

See also: `ord2`.

`ord2 (nfreq, damp, gain)` [Function File]

Creates a continuous 2nd order system with parameters:

Inputs

nfreq natural frequency [Hz]. (not in rad/s)

damp damping coefficient

gain dc-gain This is steady state value only for $\text{damp} > 0$. *gain* is assumed to be 1.0 if omitted.

Output

outsys system data structure has representation with $w = 2\pi f$:

$$G = \begin{bmatrix} \frac{1}{s} & \frac{-2w*damp}{s^2 + 2w*damp*s + w^2} & \frac{-w}{s^2 + 2w*damp*s + w^2} & \frac{1}{s} & \frac{w}{s^2 + 2w*damp*s + w^2} \\ \frac{1}{s} & \frac{1}{s} & \frac{1}{s} & \frac{1}{s} & \frac{1}{s} \end{bmatrix}, [0 \quad gain] \quad 0$$

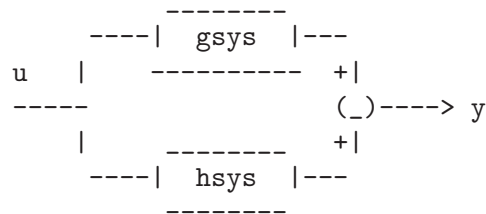
See also `jet707` (MIMO example, Boeing 707-321 aircraft model)

`sysadd (gsys, hsys)`

[Function File]

returns `sys = gsys + hsys`.

- Exits with an error if *gsys* and *hsys* are not compatibly dimensioned.
- Prints a warning message if system states have identical names; duplicate names are given a suffix to make them unique.
- *sys* input/output names are taken from *gsys*.



`sys = sysappend (syst, b, c, d, outname, inname, yd)`

[Function File]

appends new inputs and/or outputs to a system

Inputs

syst system data structure

b matrix to be appended to sys "B" matrix (empty if none)

c matrix to be appended to sys "C" matrix (empty if none)

d revised sys d matrix (can be passed as [] if the revised d is all zeros)

outname list of names for new outputs

inname list of names for new inputs

yd binary vector; $yd(ii) = 0$ indicates a continuous output; $yd(ii) = 1$ indicates a discrete output.

Outputs

sys

```

sys.b := [syst.b , b]
sys.c := [syst.c ]
        [ c      ]
sys.d := [syst.d | D12 ]
        [ D21   | D22 ]

```

where $D12$, $D21$, and $D22$ are the appropriate dimensioned blocks of the input parameter d .

- The leading block $D11$ of d is ignored.
- If *iname* and *outname* are not given as arguments, the new inputs and outputs are be assigned default names.
- *yd* is a binary vector of length `rows(c)` that indicates continuous/sampled outputs. Default value for *yd* is:
 - *sys* is continuous or mixed $yd = \text{zeros}(1, \text{rows}(c))$
 - *sys* is discrete $yd = \text{ones}(1, \text{rows}(c))$

`clsys = sysconnect (sys, out_idx, in_idx, order, tol)` [Function File]

Close the loop from specified outputs to respective specified inputs

Inputs

sys System data structure.

out_idx

in_idx Names or indices of signals to connect (see `sysidx`). The output specified by $out_idx(ii)$ is connected to the input specified by $in_idx(ii)$.

order logical flag (default = 0)

0 Leave inputs and outputs in their original order.

1 Permute inputs and outputs to the order shown in the diagram below.

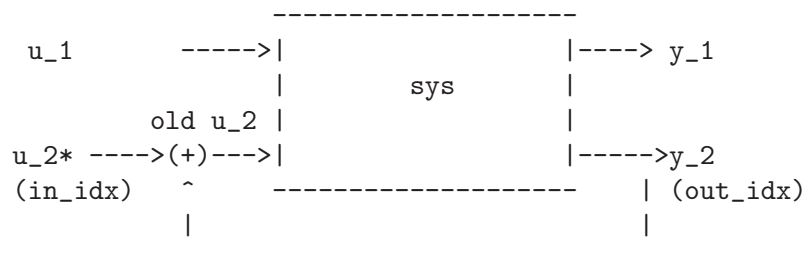
tol Tolerance for singularities in algebraic loops, default: `200eps`.

Outputs

clsys Resulting closed loop system.

Method

`sysconnect` internally permutes selected inputs, outputs as shown below, closes the loop, and then permutes inputs and outputs back to their original order



The input that has the summing junction added to it has an * added to the end of the input name.

`[csys, acd, ccd] = syscont (sys)` [Function File]

Extract the purely continuous subsystem of an input system.

Input

sys system data structure.

Outputs

csys is the purely continuous input/output connections of *sys*

acd

ccd connections from discrete states to continuous states, discrete states to continuous outputs, respectively.

returns *csys* empty if no continuous/continuous path exists

`[dsys, adc, cdc] = sysdisc (sys)` [Function File]

Input

sys System data structure.

Outputs

dsys Purely discrete portion of *sys* (returned empty if there is no purely discrete path from inputs to outputs).

adc

cdc Connections from continuous states to discrete states and discrete. outputs, respectively.

`retsys = sysdup (asys, out_idx, in_idx)` [Function File]

Duplicate specified input/output connections of a system

Inputs

asys system data structure

out_idx

in_idx indices or names of desired signals (see `sigidx`). duplicates are made of `y(out_idx(ii))` and `u(in_idx(ii))`.

Output

retsys Resulting closed loop system: duplicated i/o names are appended with a "+" suffix.

Method

`sysdup` creates copies of selected inputs and outputs as shown below. *u1*, *y1* is the set of original inputs/outputs, and *u2*, *y2* is the set of duplicated inputs/outputs in the order specified in *in_idx*, *out_idx*, respectively

```

      u1  ----->|-----> y1
                  |         |
                  |   asys  |
      u2  ----->|-----> y2
      (in_idx) ----- (out_idx)
```


sys = **sysgroup** (**asys**, **bsys**) [Function File]

Combines two systems into a single system.

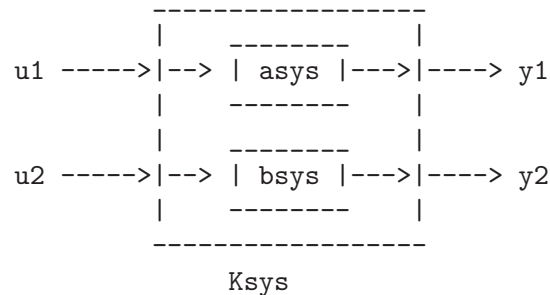
Inputs

asys

bsys System data structures.

Output

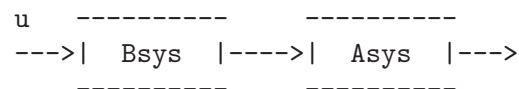
sys **sys** = **blockdiag**(**asys**, **bsys**)



The function also rearranges the internal state-space realization of **sys** so that the continuous states come first and the discrete states come last. If there are duplicate names, the second name has a unique suffix appended on to the end of the name.

sys = **sysmult** (**Asys**, **Bsys**) [Function File]

Compute **sys** = **Asys** * **Bsys** (series connection):



A warning occurs if there is direct feed-through from an input or a continuous state of **Bsys**, through a discrete output of **Bsys**, to a continuous state or output in **Asys** (system data structure does not recognize discrete inputs).

retsys = **sysprune** (**asys**, **out_idx**, **in_idx**) [Function File]

Extract specified inputs/outputs from a system

Inputs

asys system data structure

out_idx

in_idx Indices or signal names of the outputs and inputs to be kept in the returned system; remaining connections are “pruned” off. May select as [] (empty matrix) to specify all outputs/inputs.

```

retsys = sysprune (Asys, [1:3,4], "u_1");
retsys = sysprune (Asys, {"tx", "ty", "tz"}, 4);
  
```

Output

retsys Resulting system.



`pv = sysreorder (vlen, list)` [Function File]

Inputs

vlen Vector length.
list A subset of `[1:vlen]`.

Output

pv A permutation vector to order elements of `[1:vlen]` in *list* to the end of a vector.

Used internally by `sysconnect` to permute vector elements to their desired locations.

`retsys = sysscale (sys, outscale, inscale, outname, inname)` [Function File]

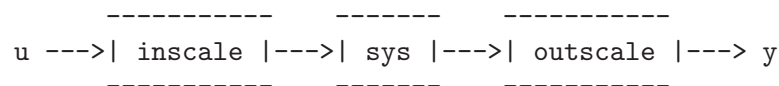
scale inputs/outputs of a system.

Inputs

sys Structured system.
outscale
inscale Constant matrices of appropriate dimension.
outname
inname Lists of strings with the names of respectively outputs and inputs.

Output

retsys resulting open loop system:



If the input names and output names (each a list of strings) are not given and the scaling matrices are not square, then default names will be given to the inputs and/or outputs.

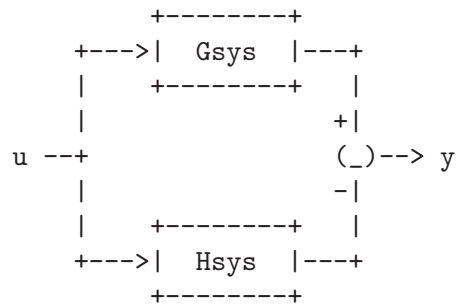
A warning message is printed if `outscale` attempts to add continuous system outputs to discrete system outputs; otherwise *yd* is set appropriately in the returned value of *sys*.

`sys = syssub (Gsys, Hsys)` [Function File]

Return $sys = Gsys - Hsys$.

Method

Gsys and *Hsys* are connected in parallel. The input vector is connected to both systems; the outputs are subtracted. Returned system names are those of *Gsys*.



ugain (*n*) [Function File]

Creates a system with unity gain, no states. This trivial system is sometimes needed to create arbitrary complex systems from simple systems with **buildssic**. Watch out if you are forming sampled systems since **ugain** does not contain a sampling period.

See also: **hinfdemo**, **jet707**.

W = **wgt1o** (*vl*, *vh*, *fc*) [Function File]

State space description of a first order weighting function.

Weighting function are needed by the $\mathcal{H}_2/\mathcal{H}_\infty$ design procedure. These function are part of the augmented plant *P* (see **hinfdemo** for an application example).

Inputs

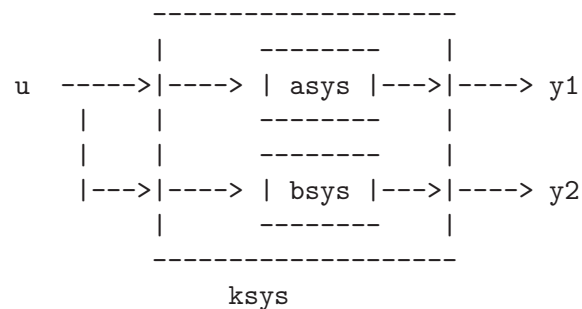
vl Gain at low frequencies.
vh Gain at high frequencies.
fc Corner frequency (in Hz, **not** in rad/sec)

Output

W Weighting function, given in form of a system data structure.

ksys = **parallel** (*asys*, *bsys*) [Function File]

Forms the parallel connection of two systems.



[retsys, nc, no] = **sysmin** (*sys*, *flg*) [Function File]

Returns a minimal (or reduced order) system

Inputs

sys System data structure

flg When equal to 0 (default value), returns minimal system, in which state names are lost; when equal to 1, returns system with physical states removed that are either uncontrollable or unobservable (cannot reduce further without discarding physical meaning of states).

Outputs

retsys Returned system.
nc Number of controllable states in the returned system.
no Number of observable states in the returned system.
cflg `is_controllable(retsys)`.
offg `is_observable(retsys)`.

31.5 Numerical Functions

x = are (a, b, c, opt) [Function File]
 Solve the Algebraic Riccati Equation

$$A^T X + X A - X B X + C = 0$$

Inputs for identically dimensioned square matrices

a *n* by *n* matrix;
b *n* by *n* matrix or *n* by *m* matrix; in the latter case *b* is replaced by $b := b * b'$;
c *n* by *n* matrix or *p* by *m* matrix; in the latter case *c* is replaced by $c := c' * c$;
opt (optional argument; default = "B"): String option passed to **balance** prior to ordered Schur decomposition.

Output

x solution of the ARE.

Method Laub's Schur method (IEEE Transactions on Automatic Control, 1979) is applied to the appropriate Hamiltonian matrix.

See also: **balance**, **dare**.

x = dare (a, b, q, r, opt) [Function File]
 Return the solution, *x* of the discrete-time algebraic Riccati equation

$$A^T X A - X + A^T X B (R + B^T X B)^{-1} B^T X A + Q = 0$$

Inputs

a *n* by *n* matrix;
b *n* by *m* matrix;

q n by n matrix, symmetric positive semidefinite, or a p by n matrix, In the latter case $q := q' * q$ is used;

r m by m , symmetric positive definite (invertible);

opt (optional argument; default = "B"): String option passed to **balance** prior to ordered QZ decomposition.

Output

x solution of DARE.

Method Generalized eigenvalue approach (Van Dooren; SIAM J. Sci. Stat. Comput., Vol 2) applied to the appropriate symplectic pencil.

See also: Ran and Rodman, *Stable Hermitian Solutions of Discrete Algebraic Riccati Equations*, Mathematics of Control, Signals and Systems, Vol 5, no 2 (1992), pp 165–194.

See also: balance, are.

`[tvals, plist] = dre(sys, q, r, qf, t0, tf, ptol, maxits)` [Function File]
Solve the differential Riccati equation

$$-\frac{dP}{dt} = A^T P + P A - P B R^{-1} B^T P + Q$$

$$P(t_f) = Q_f$$

for the LTI system `sys`. Solution of standard LTI state feedback optimization

$$\min \int_{t_0}^{t_f} x^T Q x + u^T R u dt + x(t_f)^T Q_f x(t_f)$$

optimal input is

$$u = -R^{-1} B^T P(t) x$$

Inputs

`sys` continuous time system data structure

`q` state integral penalty

`r` input integral penalty

`qf` state terminal penalty

`t0`

`tf` limits on the integral

`ptol` tolerance (used to select time samples; see below); default = 0.1

`maxits` number of refinement iterations (default=10)

Outputs

`tvals` time values at which $p(t)$ is computed

`plist` list values of $p(t)$; `plist { i }` is $p(tvals(i))$

`tvals` is selected so that:

$$\|plist_i - plist_{i-1}\| < ptol$$

for every i between 2 and `length(tvals)`.

dgram (*a*, *b*) [Function File]

Return controllability gramian of discrete time system

$$x_{k+1} = ax_k + bu_k$$

Inputs

a *n* by *n* matrix

b *n* by *m* matrix

Output

m *n* by *n* matrix, satisfies

$$ama^T - m + bb^T = 0$$

dlyap (*a*, *b*) [Function File]

Solve the discrete-time Lyapunov equation

Inputs

a *n* by *n* matrix;

b Matrix: *n* by *n*, *n* by *m*, or *p* by *n*.

Output

x matrix satisfying appropriate discrete time Lyapunov equation.

Options:

- *b* is square: solve

$$axa^T - x + b = 0$$

- *b* is not square: *x* satisfies either

$$axa^T - x + bb^T = 0$$

or

$$a^T xa - x + b^T b = 0,$$

whichever is appropriate.

Method Uses Schur decomposition method as in Kitagawa, *An Algorithm for Solving the Matrix Equation $X = FXF' + S$* , International Journal of Control, Volume 25, Number 5, pages 745–753 (1977).

Column-by-column solution method as suggested in Hammarling, *Numerical Solution of the Stable, Non-Negative Definite Lyapunov Equation*, IMA Journal of Numerical Analysis, Volume 2, pages 303–323 (1982).

gram (*a*, *b*) [Function File]

Return controllability gramian *m* of the continuous time system $dx/dt = ax + bu$.

m satisfies $am + ma' + bb' = 0$.

lyap (*a*, *b*, *c*) [Function File]

lyap (*a*, *b*) [Function File]

Solve the Lyapunov (or Sylvester) equation via the Bartels-Stewart algorithm (Communications of the ACM, 1972).

If *a*, *b*, and *c* are specified, then **lyap** returns the solution of the Sylvester equation

$$AX + XB + C = 0$$

If only (*a*, *b*) are specified, then **lyap** returns the solution of the Lyapunov equation

$$A^T X + XA + B = 0$$

If *b* is not square, then **lyap** returns the solution of either

$$A^T X + XA + B^T B = 0$$

or

$$AX + XA^T + BB^T = 0$$

whichever is appropriate.

Solves by using the Bartels-Stewart algorithm (1972).

qzval (*a*, *b*) [Function File]

Compute generalized eigenvalues of the matrix pencil $(A - \lambda B)$.

a and *b* must be real matrices.

qzval is obsolete; use **qz** instead.

y = **zgfmul** (*a*, *b*, *c*, *d*, *x*) [Function File]

Compute product of **zgep** incidence matrix *F* with vector *x*. Used by **zgepbal** (in **zgscal**) as part of generalized conjugate gradient iteration.

zgfslv (*n*, *m*, *p*, *b*) [Function File]

Solve system of equations for dense **zgep** problem.

zz = **zginit** (*a*, *b*, *c*, *d*) [Function File]

Construct right hand side vector **zz** for the zero-computation generalized eigenvalue problem balancing procedure. Called by **zgepbal**.

zgreduce (*sys*, *meps*) [Function File]

Implementation of procedure REDUCE in (Emami-Naeini and Van Dooren, Automatica, # 1982).

[**nonz**, **zer**] = **zgrownorm** (*mat*, *meps*) [Function File]

Return **nonz** = number of rows of *mat* whose two norm exceeds *meps*, and **zer** = number of rows of *mat* whose two norm is less than *meps*.

`x = zgscal (f, z, n, m, p)` [Function File]
 Generalized conjugate gradient iteration to solve zero-computation generalized eigenvalue problem balancing equation $fx = z$; called by `zgepbal`.

`[a, b] = zgsgiv (c, s, a, b)` [Function File]
 Apply givens rotation `c,s` to row vectors `a, b`. No longer used in zero-balancing (`--zgpbal--`); kept for backward compatibility.

`x = zgshsr (y)` [Function File]
 Apply householder vector based on e^m to column vector `y`. Called by `zgfslv`.

References

ZGEP Hodel, *Computation of Zeros with Balancing*, 1992, Linear Algebra and its Applications

Generalized CG

Golub and Van Loan, *Matrix Computations*, 2nd ed 1989.

31.6 System Analysis-Properties

`analdemo ()` [Function File]
 Octave Controls toolbox demo: State Space analysis demo

`[n, m, p] = abcd dim (a, b, c, d)` [Function File]
 Check for compatibility of the dimensions of the matrices defining the linear system $[A, B, C, D]$ corresponding to

$$\begin{aligned}\frac{dx}{dt} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

or a similar discrete-time system.

If the matrices are compatibly dimensioned, then `abcd dim` returns

`n` The number of system states.
`m` The number of system inputs.
`p` The number of system outputs.

Otherwise `abcd dim` returns $n = m = p = -1$.

Note: $n = 0$ (pure gain block) is returned without warning.

See also: `is_abcd`.

`ctrb (sys, b)` [Function File]
`ctrb (a, b)` [Function File]

Build controllability matrix:

$$Q_s = [BABA^2B \dots A^{n-1}B]$$

of a system data structure or the pair (a, b) .

`ctrb` forms the controllability matrix. The numerical properties of `is_controllable` are much better for controllability tests.

h2norm (*sys*) [Function File]

Computes the \mathcal{H}_2 norm of a system data structure (continuous time only).

Reference: Doyle, Glover, Khargonekar, Francis, *State-Space Solutions to Standard \mathcal{H}_2 and \mathcal{H}_∞ Control Problems*, IEEE TAC August 1989.

[g, gmin, gmax] = hinfnorm (*sys, tol, gmin, gmax, ptol*) [Function File]

Computes the \mathcal{H}_∞ norm of a system data structure.

Inputs

sys system data structure

tol \mathcal{H}_∞ norm search tolerance (default: 0.001)

gmin minimum value for norm search (default: 1e-9)

gmax maximum value for norm search (default: 1e+9)

ptol pole tolerance:

- if *sys* is continuous, poles with $|\text{real}(\text{pole})| < \text{ptol}\|H\|$ (H is appropriate Hamiltonian) are considered to be on the imaginary axis.
- if *sys* is discrete, poles with $|\text{pole} - 1| < \text{ptol}\|[s_1 s_2]\|$ (appropriate symplectic pencil) are considered to be on the unit circle.
- Default value: 1e-9

Outputs

g Computed gain, within *tol* of actual gain. *g* is returned as Inf if the system is unstable.

gmin

gmax Actual system gain lies in the interval [*gmin*, *gmax*].

References: Doyle, Glover, Khargonekar, Francis, *State-space solutions to standard \mathcal{H}_2 and \mathcal{H}_∞ control problems*, IEEE TAC August 1989; Iglesias and Glover, *State-Space approach to discrete-time \mathcal{H}_∞ control*, Int. J. Control, vol 54, no. 5, 1991; Zhou, Doyle, Glover, *Robust and Optimal Control*, Prentice-Hall, 1996.

obsv (*sys, c*) [Function File]

obsv (*a, c*) [Function File]

Build observability matrix:

$$Q_b = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

of a system data structure or the pair (*a, c*).

The numerical properties of **is_observable** are much better for observability tests.

`[zer, pol] = pzmap (sys)` [Function File]

Plots the zeros and poles of a system in the complex plane.

Input

sys System data structure.

Outputs

pol

zer if omitted, the poles and zeros are plotted on the screen. otherwise, *pol* and *zer* are returned as the system poles and zeros (see `sys2zp` for a preferable function call).

`retval = is_abcd (a, b, c, d)` [Function File]

Returns *retval* = 1 if the dimensions of *a*, *b*, *c*, *d* are compatible, otherwise *retval* = 0 with an appropriate diagnostic message printed to the screen. The matrices *b*, *c*, or *d* may be omitted.

See also: `abcdim`.

`[retval, u] = is_controllable (sys, tol)` [Function File]

`[retval, u] = is_controllable (a, b, tol)` [Function File]

Logical check for system controllability.

Inputs

sys system data structure

a

b *n* by *n*, *n* by *m* matrices, respectively

tol optional roundoff parameter. Default value: `10*eps`

Outputs

retval Logical flag; returns true (1) if the system *sys* or the pair (*a*, *b*) is controllable, whichever was passed as input arguments.

u *u* is an orthogonal basis of the controllable subspace.

Method Controllability is determined by applying Arnoldi iteration with complete re-orthogonalization to obtain an orthogonal basis of the Krylov subspace

`span ([b, a*b, ..., a^{n-1}*b]).`

The Arnoldi iteration is executed with `krylov` if the system has a single input; otherwise a block Arnoldi iteration is performed with `krylovb`.

See also: `size`, `rows`, `columns`, `length`, `ismatrix`, `isscalar`, `isvector`, `is_observable`, `is_stabilizable`, `is_detectable`, `krylov`, `krylovb`.

`retval = is_detectable (a, c, tol, dflg)` [Function File]

`retval = is_detectable (sys, tol)` [Function File]

Test for detectability (observability of unstable modes) of (*a*, *c*).

Returns 1 if the system a or the pair (a, c) is detectable, 0 if not, and -1 if the system has unobservable modes at the imaginary axis (unit circle for discrete-time systems).

See `is_stabilizable` for detailed description of arguments and computational method.

See also: `is_stabilizable`, `size`, `rows`, `columns`, `length`, `ismatrix`, `isscalar`, `isvector`.

`[retval, dgkf_struct] = is_dgkf (asys, nu, ny, tol)` [Function File]

Determine whether a continuous time state space system meets assumptions of DGKF algorithm. Partitions system into:

$$\begin{bmatrix} dx/dt \\ z \\ y \end{bmatrix} = \begin{bmatrix} A & | & Bw & Bu \\ Cz & | & Dzw & Dzu \\ Cy & | & Dyw & Dy \end{bmatrix} \begin{bmatrix} w \\ u \end{bmatrix}$$

or similar discrete-time system. If necessary, orthogonal transformations qw , qz and nonsingular transformations ru , ry are applied to respective vectors w , z , u , y in order to satisfy DGKF assumptions. Loop shifting is used if dyu block is nonzero.

Inputs

asys system data structure
nu number of controlled inputs
ny number of measured outputs
tol threshold for 0; default: 200*`eps`.

Outputs

retval true(1) if system passes check, false(0) otherwise

dgkf_struct data structure of `is_dgkf` results. Entries:

nw
nz dimensions of w , z
a system A matrix
bw $(n \times nw)$ qw -transformed disturbance input matrix
bu $(n \times nu)$ ru -transformed controlled input matrix;
 $B = [BwBu]$
cz $(nz \times n)$ Qz -transformed error output matrix
cy $(ny \times n)$ ry -transformed measured output matrix
 $C = [Cz; Cy]$
dzu
dyw off-diagonal blocks of transformed system D matrix that enter z , y from u , w respectively
ru controlled input transformation matrix
ry observed output transformation matrix

`dyu_nz` nonzero if the `dyu` block is nonzero.
`dyu` untransformed `dyu` block
`dflg` nonzero if the system is discrete-time

`is_dgkf` exits with an error if the system is mixed discrete/continuous.

References

- [1] Doyle, Glover, Khargonekar, Francis, *State Space Solutions to Standard \mathcal{H}_2 and \mathcal{H}_∞ Control Problems*, IEEE TAC August 1989.
- [2] Maciejowski, J.M., *Multivariable Feedback Design*, Addison-Wesley, 1989.

`digital = is_digital (sys, eflg)` [Function File]

Return nonzero if system is digital.

Inputs

`sys` System data structure.
`eflg` When equal to 0 (default value), exits with an error if the system is mixed (continuous and discrete components); when equal to 1, print a warning if the system is mixed (continuous and discrete); when equal to 2, operate silently.

Output

`digital` When equal to 0, the system is purely continuous; when equal to 1, the system is purely discrete; when equal to -1, the system is mixed continuous and discrete.

Exits with an error if `sys` is a mixed (continuous and discrete) system.

`[retval, u] = is_observable (a, c, tol)` [Function File]

`[retval, u] = is_observable (sys, tol)` [Function File]

Logical check for system observability.

Default: `tol = tol = 10*norm(a, 'fro')*eps`

Returns 1 if the system `sys` or the pair `(a, c)` is observable, 0 if not.

See `is_controllable` for detailed description of arguments and default values.

See also: `size`, `rows`, `columns`, `length`, `ismatrix`, `isscalar`, `isvector`.

`is_sample (ts)` [Function File]

Return true if `ts` is a valid sampling time (real, scalar, > 0).

`is_siso (sys)` [Function File]

Returns nonzero if the system data structure `sys` is single-input, single-output.

`retval = is_stabilizable (sys, tol)` [Function File]

`retval = is_stabilizable (a, b, tol, dflg)` [Function File]

Logical check for system stabilizability (i.e., all unstable modes are controllable). Returns 1 if the system is stabilizable, 0 if the system is not stabilizable, -1 if the system has non stabilizable modes at the imaginary axis (unit circle for discrete-time systems).

Test for stabilizability is performed via Hautus Lemma. If *dflg*≠0 assume that discrete-time matrices (a,b) are supplied.

See also: size, rows, columns, length, ismatrix, isscalar, isvector is_observable, is_stabilizable, is_detectable.

`is_signal_list (mylist)` [Function File]

Return true if *mylist* is a list of individual strings.

`is_stable (a, tol, dflg)` [Function File]

`is_stable (sys, tol)` [Function File]

Returns 1 if the matrix *a* or the system *sys* is stable, or 0 if not.

Inputs

tol is a roundoff parameter, set to 200***eps** if omitted.

dflg Digital system flag (not required for system data structure):

dflg != 0 stable if eig(a) is in the unit circle

dflg == 0 stable if eig(a) is in the open LHP (default)

See also: size, rows, columns, length, ismatrix, isscalar, isvector is_observable, is_stabilizable, is_detectable, krylov, krylovb.

31.7 System Analysis-Time Domain

`c2d (sys, opt, t)` [Function File]

`c2d (sys, t)` [Function File]

Converts the system data structure describing:

$$\dot{x} = A_c x + B_c u$$

into a discrete time equivalent model:

$$x_{n+1} = A_d x_n + B_d u_n$$

via the matrix exponential or bilinear transform.

Inputs

sys system data structure (may have both continuous time and discrete time subsystems)

opt string argument; conversion option (optional argument; may be omitted as shown above)

`[dsys, fidx] = dmr2d(sys, idx, sprefix, ts2, cuf1g)` [Function File]
 convert a multirate digital system to a single rate digital system states specified by *idx*, *sprefix* are sampled at *ts2*, all others are assumed sampled at *ts1* = `sysgettsam(sys)`.

Inputs

sys discrete time system; `dmr2d` exits with an error if *sys* is not discrete

idx indices or names of states with sampling time `sysgettsam(sys)` (may be empty); see `cellidx`

sprefix list of string prefixes of states with sampling time `sysgettsam(sys)` (may be empty)

ts2 sampling time of states not specified by *idx*, *sprefix* must be an integer multiple of `sysgettsam(sys)`

cuf1g "constant u flag" if *cuf1g* is nonzero then the system inputs are assumed to be constant over the revised sampling interval *ts2*. Otherwise, since the inputs can change during the interval *t* in $[kts2, (k+1)ts2]$, an additional set of inputs is included in the revised B matrix so that these intersample inputs may be included in the single-rate system. default *cuf1g* = 1.

Outputs

dsys equivalent discrete time system with sampling time *ts2*.
 The sampling time of *sys* is updated to *ts2*.
 if *cuf1g*=0 then a set of additional inputs is added to the system with suffixes *_d1*, ..., *_dn* to indicate their delay from the starting time *k ts2*, i.e. $u = [u_1; u_1_d1; \dots, u_1_dn]$ where *u_1_dk* is the input *k*ts1* units of time after *u_1* is sampled. (*ts1* is the original sampling time of the discrete time system and $ts2 = (n+1)*ts1$)

fidx indices of "formerly fast" states specified by *idx* and *sprefix*; these states are updated to the new (slower) sampling interval *ts2*.

WARNING Not thoroughly tested yet; especially when *cuf1g* == 0.

`damp(p, tsam)` [Function File]
 Displays eigenvalues, natural frequencies and damping ratios of the eigenvalues of a matrix *p* or the *A* matrix of a system *p*, respectively. If *p* is a system, *tsam* must not be specified. If *p* is a matrix and *tsam* is specified, eigenvalues of *p* are assumed to be in *z*-domain.

See also: `eig`.

`dcgain(sys, tol)` [Function File]
 Returns dc-gain matrix. If dc-gain is infinite an empty matrix is returned. The argument *tol* is an optional tolerance for the condition number of the *A* Matrix in *sys* (default *tol* = 1.0e-10)

`[y, t] = impulse (sys, inp, tstop, n)` [Function File]

Impulse response for a linear system. The system can be discrete or multivariable (or both). If no output arguments are specified, **impulse** produces a plot or the impulse response data for system *sys*.

Inputs

sys System data structure.

inp Index of input being excited

tstop The argument *tstop* (scalar value) denotes the time when the simulation should end.

n the number of data values.

Both parameters *tstop* and *n* can be omitted and will be computed from the eigenvalues of the A Matrix.

Outputs

y Values of the impulse response.

t Times of the impulse response.

See also: `step`, `--stepimp--`.

`[y, t] = step (sys, inp, tstop, n)` [Function File]

Step response for a linear system. The system can be discrete or multivariable (or both). If no output arguments are specified, **step** produces a plot or the step response data for system *sys*.

Inputs

sys System data structure.

inp Index of input being excited

tstop The argument *tstop* (scalar value) denotes the time when the simulation should end.

n the number of data values.

Both parameters *tstop* and *n* can be omitted and will be computed from the eigenvalues of the A Matrix.

Outputs

y Values of the step response.

t Times of the step response.

When invoked with the output parameter *y* the plot is not displayed.

See also: `impulse`, `--stepimp--`.

31.8 System Analysis-Frequency Domain

Demonstration/tutorial script

`frdemo ()` [Function File]

Octave Control Toolbox demo: Frequency Response demo.

`[mag, phase, w] = bode (sys, w, out_idx, in_idx)` [Function File]

If no output arguments are given: produce Bode plots of a system; otherwise, compute the frequency response of a system data structure

Inputs

`sys` a system data structure (must be either purely continuous or discrete; see `is_digital`)

`w` frequency values for evaluation.
 if `sys` is continuous, then `bode` evaluates $G(jw)$ where $G(s)$ is the system transfer function.
 if `sys` is discrete, then `bode` evaluates $G(\exp(jwT))$, where

- T is the system sampling time
- $G(z)$ is the system transfer function.

Default the default frequency range is selected as follows: (These steps are **not** performed if `w` is specified)

1. via routine `__bodquist__`, isolate all poles and zeros away from $w=0$ ($jw=0$ or $\exp(jwT)=1$) and select the frequency range based on the breakpoint locations of the frequencies.
2. if `sys` is discrete time, the frequency range is limited to jwT in $[0, 2\pi/T]$
3. A "smoothing" routine is used to ensure that the plot phase does not change excessively from point to point and that singular points (e.g., crossovers from ± 180) are accurately shown.

`out_idx`
`in_idx`

The names or indices of outputs and inputs to be used in the frequency response. See `sysprune`.

Example

```
bode(sys, [], "y_3", {"u_1", "u_4"});
```

Outputs

`mag`
`phase` the magnitude and phase of the frequency response $G(jw)$ or $G(\exp(jwT))$ at the selected frequency values.

`w` the vector of frequency values used

1. If no output arguments are given, e.g.,

```
bode(sys);
```

bode plots the results to the screen. Descriptive labels are automatically placed. Failure to include a concluding semicolon will yield some garbage being printed to the screen (`ans = []`).

2. If the requested plot is for an MIMO system, `mag` is set to $\|G(jw)\|$ or $\|G(\exp(jwT))\|$ and phase information is not computed.

```
[wmin, wmax] = bode_bounds (zer, pol, dflg, tsam) [Function File]
```

Get default range of frequencies based on cutoff frequencies of system poles and zeros. Frequency range is the interval $[10^{w_{min}}, 10^{w_{max}}]$

Used internally in `__freqresp__` (`bode`, `nyquist`)

```
freqchkw (w) [Function File]
```

Used by `__freqresp__` to check that input frequency vector `w` is valid. Returns boolean value.

```
out = ltifr (a, b, w) [Function File]
```

```
out = ltifr (sys, w) [Function File]
```

Linear time invariant frequency response of single-input systems.

Inputs

`a`
`b` coefficient matrices of $dx/dt = Ax + Bu$
`sys` system data structure
`w` vector of frequencies

Output

`out` frequency response, that is:

$$G(j\omega) = (j\omega I - A)^{-1}B$$

for complex frequencies $s = jw$.

```
[realp, imagp, w] = nyquist (sys, w, out_idx, in_idx, atol) [Function File]
```

```
nyquist (sys, w, out_idx, in_idx, atol) [Function File]
```

Produce Nyquist plots of a system; if no output arguments are given, Nyquist plot is printed to the screen.

Compute the frequency response of a system.

Inputs (pass as empty to get default values)

`sys` system data structure (must be either purely continuous or discrete; see `is_digital`)

`w` frequency values for evaluation. If `sys` is continuous, then `bode` evaluates $G(jw)$; if `sys` is discrete, then `bode` evaluates $G(\exp(jwT))$, where T is the system sampling time.

- default* the default frequency range is selected as follows: (These steps are **not** performed if *w* is specified)
1. via routine `__bodquist__`, isolate all poles and zeros away from $w=0$ ($jw=0$ or $\exp(jwT) = 1$) and select the frequency range based on the breakpoint locations of the frequencies.
 2. if *sys* is discrete time, the frequency range is limited to jwT in $[0, 2p\pi]$
 3. A “smoothing” routine is used to ensure that the plot phase does not change excessively from point to point and that singular points (e.g., crossovers from ± 180) are accurately shown.
- atol* for interactive nyquist plots: *atol* is a change-in-slope tolerance for the of asymptotes (default = 0; 1e-2 is a good choice). This allows the user to “zoom in” on portions of the Nyquist plot too small to be seen with large asymptotes.

Outputs

- realp*
imagp the real and imaginary parts of the frequency response $G(jw)$ or $G(\exp(jwT))$ at the selected frequency values.
- w* the vector of frequency values used

If no output arguments are given, `nyquist` plots the results to the screen. If *atol* $\neq 0$ and asymptotes are detected then the user is asked interactively if they wish to zoom in (remove asymptotes) Descriptive labels are automatically placed.

Note: if the requested plot is for an MIMO system, a warning message is presented; the returned information is of the magnitude $\|G(jw)\|$ or $\|G(\exp(jwT))\|$ only; phase information is not computed.

`[mag, phase, w] = nichols(sys, w, outputs, inputs)` [Function File]
 Produce Nichols plot of a system.

Inputs

- sys* System data structure (must be either purely continuous or discrete; see `is_digital`).
- w* Frequency values for evaluation.
- if *sys* is continuous, then `nichols` evaluates $G(jw)$.
 - if *sys* is discrete, then `nichols` evaluates $G(\exp(jwT))$, where $T=\text{sys.tsam}$ is the system sampling time.
 - the default frequency range is selected as follows (These steps are **not** performed if *w* is specified):
 1. via routine `__bodquist__`, isolate all poles and zeros away from $w=0$ ($jw = 0$ or $\exp(jwT) = 1$) and select the frequency range based on the breakpoint locations of the frequencies.
 2. if *sys* is discrete time, the frequency range is limited to jwT in $[0, 2p\pi]$.

3. A “smoothing” routine is used to ensure that the plot phase does not change excessively from point to point and that singular points (e.g., crossovers from +/- 180) are accurately shown.

outputs

inputs the names or indices of the output(s) and input(s) to be used in the frequency response; see `sysprune`.

Outputs

mag

phase The magnitude and phase of the frequency response $G(jw)$ or $G(\exp(jwT))$ at the selected frequency values.

w The vector of frequency values used.

If no output arguments are given, `nichols` plots the results to the screen. Descriptive labels are automatically placed. See `xlabel`, `ylabel`, and `title`.

Note: if the requested plot is for an MIMO system, *mag* is set to $\|G(jw)\|$ or $\|G(\exp(jwT))\|$ and phase information is not computed.

`[zer, gain] = tzero (a, b, c, d, opt)` [Function File]

`[zer, gain] = tzero (sys, opt)` [Function File]

Compute transmission zeros of a continuous system:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

or of a discrete one:

$$x_{k+1} = Ax_k + Bu_k$$

$$y_k = Cx_k + Du_k$$

Outputs

zer transmission zeros of the system

gain leading coefficient (pole-zero form) of SISO transfer function returns gain=0 if system is multivariable

References

1. Emami-Naeini and Van Dooren, Automatica, 1982.
2. Hodel, *Computation of Zeros with Balancing*, 1992 Lin. Alg. Appl.

`zr = tzero2 (a, b, c, d, bal)` [Function File]

Compute the transmission zeros of *a*, *b*, *c*, *d*.

bal = balancing option (see `balance`); default is "B".

Needs to incorporate `mvzero` algorithm to isolate finite zeros; use `tzero` instead.

31.9 Controller Design

dgkfdemo ()

[Function File]

Octave Controls toolbox demo: $\mathcal{H}_2/\mathcal{H}_\infty$ options demos.

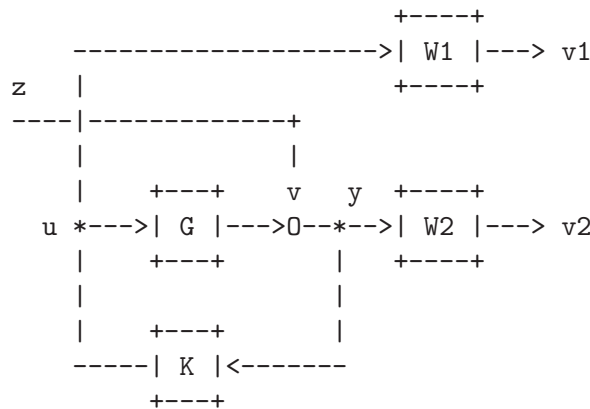
hinfdemo ()

[Function File]

\mathcal{H}_∞ design demos for continuous SISO and MIMO systems and a discrete system. The SISO system is difficult to control because it is non-minimum-phase and unstable. The second design example controls the **jet707** plant, the linearized state space model of a Boeing 707-321 aircraft at $v=80$ m/s ($M = 0.26$, $G_{a0} = -3^\circ$, $\alpha_0 = 4^\circ$, $\kappa = 50^\circ$). Inputs: (1) thrust and (2) elevator angle Outputs: (1) airspeed and (2) pitch angle. The discrete system is a stable and second order.

SISO plant:

$$G(s) = \frac{s - 2}{(s + 2)(s - 1)}$$

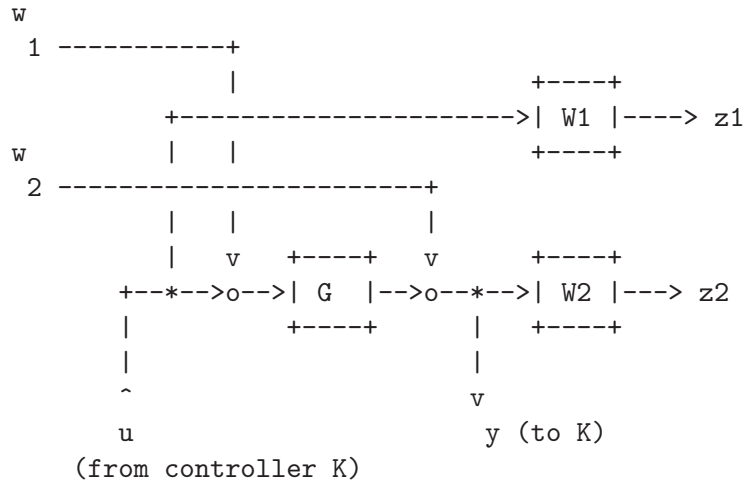


$$\min \|T_{vz}\|_\infty$$

$W1$ und $W2$ are the robustness and performance weighting functions.

MIMO plant:

The optimal controller minimizes the \mathcal{H}_∞ norm of the augmented plant P (mixed-sensitivity problem):



$$\begin{bmatrix} z_1 \\ z_2 \\ y \end{bmatrix} = P \begin{bmatrix} w_1 \\ w_2 \\ u \end{bmatrix}$$

Discrete system:

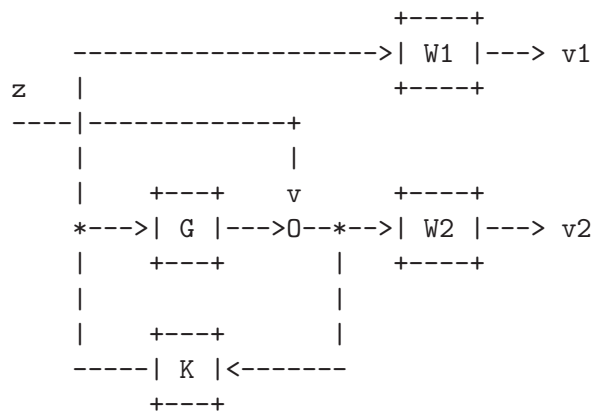
This is not a true discrete design. The design is carried out in continuous time while the effect of sampling is described by a bilinear transformation of the sampled system. This method works quite well if the sampling period is “small” compared to the plant time constants.

The continuous plant:

$$G(s) = \frac{1}{(s+2)(s+1)}$$

is discretised with a ZOH (Sampling period = $T_s = 1$ second):

$$G(z) = \frac{0.199788z + 0.073498}{(z - 0.36788)(z - 0.13534)}$$



$$\min \|T_{vz}\|_{\infty}$$

$W1$ and $W2$ are the robustness and performance weighting functions.

`[l, m, p, e] = dlqe (a, g, c, sigw, sigv, z)` [Function File]
Construct the linear quadratic estimator (Kalman filter) for the discrete time system

$$x_{k+1} = Ax_k + Bu_k + Gw_k$$

$$y_k = Cx_k + Du_k + v_k$$

where w , v are zero-mean gaussian noise processes with respective intensities $\text{sigw} = \text{cov}(w, w)$ and $\text{sigv} = \text{cov}(v, v)$.

If specified, z is $\text{cov}(w, v)$. Otherwise $\text{cov}(w, v) = 0$.

The observer structure is

$$z_{k|k} = z_{k|k-1} + l(y_k - Cz_{k|k-1} - Du_k)$$

$$z_{k+1|k} = Az_{k|k} + Bu_k$$

The following values are returned:

- l The observer gain, $(A - ALC)$. is stable.
- m The Riccati equation solution.
- p The estimate error covariance after the measurement update.
- e The closed loop poles of $(A - ALC)$.

`[k, p, e] = dlqr (a, b, q, r, z)` [Function File]
Construct the linear quadratic regulator for the discrete time system

$$x_{k+1} = Ax_k + Bu_k$$

to minimize the cost functional

$$J = \sum x^T Qx + u^T Ru$$

z omitted or

$$J = \sum x^T Qx + u^T Ru + 2x^T Zu$$

z included.

The following values are returned:

- k The state feedback gain, $(A - BK)$ is stable.
- p The solution of algebraic Riccati equation.
- e The closed loop poles of $(A - BK)$.

`[Lp, Lf, P, Z] = dkalman (A, G, C, Qw, Rv, S)` [Function File]
 Construct the linear quadratic estimator (Kalman predictor) for the discrete time system

$$x_{k+1} = Ax_k + Bu_k + Gw_k$$

$$y_k = Cx_k + Du_k + v_k$$

where w , v are zero-mean gaussian noise processes with respective intensities $Qw = \text{cov}(w, w)$ and $Rv = \text{cov}(v, v)$.

If specified, S is $\text{cov}(w, v)$. Otherwise $\text{cov}(w, v) = 0$.

The observer structure is $x_{k+1|k} = Ax_{k|k-1} + Bu_k + L_p(y_k - Cx_{k|k-1} - Du_k)$ $x_{k|k} = x_{k|k} + L_f(y_k - Cx_{k|k-1} - Du_k)$

The following values are returned:

Lp	The predictor gain, $(A - L_p C)$. is stable.
Lf	The filter gain.
P	The Riccati solution. $P = E\{(x - x_{n n-1})(x - x_{n n-1})'\}$
Z	The updated error covariance matrix. $Z = E\{(x - x_{n n})(x - x_{n n})'\}$

`[K, gain, kc, kf, pc, pf] = h2syn (asys, nu, ny, tol)` [Function File]
 Design \mathcal{H}_2 optimal controller per procedure in Doyle, Glover, Khargonekar, Francis, *State-Space Solutions to Standard \mathcal{H}_2 and \mathcal{H}_∞ Control Problems*, IEEE TAC August 1989.

Discrete-time control per Zhou, Doyle, and Glover, *Robust and optimal control*, Prentice-Hall, 1996.

Inputs

<code>asys</code>	system data structure (see <code>ss</code> , <code>sys2ss</code>) <ul style="list-style-type: none"> • controller is implemented for continuous time systems • controller is not implemented for discrete time systems
<code>nu</code>	number of controlled inputs
<code>ny</code>	number of measured outputs
<code>tol</code>	threshold for 0. Default: <code>200*eps</code>

Outputs

<code>k</code>	system controller
<code>gain</code>	optimal closed loop gain
<code>kc</code>	full information control (packed)
<code>kf</code>	state estimator (packed)
<code>pc</code>	ARE solution matrix for regulator subproblem
<code>pf</code>	ARE solution matrix for filter subproblem

$K = \text{hinf_ctr}(dgs, f, h, z, g)$ [Function File]

Called by `hinfsyn` to compute the \mathcal{H}_∞ optimal controller.

Inputs

dgs data structure returned by `is_dgkf`
 f
 h feedback and filter gain (not partitioned)
 g final gamma value

Outputs

K controller (system data structure)

Do not attempt to use this at home; no argument checking performed.

$[k, g, gw, xinf, yinf] = \text{hinfsyn}(asys, nu, ny, gmin, gmax, gtol, ptol, tol)$ [Function File]

Inputs input system is passed as either

$asys$ system data structure (see `ss`, `sys2ss`)

- controller is implemented for continuous time systems
- controller is **not** implemented for discrete time systems (see bilinear transforms in `c2d`, `d2c`)

nu number of controlled inputs
 ny number of measured outputs
 $gmin$ initial lower bound on \mathcal{H}_∞ optimal gain
 $gmax$ initial upper bound on \mathcal{H}_∞ Optimal gain.
 $gtol$ Gain threshold. Routine quits when $gmax/gmin < 1+tol$.
 $ptol$ poles with $\text{abs}(\text{real}(\text{pole})) < ptol\|H\|$ (H is appropriate Hamiltonian) are considered to be on the imaginary axis. Default: 1e-9.
 tol threshold for 0. Default: 200*eps.
 $gmax$, min , tol , and tol must all be postive scalars.

Outputs

k System controller.
 g Designed gain value.
 gw Closed loop system.
 $xinf$ ARE solution matrix for regulator subproblem.
 $yinf$ ARE solution matrix for filter subproblem.

References:

1. Doyle, Glover, Khargonekar, Francis, *State-Space Solutions to Standard \mathcal{H}_2 and \mathcal{H}_∞ Control Problems*, IEEE TAC August 1989.

2. Maciejowski, J.M., *Multivariable feedback design*, Addison-Wesley, 1989, ISBN 0-201-18243-2.
3. Keith Glover and John C. Doyle, *State-space formulae for all stabilizing controllers that satisfy an \mathcal{H}_∞ norm bound and relations to risk sensitivity*, Systems & Control Letters 11, Oct. 1988, pp 167–172.

`[retval, pc, pf] = hinfsyn_chk (a, b1, b2, c1, c2, d12, d21, g, ptol)` [Function File]

Called by `hinfsyn` to see if gain g satisfies conditions in Theorem 3 of Doyle, Glover, Khargonekar, Francis, *State Space Solutions to Standard \mathcal{H}_2 and \mathcal{H}_∞ Control Problems*, IEEE TAC August 1989.

Warning: do not attempt to use this at home; no argument checking performed.

Inputs

As returned by `is_dgkf`, except for:

g candidate gain level

$ptol$ as in `hinfsyn`

Outputs

$retval$ 1 if g exceeds optimal Hinf closed loop gain, else 0

pc solution of “regulator” \mathcal{H}_∞ ARE

pf solution of “filter” \mathcal{H}_∞ ARE

Do not attempt to use this at home; no argument checking performed.

`[xinf, x_ha_err] = hinfsyn_ric (a, bb, c1, d1dot, r, ptol)` [Function File]
Forms

```
xx = ([bb; -c1'*d1dot]/r) * [d1dot'*c1 bb'];
Ha = [a 0*a; -c1'*c1 - a'] - xx;
```

and solves associated Riccati equation. The error code `x_ha_err` indicates one of the following conditions:

- | | |
|---|--|
| 0 | successful |
| 1 | $xinf$ has imaginary eigenvalues |
| 2 | hx not Hamiltonian |
| 3 | $xinf$ has infinite eigenvalues (numerical overflow) |
| 4 | $xinf$ not symmetric |
| 5 | $xinf$ not positive definite |
| 6 | r is singular |

`[k, p, e] = lqe (a, g, c, sigw, sigv, z)` [Function File]

Construct the linear quadratic estimator (Kalman filter) for the continuous time system

$$\frac{dx}{dt} = Ax + Gu$$

$$y = Cx + v$$

where w and v are zero-mean gaussian noise processes with respective intensities

$$\text{sigw} = \text{cov}(w, w)$$

$$\text{sigv} = \text{cov}(v, v)$$

The optional argument z is the cross-covariance $\text{cov}(w, v)$. If it is omitted, $\text{cov}(w, v) = 0$ is assumed.

Observer structure is $dz/dt = A z + B u + k (y - C z - D u)$

The following values are returned:

k The observer gain, $(A - KC)$ is stable.
 p The solution of algebraic Riccati equation.
 e The vector of closed loop poles of $(A - KC)$.

`[k, q1, p1, ee, er] = lqg (sys, sigw, sigv, q, r, in_idx)` [Function File]

Design a linear-quadratic-gaussian optimal controller for the system

$$\begin{aligned} dx/dt &= A x + B u + G w & [w] &= N(0, [\text{Sigw} \ 0 \]) \\ y &= C x + v & [v] &= (\ 0 \ \text{Sigv} \) \end{aligned}$$

or

$$\begin{aligned} x(k+1) &= A x(k) + B u(k) + G w(k) & [w] &= N(0, [\text{Sigw} \ 0 \]) \\ y(k) &= C x(k) + v(k) & [v] &= (\ 0 \ \text{Sigv} \) \end{aligned}$$

Inputs

sys system data structure
 $sigw$
 $sigv$ intensities of independent Gaussian noise processes (as above)
 q
 r state, control weighting respectively. Control ARE is
 in_idx names or indices of controlled inputs (see `sysidx`, `cellidx`)
 default: last $\text{dim}(R)$ inputs are assumed to be controlled inputs, all others
 are assumed to be noise inputs.

Outputs

k system data structure format LQG optimal controller (Obtain A , B , C matrices with `sys2ss`, `sys2tf`, or `sys2zp` as appropriate).
 $p1$ Solution of control (state feedback) algebraic Riccati equation.
 $q1$ Solution of estimation algebraic Riccati equation.
 ee Estimator poles.

`es` Controller poles.

See also: `h2syn`, `lqe`, `lqr`.

`[k, p, e] = lqr (a, b, q, r, z)` [Function File]
construct the linear quadratic regulator for the continuous time system

$$\frac{dx}{dt} = Ax + Bu$$

to minimize the cost functional

$$J = \int_0^\infty x^T Q x + u^T R u$$

`z` omitted or

$$J = \int_0^\infty x^T Q x + u^T R u + 2x^T Z u$$

`z` included.

The following values are returned:

- `k` The state feedback gain, $(A - BK)$ is stable and minimizes the cost functional
- `p` The stabilizing solution of appropriate algebraic Riccati equation.
- `e` The vector of the closed loop poles of $(A - BK)$.

Reference Anderson and Moore, *Optimal control: linear quadratic methods*, Prentice-Hall, 1990, pp. 56–58.

`[y, x] = lsim (sys, u, t, x0)` [Function File]
Produce output for a linear simulation of a system; produces a plot for the output of the system, `sys`.

`u` is an array that contains the system's inputs. Each row in `u` corresponds to a different time step. Each column in `u` corresponds to a different input. `t` is an array that contains the time index of the system; `t` should be regularly spaced. If initial conditions are required on the system, the `x0` vector should be added to the argument list.

When the `lsim` function is invoked a plot is not displayed; however, the data is returned in `y` (system output) and `x` (system states).

`K = place (sys, p)` [Function File]

Computes the matrix `K` such that if the state is feedback with gain `K`, then the eigenvalues of the closed loop system (i.e. $A - BK$) are those specified in the vector `p`.

Version: Beta (May-1997): If you have any comments, please let me know. (see the file `place.m` for my address)

31.10 Miscellaneous Functions (Not yet properly filed/documentated)

axis2dlim (*axdata*) [Function File]

Determine axis limits for 2-D data (column vectors); leaves a 10% margin around the plots. Inserts margins of +/- 0.1 if data is one-dimensional (or a single point).

Input

axdata *n* by 2 matrix of data [*x*, *y*].

Output

axvec Vector of axis limits appropriate for call to **axis** function.

moddemo (*inputs*) [Function File]

Octave Control toolbox demo: Model Manipulations demo.

prompt (*str*) [Function File]

Prompt user to continue

Input

str Input string. Its default value is:
 \n ---- Press a key to continue ---

rldemo (*inputs*) [Function File]

Octave Control toolbox demo: Root Locus demo.

[rldata, k] = rlocus (*sys*[, *increment*, *min_k*, *max_k*]) [Function File]

Display root locus plot of the specified SISO system.

```

      -----
---->| + |---|k|---->| SISO |----->
      -----
      - ^
      |-----|

```

Inputs

sys system data structure

min_k Minimum value of *k*

max_k Maximum value of *k*

increment The increment used in computing gain values

Outputs

Plots the root locus to the screen.

rldata Data points plotted: in column 1 real values, in column 2 the imaginary values.

k Gains for real axis break points.

`[yy, idx] = sortcom (xx[, opt])` [Function File]
 Sort a complex vector.

Inputs

`xx` Complex vector

`opt` sorting option:

`"re"` Real part (default);

`"mag"` By magnitude;

`"im"` By imaginary part.

if `opt` is not chosen as `"im"`, then complex conjugate pairs are grouped together, $a - jb$ followed by $a + jb$.

Outputs

`yy` Sorted values

`idx` Permutation vector: `yy = xx(idx)`

`[num, den] = ss2tf (a, b, c, d)` [Function File]
 Conversion from tranfer function to state-space. The state space system:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

is converted to a transfer function:

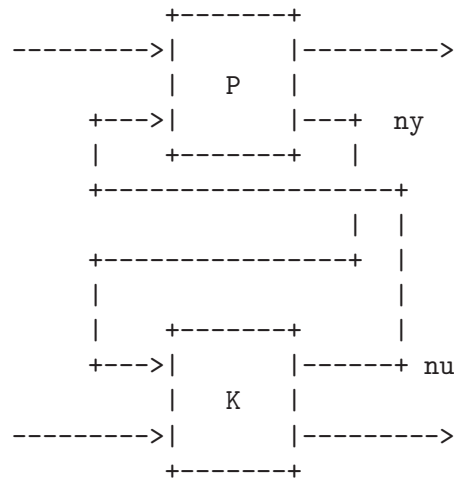
$$G(s) = \frac{\text{num}(s)}{\text{den}(s)}$$

used internally in system data structure format manipulations.

`[pol, zer, k] = ss2zp (a, b, c, d)` [Function File]
 Converts a state space representation to a set of poles and zeros; k is a gain associated with the zeros.

Used internally in system data structure format manipulations.

`starp (P, K, ny, nu)` [Function File]
 Redheffer star product or upper/lower LFT, respectively.



ny and/or nu may be negative (i.e. negative feedback).

Conversion from tranfer function to state-space. The state space system:

$$y = Cx + Du$$

is obtained from a transfer function:

$$G(s) = \frac{\text{num}(s)}{\text{den}(s)}$$

The vector *den* must contain only one row, whereas the vector *num* may contain as many rows as there are outputs *y* of the system. The state space system matrices obtained from this function will be in controllable canonical form as described in *Modern Control Theory*, (Brogan, 1991).

Converts transfer functions to poles-and-zero representations.

Returns the zeros and poles of the SISO system defined by *num/den*. *k* is a gain associated with the system zeros.

Conversion from zero / pole to state space.

Inputs

zer
pol Vectors of (possibly) complex poles and zeros of a transfer function. Complex values must come in conjugate pairs (i.e., $x + jy$ in *zer* means that $x - jy$ is also in *zer*). The number of zeros must not exceed the number of poles.

k Real scalar (leading coefficient).

Outputs

a
b
c
d The state space system, in the form:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

`[num, den] = zp2tf (zer, pol, k)` [Function File]
 Converts zeros / poles to a transfer function.

Inputs

zer
pol Vectors of (possibly complex) poles and zeros of a transfer function. Complex values must appear in conjugate pairs.

k Real scalar (leading coefficient).

32 Signal Processing

detrend (*x*, *p*) [Function File]

If *x* is a vector, **detrend** (*x*, *p*) removes the best fit of a polynomial of order *p* from the data *x*.

If *x* is a matrix, **detrend** (*x*, *p*) does the same for each column in *x*.

The second argument is optional. If it is not specified, a value of 1 is assumed. This corresponds to removing a linear trend.

fft (*a*, *n*, *dim*) [Loadable Function]

Compute the FFT of *a* using subroutines from FFTW. The FFT is calculated along the first non-singleton dimension of the array. Thus if *a* is a matrix, **fft** (*a*) computes the FFT for each column of *a*.

If called with two arguments, *n* is expected to be an integer specifying the number of elements of *a* to use, or an empty matrix to specify that its value should be ignored. If *n* is larger than the dimension along which the FFT is calculated, then *a* is resized and padded with zeros. Otherwise, if *n* is smaller than the dimension along which the FFT is calculated, then *a* is truncated.

If called with three arguments, *dim* is an integer specifying the dimension of the matrix along which the FFT is performed

See also: `ifft`, `fft2`, `fftn`, `fftw-wisdom`.

method = **fftw** ('*planner*') [Loadable Function]

fftw ('*planner*', *method*) [Loadable Function]

wisdom = **fftw** ('*dwisdom*') [Loadable Function]

wisdom = **fftw** ('*dwisdom*', *wisdom*) [Loadable Function]

Manage FFTW wisdom data. Wisdom data can be used to significantly accelerate the calculation of the FFTs but implies a initial cost in its calculation. The wisdom used by Octave can be imported directly, usually from a file `/etc/fftw/wisdom`, or *fftw* can be used to import wisdom. For example

```
wisdom = fftw ('dwisdom')
```

will save the existing wisdom used by Octave to the string *wisdom*. This string can then be saved in the usual manner. This existing wisdom can be reimported as follows

```
fftw ('dwisdom', wisdom)
```

If *wisdom* is an empty matrix, then the wisdom used is cleared.

During the calculation of fourier transforms further wisdom is generated. The fashion in which this wisdom is generated is equally controlled by the *fftw* function. There are five different manners in which the wisdom can be treated, these being

'estimate' This specifies that no run-time measurement of the optimal means of calculating a particular is performed, and a simple heuristic is used to pick a (probably sub-optimal) plan. The advantage of this method is that there is little or no overhead in the generation of the plan, which is appropriate for a fourier transform that will be calculated once.

- 'measure' In this case a range of algorithms to perform the transform is considered and the best is selected based on their execution time.
- 'patient' This is like 'measure', but a wider range of algorithms is considered.
- 'exhasutive' This is like 'meaure', but all possible algorithms that may be used to treat the transform are considered.
- 'hybrid' As run-time measurement of the algorithm can be expensive, this is a compromise where 'measure' is used for transforms upto the size of 8192 and beyond that the 'estimate' method is used.

The default method is 'estimate', and the method currently being used can be probed with

```
method = fftw ('planner')
```

and the method used can be set using

```
fftw ('planner', method)
```

Note that calculated wisdom will be lost when restarting Octave. However, the wisdom data can be reloaded if it is saved to a file as described above. Also, any system-wide wisdom file that has been found will also be used. Saved wisdom files should not be used on different platforms since they will not be efficient and the point of calculating the wisdom is lost.

See also: `fft`, `ifft`, `fft2`, `ifft2`, `fftn`, `ifftn`.

`ifft (a, n, dim)` [Loadable Function]

Compute the inverse FFT of *a* using subroutines from FFTW. The inverse FFT is calculated along the first non-singleton dimension of the array. Thus if *a* is a matrix, `ifft (a)` computes the inverse FFT for each column of *a*.

If called with two arguments, *n* is expected to be an integer specifying the number of elements of *a* to use, or an empty matrix to specify that its value should be ignored. If *n* is larger than the dimension along which the inverse FFT is calculated, then *a* is resized and padded with zeros. Otherwise, if *n* is smaller than the dimension along which the inverse FFT is calculated, then *a* is truncated.

If called with three arguments, *dim* is an integer specifying the dimension of the matrix along which the inverse FFT is performed

See also: `fft`, `ifft2`, `ifftn`, `fftw_wisdom`.

`fft2 (a, n, m)` [Loadable Function]

Compute the two dimensional FFT of *a* using subroutines from FFTW. The optional arguments *n* and *m* may be used specify the number of rows and columns of *a* to use. If either of these is larger than the size of *a*, *a* is resized and padded with zeros.

If *a* is a multi-dimensional matrix, each two-dimensional sub-matrix of *a* is treated seperately

See also: `ifft2`, `fft`, `fftn`, `fftw_wisdom`.

fft2 (*a*, *n*, *m*) [Loadable Function]

Compute the inverse two dimensional FFT of *a* using subroutines from FFTW. The optional arguments *n* and *m* may be used specify the number of rows and columns of *a* to use. If either of these is larger than the size of *a*, *a* is resized and padded with zeros.

If *a* is a multi-dimensional matrix, each two-dimensional sub-matrix of *a* is treated separately

See also: fft2, ifft, ifftn, fftw_wisdom.

fftn (*a*, *size*) [Loadable Function]

Compute the N dimensional FFT of *a* using subroutines from FFTW. The optional vector argument *size* may be used specify the dimensions of the array to be used. If an element of *size* is smaller than the corresponding dimension, then the dimension is truncated prior to performing the FFT. Otherwise if an element of *size* is larger than the corresponding dimension *a* is resized and padded with zeros.

See also: ifftn, fft, fft2, fftw_wisdom.

ifftn (*a*, *size*) [Loadable Function]

Compute the inverse N dimensional FFT of *a* using subroutines from FFTW. The optional vector argument *size* may be used specify the dimensions of the array to be used. If an element of *size* is smaller than the corresponding dimension, then the dimension is truncated prior to performing the inverse FFT. Otherwise if an element of *size* is larger than the corresponding dimension *a* is resized and padded with zeros.

See also: fftn, ifft, ifft2, fftw_wisdom.

fftconv (*a*, *b*, *n*) [Function File]

Return the convolution of the vectors *a* and *b*, as a vector with length equal to the `length(a) + length(b) - 1`. If *a* and *b* are the coefficient vectors of two polynomials, the returned value is the coefficient vector of the product polynomial.

The computation uses the FFT by calling the function `fftfilt`. If the optional argument *n* is specified, an N-point FFT is used.

fftfilt (*b*, *x*, *n*) [Function File]

With two arguments, `fftfilt` filters *x* with the FIR filter *b* using the FFT.

Given the optional third argument, *n*, `fftfilt` uses the overlap-add method to filter *x* with *b* using an N-point FFT.

If *x* is a matrix, filter each column of the matrix.

y = **filter** (*b*, *a*, *x*) [Loadable Function]

[*y*, *sf*] = **filter** (*b*, *a*, *x*, *si*) [Loadable Function]

[*y*, *sf*] = **filter** (*b*, *a*, *x*, [], *dim*) [Loadable Function]

[*y*, *sf*] = **filter** (*b*, *a*, *x*, *si*, *dim*) [Loadable Function]

Return the solution to the following linear, time-invariant difference equation:

$$\sum_{k=0}^N a_{k+1} y_{n-k} = \sum_{k=0}^M b_{k+1} x_{n-k}, \quad 1 \leq n \leq P$$

where $a \in \Re^{N-1}$, $b \in \Re^{M-1}$, and $x \in \Re^P$. over the first non-singleton dimension of x or over *dim* if supplied. An equivalent form of this equation is:

$$y_n = - \sum_{k=1}^N c_{k+1} y_{n-k} + \sum_{k=0}^M d_{k+1} x_{n-k}, \quad 1 \leq n \leq P$$

where $c = a/a_1$ and $d = b/a_1$.

If the fourth argument *si* is provided, it is taken as the initial state of the system and the final state is returned as *sf*. The state vector is a column vector whose length is equal to the length of the longest coefficient vector minus one. If *si* is not supplied, the initial state vector is set to all zeros.

In terms of the z-transform, y is the result of passing the discrete- time signal x through a system characterized by the following rational system function:

$$H(z) = \frac{\sum_{k=0}^M d_{k+1} z^{-k}}{1 + \sum_{k=1}^N c_{k+1} z^{-k}}$$

`y = filter2 (b, x)` [Function File]
`y = filter2 (b, x, shape)` [Function File]

Apply the 2-D FIR filter b to x . If the argument *shape* is specified, return an array of the desired shape. Possible values are:

'full' pad x with zeros on all sides before filtering.
 'same' unpadded x (default)
 'valid' trim x after filtering so edge effects are no included.

Note this is just a variation on convolution, with the parameters reversed and b rotated 180 degrees.

See also: conv2.

`[h, w] = freqz (b, a, n, "whole")` [Function File]

Return the complex frequency response h of the rational IIR filter whose numerator and denominator coefficients are b and a , respectively. The response is evaluated at n angular frequencies between 0 and 2π .

The output value w is a vector of the frequencies.

If the fourth argument is omitted, the response is evaluated at frequencies between 0 and π .

If n is omitted, a value of 512 is assumed.

If a is omitted, the denominator is assumed to be 1 (this corresponds to a simple FIR filter).

For fastest computation, n should factor into a small number of small primes.

`h = freqz (b, a, w)` [Function File]
 Evaluate the response at the specific frequencies in the vector `w`. The values for `w` are measured in radians.

`[...] = freqz (... , Fs)` [Function File]
 Return frequencies in Hz instead of radians assuming a sampling rate `Fs`. If you are evaluating the response at specific frequencies `w`, those frequencies should be requested in Hz rather than radians.

`freqz (...)` [Function File]
 Plot the pass band, stop band and phase response of `h` rather than returning them.

`freqz_plot (w, h)` [Function File]
 Plot the pass band, stop band and phase response of `h`.

`sinc (x)` [Function File]
 Return $\sin(\pi x)/(\pi x)$.

`b = unwrap (a, tol, dim)` [Function File]
 Unwrap radian phases by adding multiples of 2π as appropriate to remove jumps greater than `tol`. `tol` defaults to π .

Unwrap will unwrap along the first non-singleton dimension of `a`, unless the optional argument `dim` is given, in which case the data will be unwrapped along this dimension

`[a, b] = arch_fit (y, x, p, iter, gamma, a0, b0)` [Function File]
 Fit an ARCH regression model to the time series `y` using the scoring algorithm in Engle's original ARCH paper. The model is

$$\begin{aligned} y(t) &= b(1) * x(t,1) + \dots + b(k) * x(t,k) + e(t), \\ h(t) &= a(1) + a(2) * e(t-1)^2 + \dots + a(p+1) * e(t-p)^2 \end{aligned}$$

in which $e(t)$ is $N(0, h(t))$, given a time-series vector `y` up to time $t - 1$ and a matrix of (ordinary) regressors `x` up to t . The order of the regression of the residual variance is specified by `p`.

If invoked as `arch_fit (y, k, p)` with a positive integer `k`, fit an $ARCH(k, p)$ process, i.e., do the above with the t -th row of `x` given by

$$[1, y(t-1), \dots, y(t-k)]$$

Optionally, one can specify the number of iterations `iter`, the updating factor `gamma`, and initial values `a0` and `b0` for the scoring algorithm.

`arch_rnd (a, b, t)` [Function File]
 Simulate an ARCH sequence of length `t` with AR coefficients `b` and CH coefficients `a`. I.e., the result `y(t)` follows the model

$$y(t) = b(1) + b(2) * y(t-1) + \dots + b(lb) * y(t-lb+1) + e(t),$$

where $e(t)$, given `y` up to time $t - 1$, is $N(0, h(t))$, with

$$h(t) = a(1) + a(2) * e(t-1)^2 + \dots + a(la) * e(t-la+1)^2$$

`[pval, lm] = arch_test (y, x, p)` [Function File]

For a linear regression model

$$y = x * b + e$$

perform a Lagrange Multiplier (LM) test of the null hypothesis of no conditional heteroscedascity against the alternative of $CH(p)$.

I.e., the model is

$$y(t) = b(1) * x(t,1) + \dots + b(k) * x(t,k) + e(t),$$

given y up to $t-1$ and x up to t , $e(t)$ is $N(0, h(t))$ with

$$h(t) = v + a(1) * e(t-1)^2 + \dots + a(p) * e(t-p)^2,$$

and the null is $a(1) == \dots == a(p) == 0$.

If the second argument is a scalar integer, k , perform the same test in a linear autoregression model of order k , i.e., with

$$[1, y(t-1), \dots, y(t-k)]$$

as the t -th row of x .

Under the null, LM approximately has a chisquare distribution with p degrees of freedom and $pval$ is the p -value (1 minus the CDF of this distribution at LM) of the test.

If no output argument is given, the p -value is displayed.

`arma_rnd (a, b, v, t, n)` [Function File]

Return a simulation of the ARMA model

$$\begin{aligned} x(n) = & a(1) * x(n-1) + \dots + a(k) * x(n-k) \\ & + e(n) + b(1) * e(n-1) + \dots + b(l) * e(n-l) \end{aligned}$$

in which k is the length of vector a , l is the length of vector b and e is gaussian white noise with variance v . The function returns a vector of length t .

The optional parameter n gives the number of dummy $x(i)$ used for initialization, i.e., a sequence of length $t+n$ is generated and $x(n+1:t+n)$ is returned. If n is omitted, $n = 100$ is used.

`autocor (x, h)` [Function File]

Return the autocorrelations from lag 0 to h of vector x . If h is omitted, all autocorrelations are computed. If x is a matrix, the autocorrelations of each column are computed.

`autocov (x, h)` [Function File]

Return the autocovariances from lag 0 to h of vector x . If h is omitted, all autocovariances are computed. If x is a matrix, the autocovariances of each column are computed.

`autoreg_matrix (y, k)` [Function File]

Given a time series (vector) y , return a matrix with ones in the first column and the first k lagged values of y in the other columns. I.e., for $t > k$, $[1, y(t-1), \dots, y(t-k)]$ is the t -th row of the result. The resulting matrix may be used as a regressor matrix in autoregressions.

bartlett (*m*) [Function File]

Return the filter coefficients of a Bartlett (triangular) window of length *m*.

For a definition of the Bartlett window, see e.g. A. V. Oppenheim & R. W. Schaffer, "Discrete-Time Signal Processing".

blackman (*m*) [Function File]

Return the filter coefficients of a Blackman window of length *m*.

For a definition of the Blackman window, see e.g. A. V. Oppenheim & R. W. Schaffer, "Discrete-Time Signal Processing".

[d, dd] = diffpara (*x, a, b*) [Function File]

Return the estimator *d* for the differencing parameter of an integrated time series.

The frequencies from $[2 * \pi * a/t, 2 * \pi * b/T]$ are used for the estimation. If *b* is omitted, the interval $[2 * \pi/T, 2 * \pi * a/T]$ is used. If both *b* and *a* are omitted then $a = 0.5 * \text{sqrt}(T)$ and $b = 1.5 * \text{sqrt}(T)$ is used, where *T* is the sample size. If *x* is a matrix, the differencing parameter of each column is estimated.

The estimators for all frequencies in the intervals described above is returned in *dd*. The value of *d* is simply the mean of *dd*.

Reference: Brockwell, Peter J. & Davis, Richard A. Time Series: Theory and Methods Springer 1987.

durbinlevinson (*c, oldphi, oldv*) [Function File]

Perform one step of the Durbin-Levinson algorithm.

The vector *c* specifies the autocovariances [*gamma_0*, ..., *gamma_t*] from lag 0 to *t*, *oldphi* specifies the coefficients based on *c*(*t*-1) and *oldv* specifies the corresponding error.

If *oldphi* and *oldv* are omitted, all steps from 1 to *t* of the algorithm are performed.

fftshift (*v*) [Function File]

fftshift (*v, dim*) [Function File]

Perform a shift of the vector *v*, for use with the **fft** and **ifft** functions, in order the move the frequency 0 to the center of the vector or matrix.

If *v* is a vector of *N* elements corresponding to *N* time samples spaced of *Dt* each, then **fftshift** (**fft** (*v*)) corresponds to frequencies

$$f = ((1:N) - \text{ceil}(N/2)) / N / Dt$$

If *v* is a matrix, the same holds for rows and columns. If *v* is an array, then the same holds along each dimension.

The optional *dim* argument can be used to limit the dimension along which the permutation occurs.

ifftshift (*v*) [Function File]

ifftshift (*v, dim*) [Function File]

Undo the action of the **fftshift** function. For even length *v*, **fftshift** is its own inverse, but odd lengths differ slightly.

fractdiff (*x*, *d*) [Function File]
 Compute the fractional differences $(1 - L)^d x$ where L denotes the lag-operator and d is greater than -1.

hamming (*m*) [Function File]
 Return the filter coefficients of a Hamming window of length m .
 For a definition of the Hamming window, see e.g. A. V. Oppenheim & R. W. Schaffer, "Discrete-Time Signal Processing".

hanning (*m*) [Function File]
 Return the filter coefficients of a Hanning window of length m .
 For a definition of this window type, see e.g. A. V. Oppenheim & R. W. Schaffer, "Discrete-Time Signal Processing".

hurst (*x*) [Function File]
 Estimate the Hurst parameter of sample x via the rescaled range statistic. If x is a matrix, the parameter is estimated for every single column.

pp = **pchip** (*x*, *y*) [Function File]
yi = **pchip** (*x*, *y*, *xi*) [Function File]
 Piecewise Cubic Hermite interpolating polynomial. Called with two arguments, the piece-wise polynomial *pp* is returned, that may later be used with **ppval** to evaluate the polynomial at specific points.
 The variable x must be a strictly monotonic vector (either increasing or decreasing). While y can be either a vector or array. In the case where y is a vector, it must have a length of n . If y is an array, then the size of y must have the form

$$[s_1, s_2, \dots, s_k, n]$$

The array is then reshaped internally to a matrix where the leading dimension is given by

$$s_1 s_2 \cdots s_k$$

and each row this matrix is then treated separately. Note that this is exactly the opposite treatment than **interp1** and is done for compatibility.

Called with a third input argument, **pchip** evaluates the piece-wise polynomial at the points xi . There is an equivalence between **ppval** (**pchip** (x , y), xi) and **pchip** (x , y , xi).

See also: **spline**, **ppval**, **mkpp**, **unmkpp**.

periodogram (*x*) [Function File]
 For a data matrix x from a sample of size n , return the periodogram.

rectangle_lw (*n*, *b*) [Function File]
 Rectangular lag window. Subfunction used for spectral density estimation.

rectangle_sw (*n*, *b*) [Function File]
 Rectangular spectral window. Subfunction used for spectral density estimation.

sinetone (*freq*, *rate*, *sec*, *ampl*) [Function File]
 Return a sinetone of frequency *freq* with length of *sec* seconds at sampling rate *rate* and with amplitude *ampl*. The arguments *freq* and *ampl* may be vectors of common size.
 Defaults are *rate* = 8000, *sec* = 1 and *ampl* = 64.

sinewave (*m*, *n*, *d*) [Function File]
 Return an *m*-element vector with *i*-th element given by $\sin(2 * \pi * (i+d-1) / n)$.
 The default value for *d* is 0 and the default value for *n* is *m*.

spectral_adf (*c*, *win*, *b*) [Function File]
 Return the spectral density estimator given a vector of autocovariances *c*, window name *win*, and bandwidth, *b*.
 The window name, e.g., "triangle" or "rectangle" is used to search for a function called *win_sw*.
 If *win* is omitted, the triangle window is used. If *b* is omitted, $1 / \sqrt{\text{length}(x)}$ is used.

spectral_xdf (*x*, *win*, *b*) [Function File]
 Return the spectral density estimator given a data vector *x*, window name *win*, and bandwidth, *b*.
 The window name, e.g., "triangle" or "rectangle" is used to search for a function called *win_sw*.
 If *win* is omitted, the triangle window is used. If *b* is omitted, $1 / \sqrt{\text{length}(x)}$ is used.

spencer (*x*) [Function File]
 Return Spencer's 15 point moving average of every single column of *x*.

[*y*, *c*] = **stft** (*x*, *win_size*, *inc*, *num_coef*, *w_type*) [Function File]
 Compute the short-term Fourier transform of the vector *x* with *num_coef* coefficients by applying a window of *win_size* data points and an increment of *inc* points.
 Before computing the Fourier transform, one of the following windows is applied:

hanning *w_type* = 1

hamming *w_type* = 2

rectangle *w_type* = 3

The window names can be passed as strings or by the *w_type* number.

If not all arguments are specified, the following defaults are used: *win_size* = 80, *inc* = 24, *num_coef* = 64, and *w_type* = 1.

`y = stft (x, ...)` returns the absolute values of the Fourier coefficients according to the *num_coef* positive frequencies.

`[y, c] = stft (x, ...)` returns the entire STFT-matrix *y* and a 3-element vector *c* containing the window size, increment, and window type, which is needed by the synthesis function.

synthesis (*y*, *c*) [Function File]

Compute a signal from its short-time Fourier transform *y* and a 3-element vector *c* specifying window size, increment, and window type.

The values *y* and *c* can be derived by

`[y, c] = stft (x , ...)`

triangle_lw (*n*, *b*) [Function File]

Triangular lag window. Subfunction used for spectral density estimation.

triangle_sw (*n*, *b*) [Function File]

Triangular spectral window. Subfunction used for spectral density estimation.

[a, v] = yulewalker (*c*) [Function File]

Fit an AR (p)-model with Yule-Walker estimates given a vector *c* of autocovariances [*gamma_0*, ..., *gamma_p*].

Returns the AR coefficients, *a*, and the variance of white noise, *v*.

33 Image Processing

Since an image basically is a matrix Octave is a very powerful environment for processing and analysing images. To illustrate how easy it is to do image processing in Octave, the following example will load an image, smooth it by a 5-by-5 averaging filter, and compute the gradient of the smoothed image.

```
I = loadimage ("default.img");
S = conv2 (I, ones (5, 5) / 25, "same");
[Dx, Dy] = gradient (S);
```

In this example *S* contains the smoothed image, and *Dx* and *Dy* contains the partial spatial derivatives of the image.

33.1 Loading and Saving Images

The first step in most image processing tasks is to load an image into Octave. Currently Octave only support saving images in the Portable Pixmap Format (PPM), PostScript, and Octave's own format, and loading images in Octave's format. Most image processing code will follow the structure of this code

```
I = loadimage ("my_input_image.img");
J = process_my_image (I);
saveimage ("my_output_image.img", J);
```

`[x, map] = loadimage (file)` [Function File]
Load an image file and it's associated color map from the specified *file*. The image must be stored in Octave's image format.

See also: saveimage, load, save.

`saveimage (file, x, fmt, map)` [Function File]
Save the matrix *x* to *file* in image format *fmt*. Valid values for *fmt* are

"img"	Octave's image format. The current colormap is also saved in the file.
"ppm"	Portable pixmap format.
"ps"	PostScript format. Note that images saved in PostScript format can not be read back into Octave with loadimage.

If the fourth argument is supplied, the specified colormap will also be saved along with the image.

Note: if the colormap contains only two entries and these entries are black and white, the bitmap ppm and PostScript formats are used. If the image is a gray scale image (the entries within each row of the colormap are equal) the gray scale ppm and PostScript image formats are used, otherwise the full color formats are used.

See also: loadimage, save, load, colormap.

`val = IMAGE_PATH ()` [Built-in Function]
`old_val = IMAGE_PATH (new_val)` [Built-in Function]

Query or set the internal variable that specifies a colon separated list of directories in which to search for image files.

33.2 Displaying Images

A natural part of image processing is visualization of an image. The most basic function for this is the `imshow` function that shows the image given in the first input argument. This function uses an external program to show the image. If gnuplot 4.2 or later is available it will be used to display the image, otherwise the `display`, `xv`, or `xloadimage` program is used. The actual program can be selected with the `image_viewer` function.

<code>imshow (im)</code>	[Function File]
<code>imshow (im, limits)</code>	[Function File]
<code>imshow (im, map)</code>	[Function File]
<code>imshow (R, G, B, ...)</code>	[Function File]
<code>imshow (filename)</code>	[Function File]
<code>imshow (... , string_param1, value1, ...)</code>	[Function File]

Display the image *im*, where *im* can be a 2-dimensional (gray-scale image) or a 3-dimensional (RGB image) matrix. If three matrices of the same size are given as arguments, they will be concatenated into a 3-dimensional (RGB image) matrix.

If *limits* is a 2-element vector [*low*, *high*], the image is shown using a display range between *low* and *high*. If an empty matrix is passed for *limits*, the display range is computed as the range between the minimal and the maximal value in the image.

If *map* is a valid color map, the image will be shown as an indexed image using the supplied color map.

If a file name is given instead of an image, the file will be read and shown.

If given, the parameter *string_param1* has value *value1*. *string_param1* can be any of the following:

`"displayrange"`

value1 is the display range as described above.

See also: `image`, `imagesc`, `colormap`, `gray2ind`, `rgb2ind`.

<code>image (img)</code>	[Function File]
<code>image (x, y, img)</code>	[Function File]

Display a matrix as a color image. The elements of *x* are indices into the current colormap and should have values between 1 and the length of the colormap.

It first tries to use `gnuplot`, then `display` from `ImageMagick`, then `xv`, and then `xloadimage`. The actual program used can be changed using the `image_viewer` function.

The axis values corresponding to the matrix elements are specified in *x* and *y*. If you're not using gnuplot 4.2 or later, these variables are ignored.

See also: `imshow`, `imagesc`, `colormap`, `image_viewer`.

<code>imagesc (A)</code>	[Function File]
<code>imagesc (x, y, A)</code>	[Function File]
<code>imagesc (... , limits)</code>	[Function File]
<code>B = imagesc (...)</code>	[Function File]

Display a scaled version of the matrix *A* as a color image. The matrix is scaled so that its entries are indices into the current colormap. The scaled matrix is returned.

If $limits = [lo, hi]$ are given, then that range maps into the full range of the colormap rather than the minimum and maximum values of A .

The axis values corresponding to the matrix elements are specified in x and y , either as pairs giving the minimum and maximum values for the respective axes, or as values for each row and column of the matrix A .

See also: `image`, `imshow`.

`[fcn, default_zoom] = image_viewer (fcn, default_zoom)` [Function File]

Change the program or function used for viewing images and return the previous values.

When the `image` or `imshow` function is called it will launch an external program to display the image. The default behaviour is to use `gnuplot` if the installed version supports image viewing, and otherwise try the programs `display`, `xv`, and `xloadimage`. Using this function it is possible to change that behaviour.

When called with one input argument images will be displayed by saving the image to a file and the system command `command` will be called to view the image. The `command` must be a string containing `%s` and possibly `%f`. The `%s` will be replaced by the filename of the image, and the `%f` will (if present) be replaced by the zoom factor given to the `image` function. For example,

```
image_viewer ("eog %s");
```

changes the image viewer to the `eog` program.

With two input arguments, images will be displayed by calling the function `function_handle`. For example,

```
image_viewer (data, @my_image_viewer);
```

sets the image viewer function to `my_image_viewer`. The image viewer function is called with

```
my_image_viewer (x, y, im, zoom, data)
```

where x and y are the axis of the image, im is the image variable, and $data$ is extra user-supplied data to be passed to the viewer function.

With three input arguments it is possible to change the zooming. Some programs (like `xloadimage`) require the zoom factor to be between 0 and 100, and not 0 and 1 like Octave assumes. This is solved by setting the third argument to 100.

See also: `image`, `imshow`, `__img_gnuplot__`, `__img_via_file__`.

33.3 Representing Images

In general Octave supports four different kinds of images, gray-scale images, RGB images, binary images, and indexed images. A gray-scale image is represented with an M -by- N matrix in which each element corresponds to the intensity of a pixel. An RGB image is represented with an M -by- N -by-3 array where each 3-vector corresponds to the red, green, and blue intensities of each pixel.

The actual meaning of the value of a pixel in a gray-scale or RGB image depends on the class of the matrix. If the matrix is of class `double` pixel intensities are between 0 and 1, if

it is of class `uint8` intensities are between 0 and 255, and if it is of class `uint16` intensities are between 0 and 65535.

A binary image is a M-by-N matrix of class `logical`. A pixel in a binary image is black if it is `false` and white if it is `true`.

An indexed image consists of an M-by-N matrix of integers and a C-by-3 color map. Each integer corresponds to an index in the color map, and each row in the color map corresponds to a RGB color. The color map must be of class `double` with values between 0 and 1.

`[img, map] = gray2ind ()` [Function File]

Convert a gray scale intensity image to an Octave indexed image.

`ind2gray (x, map)` [Function File]

Convert an Octave indexed image to a gray scale intensity image. If `map` is omitted, the current colormap is used to determine the intensities.

See also: `gray2ind`, `rgb2ntsc`, `image`, `colormap`.

`[x, map] = rgb2ind (rgb)` [Function File]

`[x, map] = rgb2ind (r, g, b)` [Function File]

Convert and RGB image to an Octave indexed image.

See also: `ind2rgb`, `rgb2ntsc`.

`rgb = ind2rgb (x, map)` [Function File]

`[r, g, b] = ind2rgb (x, map)` [Function File]

Convert an indexed image to red, green, and blue color components. If the colormap doesn't contain enough colors, pad it with the last color in the map. If `map` is omitted, the current colormap is used for the conversion.

See also: `rgb2ind`, `image`, `imshow`, `ind2gray`, `gray2ind`.

`colormap (map)` [Function File]

`colormap ("default")` [Function File]

Set the current colormap.

`colormap (map)` sets the current colormap to `map`. The color map should be an n row by 3 column matrix. The columns contain red, green, and blue intensities respectively. All entries should be between 0 and 1 inclusive. The new colormap is returned.

`colormap ("default")` restores the default colormap (a gray scale colormap with 64 entries). The default colormap is returned.

With no arguments, `colormap` returns the current color map.

`autumn (n)` [Function File]

Create color colormap. This colormap is red through orange to yellow. The argument n should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: `colormap`.

bone (*n*) [Function File]

Create color colormap. This colormap is a gray colormap with a light blue tone. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

cool (*n*) [Function File]

Create color colormap. The colormap is cyan to magenta. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

copper (*n*) [Function File]

Create color colormap. This colormap is black to a light copper tone. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

gray (*n*) [Function File]

Return a gray colormap with *n* entries corresponding to values from 0 to *n*-1. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

hot (*n*) [Function File]

Create color colormap. This colormap is black through dark red, red, orange, yellow to white. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

hsv (*n*) [Function File]

Create color colormap. This colormap is red through yellow, green, cyan, blue, magenta to red. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

jet (*n*) [Function File]

Create color colormap. This colormap is dark blue through blue, cyan, green, yellow, red to dark red. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

ocean (*n*) [Function File]

Create color colormap. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

pink (*n*) [Function File]

Create color colormap. This colormap gives a sephia tone on black and white images. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

prism (*n*) [Function File]

Create color colormap. This colormap cycles trough red, orange, yellow, green, blue and violet. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

rainbow (*n*) [Function File]

Create color colormap. This colormap is red through orange, yellow, green, blue to violet. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

spring (*n*) [Function File]

Create color colormap. This colormap is magenta to yellow. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

summer (*n*) [Function File]

Create color colormap. This colormap is green to yellow. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

white (*n*) [Function File]

Create color colormap. This colormap is completely white. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

winter (*n*) [Function File]

Create color colormap. This colormap is blue to green. The argument *n* should be a scalar. If it is omitted, the length of the current colormap or 64 is assumed.

See also: colormap.

33.4 Plotting on top of Images

If gnuplot is being used to display images it is possible to plot on top of images. Since an image is a matrix it is indexed by row and column values. The plotting system is, however, based on the traditional (x, y) system. To minimize the difference between the two systems Octave places the origin of the coordinate system in the point corresponding to the pixel at $(1, 1)$. So, to plot points given by row and column values on top of an image, one should simply call `plot` with the column values as the first argument and the row values as the second. As an example the following code generates an image with random intensities between 0 and 1, and shows the image with red circles over pixels with an intensity above 0.99.

```
I = rand (100, 100);
[row, col] = find (I > 0.99);
hold ("on");
imshow (I);
plot (col, row, "ro");
hold ("off");
```

33.5 Color Conversion

Octave supports conversion from the RGB color system to NTSC and HSV and vice versa.

`hsv_map = rgb2hsv (rgb_map)` [Function File]

Transform a colormap from the rgb space to the hsv space.

A color in the RGB space consists of the red, green and blue intensities.

In the HSV space each color is represented by their hue, saturation and value (brightness). Value gives the amount of light in the color. Hue describes the dominant wavelength. Saturation is the amount of Hue mixed into the color.

See also: `hsv2rgb`.

`rgb_map = hsv2rgb (hsv_map)` [Function File]

Transform a colormap from the hsv space to the rgb space.

See also: `rgb2hsv`.

`rgb2ntsc (rgb)` [Function File]

Image format conversion.

`ntsc2rgb (yiq)` [Function File]

Image format conversion.

34 Audio Processing

Octave provides a few functions for dealing with audio data. An audio ‘sample’ is a single output value from an A/D converter, i.e., a small integer number (usually 8 or 16 bits), and audio data is just a series of such samples. It can be characterized by three parameters: the sampling rate (measured in samples per second or Hz, e.g. 8000 or 44100), the number of bits per sample (e.g. 8 or 16), and the number of channels (1 for mono, 2 for stereo, etc.).

There are many different formats for representing such data. Currently, only the two most popular, *linear encoding* and *mu-law encoding*, are supported by Octave. There is an excellent FAQ on audio formats by Guido van Rossum <guido@cwi.nl> which can be found at any FAQ ftp site, in particular in the directory ‘/pub/usenet/news.answers/audio-fmts’ of the archive site rtfm.mit.edu.

Octave simply treats audio data as vectors of samples (non-mono data are not supported yet). It is assumed that audio files using linear encoding have one of the extensions ‘**lin**’ or ‘**raw**’, and that files holding data in mu-law encoding end in ‘**au**’, ‘**mu**’, or ‘**snd**’.

lin2mu (*x*, *n*) [Function File]

Converts audio data from linear to mu-law. Mu-law values use 8-bit unsigned integers. Linear values use *n*-bit signed integers or floating point values in the range $-1 \leq x \leq 1$ if *n* is 0. If *n* is not specified it defaults to 0, 8 or 16 depending on the range values in *x*.

See also: mu2lin, loadaudio, saveaudio, playaudio, setaudio, record.

mu2lin (*x*, *bps*) [Function File]

Converts audio data from linear to mu-law. Mu-law values are 8-bit unsigned integers. Linear values use *n*-bit signed integers or floating point values in the range $-1 \leq y \leq 1$ if *n* is 0. If *n* is not specified it defaults to 8.

See also: lin2mu, loadaudio, saveaudio, playaudio, setaudio, record.

loadaudio (*name*, *ext*, *bps*) [Function File]

Loads audio data from the file ‘*name.ext*’ into the vector *x*.

The extension *ext* determines how the data in the audio file is interpreted; the extensions ‘**lin**’ (default) and ‘**raw**’ correspond to linear, the extensions ‘**au**’, ‘**mu**’, or ‘**snd**’ to mu-law encoding.

The argument *bps* can be either 8 (default) or 16, and specifies the number of bits per sample used in the audio file.

See also: lin2mu, mu2lin, saveaudio, playaudio, setaudio, record.

saveaudio (*name*, *x*, *ext*, *bps*) [Function File]

Saves a vector *x* of audio data to the file ‘*name.ext*’. The optional parameters *ext* and *bps* determine the encoding and the number of bits per sample used in the audio file (see **loadaudio**); defaults are ‘**lin**’ and 8, respectively.

See also: lin2mu, mu2lin, loadaudio, playaudio, setaudio, record.

The following functions for audio I/O require special A/D hardware and operating system support. It is assumed that audio data in linear encoding can be played and recorded by reading from and writing to `‘/dev/dsp’`, and that similarly `‘/dev/audio’` is used for mu-law encoding. These file names are system-dependent. Improvements so that these functions will work without modification on a wide variety of hardware are welcome.

`playaudio (name, ext)` [Function File]

`playaudio (x)` [Function File]

Plays the audio file `‘name.ext’` or the audio data stored in the vector `x`.

See also: `lin2mu`, `mu2lin`, `loadaudio`, `saveaudio`, `setaudio`, `record`.

`record (sec, sampling_rate)` [Function File]

Records `sec` seconds of audio input into the vector `x`. The default value for `sampling_rate` is 8000 samples per second, or 8kHz. The program waits until the user types `RET` and then immediately starts to record.

See also: `lin2mu`, `mu2lin`, `loadaudio`, `saveaudio`, `playaudio`, `setaudio`.

`setaudio ([w_type [, value]])` [Function File]

Execute the shell command `‘mixer [w_type [, value]]’`

`y = wavread (filename)` [Function File]

Load the RIFF/WAVE sound file `filename`, and return the samples in vector `y`. If the file contains multichannel data, then `y` is a matrix with the channels represented as columns.

`[y, Fs, bits] = wavread (filename)` [Function File]

Additionally return the sample rate (`fs`) in Hz and the number of bits per sample (`bits`).

`[...] = wavread (filename, n)` [Function File]

Read only the first `n` samples from each channel.

`[...] = wavread (filename, [n1 n2])` [Function File]

Read only samples `n1` through `n2` from each channel.

`[samples, channels] = wavread (filename, "size")` [Function File]

Return the number of samples (`n`) and channels (`ch`) instead of the audio data.

See also: `wavwrite`.

`wavwrite (filename, y)` [Function File]

Write `y` to the canonical RIFF/WAVE sound file `filename`. A sample rate of 8000 Hz and 16-bit samples are assumed. Each column of the data represents a separate channel.

`wavwrite (filename, y, fs)` [Function File]

Set the sample rate to `fs` Hz.

`wavwrite (filename, y, fs, bits)` [Function File]

Set the sample rate to `fs` Hz and resolution to `bits` bits.

See also: `wavread`.

35 Quaternions

Quaternions are hypercomplex numbers used to represent spatial rotations in three dimensions. This set of routines provides a useful basis for working with quaternions in Octave. A tutorial is in the Octave source, `scripts/quaternion/quaternion.ps`.

These functions were written by A. S. Hodel, Associate Professor, Auburn University.

```
[a, b, c, d] = quaternion (w) [Function File]
[vv, theta] = quaternion (w) [Function File]
w = quaternion (a, b, c, d) [Function File]
w = quaternion (vv, theta) [Function File]
    Construct or extract a quaternion
        w = a*i + b*j + c*k + d
    from given data.
```

```
qconj (q) [Function File]
    Conjugate of a quaternion.
        q = [w, x, y, z] = w*i + x*j + y*k + z
        qconj (q) = -w*i -x*j -y*k + z
```

```
qderiv (omega) [Function File]
    Derivative of a quaternion.
    Let Q be a quaternion to transform a vector from a fixed frame to a rotating frame.
    If the rotating frame is rotating about the [x, y, z] axes at angular rates [wx, wy, wz],
    then the derivative of Q is given by
        Q' = qderivmat (omega) * Q
    If the passive convention is used (rotate the frame, not the vector), then
        Q' = -qderivmat (omega) * Q
```

```
qderivmat (omega) [Function File]
    Derivative of a quaternion.
    Let Q be a quaternion to transform a vector from a fixed frame to a rotating frame.
    If the rotating frame is rotating about the [x, y, z] axes at angular rates [wx, wy, wz],
    then the derivative of Q is given by
        Q' = qderivmat (omega) * Q
    If the passive convention is used (rotate the frame, not the vector), then
        Q' = -qderivmat (omega) * Q.
```

```
qinv (q) [Function File]
    Return the inverse of a quaternion.
        q = [w, x, y, z] = w*i + x*j + y*k + z
        qmult (q, qinv (q)) = 1 = [0 0 0 1]
```

qmult (*a*, *b*) [Function File]

Multiply two quaternions.

$[w, x, y, z] = w*i + x*j + y*k + z$

identities:

$i^2 = j^2 = k^2 = -1$

$ij = k$

$jk = i$

$ki = j$

$kj = -i$

$ji = -k$

$ik = -j$

qtrans (*v*, *q*) [Function File]

Transform the unit quaternion *v* by the unit quaternion *q*. Returns $v = q*v/q$.

qtransv (*v*, *q*) [Function File]

Transform the 3-D vector *v* by the unit quaternion *q*. Return a column vector.

$$v_i = (2*\text{real}(q)^2 - 1)*v_b + 2*\text{imag}(q)*(\text{imag}(q)'*v_b)$$

$$+ 2*\text{real}(q)*\text{cross}(\text{imag}(q), v_b)$$

Where $\text{imag}(q)$ is a column vector of length 3.

qtransvmat (*qib*) [Function File]

Construct a 3x3 transformation matrix from quaternion *qib* that is equivalent to rotation of *th* radians about axis *vv*, where $[vv, th] = \text{quaternion}(qib)$.

qcoordinate_plot (*qf*, *qb*, *qv*) [Function File]

Plot in the current figure a set of coordinate axes as viewed from the orientation specified by quaternion *qv*. Inertial axes are also plotted:

qf Quaternion from reference (x,y,z) to inertial.

qb Quaternion from reference to body.

qv Quaternion from reference to view angle.

36 System Utilities

This chapter describes the functions that are available to allow you to get information about what is happening outside of Octave, while it is still running, and use this information in your program. For example, you can get information about environment variables, the current time, and even start other programs from the Octave prompt.

36.1 Timing Utilities

Octave's core set of functions for manipulating time values are patterned after the corresponding functions from the standard C library. Several of these functions use a data structure for time that includes the following elements:

<code>usec</code>	Microseconds after the second (0-999999).
<code>sec</code>	Seconds after the minute (0-61). This number can be 61 to account for leap seconds.
<code>min</code>	Minutes after the hour (0-59).
<code>hour</code>	Hours since midnight (0-23).
<code>mday</code>	Day of the month (1-31).
<code>mon</code>	Months since January (0-11).
<code>year</code>	Years since 1900.
<code>wday</code>	Days since Sunday (0-6).
<code>yday</code>	Days since January 1 (0-365).
<code>isdst</code>	Daylight Savings Time flag.
<code>zone</code>	Time zone.

In the descriptions of the following functions, this structure is referred to as a *tm_struct*.

`time ()` [Loadable Function]
 Return the current time as the number of seconds since the epoch. The epoch is referenced to 00:00:00 CUT (Coordinated Universal Time) 1 Jan 1970. For example, on Monday February 17, 1997 at 07:15:06 CUT, the value returned by `time` was 856163706.

See also: `strftime`, `strptime`, `localtime`, `gmtime`, `mktime`, `now`, `date`, `clock`, `datenum`, `datestr`, `datevec`, `calendar`, `weekday`.

`t = now ()` [Function File]
 Returns the current local time as the number of days since Jan 1, 0000. By this reckoning, Jan 1, 1970 is day number 719529.
 The integral part, `floor(now)` corresponds to 00:00:00 today.
 The fractional part, `rem(now, 1)` corresponds to the current time on Jan 1, 0000.
 The returned value is also called a "serial date number" (see `datenum`).

See also: `clock`, `date`, `datenum`.

`ctime (t)` [Function File]

Convert a value returned from `time` (or any other nonnegative integer), to the local time and return a string of the same form as `asctime`. The function `ctime (time)` is equivalent to `asctime (localtime (time))`. For example,

```
ctime (time ())
⇒ "Mon Feb 17 01:15:06 1997\n"
```

`gmtime (t)` [Loadable Function]

Given a value returned from `time` (or any nonnegative integer), return a time structure corresponding to CUT. For example,

```
gmtime (time ())
⇒ {
    usec = 0
    year = 97
    mon = 1
    mday = 17
    sec = 6
    zone = CST
    min = 15
    wday = 1
    hour = 7
    isdst = 0
    yday = 47
}
```

See also: `strftime`, `strptime`, `localtime`, `mktime`, `time`, `now`, `date`, `clock`, `datenum`, `datestr`, `datevec`, `calendar`, `weekday`.

`localtime (t)` [Loadable Function]

Given a value returned from `time` (or any nonnegative integer), return a time structure corresponding to the local time zone.

```
localtime (time ())
⇒ {
    usec = 0
    year = 97
    mon = 1
    mday = 17
    sec = 6
    zone = CST
    min = 15
    wday = 1
    hour = 1
    isdst = 0
    yday = 47
}
```

See also: `strftime`, `strptime`, `gmtime`, `mktime`, `time`, `now`, `date`, `clock`, `datenum`, `datestr`, `datevec`, `calendar`, `weekday`.

mktime (*tm_struct*) [Loadable Function]

Convert a time structure corresponding to the local time to the number of seconds since the epoch. For example,

```
mktime (localtime (time ()))
⇒ 856163706
```

See also: `strftime`, `strptime`, `localtime`, `gmtime`, `time`, `now`, `date`, `clock`, `datenum`, `datestr`, `datevec`, `calendar`, `weekday`.

asctime (*tm_struct*) [Function File]

Convert a time structure to a string using the following five-field format: Thu Mar 28 08:40:14 1996. For example,

```
asctime (localtime (time ()))
⇒ "Mon Feb 17 01:15:06 1997\n"
```

This is equivalent to `ctime (time ())`.

strftime (*fmt*, *tm_struct*) [Loadable Function]

Format the time structure *tm_struct* in a flexible way using the format string *fmt* that contains ‘%’ substitutions similar to those in `printf`. Except where noted, substituted fields have a fixed size; numeric fields are padded if necessary. Padding is with zeros by default; for fields that display a single number, padding can be changed or inhibited by following the ‘%’ with one of the modifiers described below. Unknown field specifiers are copied as normal characters. All other characters are copied to the output without change. For example,

```
strftime ("%r (%Z) %A %e %B %Y", localtime (time ()))
⇒ "01:15:06 AM (CST) Monday 17 February 1997"
```

Octave’s `strftime` function supports a superset of the ANSI C field specifiers.

Literal character fields:

%	% character.
n	Newline character.
t	Tab character.

Numeric modifiers (a nonstandard extension):

- (dash)	Do not pad the field.
_ (underscore)	Pad the field with spaces.

Time fields:

%H	Hour (00-23).
%I	Hour (01-12).
%k	Hour (0-23).
%l	Hour (1-12).
%M	Minute (00-59).

<code>%p</code>	Locale's AM or PM.
<code>%r</code>	Time, 12-hour (hh:mm:ss [AP]M).
<code>%R</code>	Time, 24-hour (hh:mm).
<code>%s</code>	Time in seconds since 00:00:00, Jan 1, 1970 (a nonstandard extension).
<code>%S</code>	Second (00-61).
<code>%T</code>	Time, 24-hour (hh:mm:ss).
<code>%X</code>	Locale's time representation (%H:%M:%S).
<code>%Z</code>	Time zone (EDT), or nothing if no time zone is determinable.

Date fields:

<code>%a</code>	Locale's abbreviated weekday name (Sun-Sat).
<code>%A</code>	Locale's full weekday name, variable length (Sunday-Saturday).
<code>%b</code>	Locale's abbreviated month name (Jan-Dec).
<code>%B</code>	Locale's full month name, variable length (January-December).
<code>%c</code>	Locale's date and time (Sat Nov 04 12:02:33 EST 1989).
<code>%C</code>	Century (00-99).
<code>%d</code>	Day of month (01-31).
<code>%e</code>	Day of month (1-31).
<code>%D</code>	Date (mm/dd/yy).
<code>%h</code>	Same as <code>%b</code> .
<code>%j</code>	Day of year (001-366).
<code>%m</code>	Month (01-12).
<code>%U</code>	Week number of year with Sunday as first day of week (00-53).
<code>%w</code>	Day of week (0-6).
<code>%W</code>	Week number of year with Monday as first day of week (00-53).
<code>%x</code>	Locale's date representation (mm/dd/yy).
<code>%y</code>	Last two digits of year (00-99).
<code>%Y</code>	Year (1970-).

See also: `strptime`, `localtime`, `gmtime`, `mktime`, `time`, `now`, `date`, `clock`, `datenum`, `datestr`, `datevec`, `calendar`, `weekday`.

`[tm_struct, nchars] = strptime (str, fmt)` [Loadable Function]

Convert the string *str* to the time structure *tm_struct* under the control of the format string *fmt*.

If *fmt* fails to match, *nchars* is 0; otherwise it is set to the position of last matched character plus 1. Always check for this unless you're absolutely sure the date string will be parsed correctly.

See also: `strptime`, `localtime`, `gmtime`, `mktime`, `time`, `now`, `date`, `clock`, `datenum`, `datestr`, `datevec`, `calendar`, `weekday`.

Most of the remaining functions described in this section are not patterned after the standard C library. Some are available for compatibility with MATLAB and others are provided because they are useful.

clock () [Function File]

Return a vector containing the current year, month (1-12), day (1-31), hour (0-23), minute (0-59) and second (0-61). For example,

```
clock ()  
⇒ [ 1993, 8, 20, 4, 56, 1 ]
```

The function `clock` is more accurate on systems that have the `gettimeofday` function.

date () [Function File]

Return the date as a character string in the form DD-MMM-YY. For example,

```
date ()  
⇒ "20-Aug-93"
```

etime (t1, t2) [Function File]

Return the difference (in seconds) between two time values returned from `clock`. For example:

```
t0 = clock ();  
many computations later...  
elapsed_time = etime (clock (), t0);
```

will set the variable `elapsed_time` to the number of seconds since the variable `t0` was set.

See also: `tic`, `toc`, `clock`, `cputime`.

[total, user, system] = cputime (); [Function File]

Return the CPU time used by your Octave session. The first output is the total time spent executing your process and is equal to the sum of second and third outputs, which are the number of CPU seconds spent executing in user mode and the number of CPU seconds spent executing in system mode, respectively. If your system does not have a way to report CPU time usage, `cputime` returns 0 for each of its output values. Note that because Octave used some CPU time to start, it is reasonable to check to see if `cputime` works by checking to see if the total CPU time used is nonzero.

is_leap_year (year) [Function File]

Return 1 if the given year is a leap year and 0 otherwise. If no arguments are provided, `is_leap_year` will use the current year. For example,

```
is_leap_year (2000)  
⇒ 1
```

tic () [Function File]

toc () [Function File]

These functions set and check a wall-clock timer. For example,

```
tic ();
    many computations later...
elapsed_time = toc ();
```

will set the variable `elapsed_time` to the number of seconds since the most recent call to the function `tic`.

Nested timing with `tic` and `toc` is not supported. Therefore `toc` will always return the elapsed time from the most recent call to `tic`.

If you are more interested in the CPU time that your process used, you should use the `cputime` function instead. The `tic` and `toc` functions report the actual wall clock time that elapsed between the calls. This may include time spent processing other jobs or doing nothing at all. For example,

```
tic (); sleep (5); toc ()
⇒ 5
t = cputime (); sleep (5); cputime () - t
⇒ 0
```

(This example also illustrates that the CPU timer may have a fairly coarse resolution.)

pause (*seconds*) [Built-in Function]

Suspend the execution of the program. If invoked without any arguments, Octave waits until you type a character. With a numeric argument, it pauses for the given number of seconds. For example, the following statement prints a message and then waits 5 seconds before clearing the screen.

```
fprintf (stderr, "wait please...
");
pause (5);
clc;
```

sleep (*seconds*) [Built-in Function]

Suspend the execution of the program for the given number of seconds.

usleep (*microseconds*) [Built-in Function]

Suspend the execution of the program for the given number of microseconds. On systems where it is not possible to sleep for periods of time less than one second, `usleep` will pause the execution for `round (microseconds / 1e6)` seconds.

***str* = datestr (*date*, [*f*, [*p*]])** [Function File]

Format the given date/time according to the format `f` and return the result in `str`. `date` is a serial date number (see `datenum`) or a date vector (see `datevec`). The value of `date` may also be a string or cell array of strings.

`f` can be an integer which corresponds to one of the codes in the table below, or a date format string.

`p` is the year at the start of the century in which two-digit years are to be interpreted in. If not specified, it defaults to the current year minus 50.

For example, the date 730736.65149 (2000-09-07 15:38:09.0934) would be formatted as follows:

Code	Format	Example
0	dd-mmm-yyyy HH:MM:SS	07-Sep-2000 15:38:09
1	dd-mmm-yyyy	07-Sep-2000
2	mm/dd/yy	09/07/00
3	mmm	Sep
4	m	S
5	mm	09
6	mm/dd	09/07
7	dd	07
8	ddd	Thu
9	d	T
10	yyyy	2000
11	yy	00
12	mmmyy	Sep00
13	HH:MM:SS	15:38:09
14	HH:MM:SS PM	03:38:09 PM
15	HH:MM	15:38
16	HH:MM PM	03:38 PM
17	QQ-YY	Q3-00
18	QQ	Q3
19	dd/mm	13/03
20	dd/mm/yy	13/03/95
21	mmm.dd.yyyy HH:MM:SS	Mar.03.1962 13:53:06
22	mmm.dd.yyyy	Mar.03.1962
23	mm/dd/yyyy	03/13/1962
24	dd/mm/yyyy	12/03/1962
25	yy/mm/dd	95/03/13
26	yyyy/mm/dd	1995/03/13
27	QQ-YYYY	Q4-2132
28	mmmyyyy	Mar2047
29	yyyymmdd	20470313
30	yyyymmddTHHMMSS	20470313T132603
31	yyyy-mm-dd HH:MM:SS	1047-03-13 13:26:03

If *f* is a format string, the following symbols are recognised:

Symbol	Meaning	Example
yyyy	Full year	2005
yy	Two-digit year	2005
mmmm	Full month name	December
mmm	Abbreviated month name	Dec
mm	Numeric month number (padded with zeros)	01, 08, 12
m	First letter of month name (capitalized)	D
dddd	Full weekday name	Sunday
ddd	Abbreviated weekday name	Sun
dd	Numeric day of month (padded with zeros)	11

d	First letter of weekday name (capitalized)	S
HH	Hour of day, padded with zeros if PM is set and not padded with zeros otherwise	09:00 9:00 AM
MM	Minute of hour (padded with zeros)	10:05
SS	Second of minute (padded with zeros)	10:05:03
PM	Use 12-hour time format	11:30 PM

If *f* is not specified or is -1, then use 0, 1 or 16, depending on whether the date portion or the time portion of *date* is empty.

If *p* is not specified, it defaults to the current year minus 50.

If a matrix or cell array of dates is given, a vector of date strings is returned.

See also: datenum, datevec, date, clock, now, datetick.

```

v = datevec (date) [Function File]
v = datevec (date, f) [Function File]
v = datevec (date, p) [Function File]
v = datevec (date, f, p) [Function File]
[y, m, d, h, mi, s] = datevec (...) [Function File]

```

Convert a serial date number (see `datenum`) or date string (see `datestr`) into a date vector.

A date vector is a row vector with six members, representing the year, month, day, hour, minute, and seconds respectively.

f is the format string used to interpret date strings (see `datestr`).

p is the year at the start of the century in which two-digit years are to be interpreted in. If not specified, it defaults to the current year minus 50.

See also: datenum, datestr, date, clock, now.

```

calendar (...) [Function File]
c = calendar () [Function File]
c = calendar (d) [Function File]
c = calendar (y, m) [Function File]

```

If called with no arguments, return the current monthly calendar in a 6x7 matrix.

If *d* is specified, return the calendar for the month containing the day *d*, which must be a serial date number or a date string.

If *y* and *m* are specified, return the calendar for year *y* and month *m*.

If no output arguments are specified, print the calendar on the screen instead of returning a matrix.

See also: datenum.

```

[n, s] = weekday (d, [form]) [Function File]

```

Return the day of week as a number in *n* and a string in *s*, for example [1, "Sun"], [2, "Mon"], ..., or [7, "Sat"].

d is a serial date number or a date string.

If the string *form* is given and is "long", *s* will contain the full name of the weekday; otherwise (or if *form* is "short"), *s* will contain the abbreviated name of the weekday.

See also: datenum, datevec, eomday.

`e = eomday (y, m)` [Function File]

Return the last day of the month *m* for the year *y*.

See also: datenum, datevec, weekday.

36.2 Filesystem Utilities

Octave includes the following functions for renaming and deleting files, creating, deleting, and reading directories, and for getting information about the status of files.

`[err, msg] = rename (old, new)` [Built-in Function]

Change the name of file *old* to *new*.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: ls, dir.

`[err, msg] = link (old, new)` [Built-in Function]

Create a new link (also known as a hard link) to an existing file.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: symlink.

`[err, msg] = symlink (old, new)` [Built-in Function]

Create a symbolic link *new* which contains the string *old*.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: link, readlink.

`[result, err, msg] = readlink (symlink)` [Built-in Function]

Read the value of the symbolic link *symlink*.

If successful, *result* contains the contents of the symbolic link *symlink*, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: link, symlink.

`[err, msg] = unlink (file)` [Built-in Function]

Delete the file named *file*.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

`[files, err, msg] = readdir (dir)` [Built-in Function]

Return names of the files in the directory *dir* as a cell array of strings. If an error occurs, return an empty cell array in *files*.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: `dir`, `glob`.

`[status, msg, msgid] = mkdir (dir)` [Built-in Function]

`[status, msg, msgid] = mkdir (parent, dir)` [Built-in Function]

Create a directory named *dir*.

If successful, *status* is 1, with *msg* and *msgid* empty character strings. Otherwise, *status* is 0, *msg* contains a system-dependent error message, and *msgid* contains a unique message identifier.

See also: `rmdir`.

`[status, msg, msgid] = rmdir (dir)` [Built-in Function]

`[status, msg, msgid] = rmdir (dir, "s")` [Built-in Function]

Remove the directory named *dir*.

If successful, *status* is 1, with *msg* and *msgid* empty character strings. Otherwise, *status* is 0, *msg* contains a system-dependent error message, and *msgid* contains a unique message identifier.

If the optional second parameter is supplied with value "s", recursively remove all subdirectories as well.

See also: `mkdir`, `confirm_recursive_rmdir`.

`val = confirm_recursive_rmdir ()` [Built-in Function]

`old_val = confirm_recursive_rmdir (new_val)` [Built-in Function]

Query or set the internal variable that controls whether Octave will ask for confirmation before recursively removing a directory tree.

`[err, msg] = mkfifo (name, mode)` [Built-in Function]

Create a *fifo* special file named *name* with file mode *mode*

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

`umask (mask)` [Built-in Function]

Set the permission mask for file creation. The parameter *mask* is an integer, interpreted as an octal number. If successful, returns the previous value of the mask (as an integer to be interpreted as an octal number); otherwise an error message is printed.

`[info, err, msg] = stat (file)` [Built-in Function]

`[info, err, msg] = lstat (file)` [Built-in Function]

Return a structure *s* containing the following information about *file*.

dev ID of device containing a directory entry for this file.

<code>ino</code>	File number of the file.
<code>mode</code>	File mode, as an integer. Use the functions <code>S_ISREG</code> , <code>S_ISDIR</code> , <code>S_ISCHR</code> , <code>S_ISBLK</code> , <code>S_ISFIFO</code> , <code>S_ISLNK</code> , or <code>S_ISSOCK</code> to extract information from this value.
<code>modestr</code>	File mode, as a string of ten letters or dashes as would be returned by <code>ls -l</code> .
<code>nlink</code>	Number of links.
<code>uid</code>	User ID of file's owner.
<code>gid</code>	Group ID of file's group.
<code>rdev</code>	ID of device for block or character special files.
<code>size</code>	Size in bytes.
<code>atime</code>	Time of last access in the same form as time values returned from <code>time</code> . See Section 36.1 [Timing Utilities] , page 427.
<code>mtime</code>	Time of last modification in the same form as time values returned from <code>time</code> . See Section 36.1 [Timing Utilities] , page 427.
<code>ctime</code>	Time of last file status change in the same form as time values returned from <code>time</code> . See Section 36.1 [Timing Utilities] , page 427.
<code>blksize</code>	Size of blocks in the file.
<code>blocks</code>	Number of blocks allocated for file.

If the call is successful `err` is 0 and `msg` is an empty string. If the file does not exist, or some other error occurs, `s` is an empty matrix, `err` is `-1`, and `msg` contains the corresponding system error message.

If `file` is a symbolic link, `stat` will return information about the actual file the is referenced by the link. Use `lstat` if you want information about the symbolic link itself.

For example,

```

[s, err, msg] = stat ("/vmlinuz")
⇒ s =
{
  atime = 855399756
  rdev = 0
  ctime = 847219094
  uid = 0
  size = 389218
  blksize = 4096
  mtime = 847219094
  gid = 6
  nlink = 1
  blocks = 768
  mode = -rw-r--r--
  modestr = -rw-r--r--
  ino = 9316
  dev = 2049
}
⇒ err = 0
⇒ msg =

```

`[info, err, msg] = lstat (file)`
 See `stat`.

[Built-in Function]

`[status, msg, msgid] = fileattrib (file)`
 Return information about *file*.

[Function File]

If successful, *status* is 1, with *result* containing a structure with the following fields:

Name Full name of *file*.

archive True if *file* is an archive (Windows).

system True if *file* is a system file (Windows).

hidden True if *file* is a hidden file (Windows).

directory True if *file* is a directory.

UserRead

GroupRead

OtherRead

True if the user (group; other users) has read permission for *file*.

UserWrite

GroupWrite

OtherWrite

True if the user (group; other users) has write permission for *file*.

```
UserExecute
GroupExecute
OtherExecute
```

True if the user (group; other users) has execute permission for *file*.

If an attribute does not apply (i.e., archive on a Unix system) then the field is set to NaN.

With no input arguments, return information about the current directory.

If *file* contains globbing characters, return information about all the matching files.

See also: glob.

isdir (*f*) [Function File]
Return true if *f* is a directory.

glob (*pattern*) [Built-in Function]
Given an array of strings (as a char array or a cell array) in *pattern*, return a cell array of file names that match any of them, or an empty cell array if no patterns match. Tilde expansion is performed on each of the patterns before looking for matching file names. For example,

```
glob ("/vm*")
⇒ "/vmlinuz"
```

See also: dir, ls, stat, readdir.

fnmatch (*pattern*, *string*) [Built-in Function]
Return 1 or zero for each element of *string* that matches any of the elements of the string array *pattern*, using the rules of filename pattern matching. For example,

```
fnmatch ("a*b", {"ab"; "axyzb"; "xyzab"})
⇒ [ 1; 1; 0 ]
```

file_in_path (*path*, *file*) [Built-in Function]
file_in_path (*path*, *file*, "all") [Built-in Function]

Return the absolute name of *file* if it can be found in *path*. The value of *path* should be a colon-separated list of directories in the format described for *path*. If no file is found, return an empty matrix. For example,

```
file_in_path (EXEC_PATH, "sh")
⇒ "/bin/sh"
```

If the second argument is a cell array of strings, search each directory of the path for element of the cell array and return the first that matches.

If the third optional argument "all" is supplied, return a cell array containing the list of all files that have the same name in the path. If no files are found, return an empty cell array.

See also: file_in_loadpath.

tilde_expand (*string*) [Built-in Function]

Performs tilde expansion on *string*. If *string* begins with a tilde character, ('~'), all of the characters preceding the first slash (or all characters, if there is no slash) are treated as a possible user name, and the tilde and the following characters up to the slash are replaced by the home directory of the named user. If the tilde is followed immediately by a slash, the tilde is replaced by the home directory of the user running Octave. For example,

```
tilde_expand ("~joeuser/bin")
⇒ "/home/joeuser/bin"
tilde_expand ("~/bin")
⇒ "/home/jwe/bin"
```

[*cname*, *status*, *msg*] **canonicalize_file_name** (*name*) [Built-in Function]

Return the canonical name of file *name*.

[*status*, *msg*, *msgid*] = **movefile** (*f1*, *f2*) [Function File]

Move the file *f1* to the new name *f2*. The name *f1* may contain globbing patterns. If *f1* expands to multiple file names, *f2* must be a directory.

If successful, *status* is 1, with *msg* and *msgid* empty\n\n character strings. Otherwise, *status* is 0, *msg* contains a\n\n system-dependent error message, and *msgid* contains a unique\n\n message identifier.\n\n

See also: glob.

[*status*, *msg*, *msgid*] = **copyfile** (*f1*, *f2*, *force*) [Function File]

Copy the file *f1* to the new name *f2*. The name *f1* may contain globbing patterns. If *f1* expands to multiple file names, *f2* must be a directory. If *force* is given and equals the string "f" the copy operation will be forced.

If successful, *status* is 1, with *msg* and *msgid* empty\n\n character strings. Otherwise, *status* is 0, *msg* contains a\n\n system-dependent error message, and *msgid* contains a unique\n\n message identifier.\n\n

See also: glob, movefile.

[*dir*, *name*, *ext*, *ver*] = **fileparts** (*filename*) [Function File]

Return the directory, name, extension, and version components of *filename*.

See also: fullfile.

filesep () [Built-in Function]

Return the system-dependent character used to separate directory names.

See also: pathsep, dir, ls.

filename = **fullfile** (*dir1*, *dir2*, ..., *file*) [Function File]

Return a complete filename constructed from the given components.

See also: fileparts.

`dir = fullfile ()` [Function File]
 Return the name of the system's directory for temporary files.

`filename = tempname ()` [Function File]
 This function is an alias for `tmpnam`.

`P_tmpdir ()` [Built-in Function]
 Return the default name of the directory for temporary files on this system. The name of this directory is system dependent.

36.3 File Archiving Utilities

`bunzip2 (bzfile, dir)` [Function File]
 Unpack the bzip2 archive *bzfile* to the directory *dir*. If *dir* is not specified, it defaults to the current directory.

See also: `unpack`, `bzip2`, `tar`, `untar`, `gzip`, `gunzip`, `zip`, `unzip`.

`gunzip (gzfile, dir)` [Function File]
 Unpack the gzip archive *gzfile* to the directory *dir*. If *dir* is not specified, it defaults to the current directory. If the *gzfile* is a directory, all files in the directory will be recursively gunzipped.

See also: `unpack`, `bzip2`, `bunzip2`, `tar`, `untar`, `gzip`, `gunzip`, `zip`, `unzip`.

`entries = tar (tarfile, files, root)` [Function File]
 Pack *files* into the TAR archive *tarfile*. The list of files must be a string or a cell array of strings.

The optional argument *root* changes the relative path of *files* from the current directory.

If an output argument is requested the entries in the archive are returned in a cell array.

See also: `untar`, `gzip`, `gunzip`, `zip`, `unzip`.

`untar (tarfile, dir)` [Function File]
 Unpack the TAR archive *tarfile* to the directory *dir*. If *dir* is not specified, it defaults to the current directory.

See also: `unpack`, `bzip2`, `bunzip2`, `tar`, `gzip`, `gunzip`, `zip`, `unzip`.

`entries = zip (zipfile, files)` [Function File]

`entries = zip (zipfile, files, rootdir)` [Function File]

Compress the list of files and/or directories specified in *files* into the archive *zipfiles* in the same directory. If *rootdir* is defined the *files* is located relative to *rootdir* rather than the current directory

See also: `unzip`, `tar`.

`unzip (zipfile, dir)` [Function File]
 Unpack the ZIP archive *zipfile* to the directory *dir*. If *dir* is not specified, it defaults to the current directory.

See also: unpack, bzip2, bunzip2, tar, untar, gzip, gunzip, zip.

`pack ()` [Function File]
 This function is provided for compatibility with MATLAB, but it doesn't actually do anything.

`files = unpack (file, dir)` [Function File]
`files = unpack (file, dir, filetype)` [Function File]

Unpack the archive *file* based on its extension to the directory *dir*. If *file* is a cellstr, then all files will be handled individually. If *dir* is not specified, it defaults to the current directory. It returns a list of *files* unpacked. If a directory is in the file list, then the *filetype* to unpack must also be specified.

The *files* includes the entire path to the output files.

See also: bzip2, bunzip2, tar, untar, gzip, gunzip, zip, unzip.

36.4 Networking Utilities

`s = urlread (url)` [Loadable Function]
`[s, success] = urlread (url)` [Loadable Function]
`[s, success, message] = urlread (url)` [Loadable Function]
`[...] = urlread (url, method, param)` [Loadable Function]

Download a remote file specified by its *URL* and return its content in string *s*. For example,

```
s = urlread ("ftp://ftp.octave.org/pub/octave/README");
```

The variable *success* is 1 if the download was successful, otherwise it is 0 in which case *message* contains an error message. If no output argument is specified and if an error occurs, then the error is signaled through Octave's error handling mechanism.

This function uses libcurl. Curl supports, among others, the HTTP, FTP and FILE protocols. Username and password may be specified in the URL. For example,

```
s = urlread ("http://username:password@example.com/file.txt");
```

GET and POST requests can be specified by *method* and *param*. The parameter *method* is either 'get' or 'post' and *param* is a cell array of parameter and value pairs. For example,

```
s = urlread ("http://www.google.com/search", "get",  
             {"query", "octave"});
```

See also: urlwrite.

`urlwrite (URL, localfile)` [Loadable Function]
`f = urlwrite (url, localfile)` [Loadable Function]
`[f, success] = urlwrite (url, localfile)` [Loadable Function]
`[f, success, message] = urlwrite (url, localfile)` [Loadable Function]

Download a remote file specified by its *URL* and save it as *localfile*. For example,

```
urlwrite ("ftp://ftp.octave.org/pub/octave/README", "README.txt");
```

The full path of the downloaded file is returned in *f*. The variable *success* is 1 if the download was successful, otherwise it is 0 in which case *message* contains an error message. If no output argument is specified and if an error occurs, then the error is signaled through Octave's error handling mechanism.

This function uses libcurl. Curl supports, among others, the HTTP, FTP and FILE protocols. Username and password may be specified in the URL, for example:

```
urlwrite ("http://username:password@example.com/file.txt",
         "file.txt");
```

GET and POST requests can be specified by *method* and *param*. The parameter *method* is either 'get' or 'post' and *param* is a cell array of parameter and value pairs. For example:

```
urlwrite ("http://www.google.com/search", "search.html",
         "get", {"query", "octave"});
```

See also: `urlread`.

36.5 Controlling Subprocesses

Octave includes some high-level commands like `system` and `popen` for starting subprocesses. If you want to run another program to perform some task and then look at its output, you will probably want to use these functions.

Octave also provides several very low-level Unix-like functions which can also be used for starting subprocesses, but you should probably only use them if you can't find any way to do what you need with the higher-level functions.

system (*string*, *return_output*, *type*) [Built-in Function]

Execute a shell command specified by *string*. The second argument is optional. If *type* is "async", the process is started in the background and the process id of the child process is returned immediately. Otherwise, the process is started, and Octave waits until it exits. If *type* argument is omitted, a value of "sync" is assumed.

If two input arguments are given (the actual value of *return_output* is irrelevant) and the subprocess is started synchronously, or if `system` is called with one input argument and one or more output arguments, the output from the command is returned. Otherwise, if the subprocess is executed synchronously, its output is sent to the standard output. To send the output of a command executed with `system` through the pager, use a command like

```
disp (system (cmd, 1));
```

or

```
printf ("%s\n", system (cmd, 1));
```

The `system` function can return two values. The first is the exit status of the command and the second is any output from the command that was written to the standard output stream. For example,

```
[status, output] = system ("echo foo; exit 2");
```

will set the variable `output` to the string 'foo', and the variable `status` to the integer '2'.

```
[status, text] unix (command) [Function File]
```

```
[status, text] unix (command, "-echo") [Function File]
```

Execute a system command if running under a Unix-like operating system, otherwise do nothing. Return the exit status of the program in *status* and any output sent to the standard output in *text*. If the optional second argument `"-echo"` is given, then also send the output from the command to the standard output.

See also: `isunix`, `ispc`, `system`.

```
[status, text] = dos (command) [Function File]
```

```
[status, text] = dos (command, "-echo") [Function File]
```

Execute a system command if running under a Windows-like operating system, otherwise do nothing. Return the exit status of the program in *status* and any output sent to the standard output in *text*. If the optional second argument `"-echo"` is given, then also send the output from the command to the standard output.

See also: `unix`, `isunix`, `ispc`, `system`.

```
fid = popen (command, mode) [Built-in Function]
```

Start a process and create a pipe. The name of the command to run is given by *command*. The file identifier corresponding to the input or output stream of the process is returned in *fid*. The argument *mode* may be

`"r"` The pipe will be connected to the standard output of the process, and open for reading.

`"w"` The pipe will be connected to the standard input of the process, and open for writing.

For example,

```
fid = popen ("ls -ltr / | tail -3", "r");
while (isstr (s = fgets (fid)))
  fputs (stdout, s);
endwhile
+ drwxr-xr-x 33 root root 3072 Feb 15 13:28 etc
+ drwxr-xr-x 3 root root 1024 Feb 15 13:28 lib
+ drwxrwxrwt 15 root root 2048 Feb 17 14:53 tmp
```

```
pclose (fid) [Built-in Function]
```

Close a file identifier that was opened by `popen`. You may also use `fclose` for the same purpose.

```
[in, out, pid] = popen2 (command, args) [Built-in Function]
```

Start a subprocess with two-way communication. The name of the process is given by *command*, and *args* is an array of strings containing options for the command.

The file identifiers for the input and output streams of the subprocess are returned in *in* and *out*. If execution of the command is successful, *pid* contains the process ID of the subprocess. Otherwise, *pid* is -1 .

For example,

```
[in, out, pid] = popen2 ("sort", "-nr");
fputs (in, "these\nare\nsome\nstrings\n");
fclose (in);
EAGAIN = errno ("EAGAIN");
done = false;
do
  s = fgets (out);
  if (ischar (s))
    fputs (stdout, s);
  elseif (errno () == EAGAIN)
    sleep (0.1);
    fcLEAR (out);
  else
    done = true;
  endif
until (done)
fclose (out);
+ are
+ some
+ strings
+ these
```

`val = EXEC_PATH ()` [Built-in Function]

`old_val = EXEC_PATH (new_val)` [Built-in Function]

Query or set the internal variable that specifies a colon separated list of directories to search when executing external programs. Its initial value is taken from the environment variable `OCTAVE_EXEC_PATH` (if it exists) or `PATH`, but that value can be overridden by the command line argument `--exec-path PATH`. At startup, an additional set of directories (including the shell `PATH`) is appended to the path specified in the environment or on the command line. If you use the `EXEC_PATH` function to modify the path, you should take care to preserve these additional directories.

In most cases, the following functions simply decode their arguments and make the corresponding Unix system calls. For a complete example of how they can be used, look at the definition of the function `popen2`.

`[pid, msg] = fork ()` [Built-in Function]

Create a copy of the current process.

Fork can return one of the following values:

- > 0 You are in the parent process. The value returned from `fork` is the process id of the child process. You should probably arrange to wait for any child processes to exit.

0 You are in the child process. You can call `exec` to start another process. If that fails, you should probably call `exit`.

< 0 The call to `fork` failed for some reason. You must take evasive action. A system dependent error message will be waiting in `msg`.

`[err, msg] = exec (file, args)` [Built-in Function]

Replace current process with a new process. Calling `exec` without first calling `fork` will terminate your current Octave process and replace it with the program named by `file`. For example,

```
exec ("ls" "-l")
```

will run `ls` and return you to your shell prompt.

If successful, `exec` does not return. If `exec` does return, `err` will be nonzero, and `msg` will contain a system-dependent error message.

`[read_fd, write_fd, err, msg] = pipe ()` [Built-in Function]

Create a pipe and return the reading and writing ends of the pipe into `read_fd` and `write_fd` respectively.

If successful, `err` is 0 and `msg` is an empty string. Otherwise, `err` is nonzero and `msg` contains a system-dependent error message.

`[fid, msg] = dup2 (old, new)` [Built-in Function]

Duplicate a file descriptor.

If successful, `fid` is greater than zero and contains the new file ID. Otherwise, `fid` is negative and `msg` contains a system-dependent error message.

`[pid, status, msg] = waitpid (pid, options)` [Built-in Function]

Wait for process `pid` to terminate. The `pid` argument can be:

-1 Wait for any child process.

0 Wait for any child process whose process group ID is equal to that of the Octave interpreter process.

> 0 Wait for termination of the child process with ID `pid`.

The `options` argument can be a bitwise OR of zero or more of the following constants:

0 Wait until signal is received or a child process exits (this is the default if the `options` argument is missing).

WNOHANG Do not hang if status is not immediately available.

WUNTRACED

Report the status of any child processes that are stopped, and whose status has not yet been reported since they stopped.

WCONTINUED

Return if a stopped child has been resumed by delivery of `SIGCONT`. This value may not be meaningful on all systems.

If the returned value of *pid* is greater than 0, it is the process ID of the child process that exited. If an error occurs, *pid* will be less than zero and *msg* will contain a system-dependent error message. The value of *status* contains additional system-dependent information about the subprocess that exited.

See also: WNOHANG, WUNTRACED, WCONTINUED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WCOREDUMP, WIFSTOPPED, WSTOPSIG, WIFCONTINUED.

[*err*, *msg*] = fcntl (*fid*, *request*, *arg*) [Built-in Function]

Change the properties of the open file *fid*. The following values may be passed as *request*:

F_DUPFD	Return a duplicate file descriptor.
F_GETFD	Return the file descriptor flags for <i>fid</i> .
F_SETFD	Set the file descriptor flags for <i>fid</i> .
F_GETFL	Return the file status flags for <i>fid</i> . The following codes may be returned (some of the flags may be undefined on some systems).
O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_APPEND	Append on each write.
O_CREAT	Create the file if it does not exist.
O_NONBLOCK	Nonblocking mode.
O_SYNC	Wait for writes to complete.
O_ASYNC	Asynchronous I/O.
F_SETFL	Set the file status flags for <i>fid</i> to the value specified by <i>arg</i> . The only flags that can be changed are O_APPEND and O_NONBLOCK .

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

[*err*, *msg*] = kill (*pid*, *sig*) [Built-in Function]

Send signal *sig* to process *pid*.

If *pid* is positive, then signal *sig* is sent to *pid*.

If *pid* is 0, then signal *sig* is sent to every process in the process group of the current process.

If *pid* is -1, then signal *sig* is sent to every process except process 1.

If *pid* is less than -1, then signal *sig* is sent to every process in the process group -*pid*.

If *sig* is 0, then no signal is sent, but error checking is still performed.

Return 0 if successful, otherwise return -1.

SIG () [Built-in Function]

Return a structure containing Unix signal names and their defined values.

36.6 Process, Group, and User IDs

`pgid = getpgrp ()` [Built-in Function]

Return the process group id of the current process.

`pid = getpid ()` [Built-in Function]

Return the process id of the current process.

`pid = getppid ()` [Built-in Function]

Return the process id of the parent process.

`euid = geteuid ()` [Built-in Function]

Return the effective user id of the current process.

`uid = getuid ()` [Built-in Function]

Return the real user id of the current process.

`egid = getegid ()` [Built-in Function]

Return the effective group id of the current process.

`gid = getgid ()` [Built-in Function]

Return the real group id of the current process.

36.7 Environment Variables

`getenv (var)` [Built-in Function]

Return the value of the environment variable *var*. For example,

```
getenv ("PATH")
```

returns a string containing the value of your path.

`putenv (var, value)` [Built-in Function]

Set the value of the environment variable *var* to *value*.

36.8 Current Working Directory

`cd dir` [Command]

`chdir dir` [Command]

Change the current working directory to *dir*. If *dir* is omitted, the current directory is changed to the users home directory. For example,

```
cd ~/octave
```

Changes the current working directory to ‘~/octave’. If the directory does not exist, an error message is printed and the working directory is not changed.

See also: `mkdir`, `rmdir`, `dir`.

ls *options* [Command]

List directory contents. For example,

```
ls -l
├ total 12
├ -rw-r--r--  1 jwe  users  4488 Aug 19 04:02 foo.m
├ -rw-r--r--  1 jwe  users  1315 Aug 17 23:14 bar.m
```

The **dir** and **ls** commands are implemented by calling your system's directory listing command, so the available options may vary from system to system.

See also: **dir**, **stat**, **readdir**, **glob**, **filesep**, **ls_command**.

[old_cmd = ls_command (cmd)] [Function File]

Set or return the shell command used by Octave's **ls** command. The value of *cmd* must be a character string. With no arguments, simply return the previous value.

See also: **ls**.

dir (directory) [Function File]

[list] = dir (directory) [Function File]

Display file listing for directory *directory*. If a return value is requested, return a structure array with the fields

```
name
bytes
date
isdir
statinfo
```

in which **statinfo** is the structure returned from **stat**.

If *directory* is not a directory, return information about the named *filename*. *directory* may be a list of directories specified either by name or with wildcard characters (like ***** and **?**) which will be expanded with **glob**.

See also: **ls**, **stat**, **readdir**, **glob**, **filesep**.

pwd () [Built-in Function]

Return the current working directory.

See also: **dir**, **ls**.

36.9 Password Database Functions

Octave's password database functions return information in a structure with the following fields.

name	The user name.
passwd	The encrypted password, if available.
uid	The numeric user id.
gid	The numeric group id.

gecos The GECOS field.
dir The home directory.
shell The initial shell.

In the descriptions of the following functions, this data structure is referred to as a *pw_struct*.

pw_struct = getpwent () [Loadable Function]
 Return a structure containing an entry from the password database, opening it if necessary. Once the end of the data has been reached, **getpwent** returns 0.

pw_struct = getpwuid (uid). [Loadable Function]
 Return a structure containing the first entry from the password database with the user ID *uid*. If the user ID does not exist in the database, **getpwuid** returns 0.

pw_struct = getpwnam (name) [Loadable Function]
 Return a structure containing the first entry from the password database with the user name *name*. If the user name does not exist in the database, **getpwnam** returns 0.

setpwent () [Loadable Function]
 Return the internal pointer to the beginning of the password database.

endpwent () [Loadable Function]
 Close the password database.

36.10 Group Database Functions

Octave's group database functions return information in a structure with the following fields.

name The user name.
passwd The encrypted password, if available.
gid The numeric group id.
mem The members of the group.

In the descriptions of the following functions, this data structure is referred to as a *grp_struct*.

grp_struct = getgrent () [Loadable Function]
 Return an entry from the group database, opening it if necessary. Once the end of the data has been reached, **getgrent** returns 0.

grp_struct = getgrgid (gid). [Loadable Function]
 Return the first entry from the group database with the group ID *gid*. If the group ID does not exist in the database, **getgrgid** returns 0.

`grp_struct = getgrnam (name)` [Loadable Function]
 Return the first entry from the group database with the group name *name*. If the group name does not exist in the database, `getgrname` returns 0.

`setgrent ()` [Loadable Function]
 Return the internal pointer to the beginning of the group database.

`endgrent ()` [Loadable Function]
 Close the group database.

36.11 System Information

`[c, maxsize, endian] = computer ()` [Function File]
 Print or return a string of the form *cpu-vendor-os* that identifies the kind of computer Octave is running on. If invoked with an output argument, the value is returned instead of printed. For example,

```
computer ()
⇒ i586-pc-linux-gnu

x = computer ()
⇒ x = "i586-pc-linux-gnu"
```

If two output arguments are requested, also return the maximum number of elements for an array.

If three output arguments are requested, also return the byte order of the current system as a character ("B" for big-endian or "L" for little-endian).

`[uts, err, msg] = uname ()` [Built-in Function]
 Return system information in the structure. For example,

```
uname ()
⇒ {
    sysname = x86_64
    nodename = segfault
    release = 2.6.15-1-amd64-k8-smp
    version = Linux
    machine = #2 SMP Thu Feb 23 04:57:49 UTC 2006
}
```

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

`ispc ()` [Function File]
 Return 1 if Octave is running on a Windows system and 0 otherwise.

`isunix ()` [Function File]
 Return 1 if Octave is running on a Unix-like system and 0 otherwise.

isieee () [Built-in Function]
 Return 1 if your computer claims to conform to the IEEE standard for floating point calculations.

OCTAVE_HOME () [Built-in Function]
 Return the name of the top-level Octave installation directory.

OCTAVE_VERSION () [Built-in Function]
 Return the version number of Octave, as a string.

version () [Function File]
 Return Octave's version number as a string. This is also the value of the built-in variable **OCTAVE_VERSION**.

ver () [Function File]
 Display a header containing the current Octave version number, license string and operating system.
See also: `license`, `version`.

octave_config_info (*option*) [Built-in Function]
 Return a structure containing configuration and installation information for Octave. if *option* is a string, return the configuration information for the specified option.

getrusage () [Loadable Function]
 Return a structure containing a number of statistics about the current Octave process. Not all fields are available on all systems. If it is not possible to get CPU time statistics, the CPU time slots are set to zero. Other missing data are replaced by NaN. Here is a list of all the possible fields that can be present in the structure returned by **getrusage**:

idrss	Unshared data size.
inblock	Number of block input operations.
isrss	Unshared stack size.
ixrss	Shared memory size.
majflt	Number of major page faults.
maxrss	Maximum data size.
minflt	Number of minor page faults.
msgrcv	Number of messages received.
msgsnd	Number of messages sent.
nivcsw	Number of involuntary context switches.
nsignals	Number of signals received.

<code>nswap</code>	Number of swaps.
<code>nvcs</code>	Number of voluntary context switches.
<code>oublock</code>	Number of block output operations.
<code>stime</code>	A structure containing the system CPU time used. The structure has the elements <code>sec</code> (seconds) <code>usec</code> (microseconds).
<code>utime</code>	A structure containing the user CPU time used. The structure has the elements <code>sec</code> (seconds) <code>usec</code> (microseconds).

36.12 Hashing Functions

It is often necessary to find if two strings or files are identical. This might be done by comparing them character by character and looking for differences. However, this can be slow, and so comparing a hash of the string or file can be a rapid way of finding if the files differ.

Another use of the hashing function is to check for file integrity. The user can check the hash of the file against a known value and find if the file they have is the same as the one that the original hash was produced with.

Octave supplies the `md5sum` function to perform MD5 hashes on strings and files. An example of the use of `md5sum` function might be

```
if exist (file, "file")
    hash = md5sum (file);
else
    # Treat the variable "file" as a string
    hash = md5sum (file, true);
endif
```

`md5sum (file)` [Loadable Function]

`md5sum (str, opt)` [Loadable Function]

Calculates the MD5 sum of the file *file*. If the second parameter *opt* exists and is true, then calculate the MD5 sum of the string *str*.

37 Packages

Since Octave is Free Software users are encouraged to share their programs amongst each other. To aid this sharing Octave supports the installation of extra packages. The ‘Octave-Forge’ project is a community-maintained set of packages that can be downloaded and installed in Octave. At the time of writing the ‘Octave-Forge’ project can be found on-line at <http://octave.sourceforge.net>, but since the Internet is an ever-changing place this may not be true at the time of reading. Therefore it is recommended to see the Octave website for an updated reference.

37.1 Installing and Removing Packages

Assuming a package is available in the file `image-1.0.0.tar.gz` it can be installed from the Octave prompt by writing

```
pkg install image-1.0.0.tar.gz
```

If the package is installed successfully nothing will be printed on the prompt, but if an error occurred during installation it will be reported. It is possible to install several packages at once by writing several package files after the `pkg install` command. If a different version of the package is already installed it will be removed prior to installing the new package. This makes it easy to upgrade and downgrade the version of a package, but makes it impossible to have several versions of the same package installed at once.

To see which packages are installed type

```
pkg list
+ Package Name | Version | Installation directory
+ -----+-----+-----
+          image *|    1.0.0 | /home/jwe/octave/image-1.0.0
```

In this case only version 1.0.0 of the `image` package is installed. The ‘*’ character next to the package name shows that the image package is loaded and ready for use.

It is possible to remove a package from the system using the `pkg uninstall` command like this

```
pkg uninstall image
```

If the package is removed successfully nothing will be printed in the prompt, but if an error occurred it will be reported. It should be noted that the package file used for installation is not needed for removal, and that only the package name as reported by `pkg list` should be used when removing a package. It is possible to remove several packages at once by writing several package names after the `pkg uninstall` command.

To minimize the amount of code duplication between packages it is possible that one package depends on another one. If a package depends on another, it will check if that package is installed during installation. If it is not, an error will be reported and the package will not be installed. This behaviour can be disabled by passing the `-nodeps` flag to the `pkg install` command

```
pkg install -nodeps my_package_with_dependencies.tar.gz
```

Since the installed package expects its dependencies to be installed it may not function correctly. Because of this it is not recommended to disable dependency checking.

37.2 Using Packages

By default installed packages are available from the Octave prompt, but it is possible to control this using the `pkg load` and `pkg unload` commands. The functions from a package can be removed from the Octave path by typing

```
pkg unload package_name
```

where `package_name` is the name of the package to be removed from the path.

In much the same way a package can be added to the Octave path by typing

```
pkg load package_name
```

37.3 Administrating Packages

On UNIX-like systems it is possible to make both per-user and system-wide installations of a package. If the user performing the installation is `root` the packages will be installed in a system-wide directory that defaults to `OCTAVE_HOME/share/octave/packages/`. If the user is not `root` the default installation directory is `~/octave/`. Packages will be installed in a subdirectory of the installation directory that will be named after the package. It is possible to change the installation directory by using the `pkg prefix` command

```
pkg prefix new_installation_directory
```

The current installation directory can be retrieved by typing

```
current_installation_directory = pkg prefix
```

To function properly the package manager needs to keep some information about the installed packages. For per-user packages this information is by default stored in the file `~/octave_packages` and for system-wide installations it is stored in `OCTAVE_HOME/share/octave/octave_packages`. The path to the per-user file can be changed with the `pkg local_list` command

```
pkg local_list /path/to/new_file
```

For system-wide installations this can be changed in the same way using the `pkg global_list` command. If these commands are called without a new path, the current path will be returned.

37.4 Creating Packages

Internally a package is simply a gzipped tar file that contains a top level directory of any given name. This directory will in the following be referred to as `package` and may contain the following files

`package/DESCRIPTION`

This is a required file containing information about the package. See [Section 37.4.1 \[The DESCRIPTION File\]](#), page 458, for details on this file.

`package/COPYING`

This is a required file containing the license of the package. No restrictions is made on the license in general. If however the package contains dynamically linked functions the license must be compatible with the GNU General Public License.

package/INDEX

This is an optional file describing the functions provided by the package. If this file is not given then one will be created automatically from the functions in the package and the `Categories` keyword in the `DESCRIPTION` file. See [Section 37.4.2 \[The INDEX file\]](#), page 459, for details on this file.

package/PKG_ADD

An optional file that includes commands that are run when the package is added to the users path. Note that `PKG_ADD` directives in the source code of the package will also be added to this file by the Octave package manager. Note that symbolic links are to be avoided in packages, as symbolic links do not exist on some file systems, and so a typical use for this file is the replacement of the symbolic link

```
ln -s foo.oct bar.oct
```

with an `autoload` directive like

```
autoload ('bar', which ('foo'));
```

See [Section 37.4.3 \[PKG_ADD and PKG_DEL directives\]](#), page 460, for details on `PKG_ADD` directives.

package/PKG_DEL

An optional file that includes commands that are run when the package is removed from the users path. Note that `PKG_DEL` directives in the source code of the package will also be added to this file by the Octave package manager. See [Section 37.4.3 \[PKG_ADD and PKG_DEL directives\]](#), page 460, for details on `PKG_DEL` directives.

package/pre_install.m

This is an optional script that is run prior to the installation of a package.

package/post_install.m

This is an optional script that is run after the installation of a package.

package/on_uninstall.m

This is an optional script that is run prior to the removal of a package.

Besides the above mentioned files, a package can also contain one or more of the following directories

package/inst

An optional directory containing any files that are directly installed by the package. Typically this will include any `m`-files.

package/src

An optional directory containing code that must be built prior to the packages installation. The Octave package manager will execute `./configure` in this directory if this script exists, and will then call `make` if a file `Makefile` exists in this directory. `make install` will however not be called. If a file called `FILES` exist all files listed there will be copied to the `inst` directory, so they also will be installed. If the `FILES` file doesn't exist, `src/*.m` and `src/*.oct` will be copied to the `inst` directory.

package/doc

An optional directory containing documentation for the package. The files in this directory will be directly installed in a sub-directory of the installed package for future reference.

package/bin

An optional directory containing files that will be added to the Octave `EXEC_PATH` when the package is loaded. This might contain external scripts, etc, called by functions within the package.

37.4.1 The DESCRIPTION File

The `DESCRIPTION` file contains various information about the package, such as it's name, author, and version. This file has a very simple format

- Lines starting with `#` are comments.
- Lines starting with a blank character are continuations from the previous line.
- Everything else is of the form `NameOfOption: ValueOfOption`.

The following is a simple example of a `DESCRIPTION` file

```
Name: The name of my package
Version: 1.0.0
Date: 2007-18-04
Author: The name (and possibly email) of the package author.
Maintainer: The name (and possibly email) of the current package maintainer.
Title: The title of the package
Description: A short description of the package. If this description
  gets too long for one line it can continue on the next by adding a
  space to the beginning of the following lines.
License: GPL version 2 or later
```

The package manager currently recognizes the following keywords

Name	Name of the package.
Version	Version of the package.
Date	Date of last update.
Author	Original author of the package.
Maintainer	Maintainer of the package.
Title	A one line description of the package.
Description	A one paragraph description of the package.
Categories	Optional keyword describing the package (if no <code>INDEX</code> file is given this is mandatory).
Problems	Optional list of known problems.

Url	Optional list of homepages related to the package.
Autoload	Optional field that sets the default loading behavior for the package. If set to yes , true or on , then Octave will automatically load the package when starting. Otherwise the package must be manually loaded with the <code>pkg load</code> command. This default behavior can be overridden when the package is installed.
Depends	A list of other Octave packages that this package depends on. This can include dependencies on particular versions, with a format <div style="text-align: center;">Depends: package (>= 1.0.0)</div> Possible operators are <code><</code> , <code><=</code> , <code>==</code> , <code>>=</code> or <code>></code> . If the part of the dependency in <code>()</code> is missing, any version of the package is acceptable. Multiple dependencies can be defined either as a comma separated list or on separate Depends lines.
License	An optional short description of the used license (e.g. GPL version 2 or newer). This is optional since the file <code>COPYING</code> is mandatory.

SystemRequirements

These are the external install dependencies of the package and are not checked by the package manager. This is here as a hint to the distribution packager. They follows the same conventions as the **Depends** keyword.

BuildRequires

These are the external build dependencies of the package and are not checked by the package manager. This is here as a hint to the distribution packager. They follows the same conventions as the **Depends** keyword. Note that in general, packaging systems such as **rpm** or **deb** and autoprobe the install dependencies from the build dependencies, and therefore the often a **BuildRequires** dependency removes the need for a **SystemRequirements** dependency.

The developer is free to add additional arguments to the `DESCRIPTION` file for their own purposes. One further detail to aid the packager is that the **SystemRequirements** and **BuildRequires** keywords can have distribution dependent section, and the automatic build process will use these. An example of the format of this is

```
BuildRequires: libtermcap-devel [Mandriva] libtermcap2-devel
```

where the first package name will be used as a default and if the RPMs are built on a Mandriva distribution, then the second package name will be used instead.

37.4.2 The INDEX file

The optional **INDEX** file provides an categorical view of the functions in the package. This file has a very simple format

- Lines beginning with **#** are comments.
- The first non-comment line should look like this

```
toolbox >> Toolbox name
```

- Lines beginning with an alphabetical character indicates a new category of functions.
- Lines starting with a white space character indicate that the function names on the line belong to last mentioned category.

The format can be summarized with the following example

```
# A comment
toolbox >> Toolbox name
Category Name 1
  function1 function2 function3
  function4
Category Name 2
  functio2 function5
```

37.4.3 PKG_ADD and PKG_DEL directives

If the package contains files called PKG_ADD or PKG_DEL the commands in these files will be executed when the package is added or removed from the users path. In some situations such files are a bit cumbersome to maintain, so the package manager supports automatic creation of such files. If a source file in the package contains a PKG_ADD or PKG_DEL directive they will be added to either the PKG_ADD or PKG_DEL files.

In m-files a PKG_ADD directive looks like this

```
## PKG_ADD: some_octave_command
```

Such lines should be added before the `function` keyword. In C++ files a PKG_ADD directive looks like this

```
// PKG_ADD: some_octave_command
```

In both cases `some_octave_command` should be replaced by the command that should be placed in the PKG_ADD file. PKG_DEL directives work in the same way, except the PKG_ADD keyword is replaced with PKG_DEL and the commands get added to the PKG_DEL file.

Appendix A Dynamically Linked Functions

Octave has the possibility of including compiled code as dynamically linked extensions and then using these extensions as if they were part of Octave itself. Octave has the option of directly calling C++ code through its native oct-file interface or C code through its mex interface. It can also indirectly call functions written in any other language through a simple wrapper. The reasons to write code in a compiled language might be either to link to an existing piece of code and allow it to be used within Octave, or to allow improved performance for key pieces of code.

Before going further, you should first determine if you really need to use dynamically linked functions at all. Before proceeding with writing any dynamically linked function to improve performance you should address ask yourself

- Can I get the same functionality using the Octave scripting language only.
- Is it thoroughly optimized Octave code? Vectorization of Octave code, doesn't just make it concise, it generally significantly improves its performance. Above all, if loops must be used, make sure that the allocation of space for variables takes place outside the loops using an assignment to a like matrix or zeros.
- Does it make as much use as possible of existing built-in library routines? These are highly optimized and many do not carry the overhead of being interpreted.
- Does writing a dynamically linked function represent useful investment of your time, relative to staying in Octave?

Also, as oct- and mex-files are dynamically linked to octave, they introduce to possibility of having Octave abort due to coding errors in the user code. For example a segmentation violation in the users code will cause Octave to abort.

A.1 Oct-Files

A.1.1 Getting Started with Oct-Files

The basic command to build oct-files is `mkoctfile` and it can be call from within octave or from the command line.

`mkoctfile [-options] file ...` [Function File]

The `mkoctfile` function compiles source code written in C, C++, or Fortran. Depending on the options used with `mkoctfile`, the compiled code can be called within Octave or can be used as a stand-alone application.

`mkoctfile` can be called from the shell prompt or from the Octave prompt.

`mkoctfile` accepts the following options, all of which are optional except for the file name of the code you wish to compile:

- '-I DIR' Add the include directory DIR to compile commands.
- '-D DEF' Add the definition DEF to the compiler call.
- '-l LIB' Add the library LIB to the link command.
- '-L DIR' Add the library directory DIR to the link command.

‘-M’
 ‘--depend’ Generate dependency files (.d) for C and C++ source files.
 ‘-c’ Compile but do not link.
 ‘-g’ Enable debugging options for compilers.
 ‘-o FILE’
 ‘--output FILE’ Output file name. Default extension is .oct (or .mex if -mex is specified) unless linking a stand-alone executable.
 ‘-p VAR’
 ‘--print VAR’ Print the configuration variable VAR. Recognized variables are:

ALL_CFLAGS	FFTW_LIBS
ALL_CXXFLAGS	FLIBS
ALL_FFLAGS	FPICFLAG
ALL_LDFLAGS	INCFLAGS
BLAS_LIBS	LD_FLAGS
CC	LD_CXX
CFLAGS	LD_STATIC_FLAG
CPICFLAG	LFLAGS
CPPFLAGS	LIBCRUFT
CXX	LIBOCTAVE
CXXFLAGS	LIBOCTINTERP
CXXPICFLAG	LIBREADLINE
DEPEND_EXTRA_SED_PATTERN	LIBS
DEPEND_FLAGS	OCTAVE_LIBS
DL_LD	RDYNAMIC_FLAG
DL_LDFLAGS	RLD_FLAG
F2C	SED
F2CFLAGS	XTRA_CFLAGS
F77	XTRA_CXXFLAGS
FFLAGS	

‘--link-stand-alone’ Link a stand-alone executable file.
 ‘--mex’ Assume we are creating a MEX file. Set the default output extension to ".mex".
 ‘-s’
 ‘--strip’ Strip the output file.
 ‘-v’
 ‘--verbose’ Echo commands as they are executed.
 ‘file’ The file to compile or link. Recognised file types are

```
.c      C source
.cc     C++ source
.C      C++ source
.cpp    C++ source
.f      Fortran source
.F      Fortran source
.o      object file
```

Consider the short example

```
#include <octave/oct.h>

DEFUN_DLD (helloworld, args, nargout,
  "Hello World Help String")
{
  int nargin = args.length ();
  octave_stdout << "Hello World has " << nargin
    << " input arguments and "
    << nargout << " output arguments.\n";
  return octave_value_list ();
}
```

This example although short introduces the basics of writing a C++ function that can be dynamically linked to Octave. The easiest way to make available most of the definitions that might be necessary for an oct-file in Octave is to use the `#include <octave/oct.h>` header.

The macro that defines the entry point into the dynamically loaded function is `DEFUN_DLD`. This macro takes four arguments, these being

1. The function name as it will be seen in Octave,
2. The list of arguments to the function of type `octave_value_list`,
3. The number of output arguments, which can and often is omitted if not used, and
4. The string that will be seen as the help text of the function.

The return type of functions defined with `DEFUN_DLD` is always `octave_value_list`.

There are a couple of important considerations in the choice of function name. Firstly, it must be a valid Octave function name and so must be a sequence of letters, digits and underscores, not starting with a digit. Secondly, as Octave uses the function name to define the filename it attempts to find the function in, the function name in the `DEFUN_DLD` macro must match the filename of the oct-file. Therefore, the above function should be in a file `'helloworld.cc'`, and it should be compiled to an oct-file using the command

```
mkoctfile helloworld.cc
```

This will create a file call `helloworld.oct`, that is the compiled version of the function. It should be noted that it is perfectly acceptable to have more than one `DEFUN_DLD` function in a source file. However, there must either be a symbolic link to the oct-file for each of the functions defined in the source code with the `DEFUN_DLD` macro or the `autoload` ([Section 12.7 \[Function Files\]](#), [page 124](#)) function should be used.

The rest of this function then shows how to find the number of input arguments, how to print through the octave pager, and return from the function. After compiling this function as above, an example of its use is

```
helloworld (1, 2, 3)
-| Hello World has 3 input arguments and 0 output arguments.
```

A.1.2 Matrices and Arrays in Oct-Files

Octave supports a number of different array and matrix classes, the majority of which are based on the Array class. The exception is the sparse matrix types discussed separately below. There are three basic matrix types

Matrix A double precision matrix class defined in dMatrix.h,

ComplexMatrix
 A complex matrix class defined in CMatrix.h, and

BoolMatrix
 A boolean matrix class defined in boolMatrix.h.

These are the basic two-dimensional matrix types of octave. In addition there are a number of multi-dimensional array types, these being

NDArray A double precision array class defined in 'dNDArray.h'

ComplexNDArray
 A complex array class defined in 'CNDArray.h'

boolNDArray
 A boolean array class defined in 'boolNDArray.h'

int8NDArray
int16NDArray
int32NDArray
int64NDArray
 8, 16, 32 and 64-bit signed array classes defined in 'int8NDArray.h',
 'int16NDArray.h', etc.

uint8NDArray
uint16NDArray
uint32NDArray
uint64NDArray
 8, 16, 32 and 64-bit unsigned array classes defined in 'uint8NDArray.h',
 'uint16NDArray.h', etc.

There are several basic means of constructing matrices of multi-dimensional arrays. Considering the **Matrix** type as an example

- We can create an empty matrix or array with the empty constructor. For example

```
Matrix a;
```

This can be used on all matrix and array types

- Define the dimensions of the matrix or array with a dim_vector. For example

```
dim_vector dv (2);
dv(0) = 2; dv(1) = 2;
Matrix a (dv);
```

This can be used on all matrix and array types

- Define the number of rows and columns in the matrix. For example

```
Matrix a (2, 2)
```

However, this constructor can only be used with the matrix types.

These types all share a number of basic methods and operators, a selection of which include

T& operator () (*octave_idx_type*) [Method]

T& elem (*octave_idx_type*) [Method]

The () operator or **elem** method allow the values of the matrix or array to be read or set. These can take a single argument, which is of type **octave_idx_type**, that is the index into the matrix or array. Additionally, the matrix type allows two argument versions of the () operator and elem method, giving the row and column index of the value to obtain or set.

Note that these function do significant error checking and so in some circumstances the user might prefer the access the data of the array or matrix directly through the **fortran_vec** method discussed below.

octave_idx_type nelem (*void*) *const* [Method]

The total number of elements in the matrix or array.

size_t byte_size (*void*) *const* [Method]

The number of bytes used to store the matrix or array.

dim_vector dims (*void*) *const* [Method]

The dimensions of the matrix or array in value of type **dim_vector**.

void resize (*const dim_vector&*) [Method]

A method taking either an argument of type **dim_vector**, or in the case of a matrix two arguments of type **octave_idx_type** defining the number of rows and columns in the matrix.

T* fortran_vec (*void*) [Method]

This method returns a pointer to the underlying data of the matrix or a array so that it can be manipulated directly, either within Octave or by an external library.

Operators such an **+**, **-**, or ***** can be used on the majority of the above types. In addition there are a number of methods that are of interest only for matrices such as **transpose**, **hermitian**, **solve**, etc.

The typical way to extract a matrix or array from the input arguments of **DEFUN_DLD** function is as follows

```

#include <octave/oct.h>

DEFUN_DLD (addtwomatrices, args, , "Add A to B")
{
  int nargin = args.length ();
  if (nargin != 2)
    print_usage ();
  else
    {
      NDAarray A = args(0).array_value ();
      NDAarray B = args(1).array_value ();
      if (! error_state)
        return octave_value (A + B);
    }
  return octave_value_list ();
}

```

To avoid segmentation faults causing Octave to abort, this function explicitly checks that there are sufficient arguments available before accessing these arguments. It then obtains two multi-dimensional arrays of type `NDAarray` and adds these together. Note that the `array_value` method is called without using the `is_matrix_type` type, and instead the `error_state` is checked before returning `A + B`. The reason to prefer this is that the arguments might be a type that is not an `NDAarray`, but it would make sense to convert it to one. The `array_value` method allows this conversion to be performed transparently if possible, and sets `error_state` if it is not.

`A + B`, operating on two `NDAarray`'s returns an `NDAarray`, which is cast to an `octave_value` on the return from the function. An example of the use of this demonstration function is

```

addtwomatrices (ones (2, 2), ones (2, 2))
⇒  2  2
   2  2

```

A list of the basic `Matrix` and `Array` types, the methods to extract these from an `octave_value` and the associated header is listed below.

<code>RowVector</code>	<code>row_vector_value</code>	<code>'dRowVector.h'</code>
<code>ComplexRowVector</code>	<code>complex_row_vector_value</code>	<code>'CRowVector.h'</code>
<code>ColumnVector</code>	<code>column_vector_value</code>	<code>'dColVector.h'</code>
<code>ComplexColumnVector</code>	<code>complex_column_vector_value</code>	<code>'CColVector.h'</code>
<code>Matrix</code>	<code>matrix_value</code>	<code>'dMatrix.h'</code>
<code>ComplexMatrix</code>	<code>complex_matrix_value</code>	<code>'CMatrix.h'</code>
<code>boolMatrix</code>	<code>bool_matrix_value</code>	<code>'boolMatrix.h'</code>
<code>charMatrix</code>	<code>char_matrix_value</code>	<code>'chMatrix.h'</code>
<code>NDAarray</code>	<code>array_value</code>	<code>'dNDAarray.h'</code>
<code>ComplexNDAarray</code>	<code>complex_array_value</code>	<code>'CNDArray.h'</code>
<code>boolNDAarray</code>	<code>bool_array_value</code>	<code>'boolNDAarray.h'</code>
<code>charNDAarray</code>	<code>char_array_value</code>	<code>'charNDAarray.h'</code>
<code>int8NDAarray</code>	<code>int8_array_value</code>	<code>'int8NDAarray.h'</code>
<code>int16NDAarray</code>	<code>int16_array_value</code>	<code>'int16NDAarray.h'</code>
<code>int32NDAarray</code>	<code>int32_array_value</code>	<code>'int32NDAarray.h'</code>

<code>int64NDArray</code>	<code>int64_array_value</code>	<code>'int64NDArray.h'</code>
<code>uint8NDArray</code>	<code>uint8_array_value</code>	<code>'uint8NDArray.h'</code>
<code>uint16NDArray</code>	<code>uint16_array_value</code>	<code>'uint16NDArray.h'</code>
<code>uint32NDArray</code>	<code>uint32_array_value</code>	<code>'uint32NDArray.h'</code>
<code>uint64NDArray</code>	<code>uint64_array_value</code>	<code>'uint64NDArray.h'</code>

A.1.3 Character Strings in Oct-Files

In Octave a character string is just a special `Array` class. Consider the example

```
#include <octave/oct.h>

DEFUN_DLD (stringdemo, args, , "String Demo")
{
    int nargin = args.length();
    octave_value_list retval;

    if (nargin != 1)
        print_usage ();
    else
    {
        charMatrix ch = args(0).char_matrix_value ();

        if (! error_state)
        {
            if (args(0).is_sq_string ())
                retval(1) = octave_value (ch, true);
            else
                retval(1) = octave_value (ch, true, '\\');

            octave_idx_type nr = ch.rows();
            for (octave_idx_type i = 0; i < nr / 2; i++)
            {
                std::string tmp = ch.row_as_string (i);
                ch.insert (ch.row_as_string(nr-i-1).c_str(), i, 0);
                ch.insert (tmp.c_str(), nr-i-1, 0);
            }
            retval(0) = octave_value (ch, true);
        }
    }
    return retval;
}
```

An example of the use of this function is

```

s0 = ["First String"; "Second String"];
[s1,s2] = stringdemo (s0)
⇒ s1 = Second String
    First String

⇒ s2 = First String
    Second String

typeinfo (s2)
⇒ sq_string
typeinfo (s1)
⇒ string

```

One additional complication of strings in Octave is the difference between single quoted and double quoted strings. To find out if an `octave_value` contains a single or double quoted string an example is

```

if (args(0).is_sq_string ())
    octave_stdout << "First argument is a singularly quoted string\n";
else if (args(0).is_dq_string ())
    octave_stdout << "First argument is a doubly quoted string\n";

```

Note however, that both types of strings are represented by the `charNDArray` type, and so when assigning to an `octave_value`, the type of string should be specified. For example

```

octave_value_list retval;
charNDArray c;
...
// Create single quoted string
retval(1) = octave_value (ch, true, '\''');

// Create a double quoted string
retval(0) = octave_value (ch, true);

```

A.1.4 Cell Arrays in Oct-Files

Octave's cell type is equally accessible within an oct-files. A cell array is just an array of `octave_values`, and so each element of the cell array can then be treated just like any other `octave_value`. A simple example is


```

#include <octave/oct.h>
#include <octave/Cell.h>

DEFUN_DLD (celldemo, args, , "Cell Demo")
{
    octave_value_list retval;
    int nargin = args.length ();

    if (nargin != 1)
        print_usage ();
    else
    {
        Cell c = args (0).cell_value ();
        if (! error_state)
            for (octave_idx_type i = 0; i < c.nelem (); i++)
                retval(i) = c.elem (i);
    }
    return retval;
}

```

Note that cell arrays are used less often in standard oct-files and so the ‘Cell.h’ header file must be explicitly included. The rest of this example extracts the `octave_values` one by one from the cell array and returns be as individual return arguments. For example consider

```

[b1, b2, b3] = celldemo ({1, [1, 2], "test"})
⇒
b1 =  1
b2 =

    1    2

b3 = test

```

A.1.5 Structures in Oct-Files

A structure in Octave is map between a number of fields represented and their values. The Standard Template Library `map` class is used, with the pair consisting of a `std::string` and an octave `Cell` variable.

A simple example demonstrating the use of structures within oct-files is

```

#include <octave/oct.h>
#include <octave/ov-struct.h>

DEFUN_DLD (structdemo, args, , "Struct demo.")
{
  int nargin = args.length ();
  octave_value retval;

  if (nargin != 2)
    print_usage ();
  else
    {
      Octave_map arg0 = args(0).map_value ();
      std::string arg1 = args(1).string_value ();

      if (! error_state && arg0.contains (arg1))
        {
          // The following two lines might be written as
          //   octave_value tmp;
          //   for (Octave_map::iterator p0 = arg0.begin() ;
          //        p0 != arg0.end(); p0++ )
          //     if (arg0.key (p0) == arg1)
          //       {
          //         tmp = arg0.contents (p0) (0);
          //         break;
          //       }
          // though using seek is more concise.
          Octave_map::const_iterator p1 = arg0.seek (arg1);
          octave_value tmp = arg0.contents(p1)(0);
          Octave_map st;
          st.assign ("selected", tmp);
          retval = octave_value (st);
        }
    }
  return retval;
}

```

An example of its use is

```

x.a = 1; x.b = "test"; x.c = [1, 2];
structdemo (x, "b")
⇒ selected = test

```

The commented code above demonstrates how to iterated over all of the fields of the structure, where as the following code demonstrates finding a particular field in a more concise manner.

As can be seen the `contents` method of the `Octave_map` class returns a `Cell` which allows structure arrays to be represented. Therefore, to obtain the underlying `octave_value` we write

```
octave_value tmp = arg0.contents (p1) (0);
```

where the trailing `(0)` is the `()` operator on the `Cell` object. We can equally iterate of the elements of the `Cell` array to address the elements of the structure array.

A.1.6 Sparse Matrices in Oct-Files

There are three classes of sparse objects that are of interest to the user.

SparseMatrix

A double precision sparse matrix class

SparseComplexMatrix

A complex sparse matrix class

SparseBoolMatrix

A boolean sparse matrix class

All of these classes inherit from the `Sparse<T>` template class, and so all have similar capabilities and usage. The `Sparse<T>` class was based on Octave `Array<T>` class, and so users familiar with Octave's `Array` classes will be comfortable with the use of the sparse classes.

The sparse classes will not be entirely described in this section, due to their similarity with the existing `Array` classes. However, there are a few differences due the different nature of sparse objects, and these will be described. Firstly, although it is fundamentally possible to have N-dimensional sparse objects, the Octave sparse classes do not allow them at this time. So all operations of the sparse classes must be 2-dimensional. This means that in fact `SparseMatrix` is similar to Octave's `Matrix` class rather than its `NDArray` class.

A.1.6.1 The Differences between the Array and Sparse Classes

The number of elements in a sparse matrix is considered to be the number of non-zero elements rather than the product of the dimensions. Therefore

```
SparseMatrix sm;
...
int nel = sm.nelem ();
```

returns the number of non-zero elements. If the user really requires the number of elements in the matrix, including the non-zero elements, they should use `numel` rather than `nelem`. Note that for very large matrices, where the product of the two dimensions is large that the representation of the an unsigned int, then `numel` can overflow. An example is `speye(1e6)` which will create a matrix with a million rows and columns, but only a million non-zero elements. Therefore the number of rows by the number of columns in this case is more than two hundred times the maximum value that can be represented by an unsigned int. The use of `numel` should therefore be avoided unless it is known it won't overflow.

Extreme care must be take with the `elem` method and the `"()"` operator, which perform basically the same function. The reason is that if a sparse object is non-const, then Octave will assume that a request for a zero element in a sparse matrix is in fact a request to create this element so it can be filled. Therefore a piece of code like

```

SparseMatrix sm;
...
for (int j = 0; j < nc; j++)
  for (int i = 0; i < nr; i++)
    std::cerr << " (" << i << ", " << j << "): " << sm(i,j)
    << std::endl;

```

is a great way of turning the sparse matrix into a dense one, and a very slow way at that since it reallocates the sparse object at each zero element in the matrix.

An easy way of preventing the above from happening is to create a temporary constant version of the sparse matrix. Note that only the container for the sparse matrix will be copied, while the actual representation of the data will be shared between the two versions of the sparse matrix. So this is not a costly operation. For example, the above would become

```

SparseMatrix sm;
...
const SparseMatrix tmp (sm);
for (int j = 0; j < nc; j++)
  for (int i = 0; i < nr; i++)
    std::cerr << " (" << i << ", " << j << "): " << tmp(i,j)
    << std::endl;

```

Finally, as the sparse types aren't just represented as a contiguous block of memory, the `fortran_vec` method of the `Array<T>` is not available. It is however replaced by three separate methods `ridx`, `cidx` and `data`, that access the raw compressed column format that the Octave sparse matrices are stored in. Additionally, these methods can be used in a manner similar to `elem`, to allow the matrix to be accessed or filled. However, in that case it is up to the user to respect the sparse matrix compressed column format discussed previous.

A.1.6.2 Creating Sparse Matrices in Oct-Files

You have several alternatives for creating a sparse matrix. You can first create the data as three vectors representing the row and column indexes and the data, and from those create the matrix. Or alternatively, you can create a sparse matrix with the appropriate amount of space and then fill in the values. Both techniques have their advantages and disadvantages.

Here is an example of how to create a small sparse matrix with the first technique

```

int nz = 4, nr = 3, nc = 4;

ColumnVector ridx (nz);
ColumnVector cidx (nz);
ColumnVector data (nz);

ridx(0) = 0; ridx(1) = 0; ridx(2) = 1; ridx(3) = 2;
cidx(0) = 0; cidx(1) = 1; cidx(2) = 3; cidx(3) = 3;
data(0) = 1; data(1) = 2; data(2) = 3; data(3) = 4;

SparseMatrix sm (data, ridx, cidx, nr, nc);

```

which creates the matrix given in section [Section 21.1.1 \[Storage\]](#), page 245. Note that the compressed matrix format is not used at the time of the creation of the matrix itself, however it is used internally.

As previously mentioned, the values of the sparse matrix are stored in increasing column-major ordering. Although the data passed by the user does not need to respect this requirement, the pre-sorting the data significantly speeds up the creation of the sparse matrix.

The disadvantage of this technique of creating a sparse matrix is that there is a brief time where two copies of the data exists. Therefore for extremely memory constrained problems this might not be the right technique to create the sparse matrix.

The alternative is to first create the sparse matrix with the desired number of non-zero elements and then later fill those elements in. The easiest way to do this is

```
int nz = 4, nr = 3, nc = 4;
SparseMatrix sm (nr, nc, nz);
sm(0,0) = 1; sm(0,1) = 2; sm(1,3) = 3; sm(2,3) = 4;
```

That creates the same matrix as previously. Again, although it is not strictly necessary, it is significantly faster if the sparse matrix is created in this manner that the elements are added in column-major ordering. The reason for this is that if the elements are inserted at the end of the current list of known elements then no element in the matrix needs to be moved to allow the new element to be inserted. Only the column indexes need to be updated.

There are a few further points to note about this technique of creating a sparse matrix. Firstly, it is possible to create a sparse matrix with fewer elements than are actually inserted in the matrix. Therefore

```
int nz = 4, nr = 3, nc = 4;
SparseMatrix sm (nr, nc, 0);
sm(0,0) = 1; sm(0,1) = 2; sm(1,3) = 3; sm(2,3) = 4;
```

is perfectly valid. However it is a very bad idea. The reason is that as each new element is added to the sparse matrix the space allocated to it is increased by reallocating the memory. This is an expensive operation, that will significantly slow this means of creating a sparse matrix. Furthermore, it is possible to create a sparse matrix with too much storage, so having *nz* above equaling 6 is also valid. The disadvantage is that the matrix occupies more memory than strictly needed.

It is not always easy to know the number of non-zero elements prior to filling a matrix. For this reason the additional storage for the sparse matrix can be removed after its creation with the *maybe_compress* function. Furthermore, the *maybe_compress* can deallocate the unused storage, but it can equally remove zero elements from the matrix. The removal of zero elements from the matrix is controlled by setting the argument of the *maybe_compress* function to be 'true'. However, the cost of removing the zeros is high because it implies resorting the elements. Therefore, if possible it is better if the user doesn't add the zeros in the first place. An example of the use of *maybe_compress* is

```

int nz = 6, nr = 3, nc = 4;

SparseMatrix sm1 (nr, nc, nz);
sm1(0,0) = 1; sm1(0,1) = 2; sm1(1,3) = 3; sm1(2,3) = 4;
sm1.maybe_compress (); // No zero elements were added

SparseMatrix sm2 (nr, nc, nz);
sm2(0,0) = 1; sm2(0,1) = 2; sm(0,2) = 0; sm(1,2) = 0;
sm1(1,3) = 3; sm1(2,3) = 4;
sm2.maybe_compress (true); // Zero elements were added

```

The use of the *maybe_compress* function should be avoided if possible, as it will slow the creation of the matrices.

A third means of creating a sparse matrix is to work directly with the data in compressed row format. An example of this technique might be

```

octave_value arg;
...
int nz = 6, nr = 3, nc = 4; // Assume we know the max no nz
SparseMatrix sm (nr, nc, nz);
Matrix m = arg.matrix_value ();

int ii = 0;
sm.cidx (0) = 0;
for (int j = 1; j < nc; j++)
{
    for (int i = 0; i < nr; i++)
    {
        double tmp = foo (m(i,j));
        if (tmp != 0.)
        {
            sm.data(ii) = tmp;
            sm.ridx(ii) = i;
            ii++;
        }
    }
    sm.cidx(j+1) = ii;
}
sm.maybe_compress (); // If don't know a-priori the final no of nz.

```

which is probably the most efficient means of creating the sparse matrix.

Finally, it might sometimes arise that the amount of storage initially created is insufficient to completely store the sparse matrix. Therefore, the method **change_capacity** exists to reallocate the sparse memory. The above example would then be modified as

```

octave_value arg;
...
int nz = 6, nr = 3, nc = 4;    // Assume we know the max no nz
SparseMatrix sm (nr, nc, nz);
Matrix m = arg.matrix_value ();

int ii = 0;
sm.cidx (0) = 0;
for (int j = 1; j < nc; j++)
{
    for (int i = 0; i < nr; i++)
    {
        double tmp = foo (m(i,j));
        if (tmp != 0.)
        {
            if (ii == nz)
            {
                nz += 2;    // Add 2 more elements
                sm.change_capacity (nz);
            }
            sm.data(ii) = tmp;
            sm.ridx(ii) = i;
            ii++;
        }
    }
    sm.cidx(j+1) = ii;
}
sm.maybe_mutate ();    // If don't know a-priori the final no of nz.

```

Note that both increasing and decreasing the number of non-zero elements in a sparse matrix is expensive, as it involves memory reallocation. Also as parts of the matrix, though not its entirety, exist as the old and new copy at the same time, additional memory is needed. Therefore if possible this should be avoided.

A.1.6.3 Using Sparse Matrices in Oct-Files

Most of the same operators and functions on sparse matrices that are available from the Octave are equally available with oct-files. The basic means of extracting a sparse matrix from an `octave_value` and returning them as an `octave_value`, can be seen in the following example

```

octave_value_list retval;

SparseMatrix sm = args(0).sparse_matrix_value ();
SparseComplexMatrix scm = args(1).sparse_complex_matrix_value ();
SparseBoolMatrix sbm = args(2).sparse_bool_matrix_value ();
...
retval(2) = sbm;
retval(1) = scm;
retval(0) = sm;

```

The conversion to an octave-value is handled by the sparse `octave_value` constructors, and so no special care is needed.

A.1.7 Accessing Global Variables in Oct-Files

Global variables allow variables in the global scope to be accessed. Global variables can easily be accessed with oct-files using the support functions `get_global_value` and `set_global_value`. `get_global_value` takes two arguments, the first is a string representing the variable name to obtain. The second argument is a boolean argument specifying what to do in the case that no global variable of the desired name is found. An example of the use of these two functions is

```

#include <octave/oct.h>

DEFUN_DLD (globaldemo, args, , "Global demo.")
{
  int nargin = args.length ();
  octave_value retval;

  if (nargin != 1)
    print_usage ();
  else
  {
    std::string s = args(0).string_value ();
    if (! error_state)
    {
      octave_value tmp = get_global_value (s, true);
      if (tmp.is_defined ())
        retval = tmp;
      else
        retval = "Global variable not found";

      set_global_value ("a", 42.0);
    }
  }
  return retval;
}

```

An example of its use is


```
global a b
b = 10;
globaldemo ("b")
⇒ 10
globaldemo ("c")
⇒ "Global variable not found"
num2str (a)
⇒ 42
```

A.1.8 Calling Octave Functions from Oct-Files

There is often a need to be able to call another octave function from within an oct-file, and there are many examples of such within octave itself. For example the `quad` function is an oct-file that calculates the definite integral by quadrature over a user supplied function.

There are also many ways in which a function might be passed. It might be passed as one of

1. Function Handle
2. Anonymous Function Handle
3. Inline Function
4. String

The example below demonstrates an example that accepts all four means of passing a function to an oct-file.

```

#include <octave/oct.h>
#include <octave/parse.h>

DEFUN_DLD (funcdemo, args, nargout, "Function Demo")
{
  int nargin = args.length();
  octave_value_list retval;

  if (nargin < 2)
    print_usage ();
  else
    {
      octave_value_list newargs;
      for (octave_idx_type i = nargin - 1; i > 0; i--)
        newargs (i - 1) = args(i);
      if (args(0).is_function_handle ()
          || args(0).is_inline_function ())
        {
          octave_function *fcn = args(0).function_value ();
          if (! error_state)
            retval = feval (fcn, newargs, nargout);
        }
      else if (args(0).is_string ())
        {
          std::string fcn = args (0).string_value ();
          if (! error_state)
            retval = feval (fcn, newargs, nargout);
        }
      else
        error ("funcdemo: expected string, inline or function handle");
    }
  return retval;
}

```

The first argument to this demonstration is the user supplied function and the following arguments are all passed to the user function.

```

funcdemo (@sin,1)
⇒ 0.84147
funcdemo (@(x) sin(x), 1)
⇒ 0.84147
funcdemo (inline ("sin(x)"), 1)
⇒ 0.84147
funcdemo ("sin",1)
⇒ 0.84147
funcdemo (@atan2, 1, 1)
⇒ 0.78540

```

When the user function is passed as a string, the treatment of the function is different. In some cases it is necessary to always have the user supplied function as an `octave_function` object. In that case the string argument can be used to create a temporary function like

```
std::octave fcn_name = unique_symbol_name ("__fcn__");
std::string fname = "function y = ";
fname.append (fcn_name);
fname.append ("(x) y = ");
fcn = extract_function (args(0), "funcdemo", fcn_name,
                       fname, "; endfunction");
...
if (fcn_name.length ())
    clear_function (fcn_name);
```

There are two important things to know in this case. The number of input arguments to the user function is fixed, and in the above is a single argument, and secondly to avoid leaving the temporary function in the Octave symbol table it should be cleared after use.

A.1.9 Calling External Code from Oct-Files

Linking external C code to Octave is relatively simple, as the C functions can easily be called directly from C++. One possible issue is the declarations of the external C functions might need to be explicitly defined as C functions to the compiler. If the declarations of the external C functions are in the header `foo.h`, then the manner in which to ensure that the C++ compiler treats these declarations as C code is

```
#ifdef __cplusplus
extern "C"
{
#endif
#include "foo.h"
#ifdef __cplusplus
} /* end extern "C" */
#endif
```

Calling Fortran code however can pose some difficulties. This is due to differences in the manner in which compilers treat the linking of Fortran code with C or C++ code. Octave supplies a number of macros that allow consistent behavior across a number of compilers.

The underlying Fortran code should use the `XSTOPX` function to replace the Fortran `STOP` function. `XSTOPX` uses the Octave exception handler to treat failing cases in the Fortran code explicitly. Note that Octave supplies its own replacement blas `XERBLA` function, which uses `XSTOPX`.

If the underlying code calls `XSTOPX`, then the `F77_XFCN` macro should be used to call the underlying Fortran function. The Fortran exception state can then be checked with the global variable `f77_exception_encountered`. If `XSTOPX` will not be called, then the `F77_FCN` macro should be used instead to call the Fortran code.

There is no harm in using `F77_XFCN` in all cases, except that for Fortran code that is short running and executes a large number of times, there is potentially an overhead in doing so. However, if `F77_FCN` is used with code that calls `XSTOP`, Octave can generate a segmentation fault.

An example of the inclusion of a Fortran function in an oct-file is given in the following example, where the C++ wrapper is

```
#include <octave/oct.h>
#include <octave/f77-fcn.h>

extern "C"
{
    F77_RET_T
    F77_FUNC (fortsub, FORTSUB)
        (const int&, double*, F77_CHAR_ARG_DECL
         F77_CHAR_ARG_LEN_DECL);
}

DEFUN_DLD (fortdemo , args , , "Fortran Demo.")
{
    octave_value_list retval;
    int nargin = args.length();
    if (nargin != 1)
        print_usage ();
    else
    {
        NDArray a = args(0).array_value ();
        if (! error_state)
        {
            double *av = a.fortran_vec ();
            octave_idx_type na = a.nelem ();
            OCTAVE_LOCAL_BUFFER (char, ctmp, 128);

            F77_XFCN (fortsub, FORTSUB, (na, av, ctmp
                                         F77_CHAR_ARG_LEN (128)));

            if (f77_exception_encountered)
                error ("fortdemo: error in fortran");
            else
            {
                retval(1) = std::string (ctmp);
                retval(0) = a;
            }
        }
    }
    return retval;
}
```

and the fortran function is

```

subroutine fortdemo (n, a, s)
implicit none
character*(*) s
real*8 a(*)
integer*4 i, n, ioerr
do i = 1, n
  if (a(i) .eq. 0d0) then
    call xstopx ('fortdemo: divide by zero')
  else
    a(i) = 1d0 / a(i)
  endif
enddo
write (unit = s, fmt = '(a,i3,a,a)', iostat = ioerr)
$      'There are ', n, ' values in the input vector',
$      char(0)
if (ioerr .ne. 0) then
  call xstopx ('fortdemo: error writing string')
endif
return
end

```

This example demonstrates most of the features needed to link to an external Fortran function, including passing arrays and strings, as well as exception handling. An example of the behavior of this function is

```

[b, s] = fortdemo (1:3)
⇒
b = 1.00000    0.50000    0.33333
s = There are    3 values in the input vector
[b, s] = fortdemo(0:3)
error: fortdemo:divide by zero
error: exception encountered in Fortran subroutine fortdemo_
error: fortdemo: error in fortran

```

A.1.10 Allocating Local Memory in Oct-Files

Allocating memory within an oct-file might seem easy as the C++ new/delete operators can be used. However, in that case care must be taken to avoid memory leaks. The preferred manner in which to allocate memory for use locally is to use the `OCTAVE_LOCAL_BUFFER` macro. An example of its use is

```

OCTAVE_LOCAL_BUFFER (double, tmp, len)

that returns a pointer tmp of type double * of length len.

```

A.1.11 Input Parameter Checking in Oct-Files

As oct-files are compiled functions they have the possibility of causing Octave to abort abnormally. It is therefore important that the input parameters to each and every function has the minimum of parameter checking needed to ensure that Octave behaves well.

The minimum requirement, as previously discussed, is to check the number of input arguments before using them to avoid referencing a non existent argument. However, in some cases this might not be sufficient as the underlying code imposes further constraints. For example an external function call might be undefined if the input arguments are not integers, or if one of the arguments is zero. Therefore, oct-files often need additional input parameter checking.

There are several functions within Octave that might be useful for the purposes of parameter checking. These include the methods of the `octave_value` class like `is_real_matrix`, etc, but equally include more specialized functions. Some of the more common ones are demonstrated in the following example

```
#include <octave/oct.h>

DEFUN_DLD (paramdemo, args, nargout, "Parameter Check Demo.")
{
    int nargin = args.length ();
    octave_value retval;

    if (nargin != 1)
        print_usage();
    else if (nargout != 0)
        error ("paramdemo: function has no output arguments");
    else
    {
        NDArray m = args(0).array_value();
        double min_val = -10.0;
        double max_val = 10.0;
        octave_stdout << "Properties of input array:\n";
        if (m.any_element_is_negative ())
            octave_stdout << "    includes negative values\n";
        if (m.any_element_is_inf_or_nan())
            octave_stdout << "    includes Inf or NaN values\n";
        if (m.any_element_not_one_or_zero())
            octave_stdout << "    includes other values than 1 and 0\n";
        if (m.all_elements_are_int_or_inf_or_nan())
            octave_stdout << "    includes only int, Inf or NaN values\n";
        if (m.all_integers (min_val, max_val))
            octave_stdout << "    includes only integers in [-10,10]\n";
    }
    return retval;
}
```

and an example of its use is

```
paramdemo ([1, 2, NaN, Inf])
⇒ Properties of input array:
    includes Inf or NaN values
    includes other values than 1 and 0
    includes only int, Inf or NaN values
```

A.1.12 Exception and Error Handling in Oct-Files

Another important feature of Octave is its ability to react to the user typing *Control-C* even during calculations. This ability is based on the C++ exception handler, where memory allocated by the C++ new/delete methods are automatically released when the exception is treated. When writing an oct-file, to allow Octave to treat the user typing *Control-C*, the `OCTAVE_QUIT` macro is supplied. For example

```
for (octave_idx_type i = 0; i < a.nelem (); i++)
{
    OCTAVE_QUIT;
    b.elem(i) = 2. * a.elem(i);
}
```

The presence of the `OCTAVE_QUIT` macro in the inner loop allows Octave to treat the user request with the *Control-C*. Without this macro, the user must either wait for the function to return before the interrupt is processed, or press *Control-C* three times to force Octave to exit.

The `OCTAVE_QUIT` macro does impose a very small speed penalty, and so for loops that are known to be small it might not make sense to include `OCTAVE_QUIT`.

When creating an oct-file that uses an external libraries, the function might spend a significant portion of its time in the external library. It is not generally possible to use the `OCTAVE_QUIT` macro in this case. The alternative in this case is

```
BEGIN_INTERRUPT_IMMEDIATELY_IN_FOREIGN_CODE;
... some code that calls a "foreign" function ...
END_INTERRUPT_IMMEDIATELY_IN_FOREIGN_CODE;
```

The disadvantage of this is that if the foreign code allocates any memory internally, then this memory might be lost during an interrupt, without being deallocated. Therefore, ideally Octave itself should allocate any memory that is needed by the foreign code, with either the `fortran_vec` method or the `OCTAVE_LOCAL_BUFFER` macro.

The Octave `unwind_protect` mechanism ([Section 11.8 \[The `unwind_protect` Statement\]](#), [page 114](#)) can also be used in oct-files. In conjunction with the exception handling of Octave, it is important to enforce that certain code is run to allow variables, etc to be restored even if an exception occurs. An example of the use of this mechanism is

```

#include <octave/oct.h>
#include <octave/unwind-prot.h>

void
err_hand (const char *fmt, ...)
{
    // Do nothing!!
}

DEFUN_DLD (unwinddemo, args, nargout, "Unwind Demo")
{
    int nargin = args.length();
    octave_value retval;
    if (nargin < 2)
        print_usage ();
    else
    {
        NDArray a = args(0).array_value ();
        NDArray b = args(1).array_value ();

        if (! error_state)
        {
            unwind_protect::begin_frame ("Funwinddemo");
            unwind_protect_ptr (current_liboctave_warning_handler);
            set_liboctave_warning_handler(err_hand);
            retval = octave_value (quotient (a, b));
            unwind_protect::run_frame ("Funwinddemo");
        }
    }
    return retval;
}

```

As can be seen in the example

```

unwinddemo (1, 0)
⇒ Inf
1 / 0
⇒ warning: division by zero
Inf

```

The division by zero (and in fact all warnings) is disabled in the `unwinddemo` function.

A.1.13 Documentation and Test of Oct-Files

The documentation of an oct-file is the fourth string parameter of the `DEFUN_DLD` macro. This string can be formatted in the same manner as the help strings for user functions ([Section C.5 \[Documentation Tips\]](#), page 512), however there are some issues that are particular to the formatting of help strings within oct-files.

The major issue is that the help string will typically be longer than a single line of text, and so the formatting of long help strings need to be taken into account. There are several

manner in which to treat this issue, but the most common is illustrated in the following example

```
DEFUN_DLD (do_what_i_want, args, nargout,
  "/*- texinfo -*-\n\
  @deftypefn {Function File} {} do_what_i_say (@var{n})\n\
  A function that does what the user actually wants rather than what\n\
  they requested.\n\
  @end deftypefn")
{
  ...
}
```

where, as can be seen, end line of text within the help string is terminated by `\n\` which is an an embedded new-line in the string together with a C++ string continuation character. Note that the final `\` must be the last character on the line.

Octave also includes the ability to embed the test and demonstration code for a function within the code itself ([Appendix B \[Test and Demo Functions\]](#), page 501). This can be used from within oct-files (or in fact any file) with certain provisos. Firstly, the test and demo functions of Octave look for a `%!` as the first characters on a new-line to identify test and demonstration code. This is equally a requirement for oct-files. Furthermore the test and demonstration code must be included in a comment block of the compiled code to avoid it being interpreted by the compiler. Finally, the Octave test and demonstration code must have access to the source code of the oct-file and not just the compiled code as the tests are stripped from the compiled code. An example in an oct-file might be

```
/*

%!error (sin())
%!error (sin(1,1))
%!assert (sin([1,2]),[sin(1),sin(2)])

*/
```

A.2 Mex-Files

Octave includes an interface to allow legacy mex-files to be compiled and used with Octave. This interface can also be used to share code between Octave and non Octave users. However, as mex-files expose the internal API of an alternative product to Octave, and the internal structure of Octave is different to this product, a mex-file can never have the same performance in Octave as the equivalent oct-file. In particular to support the manner in which mex-files access the variables passed to mex functions, there are a significant number of additional copies of memory when calling or returning from a mex function. For this reason, new code should be written using the oct-file interface discussed above if possible.

A.2.1 Getting Started with Mex-Files

The basic command to build a mex-file is either `mkoctfile --mex` or `mex`. The first can either be used from within Octave or from the commandline. However, to avoid issues with the installation of other products, the use of the command `mex` is limited to within Octave.

mex [*options*] *file* ... [Function File]
 Compile source code written in C, C++, or Fortran, to a MEX file. This is equivalent to `mkoctfile --mex [options] file`.

See also: `mkoctfile`.

mexext () [Function File]
 Return the filename extension used for MEX files.

One important difference between the use of `mex` with other products and with Octave is that the header file "matrix.h" is implicitly included through the inclusion of "mex.h". This is to avoid a conflict with the Octave file "Matrix.h" with operating systems and compilers that don't distinguish between filenames in upper and lower case

Consider the short example

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    mxArray *v = mxCreateDoubleMatrix (1, 1, mxREAL);
    double *data = mxGetPr (v);
    *data = 1.23456789;
    plhs[0] = v;
}
```

This simple example demonstrates the basics of writing a mex-file. The entry point into the mex-file is defined by `mexFunction`. Note that the function name is not explicitly included in the `mexFunction` and so there can only be a single `mexFunction` entry point per-file. Also the name of the function is determined by the name of the mex-file itself. Therefore if the above function is in the file 'firstmexdemo.c', it can be compiled with

```
mkoctfile --mex firstmexdemo.c
```

which creates a file 'firstmexdemo.mex'. The function can then be run from Octave as

```
firstmexdemo()
⇒ 1.2346
```

It should be noted that the mex-file contains no help string for the functions it contains. To document mex-files, there should exist an m-file in the same directory as the mex-file itself. Taking the above as an example, we would therefore have a file 'firstmexdemo.m' that might contain the text

```
%FIRSTMEDEMO Simple test of the functionality of a mex-file.
```

In this case, the function that will be executed within Octave will be given by the mex-file, while the help string will come from the m-file. This can also be useful to allow a sample implementation of the mex-file within the Octave language itself for testing purposes.

Although we can not have multiple entry points into a single mex-file, we can use the `mexFunctionName` function to determine what name the mex-file was called with. This can be used to alter the behavior of the mex-file based on the function name. For example if

```

#include "mex.h"

void
mexFunction (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    const char *nm;
    nm = mexFunctionName ();
    mexPrintf ("You called function: %s\n", nm);
    if (strcmp (nm, "myfunc") == 0)
        mexPrintf ("This is the principal function\n", nm);
    return;
}

```

is in file ‘myfunc.c’, and it is compiled with

```

mkoctfile --mex myfunc.c
ln -s myfunc.mex myfunc2.mex

```

Then as can be seen by

```

myfunc()
⇒ You called function: myfunc
   This is the principal function
myfunc2()
⇒ You called function: myfunc2

```

the behavior of the mex-file can be altered depending on the functions name.

Allow the user should only include `mex.h` in their code, Octave declares additional functions, typedefs, etc, available to the user to write mex-files in the headers `mexproto.h` and `mxarray.h`.

A.2.2 Working with Matrices and Arrays in Mex-Files

The basic mex type of all variables is `mxArray`. All variables, such as matrices, cell arrays or structures are all stored in this basic type, and this type serves basically the same purpose as the `octave_value` class in oct-files. That is it acts as a container for the more specialized types.

The `mxArray` structure contains at a minimum, the variable it represents name, its dimensions, its type and whether the variable is real or complex. It can however contain a number of additional fields depending on the type of the `mxArray`. There are a number of functions to create `mxArray` structures, including `mxCreateCellArray`, `mxCreateSparse` and the generic `mxCreateNumericArray`.

The basic functions to access the data contained in an array is `mxGetPr`. As the mex interface assumes that the real and imaginary parts of a complex array are stored separately, there is an equivalent function `mxGetPi` that get the imaginary part. Both of these functions are for use only with double precision matrices. There also exists the generic function `mxGetData` and `mxGetImagData` that perform the same operation on all matrix types. For example

```
mxArray *m;  
mwSize *dims;  
UINT32_T *pr;  
  
dims = (mwSize *) mxMalloc (2 * sizeof(mwSize));  
dims[0] = 2;  
dims[1] = 2;  
m = mxCreateNumericArray (2, dims, mxUINT32_CLASS, mxREAL);  
pr = (UINT32_T *) mxGetData (m);
```

There are also the functions `mxSetPr`, etc, that perform the inverse, and set the data of an Array to use the block of memory pointed to by the argument of `mxSetPr`.

Note the type `mwSize` used above, and `mwIndex` are defined as the native precision of the indexing in Octave on the platform on which the mex-file is built. This allows both 32- and 64-bit platforms to support mex-files. `mwSize` is used to define array dimension and maximum number of elements, while `mwIndex` is used to define indexing into arrays.

An example that demonstrates how to work with arbitrary real or complex double precision arrays is given by the file ‘`mypow2.c`’ as given below.

```

#include "mex.h"

void
mexFunction (int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
{
    mwIndex i;
    mwSize n;
    double *vri, *vro;

    if (nrhs != 1 || ! mxIsNumeric (prhs[0]))
        mexErrMsgTxt ("expects matrix");

    n = mxGetNumberOfElements (prhs[0]);
    plhs[0] = (mxArray *) mxCreateNumericArray
        (mxGetNumberOfDimensions (prhs[0]),
         mxGetDimensions (prhs[0]), mxGetClassID (prhs[0]),
         mxIsComplex (prhs[0]));
    vri = mxGetPr (prhs[0]);
    vro = mxGetPr (plhs[0]);

    if (mxIsComplex (prhs[0]))
    {
        double *vii, *vio;
        vii = mxGetPi (prhs[0]);
        vio = mxGetPi (plhs[0]);

        for (i = 0; i < n; i++)
        {
            vro [i] = vri [i] * vri [i] - vii [i] * vii [i];
            vio [i] = 2 * vri [i] * vii [i];
        }
    }
    else
    {
        for (i = 0; i < n; i++)
            vro [i] = vri [i] * vri [i];
    }
}

```

with an example of its use

```

b = randn(4,1) + 1i * randn(4,1);
all(b.^2 == mypow2(b))
⇒ 1

```

The example above uses the `mxGetNumberOfElements`, `mxGetNumberOfDimensions` and `mxGetDimensions`, to work with the dimensional parameters of multi-dimensional arrays. There also exists the functions `mxGetM`, and `mxGetN` that probe the number of rows and columns in a matrix.

A.2.3 Character Strings in Mex-Files

As mex-files do not make the distinction between single and double quoted strings within Octave, there is perhaps less complexity in the use of strings and character matrices in mex-files. An example of their use, that parallels the demo in ‘stringdemo.cc’, is given in the file ‘mystring.c’, as seen below.

```
#include <string.h>
#include "mex.h"

void
mexFunction (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    mwIndex i, j;
    mwSize m, n;
    mxChar *pi, *po;

    if (nrhs != 1 || ! mxIsChar (prhs[0]) ||
        mxGetNumberOfDimensions (prhs[0]) > 2)
        mexErrMsgTxt ("expecting char matrix");

    m = mxGetM (prhs[0]);
    n = mxGetN (prhs[0]);
    pi = mxGetChars (prhs[0]);
    plhs[0] = mxCreateNumericMatrix (m, n, mxCHAR_CLASS, mxREAL);
    po = mxGetChars (plhs[0]);

    for (j = 0; j < n; j++)
        for (i = 0; i < m; i++)
            po [j*m + m - 1 - i] = pi [j*m + i];
}
```

An example of its expected output is

```
mystring(["First String"; "Second String"])
⇒ s1 = Second String
    First String
```

There are a couple of additional functions available in mex-files of interest in the treatment of strings. These are `mxCreateString`, `mxArrayToString` and `mxCreateCharMatrixFromStrings`. A string in a mex-file is considered to be a vector rather than a matrix. This is perhaps an arbitrary distinction as the data in the `mxArray` for the matrix is consecutive in any case.

A.2.4 Cell Arrays with Mex-Files

We can perform exactly the same operations in Cell arrays in mex-files as we can in oct-files. An example that reduplicates the functional of the ‘celldemo.cc’ oct-file in a mex-file is given by ‘mycell.c’ as below

```

#include "mex.h"

void
mexFunction (int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
{
    mwSize n;
    mwIndex i;

    if (nrhs != 1 || ! mxIsCell (prhs[0]))
        mexErrMsgTxt ("expects cell");

    n = mxGetNumberOfElements (prhs[0]);
    n = (n > nlhs ? nlhs : n);

    for (i = 0; i < n; i++)
        plhs[i] = mxDuplicateArray (mxGetCell (prhs[0], i));
}

```

which as can be seen below has exactly the same behavior as the oct-file version.

```

[b1, b2, b3] = mycell ({1, [1, 2], "test"})
⇒
b1 = 1
b2 =

    1    2

b3 = test

```

Note in the example the use of the `mxDuplicateArray` function. This is needed as the `mxArray` pointer returned by `mxGetCell` might be deallocated. The inverse function to `mxGetCell` is `mxSetCell` and is defined as

```
void mxSetCell (mxArray *ptr, int idx, mxArray *val);
```

Finally, to create a cell array or matrix, the appropriate functions are

```
mxArray *mxCreateCellArray (int ndims, const int *dims);
mxArray *mxCreateCellMatrix (int m, int n);
```

A.2.5 Structures with Mex-Files

The basic function to create a structure in a mex-file is `mxCreateStructMatrix`, which creates a structure array with a two dimensional matrix, or `mxCreateStructArray`.

```

mxArray *mxCreateStructArray (int ndims, int *dims, int num_keys,
                             const char **keys);
mxArray *mxCreateStructMatrix (int rows, int cols, int num_keys,
                              const char **keys);

```

Accessing the fields of the structure can then be performed with the `mxGetField` and `mxSetField` or alternatively with the `mxGetFieldByNumber` and `mxSetFieldByNumber` functions.

```

mxArray *mxGetField (const mxArray *ptr, mwIndex index, const char *key);
mxArray *mxGetFieldByNumber (const mxArray *ptr,
                             mwIndex index, int key_num);
void mxSetField (mxArray *ptr, mwIndex index,
                 const char *key, mxArray *val);
void mxSetFieldByNumber (mxArray *ptr, mwIndex index,
                         int key_num, mxArray *val);

```

A difference between the oct-file interface to structures and the mex-file version is that the functions to operate on structures in mex-files directly include an `index` over the elements of the arrays of elements per `field`. Whereas the oct-file structure includes a Cell Array per field of the structure.

An example that demonstrates the use of structures in mex-file can be found in the file `'mystruct.c'`, as seen below


```

#include "mex.h"

void
mexFunction (int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
{
    int i;
    mwIndex j;
    mxArray *v;
    const char *keys[] = { "this", "that" };

    if (nrhs != 1 || ! mxIsStruct (prhs[0]))
        mexErrMsgTxt ("expects struct");

    for (i = 0; i < mxGetNumberOfFields (prhs[0]); i++)
        for (j = 0; j < mxGetNumberOfElements (prhs[0]); j++)
        {
            mexPrintf ("field %s(%d) = ",
                        mxGetFieldNameByNumber (prhs[0], i), j);
            v = mxGetFieldByNumber (prhs[0], j, i);
            mexCallMATLAB (0, 0, 1, &v, "disp");
        }

    v = mxCreateStructMatrix (2, 2, 2, keys);

    mxSetFieldByNumber (v, 0, 0, mxCreateString ("this1"));
    mxSetFieldByNumber (v, 0, 1, mxCreateString ("that1"));
    mxSetFieldByNumber (v, 1, 0, mxCreateString ("this2"));
    mxSetFieldByNumber (v, 1, 1, mxCreateString ("that2"));
    mxSetFieldByNumber (v, 2, 0, mxCreateString ("this3"));
    mxSetFieldByNumber (v, 2, 1, mxCreateString ("that3"));
    mxSetFieldByNumber (v, 3, 0, mxCreateString ("this4"));
    mxSetFieldByNumber (v, 3, 1, mxCreateString ("that4"));

    if (nlhs)
        plhs[0] = v;
}

```

An example of the behavior of this function within Octave is then

```

a(1).f1 = "f11"; a(1).f2 = "f12"; a(2).f1 = "f21"; a(2).f2 = "f22";
b = mystruct(a)
⇒ field f1(0) = f11
   field f1(1) = f21
   field f2(0) = f12
   field f2(1) = f22
b =
{
  this =

    (,
      [1] = this1
      [2] = this2
      [3] = this3
      [4] = this4
    ,)

  that =

    (,
      [1] = that1
      [2] = that2
      [3] = that3
      [4] = that4
    ,)

}

```

A.2.6 Sparse Matrices with Mex-Files

The Octave format for sparse matrices is identical to the mex format in that it is a compressed column sparse format. Also in both, sparse matrices are required to be two dimensional. The only difference is that the real and imaginary parts of the matrix are stored separately.

The mex-file interface, as well as using `mxGetM`, `mxGetN`, `mxSetM`, `mxSetN`, `mxGetPr`, `mxGetPi`, `mxSetPr` and `mxSetPi`, the mex-file interface supplies the functions

```

mwIndex *mxGetIr (const mxArray *ptr);
mwIndex *mxGetJc (const mxArray *ptr);
mwSize mxGetNzmax (const mxArray *ptr);

void mxSetIr (mxArray *ptr, mwIndex *ir);
void mxSetJc (mxArray *ptr, mwIndex *jc);
void mxSetNzmax (mxArray *ptr, mwSize nzmax);

```

`mxGetNzmax` gets the maximum number of elements that can be stored in the sparse matrix. This is not necessarily the number of non-zero elements in the sparse matrix. `mxGetJc` returns an array with one additional value than the number of columns in the sparse matrix.

The difference between consecutive values of the array returned by `mxGetJc` define the number of non-zero elements in each column of the sparse matrix. Therefore

```
mwSize nz, n;
mwIndex *Jc;
mxArray *m;
...
n = mxGetN (m);
Jc = mxGetJc (m);
nz = Jc[n];
```

returns the actual number of non-zero elements stored in the matrix in `nz`. As the arrays returned by `mxGetPr` and `mxGetPi` only contain the non-zero values of the matrix, we also need a pointer to the rows of the non-zero elements, and this is given by `mxGetIr`. A complete example of the use of sparse matrices in mex-files is given by the file ‘`myparse.c`’ as seen below

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    mwSize n, m, nz;
    mxArray *v;
    mwIndex i;
    double *pr, *pi;
    double *pr2, *pi2;
    mwIndex *ir, *jc;
    mwIndex *ir2, *jc2;

    if (nrhs != 1 || ! mxIsSparse (prhs[0]))
        mexErrMsgTxt ("expects sparse matrix");

    m = mxGetM (prhs [0]);
    n = mxGetN (prhs [0]);
    nz = mxGetNzmax (prhs [0]);

    if (mxIsComplex (prhs[0]))
    {
        mexPrintf ("Matrix is %d-by-%d complex sparse matrix", m, n);
        mexPrintf (" with %d elements\n", nz);

        pr = mxGetPr (prhs[0]);
        pi = mxGetPi (prhs[0]);
        ir = mxGetIr (prhs[0]);
        jc = mxGetJc (prhs[0]);

        i = n;
        while (jc[i] == jc[i-1] && i != 0) i--;
```

```

mexPrintf ("last non-zero element (%d, %d) = (%g, %g)\n",
           ir[nz-1]+ 1, i, pr[nz-1], pi[nz-1]);

v = mxCreateSparse (m, n, nz, mxCOMPLEX);
pr2 = mxGetPr (v);
pi2 = mxGetPi (v);
ir2 = mxGetIr (v);
jc2 = mxGetJc (v);

for (i = 0; i < nz; i++)
{
    pr2[i] = 2 * pr[i];
    pi2[i] = 2 * pi[i];
    ir2[i] = ir[i];
}
for (i = 0; i < n + 1; i++)
    jc2[i] = jc[i];

if (nlhs > 0)
    plhs[0] = v;
}
else if (mxIsLogical (prhs[0]))
{
    bool *pbr, *pbr2;
    mexPrintf ("Matrix is %d-by-%d logical sparse matrix", m, n);
    mexPrintf (" with %d elements\n", nz);

    pbr = mxGetLogicals (prhs[0]);
    ir = mxGetIr (prhs[0]);
    jc = mxGetJc (prhs[0]);

    i = n;
    while (jc[i] == jc[i-1] && i != 0) i--;
    mexPrintf ("last non-zero element (%d, %d) = %d\n", ir[nz-1]+ 1,
               i, pbr[nz-1]);

    v = mxCreateSparseLogicalMatrix (m, n, nz);
    pbr2 = mxGetLogicals (v);
    ir2 = mxGetIr (v);
    jc2 = mxGetJc (v);

    for (i = 0; i < nz; i++)
    {
        pbr2[i] = pbr[i];
        ir2[i] = ir[i];
    }
    for (i = 0; i < n + 1; i++)

```

```

        jc2[i] = jc[i];

    if (nlhs > 0)
        plhs[0] = v;
    }
else
    {
        mexPrintf ("Matrix is %d-by-%d real sparse matrix", m, n);
        mexPrintf (" with %d elements\n", nz);

        pr = mxGetPr (prhs[0]);
        ir = mxGetIr (prhs[0]);
        jc = mxGetJc (prhs[0]);

        i = n;
        while (jc[i] == jc[i-1] && i != 0) i--;
        mexPrintf ("last non-zero element (%d, %d) = %g\n", ir[nz-1]+ 1,
                    i, pr[nz-1]);

        v = mxCreateSparse (m, n, nz, mxREAL);
        pr2 = mxGetPr (v);
        ir2 = mxGetIr (v);
        jc2 = mxGetJc (v);

        for (i = 0; i < nz; i++)
        {
            pr2[i] = 2 * pr[i];
            ir2[i] = ir[i];
        }
        for (i = 0; i < n + 1; i++)
            jc2[i] = jc[i];

        if (nlhs > 0)
            plhs[0] = v;
    }
}

```

A.2.7 Calling Other Functions in Mex-Files

It is also possible call other Octave functions from within a mex-file using `mexCallMATLAB`. An example of the use of `mexCallMATLAB` can be see in the example below

```

#include "mex.h"

void
mexFunction (int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
{
    char *str;

    mexPrintf ("Hello, World!\n");

    mexPrintf ("I have %d inputs and %d outputs\n", nrhs, nlhs);

    if (nrhs < 1 || ! mxIsString (prhs[0]))
        mexErrMsgTxt ("function name expected");

    str = mxArrayToString (prhs[0]);

    mexPrintf ("I'm going to call the interpreter function %s\n", str);

    mexCallMATLAB (nlhs, plhs, nrhs-1, prhs+1, str);

    mxFree (str);
}

```

If this code is in the file ‘myfeval.c’, and is compiled to ‘myfeval.mex’, then an example of its use is

```

myfeval("sin", 1)
a = myfeval("sin", 1)
⇒ Hello, World!
   I have 2 inputs and 1 outputs
   I'm going to call the interpreter function sin
   a =  0.84147

```

Note that it is not possible to use function handles or inline functions within a mex-file.

A.3 Standalone Programs

The libraries Octave itself uses, can be utilized in standalone applications. These applications then have access, for example, to the array and matrix classes as well as to all the Octave algorithms. The following C++ program, uses class `Matrix` from `liboctave.a` or `liboctave.so`.

```
#include <iostream>
#include <octave/oct.h>
int
main (void)
{
    std::cout << "Hello Octave world!\n";
    int n = 2;
    Matrix a_matrix = Matrix (n, n);
    for (octave_idx_type i = 0; i < n; i++)
    {
        for (octave_idx_type j = 0; j < n; j++)
        {
            a_matrix(row,column) = (i+1)*10 + (j+1);
        }
    }
    std::cout << a_matrix;
    return 0;
}
```

mkcoffile can then be used to build a standalone application with a command like

```
$ mkcoffile --link-stand-alone hello.cc -o hello
$ ./hello
Hello Octave world!
  11 12
  21 22
$
```

Note that the application `hello` will be dynamically linked against the octave libraries and any octave support libraries.

Appendix B Test and Demo Functions

Octave includes a number of functions to allow the integration of testing and demonstration code in the source code of the functions themselves.

B.1 Test Functions

<code>test name</code>	[Function File]
<code>test name quiet normal verbose</code>	[Function File]
<code>test ('name', 'quiet normal verbose', fid)</code>	[Function File]
<code>test ([], 'explain', fid)</code>	[Function File]
<code>success = test (...)</code>	[Function File]
<code>[n, max] = test (...)</code>	[Function File]
<code>[code, idx] = test ('name', 'grabdemo')</code>	[Function File]

Perform tests from the first file in the loadpath matching *name*. `test` can be called as a command or as a function. Called with a single argument *name*, the tests are run interactively and stop after the first error is encountered.

With a second argument the tests which are performed and the amount of output is selected.

'quiet'	Don't report all the tests as they happen, just the errors.
'normal'	Report all tests as they happen, but don't do tests which require user interaction.
'verbose'	Do tests which require user interaction.

The argument *fid* can be used to allow batch processing. Errors can be written to the already open file defined by *fid*, and hopefully when octave crashes this file will tell you what was happening when it did. You can use `stdout` if you want to see the results as they happen. You can also give a file name rather than an *fid*, in which case the contents of the file will be replaced with the log from the current test.

Called with a single output argument *success*, `test` returns true if all of the tests were successful. Called with two output arguments *n* and *max*, the number of successful test and the total number of tests in the file *name* are returned.

If the second argument is the string 'grabdemo', the contents of the demo blocks are extracted but not executed. Code for all code blocks is concatenated and returned as *code* with *idx* being a vector of positions of the ends of the demo blocks.

If the second argument is 'explain', then *name* is ignored and an explanation of the line markers used is written to the file *fid*.

See also: error, assert, fail, demo, example.

`test` scans the named script file looking for lines which start with `%!`. The prefix is stripped off and the rest of the line is processed through the octave interpreter. If the code generates an error, then the test is said to fail.

Since `eval()` will stop at the first error it encounters, you must divide your tests up into blocks, with anything in a separate block evaluated separately. Blocks are introduced by the keyword `test` immediately following the `%!`. For example,

```

%!test error ("this test fails!");
%!test "this test doesn't fail since it doesn't generate an error";

```

When a test fails, you will see something like:

```

***** test error ('this test fails!')
!!!! test failed
this test fails!

```

Generally, to test if something works, you want to assert that it produces a correct value. A real test might look something like

```

%!test
%! a = [1, 2, 3; 4, 5, 6]; B = [1; 2];
%! expect = [ a ; 2*a ];
%! get = kron (b, a);
%! if (any(size(expect) != size(get)))
%!     error ("wrong size: expected %d,%d but got %d,%d",
%!           size(expect), size(get));
%! elseif (any(any(expect!=get)))
%!     error ("didn't get what was expected.");
%! endif

```

To make the process easier, use the `assert` function. For example, with `assert` the previous test is reduced to:

```

%!test
%! a = [1, 2, 3; 4, 5, 6]; b = [1; 2];
%! assert (kron (b, a), [ a ; 2*a ]);

```

`assert` can accept a tolerance so that you can compare results absolutely or relatively. For example, the following all succeed:

```

%!test assert (1+eps, 1, 2*eps)           # absolute error
%!test assert (100+100*eps, 100, -2*eps) # relative error

```

You can also do the comparison yourself, but still have `assert` generate the error:

```

%!test assert (isempty([]))
%!test assert ([ 1,2; 3,4 ] > 0)

```

Because `assert` is so frequently used alone in a test block, there is a shorthand form:

```

%!assert (...)

```

which is equivalent to:

```

%!test assert (...)

```

Sometimes during development there is a test that should work but is known to fail. You still want to leave the test in because when the final code is ready the test should pass, but you may not be able to fix it immediately. To avoid unnecessary bug reports for these known failures, mark the block with `xtest` rather than `test`:

```

%!xtest assert (1==0)
%!xtest fail ('success=1','error')

```

Another use of `xtest` is for statistical tests which should pass most of the time but are known to fail occasionally.

Each block is evaluated in its own function environment, which means that variables defined in one block are not automatically shared with other blocks. If you do want to share variables, then you must declare them as **shared** before you use them. For example, the following declares the variable *a*, gives it an initial value (default is empty), then uses it in several subsequent tests.

```
%!shared a
%! a = [1, 2, 3; 4, 5, 6];
%!assert (kron ([1; 2], a), [ a; 2*a ]);
%!assert (kron ([1, 2], a), [ a, 2*a ]);
%!assert (kron ([1,2; 3,4], a), [ a,2*a; 3*a,4*a ]);
```

You can share several variables at the same time:

```
%!shared a, b
```

You can also share test functions:

```
%!function a = fn(b)
%! a = 2*b;
%!assert (a(2),4);
```

Note that all previous variables and values are lost when a new shared block is declared.

Error and warning blocks are like test blocks, but they only succeed if the code generates an error. You can check the text of the error is correct using an optional regular expression **<pattern>**. For example:

```
%!error <passes!> error('this test passes!');
```

If the code doesn't generate an error, the test fails. For example,

```
%!error "this is an error because it succeeds.";
```

produces

```
***** error "this is an error because it succeeds.";
!!!!! test failed: no error
```

It is important to automate the tests as much as possible, however some tests require user interaction. These can be isolated into demo blocks, which if you are in batch mode, are only run when called with **demo** or **verbose**. The code is displayed before it is executed. For example,

```
%!demo
%! t=[0:0.01:2*pi]; x=sin(t);
%! plot(t,x);
%! you should now see a sine wave in your figure window
```

produces

```
> t=[0:0.01:2*pi]; x=sin(t);
> plot(t,x);
> you should now see a sine wave in your figure window
Press <enter> to continue:
```

Note that demo blocks cannot use any shared variables. This is so that they can be executed by themselves, ignoring all other tests.

If you want to temporarily disable a test block, put **#** in place of the block type. This creates a comment block which is echoed in the log file, but is not executed. For example:

```

%!#demo
%! t=[0:0.01:2*pi]; x=sin(t);
%! plot(t,x);
%! you should now see a sine wave in your figure window

```

Block type summary:

```

%!test      check that entire block is correct
%!error     check for correct error message
%!warning
            check for correct warning message
%!demo     demo only executes in interactive mode
%!#        comment: ignore everything within the block
%!shared x,y,z
            declares variables for use in multiple tests
%!function
            defines a function value for a shared variable
%!assert (x, y, tol)
            shorthand for %!test assert (x, y, tol)

```

You can also create test scripts for builtins and your own C++ functions. Just put a file of the function name on your path without any extension and it will be picked up by the test procedure. You can even embed tests directly in your C++ code:

```

    #if 0
    %!test disp('this is a test')
    #endif
or
    /*
    %!test disp('this is a test')
    */

```

but then the code will have to be on the load path and the user will have to remember to type `test('name.cc')`. Conversely, you can separate the tests from normal octave script files by putting them in plain files with no extension rather than in script files.

<code>assert (cond)</code>	[Function File]
<code>assert (observed,expected)</code>	[Function File]
<code>assert (observed,expected,tol)</code>	[Function File]

Produces an error if the condition is not met. **assert** can be called in three different ways.

assert (cond)
 Called with a single argument *cond*, **assert** produces an error if *cond* is zero.

assert (observed, expected)
 Produce an error if observed is not the same as expected. Note that observed and expected can be strings, scalars, vectors, matrices, lists or structures.

```
assert(observed, expected, tol)
```

Produce an error if relative error is less than tolerance. That is,
 $\text{abs}(\text{observed} - \text{expected}) > \text{tol} * \text{expected}$. Absolute error
 $\text{abs}(\text{observed} - \text{expected}) > \text{abs}(\text{tol})$ will be used when tolerance is
 negative or when the expected value is zero.

See also: test.

```
fail (code,pattern) [Function File]
```

```
fail (code, 'warning', pattern) [Function File]
```

Return true if *code* fails with an error message matching *pattern*, otherwise produce an error. Note that *code* is a string and if *code* runs successfully, the error produced is:

```
expected error but got none
```

If the code fails with a different error, the message produced is:

```
expected <pattern>  
but got <text of actual error>
```

The angle brackets are not part of the output.

Called with three arguments, the behavior is similar to `fail(code, pattern)`, but produces an error if no warning is given during code execution or if the code fails.

B.2 Demonstration Functions

```
demo ('name',n) [Function File]
```

Runs any examples associated with the function '*name*'. Examples are stored in the script file, or in a file with the same name but no extension somewhere on your path. To keep them separate from the usual script code, all lines are prefixed by `%!` . Each example is introduced by the keyword 'demo' flush left to the prefix, with no intervening spaces. The remainder of the example can contain arbitrary octave code. For example:

```
%!demo  
%! t=0:0.01:2*pi; x = sin(t);  
%! plot(t,x)  
%! %-----  
%! % the figure window shows one cycle of a sine wave
```

Note that the code is displayed before it is executed, so a simple comment at the end suffices. It is generally not necessary to use `disp` or `printf` within the demo.

Demos are run in a function environment with no access to external variables. This means that all demos in your function must use separate initialization code. Alternatively, you can combine your demos into one huge demo, with the code:

```
%! input("Press <enter> to continue: ", "s");
```

between the sections, but this is discouraged. Other techniques include using multiple plots by saying figure between each, or using subplot to put multiple plots in the same window.

Also, since `demo` evaluates inside a function context, you cannot define new functions inside a demo. Instead you will have to use `eval(example('function',n))` to see them. Because `eval` only evaluates one line, or one statement if the statement crosses multiple lines, you must wrap your demo in "if 1 <demo stuff> endif" with the 'if' on the same line as 'demo'. For example,

```

%!demo if 1
%!  function y=f(x)
%!    y=x;
%!  endfunction
%!  f(3)
%! endif

```

See also: `test`, `example`.

`example ('name',n)` [Function File]
`[x, idx] = example ('name',n)` [Function File]

Display the code for example *n* associated with the function '*name*', but do not run it. If *n* is not given, all examples are displayed.

Called with output arguments, the examples are returned in the form of a string *x*, with *idx* indicating the ending position of the various examples.

See `demo` for a complete explanation.

See also: `demo`, `test`.

`speed (f, init, max_n, f2, tol)` [Function File]
`[order, n, T_f, T_f2] = speed (...)` [Function File]

Determine the execution time of an expression for various *n*. The *n* are log-spaced from 1 to *max_n*. For each *n*, an initialization expression is computed to create whatever data are needed for the test. If a second expression is given, the execution times of the two expressions will be compared. Called without output arguments the results are presented graphically.

f The expression to evaluate.

max_n The maximum test length to run. Default value is 100. Alternatively, use `[min_n,max_n]` or for complete control, `[n1,n2,...,nk]`.

init Initialization expression for function argument values. Use *k* for the test number and *n* for the size of the test. This should compute values for all variables listed in args. Note that *init* will be evaluated first for *k*=0, so things which are constant throughout the test can be computed then. The default value is `x = randn (n, 1);`.

f2 An alternative expression to evaluate, so the speed of the two can be compared. Default is `[]`.

tol If *tol* is `Inf`, then no comparison will be made between the results of expression *f* and expression *f2*. Otherwise, expression *f* should produce a value *v* and expression *f2* should produce a value *v2*, and these shall be compared using `assert(v,v2,tol)`. If *tol* is positive, the tolerance is

	assumed to be absolute. If <i>tol</i> is negative, the tolerance is assumed to be relative. The default is <code>eps</code> .
<i>order</i>	The time complexity of the expression $O(a n^p)$. This is a structure with fields <i>a</i> and <i>p</i> .
<i>n</i>	The values <i>n</i> for which the expression was calculated and the execution time was greater than zero.
<i>T_f</i>	The nonzero execution times recorded for the expression <i>f</i> in seconds.
<i>T_{f2}</i>	The nonzero execution times recorded for the expression <i>f2</i> in seconds. If it is needed, the mean time ratio is just <code>mean(T_f./T_{f2})</code> .

The slope of the execution time graph shows the approximate power of the asymptotic running time $O(n^p)$. This power is plotted for the region over which it is approximated (the latter half of the graph). The estimated power is not very accurate, but should be sufficient to determine the general order of your algorithm. It should indicate if for example your implementation is unexpectedly $O(n^2)$ rather than $O(n)$ because it extends a vector each time through the loop rather than preallocating one which is big enough. For example, in the current version of Octave, the following is not the expected $O(n)$:

```
speed("for i=1:n,y{i}=x(i); end", "", [1000,10000])
```

but it is if you preallocate the cell array *y*:

```
speed("for i=1:n,y{i}=x(i);end", ...
      "x=rand(n,1);y=cell(size(x));", [1000,10000])
```

An attempt is made to approximate the cost of the individual operations, but it is wildly inaccurate. You can improve the stability somewhat by doing more work for each *n*. For example:

```
speed("airy(x)", "x=rand(n,10)", [10000,100000])
```

When comparing a new and original expression, the line on the speedup ratio graph should be larger than 1 if the new expression is faster. Better algorithms have a shallow slope. Generally, vectorizing an algorithm will not change the slope of the execution time graph, but it will shift it relative to the original. For example:

```
speed("v=sum(x)", "", [10000,100000], ...
      "v=0;for i=1:length(x),v+=x(i);end")
```

A more complex example, if you had an original version of `xcorr` using for loops and another version using an FFT, you could compare the run speed for various lags as follows, or for a fixed lag with varying vector lengths as follows:

```
speed("v=xcorr(x,n)", "x=rand(128,1);", 100, ...
      "v2=xcorr_orig(x,n)", -100*eps)
speed("v=xcorr(x,15)", "x=rand(20+n,1);", 100, ...
      "v2=xcorr_orig(x,n)", -100*eps)
```

Assuming one of the two versions is in `xcorr_orig`, this would compare their speed and their output values. Note that the FFT version is not exact, so we specify an acceptable tolerance on the comparison `100*eps`, and the errors should be computed relatively, as `abs((x - y)./y)` rather than absolutely as `abs(x - y)`.

Type `example('speed')` to see some real examples. Note for obscure reasons, you can't run examples 1 and 2 directly using `demo('speed')`. Instead use, `eval(example('speed',1))` and `eval(example('speed',2))`.

Appendix C Tips and Standards

This chapter describes no additional features of Octave. Instead it gives advice on making effective use of the features described in the previous chapters.

C.1 Writing Clean Octave Programs

Here are some tips for avoiding common errors in writing Octave code intended for widespread use:

- Since all global variables share the same name space, and all functions share another name space, you should choose a short word to distinguish your program from other Octave programs. Then take care to begin the names of all global variables, constants, and functions with the chosen prefix. This helps avoid name conflicts.

If you write a function that you think ought to be added to Octave under a certain name, such as `fiddle_matrix`, don't call it by that name in your program. Call it `mylib_fiddle_matrix` in your program, and send mail to maintainers@octave.org suggesting that it be added to Octave. If and when it is, the name can be changed easily enough.

If one prefix is insufficient, your package may use two or three alternative common prefixes, so long as they make sense.

Separate the prefix from the rest of the symbol name with an underscore `'_'`. This will be consistent with Octave itself and with most Octave programs.

- When you encounter an error condition, call the function `error` (or `usage`). The `error` and `usage` functions do not return. See [Section 2.5 \[Errors\]](#), page 26.
- Please put a copyright notice on the file if you give copies to anyone. Use the same lines that appear at the top of the function files distributed with Octave. If you have not signed papers to assign the copyright to anyone else, then place your name in the copyright notice.

C.2 Tips for Making Code Run Faster.

Here are some ways of improving the execution speed of Octave programs.

- Avoid looping wherever possible.
- Use iteration rather than recursion whenever possible. Function calls are slow in Octave.
- Avoid resizing matrices unnecessarily. When building a single result matrix from a series of calculations, set the size of the result matrix first, then insert values into it. Write

```
result = zeros (big_n, big_m)
for i = over:and_over
    r1 = ...
    r2 = ...
    result (r1, r2) = new_value ();
endfor
```

instead of

```

result = [];
for i = ever:and_ever
    result = [ result, new_value() ];
endfor

```

- Avoid calling `eval` or `feval` whenever possible, because they require Octave to parse input or look up the name of a function in the symbol table.

If you are using `eval` as an exception handling mechanism and not because you need to execute some arbitrary text, use the `try` statement instead. See [Section 11.9 \[The try Statement\]](#), page 114.

- If you are calling lots of functions but none of them will need to change during your run, set the variable `ignore_function_time_stamp` to "all" so that Octave doesn't waste a lot of time checking to see if you have updated your function files.

C.3 Tips on Writing Comments

Here are the conventions to follow when writing comments.

- ‘#’ Comments that start with a single sharp-sign, ‘#’, should all be aligned to the same column on the right of the source code. Such comments usually explain how the code on the same line does its job. In the Emacs mode for Octave, the `M-;` (`indent-for-comment`) command automatically inserts such a ‘#’ in the right place, or aligns such a comment if it is already present.
- ‘##’ Comments that start with two semicolons, ‘##’, should be aligned to the same level of indentation as the code. Such comments usually describe the purpose of the following lines or the state of the program at that point.

The indentation commands of the Octave mode in Emacs, such as `M-;` (`indent-for-comment`) and `TAB` (`octave-indent-line`) automatically indent comments according to these conventions, depending on the number of semicolons. See [section “Manipulating Comments”](#) in *The GNU Emacs Manual*.

C.4 Conventional Headers for Octave Functions

Octave has conventions for using special comments in function files to give information such as who wrote them. This section explains these conventions.

The top of the file should contain a copyright notice, followed by a block of comments that can be used as the help text for the function. Here is an example:

```

## Copyright (C) 1996, 1997, 2007 John W. Eaton
##
## This file is part of Octave.
##
## Octave is free software; you can redistribute it and/or
## modify it under the terms of the GNU General Public
## License as published by the Free Software Foundation;
## either version 2, or (at your option) any later version.
##
## Octave is distributed in the hope that it will be useful,

```

```

## but WITHOUT ANY WARRANTY; without even the implied
## warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
## PURPOSE. See the GNU General Public License for more
## details.
##
## You should have received a copy of the GNU General Public
## License along with Octave; see the file COPYING. If not,
## write to the Free Software Foundation, Inc., 51 Franklin Street,
## Fifth Floor, Boston, MA 02110-1301, USA.

## usage: [IN, OUT, PID] = popen2 (COMMAND, ARGS)
##
## Start a subprocess with two-way communication. COMMAND
## specifies the name of the command to start. ARGS is an
## array of strings containing options for COMMAND. IN and
## OUT are the file ids of the input and streams for the
## subprocess, and PID is the process id of the subprocess,
## or -1 if COMMAND could not be executed.
##
## Example:
##
## [in, out, pid] = popen2 ("sort", "-nr");
## fputs (in, "these\nare\nsome\nstrings\n");
## fclose (in);
## while (isstr (s = fgets (out)))
##     fputs (stdout, s);
## endwhile
## fclose (out);

```

Octave uses the first block of comments in a function file that do not appear to be a copyright notice as the help text for the file. For Octave to recognize the first comment block as a copyright notice, it must start with the word ‘Copyright’ after stripping the leading comment characters.

After the copyright notice and help text come several *header comment* lines, each beginning with ‘## *header-name*:’. For example,

```

## Author: jwe
## Keywords: subprocesses input-output
## Maintainer: jwe

```

Here is a table of the conventional possibilities for *header-name*:

‘Author’ This line states the name and net address of at least the principal author of the library.

```
## Author: John W. Eaton <jwe@bevo.che.wisc.edu>
```

‘Maintainer’

This line should contain a single name/address as in the Author line, or an address only, or the string ‘jwe’. If there is no maintainer line, the person(s)

in the Author field are presumed to be the maintainers. The example above is mildly bogus because the maintainer line is redundant.

The idea behind the ‘Author’ and ‘Maintainer’ lines is to make possible a function to “send mail to the maintainer” without having to mine the name out by hand.

Be sure to surround the network address with ‘<...>’ if you include the person’s full name as well as the network address.

‘Created’ This optional line gives the original creation date of the file. For historical interest only.

‘Version’ If you wish to record version numbers for the individual Octave program, put them in this line.

‘Adapted-By’
In this header line, place the name of the person who adapted the library for installation (to make it fit the style conventions, for example).

‘Keywords’
This line lists keywords. Eventually, it will be used by an apropos command to allow people will find your package when they’re looking for things by topic area. To separate the keywords, you can use spaces, commas, or both.

Just about every Octave function ought to have the ‘Author’ and ‘Keywords’ header comment lines. Use the others if they are appropriate. You can also put in header lines with other header names—they have no standard meanings, so they can’t do any harm.

C.5 Tips for Documentation Strings

As noted above, documentation is typically in a commented header block on an Octave function following the copyright statement. The help string shown above is an unformatted string and will be displayed as is by Octave. Here are some tips for the writing of documentation strings.

- Every command, function, or variable intended for users to know about should have a documentation string.
- An internal variable or subroutine of an Octave program might as well have a documentation string.
- The first line of the documentation string should consist of one or two complete sentences that stand on their own as a summary.

The documentation string can have additional lines that expand on the details of how to use the function or variable. The additional lines should also be made up of complete sentences.

- For consistency, phrase the verb in the first sentence of a documentation string as an infinitive with “to” omitted. For instance, use “Return the frob of A and B.” in preference to “Returns the frob of A and B.” Usually it looks good to do likewise for the rest of the first paragraph. Subsequent paragraphs usually look better if they have proper subjects.

- Write documentation strings in the active voice, not the passive, and in the present tense, not the future. For instance, use “Return a list containing A and B.” instead of “A list containing A and B will be returned.”
- Avoid using the word “cause” (or its equivalents) unnecessarily. Instead of, “Cause Octave to display text in boldface,” write just “Display text in boldface.”
- Do not start or end a documentation string with whitespace.
- Format the documentation string so that it fits in an Emacs window on an 80-column screen. It is a good idea for most lines to be no wider than 60 characters.

However, rather than simply filling the entire documentation string, you can make it much more readable by choosing line breaks with care. Use blank lines between topics if the documentation string is long.

- **Do not** indent subsequent lines of a documentation string so that the text is lined up in the source code with the text of the first line. This looks nice in the source code, but looks bizarre when users view the documentation. Remember that the indentation before the starting double-quote is not part of the string!
- The documentation string for a variable that is a yes-or-no flag should start with words such as “Nonzero means...”, to make it clear that all nonzero values are equivalent and indicate explicitly what zero and nonzero mean.
- When a function’s documentation string mentions the value of an argument of the function, use the argument name in capital letters as if it were a name for that value. Thus, the documentation string of the operator / refers to its second argument as ‘DIVISOR’, because the actual argument name is `divisor`.

Also use all caps for meta-syntactic variables, such as when you show the decomposition of a list or vector into subunits, some of which may vary.

Octave also allows extensive formatting of the help string of functions using Texinfo. The effect on the online documentation is relatively small, but makes the help string of functions conform to the help of Octave’s own functions. However, the effect on the appearance of printed or online documentation will be greatly improved.

The fundamental building block of Texinfo documentation strings is the Texinfo-macro `@deftypefn`, which takes three arguments: The class the function is in, its output arguments, and the function’s signature. Typical classes for functions include **Function File** for standard Octave functions, and **Loadable Function** for dynamically linked functions. A skeletal Texinfo documentation string therefore looks like this

```

-*- texinfo -*-
@deftypefn{Function File} {@var{return_value} = } function_name (...)
@cindex index term
Help text in Texinfo format. Code samples should be marked like
@code{sample of code} and variables should be marked as
@var{variable}.
@seealso{function2}
@end deftypefn

```

This help string must be commented in user functions, or in the help string of the `DEFUN_DLD` macro for dynamically loadable functions. The important aspects of the documentation string are

`-*- texinfo -*-`

This string signals Octave that the follow text is in Texinfo format, and should be the first part of any help string in Texinfo format.

`@deftypefn{class} . . . @end deftypefn`

The entire help string should be enclosed within the block defined by `deftypefn`.

`@cindex index term`

This generates an index entry, and can be useful when the function is included as part of a larger piece of documentation. It is ignored within Octave's help viewer.

`@var{variable}`

All variables should be marked with this macro. The markup of variables is then changed appropriately for display.

`@code{sample of code}`

All samples of code should be marked with this macro for the same reasons as the `@var` macro.

`@seealso{function2}`

This is a comma separated list of function names that allows cross referencing from one function documentation string to another.

Texinfo format has been designed to generate output for online viewing with text-terminals as well as generating high-quality printed output. To these ends, Texinfo has commands which control the diversion of parts of the document into a particular output processor. Three formats are of importance: info, html and TeX. These are selected with

```
@ifinfo
Text area for info only
@end ifinfo

@ifhtml
Text area for html only
@end ifhtml

@iftex
@tex
text for TeX only
@end tex
@end iftex
```

Note that often TeX output can be used in html documents and so often the `@ifhtml` blocks are unnecessary. If no specific output processor is chosen, by default, the text goes into all output processors. It is usual to have the above blocks in pairs to allow the same information to be conveyed in all output formats, but with a different markup.

Another important feature of Texinfo that is often used in Octave help strings is the `@example` environment. An example of its use is

```

@example
@group
@code{2 * 2}
@result{} 4
@end group
@end example

```

which produces

```

2 * 2
⇒ 4

```

The `@group` block prevents the example from being split across a page boundary, while the `@result{}` macro produces a right arrow signifying the result of a command.

In many cases a function has multiple means in which it can be called, and the `@deftypefnx` macro can be used to give alternatives. For example

```

-*- texinfo -*-
@deftypefn{Function File} {@var{a} = } function_name (@var{x}, ...)
@deftypefnx{Function File} {@var{a} = } function_name (@var{y}, ...)
Help text in Texinfo format.
@end deftypefn

```

Many complete examples of Texinfo documentation can be taken from the help strings for the Octave functions themselves. A relatively complete example of which is the `nchoosek` function. The Texinfo documentation string of `nchoosek` is

```

-- texinfo --
@deftypefn {Function File} {@var{c} =} nchoosek (@var{n}, @var{k})

Compute the binomial coefficient or all combinations of @var{n}.
If @var{n} is a scalar then, calculate the binomial coefficient
of @var{n} and @var{k}, defined as

@iftex
@tex
$$
{n \choose k} = {n (n-1) (n-2) \cdots (n-k+1) \over k!}
$$
@end tex
@end iftex
@ifinfo

@example
@group
/ \
| n |   n (n-1) (n-2) ... (n-k+1)
|   | = -----
| k |           k!
\   /
@end group
@end example
@end ifinfo

If @var{n} is a vector generate all combinations of the elements
of @var{n}, taken @var{k} at a time, one row per combination. The
resulting @var{c} has size @code{[nchoosek (length (@var{n}),
@var{k}), @var{k}]}.

@seealso{bincoeff}
@end deftypefn

```

which demonstrates most of the concepts discussed above. This documentation string renders as

-- Function File: `C = nchoosek (N, K)`

Compute the binomial coefficient or all combinations of `N`. If `N` is a scalar then, calculate the binomial coefficient of `N` and `K`, defined as

$$\frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\dots(n-k+1)}{k!}$$

If `N` is a vector generate all combinations of the elements of `N`, taken `K` at a time, one row per combination. The resulting `C` has size `'[nchoosek (length (N), K), K]'`.

See also: `bincoeff`.

using `info`, whereas in a printed documentation using TeX it will appear as

`c = nchoosek (n, k)` [Function File]

Compute the binomial coefficient or all combinations of `n`. If `n` is a scalar then, calculate the binomial coefficient of `n` and `k`, defined as

$$\binom{n}{k} = \frac{n(n-1)(n-2)\dots(n-k+1)}{k!}$$

If `n` is a vector generate all combinations of the elements of `n`, taken `k` at a time, one row per combination. The resulting `c` has size `[nchoosek (length (n), k), k]`.

See also: `bincoeff`.

Appendix D Known Causes of Trouble

This section describes known problems that affect users of Octave. Most of these are not Octave bugs per se—if they were, we would fix them. But the result for a user may be like the result of a bug.

Some of these problems are due to bugs in other software, some are missing features that are too much work to add, and some are places where people’s opinions differ as to what is best.

D.1 Actual Bugs We Haven’t Fixed Yet

- Output that comes directly from Fortran functions is not sent through the pager and may appear out of sequence with other output that is sent through the pager. One way to avoid this is to force pending output to be flushed before calling a function that will produce output from within Fortran functions. To do this, use the command

```
fflush (stdout)
```

Another possible workaround is to use the command

```
page_screen_output = "false"
```

to turn the pager off.

A list of ideas for future enhancements is distributed with Octave. See the file ‘PROJECTS’ in the top level directory in the source distribution.

D.2 Reporting Bugs

Your bug reports play an essential role in making Octave reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See [Appendix D \[Trouble\], page 519](#). If it isn’t known, then you should report the problem.

Reporting a bug may help you by bringing a solution to your problem, or it may not. In any case, the principal function of a bug report is to help the entire community by making the next version of Octave work better. Bug reports are your contribution to the maintenance of Octave.

In order for a bug report to serve its purpose, you must include the information that makes it possible to fix the bug.

If you have Octave working at all, the easiest way to prepare a complete bug report is to use the Octave function `bug_report`. When you execute this function, Octave will prompt you for a subject and then invoke the editor on a file that already contains all the configuration information. When you exit the editor, Octave will mail the bug report for you.

`bug_report ()` [Function File]

Have Octave create a bug report template file, invoke your favorite editor, and submit the report to the bug-octave mailing list when you are finished editing.

D.3 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If Octave gets a fatal signal, for any input whatever, that is a bug. Reliable interpreters never crash.
- If Octave produces incorrect results, for any input whatever, that is a bug.
- Some output may appear to be incorrect when it is in fact due to a program whose behavior is undefined, which happened by chance to give the desired results on another system. For example, the range operator may produce different results because of differences in the way floating point arithmetic is handled on various systems.
- If Octave produces an error message for valid input, that is a bug.
- If Octave does not produce an error message for invalid input, that is a bug. However, you should note that your idea of “invalid input” might be my idea of “an extension” or “support for traditional practice”.
- If you are an experienced user of programs like Octave, your suggestions for improvement are welcome in any case.

D.4 Where to Report Bugs

If you have Octave working at all, the easiest way to prepare a complete bug report is to use the Octave function `bug_report`. When you execute this function, Octave will prompt you for a subject and then invoke the editor on a file that already contains all the configuration information. When you exit the editor, Octave will mail the bug report for you.

If for some reason you cannot use Octave’s `bug_report` function, send bug reports for Octave to bug@octave.org.

Do not send bug reports to ‘help-octave’. Most users of Octave do not want to receive bug reports. Those that do have asked to be on the mailing list.

As a last resort, send bug reports on paper to:

Octave Bugs c/o John W. Eaton
University of Wisconsin-Madison
Department of Chemical Engineering
1415 Engineering Drive
Madison, Wisconsin 53706 USA

D.5 How to Report Bugs

Send bug reports for Octave to one of the addresses listed in [Section D.4 \[Bug Lists\]](#), [page 520](#).

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don’t matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn’t, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the

contents of that location would fool the interpreter into doing the right thing despite the bug. Play it safe and give a specific, complete example.

Keep in mind that the purpose of a bug report is to enable someone to fix the bug if it is not known. Always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, “Does this ring a bell?” This cannot help us fix a bug. It is better to send a complete bug report to begin with.

Try to make your bug report self-contained. If we have to ask you for more information, it is best if you include all the previous information in your response, as well as the information that was missing.

To enable someone to investigate the bug, you should include all these things:

- The version of Octave. You can get this by noting the version number that is printed when Octave starts, or running it with the ‘-v’ option.
- A complete input file that will reproduce the bug.

A single statement may not be enough of an example—the bug might depend on other details that are missing from the single statement where the error finally occurs.

- The command arguments you gave Octave to execute that example and observe the bug. To guarantee you won’t omit something important, list all the options.

If we were to try to guess the arguments, we would probably guess wrong and then we would not encounter the bug.

- The type of machine you are using, and the operating system name and version number.
- The command-line arguments you gave to the `configure` command when you installed the interpreter.
- A complete list of any modifications you have made to the interpreter source.

Be precise about these changes—show a context diff for them.

- Details of any other deviations from the standard procedure for installing Octave.
- A description of what behavior you observe that you believe is incorrect. For example, "The interpreter gets a fatal signal," or, "The output produced at line 208 is incorrect."

Of course, if the bug is that the interpreter gets a fatal signal, then one can’t miss it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong.

Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the interpreter is out of synch, or you have encountered a bug in the C library on your system. Your copy might crash and the copy here would not. If you said to expect a crash, then when the interpreter here fails to crash, we would know that the bug was not happening. If you don’t say to expect a crash, then we would not know whether the bug was happening. We would not be able to draw any conclusion from our observations.

Often the observed symptom is incorrect output when your program is run. Unfortunately, this is not enough information unless the program is short and simple. It is very helpful if you can include an explanation of the expected output, and why the actual output is incorrect.

- If you wish to suggest changes to the Octave source, send them as context diffs. If you even discuss something in the Octave source, refer to it by context, not by line number,

because the line numbers in the development sources probably won't match those in your sources.

Here are some things that are not necessary:

- A description of the envelope of the bug.
Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it. Such information is usually not necessary to enable us to fix bugs in Octave, but if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc. Most Octave bugs involve just one function, so the most straightforward way to simplify an example is to delete all the function definitions except the one in which the bug occurs.
However, simplification is not vital; if you don't want to do this, report the bug anyway and send the entire test case you used.
- A patch for the bug. Patches can be helpful, but if you find a bug, you should report it, even if you cannot send a fix for the problem.

D.6 Sending Patches for Octave

If you would like to write bug fixes or improvements for Octave, that is very helpful. When you send your changes, please follow these guidelines to avoid causing extra work for us in studying the patches.

If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining Octave is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a bug fix, just include a copy of the bug report, and explain why the change fixes the bug.
- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is right, we might have trouble judging it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one.

- Use `'diff -c'` to make your diffs. Diffs without context are hard for us to install reliably. More than that, they make it hard for us to study the diffs to decide whether we want to install them. Unidiff format is better than contextless diffs, but not as easy to read as `'-c'` format.

If you have GNU diff, use `'diff -cp'`, which shows the name of the function that each change occurs in.

- Write the change log entries for your changes.

Read the `'ChangeLog'` file to see what sorts of information to put in, and to learn the style that we use. The purpose of the change log is to show people where to find what

was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was made.

On the other hand, once you have shown people where to find the change, you need not explain its purpose. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does—but the explanation will be much more useful if you put it in comments in the code.

If you would like your name to appear in the header line for who made the change, send us the header line.

D.7 How To Get Help with Octave

The mailing list help@octave.org exists for the discussion of matters related to using and installing Octave. If you would like to join the discussion, please send a short note to help-request@octave.org.

Please do not send requests to be added or removed from the mailing list, or other administrative trivia to the list itself.

If you think you have found a bug in the installation procedure, however, you should send a complete bug report for the problem to bug@octave.org. See [Section D.5 \[Bug Reporting\]](#), [page 520](#), for information that will help you to submit a useful report.

Appendix E Installing Octave

Here is the procedure for installing Octave from scratch on a Unix system.

- Run the shell script ‘`configure`’. This will determine the features your system has (or doesn’t have) and create a file named ‘`Makefile`’ from each of the files named ‘`Makefile.in`’.

Here is a summary of the configure options that are most frequently used when building Octave:

`--prefix=prefix`

Install Octave in subdirectories below *prefix*. The default value of *prefix* is ‘`/usr/local`’.

`--srcdir=dir`

Look for Octave sources in the directory *dir*.

`--with-f2c`

Use `f2c` even if a Fortran compiler is available.

`--with-f77`

Use `f77` to compile Fortran code. You may also specify the name of the compiler to use as an optional argument. For example, `--with-f77=g77` sets the name of the Fortran compiler to `g77`.

`--enable-shared`

Create shared libraries. If you are planning to use `--enable-lite-kernel` or the dynamic loading features, you will probably want to use this option. It will make your ‘`.oct`’ files much smaller and on some systems it may be necessary to build shared libraries in order to use dynamically linked functions.

You may also want to build a shared version of `libstdc++`, if your system doesn’t already have one. Note that a patch is needed to build shared versions of version 2.7.2 of `libstdc++` on the HP-PA architecture. You can find the patch at <ftp://ftp.cygnus.com/pub/g++/libg++-2.7.2-hppa-gcc-fix>.

`--enable-dl`

Use `dlopen` and friends to make Octave capable of dynamically linking externally compiled functions. This only works on systems that actually have these functions. If you plan on using this feature, you should probably also use `--enable-shared` to reduce the size of your ‘`.oct`’ files.

`--enable-shl`

Use `shl_load` and friends to make Octave capable of dynamically linking externally compiled functions. This only works on systems that actually have these functions (only HP-UX systems). If you plan on using this feature, you should probably also use `--enable-shared` to reduce the size of your ‘`.oct`’ files.

--enable-lite-kernel

Compile smaller kernel. This currently requires the dynamic linking functions `dlopen` or `shl_load` and friends so that Octave can load functions at run time that are not loaded at compile time.

--without-blas

Compile and use the generic BLAS and LAPACK versions included with Octave. By default, configure first looks for BLAS and LAPACK matrix libraries on your system, including optimized BLAS implementations such as the free ATLAS 3.0, as well as vendor-tuned libraries. (The use of an optimized BLAS will generally result in several-times faster matrix operations.) Only use this option if your system has BLAS/LAPACK libraries that cause problems for some reason. You can also use **--with-blas=lib** to specify a particular BLAS library `-llib` that configure doesn't check for automatically.

--help Print a summary of the options recognized by the configure script.

See the file 'INSTALL' for more information about the command line options used by configure. That file also contains instructions for compiling in a directory other than where the source is located.

- Run make.

You will need a recent version of GNU Make. Modifying Octave's makefiles to work with other make programs is probably not worth your time. We recommend you get and compile GNU Make instead.

For plotting, you will need to have gnuplot installed on your system. Gnuplot is a command-driven interactive function plotting program. Gnuplot is copyrighted, but freely distributable. The 'gnu' in gnuplot is a coincidence—it is not related to the GNU project or the FSF in any but the most peripheral sense.

To compile Octave, you will need a recent version of GNU Make. You will also need g++ 2.7.2 or later. Version 2.8.0 or egcs 1.0.x should work. Later versions may work, but C++ is still evolving, so don't be too surprised if you run into some trouble.

It is no longer necessary to have `libg++`, but you do need to have the GNU implementation of `libstdc++`. If you are using g++ 2.7.2, `libstdc++` is distributed along with `libg++`, but for later versions, `libstdc++` is distributed separately. For egcs, `libstdc++` is included with the compiler distribution.

If you plan to modify the parser you will also need GNU bison and flex. If you modify the documentation, you will need GNU Texinfo, along with the patch for the `makeinfo` program that is distributed with Octave.

GNU Make, gcc, and `libstdc++`, gnuplot, bison, flex, and Texinfo are all available from many anonymous ftp archives. The primary site is <ftp.gnu.org>, but it is often very busy. A list of sites that mirror the software on <ftp.gnu.org> is available by anonymous ftp from <ftp://ftp.gnu.org/pub/gnu/GNUinfo/FTP>.

If you don't have a Fortran compiler, or if your Fortran compiler doesn't work like the traditional Unix f77, you will need to have the Fortran to C translator `f2c`. You can get `f2c` from any number of anonymous ftp archives. The most recent version of `f2c` is always available from netlib.att.com.

On an otherwise idle Pentium 133 running Linux, it will take somewhere between 1-1/2 to 3 hours to compile everything, depending on whether you are building shared libraries. You will need about 100 megabytes of disk storage to work with (considerably less if you don't compile with debugging symbols). To do that, use the command

```
make CFLAGS=-O CXXFLAGS=-O LDFLAGS=
```

instead of just 'make'.

- If you encounter errors while compiling Octave, first check the list of known problems below to see if there is a workaround or solution for your problem. If not, see [Appendix D \[Trouble\]](#), page 519, for information about how to report bugs.
- Once you have successfully compiled Octave, run 'make install'.

This will install a copy of octave, its libraries, and its documentation in the destination directory. As distributed, Octave is installed in the following directories. In the table below, *prefix* defaults to '/usr/local', *version* stands for the current version number of the interpreter, and *arch* is the type of computer on which Octave is installed (for example, 'i586-unknown-gnu').

'*prefix*/bin'

Octave and other binaries that people will want to run directly.

'*prefix*/lib'

Libraries like libcruft.a and liboctave.a.

'*prefix*/share'

Architecture-independent data files.

'*prefix*/include/octave'

Include files distributed with Octave.

'*prefix*/man/man1'

Unix-style man pages describing Octave.

'*prefix*/info'

Info files describing Octave.

'*prefix*/share/octave/*version*/m'

Function files distributed with Octave. This includes the Octave version, so that multiple versions of Octave may be installed at the same time.

'*prefix*/lib/octave/*version*/exec/*arch*'

Executables to be run by Octave rather than the user.

'*prefix*/lib/octave/*version*/oct/*arch*'

Object files that will be dynamically loaded.

'*prefix*/share/octave/*version*/imagelib'

Image files that are distributed with Octave.

E.1 Installation Problems

This section contains a list of problems (and some apparent problems that don't really mean anything is wrong) that may show up during installation of Octave.

- On some SCO systems, `info` fails to compile if `HAVE_TERMIOS_H` is defined in `'config.h'`. Simply removing the definition from `'info/config.h'` should allow it to compile.
- If `configure` finds `dlopen`, `dlsym`, `dlclose`, and `dlerror`, but not the header file `'dlfcn.h'`, you need to find the source for the header file and install it in the directory `'usr/include'`. This is reportedly a problem with Slackware 3.1. For Linux/GNU systems, the source for `'dlfcn.h'` is in the `ldso` package.
- Building `' .oct'` files doesn't work.

You should probably have a shared version of `libstdc++`. A patch is needed to build shared versions of version 2.7.2 of `libstdc++` on the HP-PA architecture. You can find the patch at <ftp://ftp.cygnum.com/pub/g++/libg++-2.7.2-hppa-gcc-fix>.

- On some alpha systems there may be a problem with the `libdxml` library, resulting in floating point errors and/or segmentation faults in the linear algebra routines called by Octave. If you encounter such problems, then you should modify the `configure` script so that `SPECIAL_MATH_LIB` is not set to `-ldxml`.
- On FreeBSD systems Octave may hang while initializing some internal constants. The fix appears to be to use

```
options      GPL_MATH_EMULATE
```

rather than

```
options      MATH_EMULATE
```

in the kernel configuration files (typically found in the directory `'/sys/i386/conf'`). After making this change, you'll need to rebuild the kernel, install it, and reboot.

- If you encounter errors like

```
passing 'void (*)()' as argument 2 of
      'octave_set_signal_handler(int, void (*)(int))'
```

or

```
warning: ANSI C++ prohibits conversion from '(int)' to '(...)'
```

while compiling `'sighandlers.cc'`, you may need to edit some files in the `gcc` include subdirectory to add proper prototypes for functions there. For example, Ultrix 4.2 needs proper declarations for the `signal` function and the `SIG_IGN` macro in the file `'signal.h'`.

On some systems the `SIG_IGN` macro is defined to be something like this:

```
#define SIG_IGN (void (*)())1
```

when it should really be something like:

```
#define SIG_IGN (void (*)(int))1
```

to match the prototype declaration for the `signal` function. This change should also be made for the `SIG_DFL` and `SIG_ERR` symbols. It may be necessary to change the definitions in `'sys/signal.h'` as well.

The `gcc` `fixincludes` and `fixproto` scripts should probably fix these problems when `gcc` installs its modified set of header files, but I don't think that's been done yet.

You should not change the files in `'/usr/include'`. You can find the `gcc` include directory tree by running the command

```
gcc -print-libgcc-file-name
```

The directory of `gcc` include files normally begins in the same directory that contains the file `'libgcc.a'`.

- Some of the Fortran subroutines may fail to compile with older versions of the Sun Fortran compiler. If you get errors like

```
zgemm.f:
zgemm:
warning: unexpected parent of complex expression subtree
zgemm.f, line 245: warning: unexpected parent of complex
expression subtree
warning: unexpected parent of complex expression subtree
zgemm.f, line 304: warning: unexpected parent of complex
expression subtree
warning: unexpected parent of complex expression subtree
zgemm.f, line 327: warning: unexpected parent of complex
expression subtree
pcc_binval: missing IR_CONV in complex op
make[2]: *** [zgemm.o] Error 1
```

when compiling the Fortran subroutines in the `'libcruft'` subdirectory, you should either upgrade your compiler or try compiling with optimization turned off.

- On NeXT systems, if you get errors like this:

```
/usr/tmp/cc007458.s:unknown:Undefined local symbol LBB7656
/usr/tmp/cc007458.s:unknown:Undefined local symbol LBE7656
```

when compiling `'Array.cc'` and `'Matrix.cc'`, try recompiling these files without `-g`.

- Some people have reported that calls to `shell_cmd` and the pager do not work on SunOS systems. This is apparently due to having `G_HAVE_SYS_WAIT` defined to be 0 instead of 1 when compiling `libg++`.
- On NeXT systems, linking to `'libsys.s.a'` may fail to resolve the following functions

```
_tcgetattr
_tcsetattr
_tcflow
```

which are part of `'libposix.a'`. Unfortunately, linking Octave with `-posix` results in the following undefined symbols.

```
.destructors_used
.constructors_used
_objc_msgSend
_NXGetDefaultValue
_NXRegisterDefaults
_objc_class_name_NXStringTable
_objc_class_name_NXBundle
```

One kluge around this problem is to extract `'termios.o'` from `'libposix.a'`, put it in Octave's `'src'` directory, and add it to the list of files to link together in the makefile. Suggestions for better ways to solve this problem are welcome!

- If Octave crashes immediately with a floating point exception, it is likely that it is failing to initialize the IEEE floating point values for infinity and NaN.

If your system actually does support IEEE arithmetic, you should be able to fix this problem by modifying the function `octave_ieee_init` in the file `'lo-ieee.cc'` to correctly initialize Octave's internal infinity and NaN variables.

If your system does not support IEEE arithmetic but Octave's configure script incorrectly determined that it does, you can work around the problem by editing the file `'config.h'` to not define `HAVE_ISINF`, `HAVE_FINITE`, and `HAVE_ISNAN`.

In any case, please report this as a bug since it might be possible to modify Octave's configuration script to automatically determine the proper thing to do.

Appendix F Emacs Octave Support

The development of Octave code can greatly be facilitated using Emacs with Octave mode, a major mode for editing Octave files which can e.g. automatically indent the code, do some of the typing (with Abbrev mode) and show keywords, comments, strings, etc. in different faces (with Font-lock mode on devices that support it).

It is also possible to run Octave from within Emacs, either by directly entering commands at the prompt in a buffer in Inferior Octave mode, or by interacting with Octave from within a file with Octave code. This is useful in particular for debugging Octave code.

Finally, you can convince Octave to use the Emacs info reader for *help -i*.

All functionality is provided by the Emacs Lisp package EOS (for “Emacs Octave Support”). This chapter describes how to set up and use this package.

Please contact <Kurt.Hornik@wu-wien.ac.at> if you have any questions or suggestions on using EOS.

F.1 Installing EOS

The Emacs package EOS consists of the three files ‘octave-mod.el’, ‘octave-inf.el’, and ‘octave-hlp.el’. These files, or better yet their byte-compiled versions, should be somewhere in your Emacs load-path.

If you have GNU Emacs with a version number at least as high as 19.35, you are all set up, because EOS is respectively will be part of GNU Emacs as of version 19.35.

Otherwise, copy the three files from the ‘emacs’ subdirectory of the Octave distribution to a place where Emacs can find them (this depends on how your Emacs was installed). Byte-compile them for speed if you want.

F.2 Using Octave Mode

If you are lucky, your sysadmins have already arranged everything so that Emacs automatically goes into Octave mode whenever you visit an Octave code file as characterized by its extension ‘.m’. If not, proceed as follows.

1. To begin using Octave mode for all ‘.m’ files you visit, add the following lines to a file loaded by Emacs at startup time, typically your ‘~/ .emacs’ file:

```
(autoload 'octave-mode "octave-mod" nil t)
(setq auto-mode-alist
      (cons '("\\.m$" . octave-mode) auto-mode-alist))
```

2. Finally, to turn on the abbrevs, auto-fill and font-lock features automatically, also add the following lines to one of the Emacs startup files:

```
(add-hook 'octave-mode-hook
  (lambda ()
    (abbrev-mode 1)
    (auto-fill-mode 1)
    (if (eq window-system 'x)
        (font-lock-mode 1))))
```

See the Emacs manual for more information about how to customize Font-lock mode.

In Octave mode, the following special Emacs commands can be used in addition to the standard Emacs commands.

<i>C-h m</i>	Describe the features of Octave mode.
<i>LFD</i>	Reindent the current Octave line, insert a newline and indent the new line (<code>octave-reindent-then-newline-and-indent</code>). An abbrev before point is expanded if <code>abbrev-mode</code> is non-nil.
<i>TAB</i>	Indents current Octave line based on its contents and on previous lines (<code>indent-according-to-mode</code>).
<i>;</i>	Insert an “electric” semicolon (<code>octave-electric-semi</code>). If <code>octave-auto-indent</code> is non-nil, reindent the current line. If <code>octave-auto-newline</code> is non-nil, automatically insert a newline and indent the new line.
<i>‘</i>	Start entering an abbreviation (<code>octave-abbrev-start</code>). If Abbrev mode is turned on, typing <i>C-h</i> or <i>‘?</i> lists all abbrevs. Any other key combination is executed normally. Note that all Octave abbrevs start with a grave accent.
<i>M-LFD</i>	Break line at point and insert continuation marker and alignment (<code>octave-split-line</code>).
<i>M-TAB</i>	Perform completion on Octave symbol preceding point, comparing that symbol against Octave’s reserved words and built-in variables (<code>octave-complete-symbol</code>).
<i>M-C-a</i>	Move backward to the beginning of a function (<code>octave-beginning-of-defun</code>). With prefix argument <i>N</i> , do it that many times if <i>N</i> is positive; otherwise, move forward to the <i>N</i> -th following beginning of a function.
<i>M-C-e</i>	Move forward to the end of a function (<code>octave-end-of-defun</code>). With prefix argument <i>N</i> , do it that many times if <i>N</i> is positive; otherwise, move back to the <i>N</i> -th preceding end of a function.
<i>M-C-h</i>	Puts point at beginning and mark at the end of the current Octave function, i.e., the one containing point or following point (<code>octave-mark-defun</code>).
<i>M-C-q</i>	Properly indents the Octave function which contains point (<code>octave-indent-defun</code>).
<i>M-;</i>	If there is no comment already on this line, create a code-level comment (started by two comment characters) if the line is empty, or an in-line comment (started by one comment character) otherwise (<code>octave-indent-for-comment</code>). Point is left after the start of the comment which is properly aligned.
<i>C-c ;</i>	Puts the comment character <i>#</i> (more precisely, the string value of <code>octave-comment-start</code>) at the beginning of every line in the region (<code>octave-comment-region</code>). With just <i>C-u</i> prefix argument, uncomment each line in the region. A numeric prefix argument <i>N</i> means use <i>N</i> comment characters.
<i>C-c :</i>	Uncomments every line in the region (<code>octave-uncomment-region</code>).
<i>C-c C-p</i>	Move one line of Octave code backward, skipping empty and comment lines (<code>octave-previous-code-line</code>). With numeric prefix argument <i>N</i> , move that many code lines backward (forward if <i>N</i> is negative).

- C-c C-n** Move one line of Octave code forward, skipping empty and comment lines (`octave-next-code-line`). With numeric prefix argument *N*, move that many code lines forward (backward if *N* is negative).
- C-c C-a** Move to the ‘real’ beginning of the current line (`octave-beginning-of-line`). If point is in an empty or comment line, simply go to its beginning; otherwise, move backwards to the beginning of the first code line which is not inside a continuation statement, i.e., which does not follow a code line ending in ‘...’ or ‘\’, or is inside an open parenthesis list.
- C-c C-e** Move to the ‘real’ end of the current line (`octave-end-of-line`). If point is in a code line, move forward to the end of the first Octave code line which does not end in ‘...’ or ‘\’ or is inside an open parenthesis list. Otherwise, simply go to the end of the current line.
- C-c M-C-n** Move forward across one balanced begin-end block of Octave code (`octave-forward-block`). With numeric prefix argument *N*, move forward across *n* such blocks (backward if *N* is negative).
- C-c M-C-p** Move back across one balanced begin-end block of Octave code (`octave-backward-block`). With numeric prefix argument *N*, move backward across *N* such blocks (forward if *N* is negative).
- C-c M-C-d** Move forward down one begin-end block level of Octave code (`octave-down-block`). With numeric prefix argument, do it that many times; a negative argument means move backward, but still go down one level.
- C-c M-C-u** Move backward out of one begin-end block level of Octave code (`octave-backward-up-block`). With numeric prefix argument, do it that many times; a negative argument means move forward, but still to a less deep spot.
- C-c M-C-h** Put point at the beginning of this block, mark at the end (`octave-mark-block`). The block marked is the one that contains point or follows point.
- C-c]** Close the current block on a separate line (`octave-close-block`). An error is signaled if no block to close is found.
- C-c f** Insert a function skeleton, prompting for the function’s name, arguments and return values which have to be entered without parens (`octave-insert-defun`).
- C-c C-h** Search the function, operator and variable indices of all info files with documentation for Octave for entries (`octave-help`). If used interactively, the entry is prompted for with completion. If multiple matches are found, one can cycle through them using the standard ‘,’ (`Info-index-next`) command of the Info reader.

The variable `octave-help-files` is a list of files to search through and defaults to `("octave")`. If there is also an Octave Local Guide with corresponding info file, say, `'octave-LG'`, you can have `octave-help` search both files by

```
(setq octave-help-files '("octave" "octave-LG"))
```

in one of your Emacs startup files.

A common problem is that the `(RET)` key does *not* indent the line to where the new text should go after inserting the newline. This is because the standard Emacs convention

is that `(RET)` (aka `C-m`) just adds a newline, whereas `(LFD)` (aka `C-j`) adds a newline and indents it. This is particularly inconvenient for users with keyboards which do not have a special `(LFD)` key at all; in such cases, it is typically more convenient to use `(RET)` as the `(LFD)` key (rather than typing `C-j`).

You can make `(RET)` do this by adding

```
(define-key octave-mode-map "\C-m"
  'octave-reindent-then-newline-and-indent)
```

to one of your Emacs startup files. Another, more generally applicable solution is

```
(defun RET-behaves-as-LFD ()
  (let ((x (key-binding "\C-j")))
    (local-set-key "\C-m" x)))
(add-hook 'octave-mode-hook 'RET-behaves-as-LFD)
```

(this works for all modes by adding to the startup hooks, without having to know the particular binding of `(RET)` in that mode!). Similar considerations apply for using `(M-RET)` as `(M-LFD)`. As Barry A. Warsaw <bwarshaw@cnri.reston.va.us> says in the documentation for his `cc-mode`, “This is a very common question. :-) If you want this to be the default behavior, don’t lobby me, lobby RMS!”

The following variables can be used to customize Octave mode.

`octave-auto-indent`

Non-nil means auto-indent the current line after a semicolon or space. Default is nil.

`octave-auto-newline`

Non-nil means auto-insert a newline and indent after semicolons are typed. The default value is nil.

`octave-blink-matching-block`

Non-nil means show matching begin of block when inserting a space, newline or ‘;’ after an else or end keyword. Default is t. This is an extremely useful feature for automatically verifying that the keywords match—if they don’t, an error message is displayed.

`octave-block-offset`

Extra indentation applied to statements in block structures. Default is 2.

`octave-continuation-offset`

Extra indentation applied to Octave continuation lines. Default is 4.

`octave-continuation-string`

String used for Octave continuation lines. Normally ‘\’.

`octave-mode-startup-message`

If t (default), a startup message is displayed when Octave mode is called.

If Font Lock mode is enabled, Octave mode will display

- strings in `font-lock-string-face`
- comments in `font-lock-comment-face`
- the Octave reserved words (such as all block keywords) and the text functions (such as ‘cd’ or ‘who’) which are also reserved using `font-lock-keyword-face`

- the built-in operators ('&&', '<>', ...) using `font-lock-reference-face`
- and the function names in function declarations in `font-lock-function-name-face`.

There is also rudimentary support for Imenu (currently, function names can be indexed).

You can generate TAGS files for Emacs from Octave '.m' files using the shell script `octave-tags` that is installed alongside your copy of Octave.

Customization of Octave mode can be performed by modification of the variable `octave-mode-hook`. If the value of this variable is non-`nil`, turning on Octave mode calls its value.

If you discover a problem with Octave mode, you can conveniently send a bug report using `C-c C-b` (`octave-submit-bug-report`). This automatically sets up a mail buffer with version information already added. You just need to add a description of the problem, including a reproducible test case and send the message.

F.3 Running Octave From Within Emacs

The package 'octave' provides commands for running an inferior Octave process in a special Emacs buffer. Use

`M-x run-octave`

to directly start an inferior Octave process. If Emacs does not know about this command, add the line

```
(autoload 'run-octave "octave-inf" nil t)
```

to your '.emacs' file.

This will start Octave in a special buffer the name of which is specified by the variable `inferior-octave-buffer` and defaults to "`*Inferior Octave*`". From within this buffer, you can interact with the inferior Octave process 'as usual', i.e., by entering Octave commands at the prompt. The buffer is in Inferior Octave mode, which is derived from the standard Comint mode, a major mode for interacting with an inferior interpreter. See the documentation for `comint-mode` for more details, and use `C-h b` to find out about available special keybindings.

You can also communicate with an inferior Octave process from within files with Octave code (i.e., buffers in Octave mode), using the following commands.

- | | |
|----------------------|--|
| <code>C-c i l</code> | Send the current line to the inferior Octave process (<code>octave-send-line</code>). With positive prefix argument <i>N</i> , send that many lines. If <code>octave-send-line-auto-forward</code> is non- <code>nil</code> , go to the next unsent code line. |
| <code>C-c i b</code> | Send the current block to the inferior Octave process (<code>octave-send-block</code>). |
| <code>C-c i f</code> | Send the current function to the inferior Octave process (<code>octave-send-defun</code>). |
| <code>C-c i r</code> | Send the region to the inferior Octave process (<code>octave-send-region</code>). |
| <code>C-c i s</code> | Make sure that 'inferior-octave-buffer' is displayed (<code>octave-show-process-buffer</code>). |
| <code>C-c i h</code> | Delete all windows that display the inferior Octave buffer (<code>octave-hide-process-buffer</code>). |
| <code>C-c i k</code> | Kill the inferior Octave process and its buffer (<code>octave-kill-process</code>). |

The effect of the commands which send code to the Octave process can be customized by the following variables.

`octave-send-echo-input`

Non-`nil` means echo input sent to the inferior Octave process. Default is `t`.

`octave-send-show-buffer`

Non-`nil` means display the buffer running the Octave process after sending a command (but without selecting it). Default is `t`.

If you send code and there is no inferior Octave process yet, it will be started automatically.

The startup of the inferior Octave process is highly customizable. The variable `inferior-octave-startup-args` can be used for specifying command line arguments to be passed to Octave on startup as a list of strings. For example, to suppress the startup message and use ‘traditional’ mode, set this to `'("-q" "--traditional")`. You can also specify a startup file of Octave commands to be loaded on startup; note that these commands will not produce any visible output in the process buffer. Which file to use is controlled by the variable `inferior-octave-startup-file`. If this is `nil`, the file `‘~/.emacs-octave’` is used if it exists.

And finally, `inferior-octave-mode-hook` is run after starting the process and putting its buffer into Inferior Octave mode. Hence, if you like the up and down arrow keys to behave in the interaction buffer as in the shell, and you want this buffer to use nice colors, add

```
(add-hook 'inferior-octave-mode-hook
  (lambda ()
    (turn-on-font-lock)
    (define-key inferior-octave-mode-map [up]
      'comint-previous-input)
    (define-key inferior-octave-mode-map [down]
      'comint-next-input)))
```

to your `‘.emacs’` file. You could also swap the roles of `C-a` (`beginning-of-line`) and `C-c C-a` (`comint-bol`) using this hook.

Note that if you set your Octave prompts to something different from the defaults, make sure that `inferior-octave-prompt` matches them. Otherwise, *nothing* will work, because Emacs will not know when Octave is waiting for input, or done sending output.

F.4 Using the Emacs Info Reader for Octave

You can also set up the Emacs Info reader for dealing with the results of Octave’s `‘help -i’`. For this, the package `‘gnuserv’` needs to be installed, which unfortunately still does not come with GNU Emacs (it does with XEmacs). It can be retrieved from any GNU Emacs Lisp Code Directory archive, e.g. <http://ftp.cis.ohio-state.edu/pub/gnu/emacs/elisp-archive>, in the `‘packages’` subdirectory. A recent version of `gnuserv` is available from <http://www.meltin.net/hacks/emacs/src/gnuserv-3.12.2.tar.gz>.

If `‘gnuserv’` is installed, add the lines

```
(autoload 'octave-help "octave-hlp" nil t)
(require 'gnuserv)
(gnuserv-start)
```

to your `‘.emacs’` file.

You can use either ‘plain’ Emacs Info or the function `octave-help` as your Octave info reader (for `‘help -i’`). In the former case, use `info_program ("info-emacs-info")`. The latter is perhaps more attractive because it allows to look up keys in the indices of *several* info files related to Octave (provided that the Emacs variable `octave-help-files` is set correctly). In this case, use `info_program ("info-emacs-octave-help")`.

If you use Octave from within Emacs, these settings are best done in the `‘~/.emacs-octave’` startup file (or the file pointed to by the Emacs variable `inferior-octave-startup-file`).

Appendix G GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 - b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
 - c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software

which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) yyyy  name of author
```

```
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy  name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.
```

```
signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

Concept Index

#

'#' 28
'#!' 27

%

'%' 28

-

--braindead 14
--debug 13
--echo-commands 13
--exec-path *path* 13
--help 13
--image-path *path* 13
--info-file *filename* 13
--info-program *program* 14
--interactive 14
--no-init-file 14
--no-line-editing 14
--no-site-file 14
--norc 14
--path *path* 14
--quiet 14
--silent 14
--traditional 14
--verbose 15
--version 15
-? 13
-d 13
-f 14
-h 13
-i 14
-p *path* 14
-q 14
-v 15
-V 15
-x 13

.

..... 121
... continuation marker 115
.octaverc 16

\

\ continuation marker 115

~

~/octaverc 16

A

acknowledgements 1
addition 91
and operator 94
anonymous functions 131
answers, incorrect 520, 521
any key 8
arguments in function call 89
arithmetic operators 91
assignment expressions 96
assignment operators 96

B

body of a loop 109
boolean expressions 94
boolean operators 94
break statement 112
bug criteria 520
bug report mailing lists 520
bugs 519
bugs, investigating 522
bugs, known 519
bugs, reporting 520
built-in data types 31
built-in function 10

C

case statement 107
catch 114
cell arrays 32, 69
character strings 32, 47
Cholesky factorization 235, 267, 268
clearing the screen 20
coding standards 509
command and output logs 25
command completion 21
command descriptions 10
command echoing 25
command history 21
command options 13
command-line editing 19
comments 28
comparison expressions 93
complex-conjugate transpose 91
compounding, value of 331
containers 69
continuation lines 115
continue statement 113
contributing to Octave 3
contributors 1
conversion specifications (**printf**) 166
conversion specifications (**scanf**) 171

copyright	537
core dump	520
customizing the prompt	24

D

DAE	283
data structures	32, 63
data types	31
data types, built-in	31
data types, user-defined	32
decrement operator	98
default arguments	124
defining functions	117
description format	9
diary of commands and output	25
Differential Equations	283
diffs, submitting	522
distribution of Octave	4
division	91
do-until statement	110
documentation notation	8
documenting Octave programs	28
Dulmage-Mendelsohn decomposition	264
dynamic-linking	461

E

echoing executing commands	25
editing the command line	19
element-by-element evaluation	94
else statement	105
elseif statement	105
Emacs TAGS files	535
end statement	105
end_try_catch	114
end_unwind_protect	114
endfor statement	110
endfunction statement	117
endif statement	105
endswitch statement	107
endwhile statement	109
equality operator	93
equality, tests for	93
equations, nonlinear	243
erroneous messages	520
erroneous results	520, 521
error message notation	9
error messages	26
error messages, incorrect	520
escape sequence notation	47
evaluation notation	8
executable scripts	27
execution speed	509
exiting octave	5, 16
exponentiation	91
expression, range	39
expressions	87

expressions, assignment	96
expressions, boolean	94
expressions, comparison	93
expressions, logical	94

F

factorial function	91
fatal signal	520
financial functions	331
flag character (printf)	167
flag character (scanf)	171
flying high and fast	77
fonts	8
for statement	110
Fordyce, A. P.	102
Frobenius norm	234
function descriptions	9
function file	10, 124
function statement	117
functions, user-defined	117
funding Octave development	3

G

getting a good job	77
global statement	77
global variables	77
graphics	179
greater than operator	93

H

handle, function handles	131
header comments	510
help, on-line	17
help, where to find	523
Hermitian operator	91
Hessenberg decomposition	235
history	1
history of commands	21

I

if statement	105
improving Octave	520, 522
incorrect error messages	520
incorrect output	520, 521
incorrect results	520, 521
increment operator	98
infinity norm	234
initialization	16
inline, inline functions	131
input conversions, for scanf	172
input history	21
installation trouble	519
installing Octave	525
introduction	5

invalid input 520

J

job hunting 77

K

known causes of trouble 519

L

less than operator 93
loadable function 10
loading data 156
logging commands and output 25
logical expressions 94
logical operators 94
loop 109
looping over structure elements 111
LP 293
LU decomposition 236, 268, 275
lvalue 96

M

mapping function 10
matching failure, in `scanf` 171
matrices 36
matrix multiplication 91
maximum field width (`scanf`) 172
messages, error 26
mex 485
mex-files 485
minimum field width (`printf`) 167
missing data 32
mkocfile 461
money 331
multiplication 91

N

negation 91
NLP 293
nonlinear equations 243
nonlinear programming 293
not operator 94
numeric constant 31, 35
numeric value 31, 35

O

oct 461
oct-files 461
Octave command options 13
`octave-tags` 535
ODE 283
on-line help 17

operator precedence 99
operators, arithmetic 91
operators, assignment 96
operators, boolean 94
operators, decrement 98
operators, increment 98
operators, logical 94
operators, relational 93
optimization 293
options, Octave command 13
or operator 94
oregonator 285
`otherwise` statement 107
output conversions, for `printf` 167

P

partial fraction expansion 337
patches, submitting 522
`persistent` statement 78
persistent variables 78
plotting 179
precision (`printf`) 167
printing notation 9
program, self contained 27
programs 28
prompt customization 24

Q

QP 293
QR factorization 236, 269
quadratic programming 293
quitting octave 5, 16
quotient 91

R

range expressions 39
relational operators 93
reporting bugs 519, 520
results, incorrect 520, 521

S

saving data 156
Schur decomposition 238
script files 117
scripts 27
self contained programs 27
short-circuit evaluation 95
side effect 96
singular value decomposition 239
speedups 509
standards of coding style 509
startup 16
startup files 16
statements 105

strings..... 32, 47
 Structural Rank..... 270
 structure elements, looping over..... 111
 structures..... 32, 63
 submitting diffs..... 522
 submitting patches..... 522
 subtraction..... 91
 suggestions..... 520
switch statement..... 107

T

TAGS..... 535
 test functions..... 501
 tests for equality..... 93
 tips..... 509
 transpose..... 91
 transpose, complex-conjugate..... 91
 troubleshooting..... 519
try statement..... 114

U

unary minus..... 91

undefined behavior..... 520
 undefined function value..... 520
unwind_protect statement..... 114
unwind_protect_cleanup..... 114
 use of comments..... 28
 user-defined data types..... 32
 user-defined functions..... 117
 user-defined variables..... 77
 Utility Functions..... 216, 249

V

variable descriptions..... 10
 variable-length argument lists..... 121
 variable-length return lists..... 122
 variables, global..... 77
 variables, persistent..... 78
 variables, user-defined..... 77

W

warranty..... 537
while statement..... 109
 wrong answers..... 520, 521

Variable Index

A

ans 151

B

beep_on_error 82

C

completion_append_char 82

crash_dumps_octave_core 83

D

default_save_options 83

do_what_i_mean_not_what_i_say 10

E

EDITOR 82, 84

EXEC_PATH 82

F

F_DUPFD 447

F_GETFD 447

F_GETFL 447

F_SETFD 447

F_SETFL 447

fixed_point_format 83

G

gnuplot_binary 83

H

history_file 83

history_size 83

I

ignore_function_time_stamp 83

M

max_recursion_depth 83

O

O_APPEND 447

O_ASYNC 447

O_CREAT 447

O_NONBLOCK 447

O_RDONLY 447

O_RDWR 447

O_SYNC 447

O_WRONLY 447

OCTAVE_EXEC_PATH 84

OCTAVE_HISTFILE 85

OCTAVE_HISTSIZE 85

OCTAVE_HOME 82

OCTAVE_INFO_FILE 85

OCTAVE_INFO_PROGRAM 85

OCTAVE_PATH 85

output_max_field_width 83

output_precision 83

P

page_screen_output 83

PAGER 82

print_answer_id_name 83

print_empty_dimensions 83

PS1 82

PS2 82

PS4 82

R

return_last_computed_value 84

S

save_precision 84

saving_history 84

sigdump_dumps_octave_core 84

sigterm_dumps_octave_core 84

silent_functions 84

split_long_rows 84

struct_levels_to_print 84

suppress_verbose_help_message 84

Function Index

(
() 15

[
[fun, helpstring] = 18

A

abscddim 380
abs 218
accumarray 222
acos 219
acosd 221
acosh 220
acot 219
acotd 221
acoth 220
acsc 219
acscd 221
acsch 220
addpath 126
airy 224
all 195
alaldemo 380
angle 218
anova 310
any 195
arch_fit 409
arch_rnd 409
arch_test 410
are 376
arg 218
argnames 132
argv 15
arma_rnd 410
arrayfun 203
asctime 429
asec 219
asecd 221
asech 220
asin 219
asind 221
asinh 220
assert 504
assignin 104
atan 219
atan2 220
atand 221
atanh 220
atexit 17
autocor 410
autocov 410
autoload 128

autoreg_matrix 410
autumn 418
axes 179
axis 188
axis2dlim 401

B

balance 231
bar 182
bartlett 411
bartlett_test 310
base2dec 58
bddemo 368
beep 139
beep_on_error 139
besselh 223
besseli 223
besselj 223
besselk 223
bessely 223
beta 224
betacdf 318
betainc 224
betainv 318
betaln 224
betapdf 319
betarnd 325
bicubic 348
bin2dec 56
bincoeff 225
binocdf 319
binoinv 319
binopdf 319
binornd 325
bitand 43
bitcmp 43
bitget 42
bitmax 43
bitor 43
bitset 42
bitshift 43
bitxor 43
blackman 411
blanks 48
blkdiag 203
bode 389
bode_bounds 390
bone 419
box 191
bug_report 519, 520
buildssic 368
builtin 128
bunzip2 441
byte_size 465

C

<code>c =</code>	182
<code>c2d</code>	385
<code>calendar</code>	434
<code>canonicalize_file_name</code>	440
<code>cart2pol</code>	227
<code>cart2sph</code>	227
<code>cast</code>	31
<code>cat</code>	199
<code>cauchy_cdf</code>	319
<code>cauchy_inv</code>	319
<code>cauchy_pdf</code>	319
<code>cauchy_rnd</code>	326
<code>ccolamd</code>	260
<code>cd</code>	10, 448
<code>ceil</code>	213
<code>cell</code>	70
<code>cell2mat</code>	75
<code>cell2struct</code>	75
<code>cellfun</code>	74
<code>cellidx</code>	73
<code>cellstr</code>	73
<code>center</code>	306
<code>char</code>	48
<code>chdir</code>	10, 448
<code>chi2cdf</code>	319
<code>chi2inv</code>	319
<code>chi2rnd</code>	326
<code>chisquare_pdf</code>	319
<code>chisquare_test_homogeneity</code>	311
<code>chisquare_test_independence</code>	311
<code>chol</code>	235
<code>chol2inv</code>	235
<code>cholin</code>	235
<code>circshift</code>	200
<code>class</code>	31
<code>clear</code>	81
<code>clf</code>	190
<code>clock</code>	431
<code>cloglog</code>	308
<code>close</code>	190
<code>closereq</code>	190
<code>colamd</code>	261, 262
<code>colloc</code>	281
<code>colormap</code>	418
<code>colperm</code>	263
<code>columns</code>	33
<code>common_size</code>	197
<code>commutation_matrix</code>	226
<code>compan</code>	335
<code>complement</code>	334
<code>completion_append_char</code>	21
<code>completion_matches</code>	21
<code>complex</code>	35
<code>computer</code>	451
<code>cond</code>	232
<code>confirm_recursive_rmdir</code>	436
<code>conj</code>	218

<code>contourc</code>	182
<code>conv</code>	335
<code>conv2</code>	335
<code>cool</code>	419
<code>copper</code>	419
<code>copyfile</code>	440
<code>cor</code>	305
<code>cor_test</code>	311
<code>corrcoef</code>	305
<code>cos</code>	219
<code>cosd</code>	221
<code>cosh</code>	219
<code>cot</code>	219
<code>cotd</code>	221
<code>coth</code>	220
<code>cov</code>	304
<code>cplxpair</code>	213
<code>cputime</code>	431
<code>crash_dumps_octave_core</code>	160
<code>create_set</code>	333
<code>cross</code>	226
<code>csc</code>	219
<code>cscd</code>	221
<code>csch</code>	220
<code>csymamd</code>	263
<code>ctime</code>	428
<code>ctrb</code>	380
<code>cumprod</code>	222
<code>cumsum</code>	222
<code>cumtrapz</code>	281
<code>cut</code>	309

D

<code>d2c</code>	386
<code>damp</code>	387
<code>dare</code>	376
<code>daspk</code>	285
<code>daspk_options</code>	286
<code>dasrt</code>	289
<code>dasrt_options</code>	290
<code>date</code>	431
<code>datestr</code>	432
<code>datevec</code>	434
<code>dbc_clear</code>	148
<code>dbstatus</code>	148
<code>dbstop</code>	147
<code>dbtype</code>	149
<code>dbwhere</code>	149
<code>dcgain</code>	387
<code>deal</code>	123
<code>deblank</code>	52
<code>debug_on_error</code>	147
<code>debug_on_interrupt</code>	147
<code>debug_on_warning</code>	147
<code>dec2base</code>	57
<code>dec2bin</code>	56
<code>dec2hex</code>	57

deconv.....	335
default_save_options.....	159
del2.....	213
delete.....	190
demo.....	505
DEMOcontrol.....	351
det.....	232
detrend.....	405
dgkfdemo.....	393
dgram.....	378
diag.....	208
diary.....	25
diff.....	196
diffpara.....	411
dims.....	465
dir.....	449
discrete_cdf.....	319
discrete_inv.....	320
discrete_pdf.....	320
discrete_rnd.....	326
disp.....	151
dispatch.....	128
dkalman.....	396
dlqe.....	395
dlqr.....	395
dlyap.....	378
dmperm.....	264
dmr2d.....	387
dmult.....	232
do_string_escapes.....	59
document.....	82
dos.....	444
dot.....	232
double.....	35
drawnow.....	181
dre.....	377
dup2.....	446
duplication_matrix.....	227
durbinlevinson.....	411

E

e.....	228
echo.....	25
echo_executing_commands.....	26
edit_history.....	22
EDITOR.....	23
eig.....	232
elem.....	465
empirical_cdf.....	320
empirical_inv.....	320
empirical_pdf.....	320
empirical_rnd.....	326
endgrent.....	451
endpwent.....	450
eomday.....	435
eps.....	228
erf.....	225

erfc.....	225
erfinv.....	225
errno.....	141
errno_list.....	141
error.....	137
errorbar.....	185
etime.....	431
etree.....	256
etreeplot.....	256
eval.....	101
evalin.....	104
example.....	506
exec.....	446
EXEC_PATH.....	445
exist.....	80
exit.....	16
exp.....	214
expcdf.....	320
expinv.....	320
expm.....	240
exppdf.....	320
exprnd.....	326
eye.....	204

F

f_test_regression.....	312
factor.....	214
factorial.....	214
fail.....	505
false.....	45
fcdf.....	320
fclose.....	164
fcntl.....	447
fdisp.....	160
feof.....	177
ferror.....	177
feval.....	102
fflush.....	155
fft.....	405
fft2.....	406, 407
fftconv.....	407
fftfilt.....	407
fftn.....	407
fftshift.....	411
fftw.....	405
fgetl.....	165
fgets.....	165
fieldnames.....	66
figure.....	192
file_in_loadpath.....	127
file_in_path.....	439
fileattrib.....	438
fileparts.....	440
filesep.....	440
filter.....	407
filter2.....	408
find.....	196

findstr.....	52
finite.....	196
finv.....	320
fir2sys.....	353
fix.....	214
fixed_point_format.....	38
flipdim.....	198
fliplr.....	197
flipud.....	197
floor.....	214
fmod.....	214
fnmatch.....	439
foo.....	9
fopen.....	163
fork.....	445
format.....	151
formula.....	133
fortran_vec.....	465
fpdf.....	320
fplot.....	181
fprintf.....	165
fputs.....	164
fractdiff.....	412
frdemo.....	389
fread.....	173
freport.....	177
freqchkw.....	390
freqz.....	408, 409
freqz_plot.....	409
frewind.....	178
frnd.....	327
fscanf.....	170
fseek.....	178
fsolve.....	243
fsolve_options.....	243
ftell.....	178
full.....	252
fullfile.....	440, 441
func2str.....	131
function_name.....	17
functions.....	131
fv.....	331
fvl.....	331
fwrite.....	176

G

gamcdf.....	321
gaminv.....	321
gamma.....	225
gammainc.....	225
gammaln.....	226
gampdf.....	321
gamrnd.....	327
gca.....	189
gcd.....	214
gcf.....	189
genpath.....	126

geocdf.....	321
geoinv.....	321
geopdf.....	321
geornd.....	327
get.....	189
getegid.....	448
getenv.....	448
geteuid.....	448
getfield.....	66
getgid.....	448
getgrent.....	450
getgrgid.....	450
getgrnam.....	451
getgrp.....	448
getpid.....	448
getppid.....	448
getpwent.....	450
getpwnam.....	450
getpwuid.....	450
getrusage.....	452
getuid.....	448
givens.....	232
glob.....	439
glpk.....	293
gls.....	302
gmtime.....	428
gnuplot_binary.....	194
gnuplot_use_title_option.....	194
gplot.....	257
gradient.....	215
gram.....	378
gray.....	419
gray2ind.....	418
grid.....	191
gunzip.....	441

H

h2norm.....	381
h2syn.....	396
hadamard.....	209
hamming.....	412
hankel.....	210
hanning.....	412
help.....	17
hess.....	235
hex2dec.....	57
hilb.....	210
hinf_ctr.....	397
hinfdemo.....	393
hinfnorm.....	381
hinfosyn.....	397
hinfosyn_chk.....	398
hinfosyn_ric.....	398
hist.....	183
history.....	22
history_file.....	23
history_size.....	23

history_timestamp_format_string.....	23
hold.....	181
horzcat.....	199
hot.....	419
hotelling_test.....	312
hotelling_test_2.....	312
housh.....	239
hsv.....	419
hsv2rgb.....	421
hurst.....	412
hygecdf.....	321
hygeinv.....	321
hygepdf.....	321
hygernd.....	327

I

I.....	227
ifft.....	406
ifftn.....	407
ifftshift.....	411
ignore_function_time_stamp.....	125
imag.....	218
image.....	416
IMAGE_PATH.....	415
image_viewer.....	417
imagesc.....	416
impulse.....	388
imshow.....	416
ind2gray.....	418
ind2rgb.....	418
ind2sub.....	89
index.....	52
Inf.....	228
info_file.....	18
info_program.....	18
inline.....	132
input.....	155
inputname.....	119
int16.....	40
int2str.....	50
int32.....	40
int64.....	40
int8.....	40
interp1.....	341
interp2.....	345
interp3.....	346
interpft.....	343
interp.....	346
intersect.....	333
intmax.....	41
intmin.....	41
inv.....	233
inverse.....	233
invhilb.....	210
ipermute.....	200
iqr.....	308
irr.....	331

is_abcd.....	382
is_controllable.....	382
is_detectable.....	382
is_dgkf.....	383
is_digital.....	384
is_duplicate_entry.....	195
is_leap_year.....	431
is_observable.....	384
is_sample.....	384
is_signal_list.....	385
is_siso.....	384
is_stabilizable.....	385
is_stable.....	385
isa.....	31
isalnum.....	60
isalpha.....	60
isascii.....	60
iscell.....	70
iscellstr.....	73
ischar.....	49
iscntrl.....	60
iscommand.....	133
iscomplex.....	45
isdefinite.....	46
isdigit.....	60
isdir.....	439
isempty.....	33
isequal.....	93
isequalwithqualnans.....	93
isfield.....	66
isfigure.....	190
isglobal.....	78
isgraph.....	60
ishold.....	181
isieee.....	452
isinf.....	196
isinteger.....	40
isletter.....	60
islogical.....	46
islower.....	60
ismatrix.....	45
ismember.....	333
isna.....	32
isnan.....	196
isnumeric.....	45
ispc.....	451
isprime.....	46
isprint.....	60
ispunct.....	60
israwcommand.....	134
isreal.....	45
isscalar.....	45
isspace.....	61
issparse.....	253
issquare.....	46
isstruct.....	65
issymmetric.....	46
isunix.....	451

isupper.....	61
isvarname.....	77
isvector.....	45
isxdigit.....	61

J

jet.....	419
jet707.....	369

K

kbhit.....	156
kendall.....	308
keyboard.....	149
kill.....	447
kolmogorov_smirnov_cdf.....	322
kolmogorov_smirnov_test.....	312
kolmogorov_smirnov_test_2.....	313
kron.....	241
kruskal_wallis_test.....	313
krylov.....	240
kurtosis.....	305

L

laplace_cdf.....	322
laplace_inv.....	322
laplace_pdf.....	322
laplace_rnd.....	327
lasterr.....	140
lasterror.....	140
lastwarn.....	142
lcm.....	215
legend.....	190
legendre.....	226
length.....	33
lgamma.....	226
lin2mu.....	423
line.....	181
link.....	435
linspace.....	209
load.....	158
loadaudio.....	423
loadimage.....	415
localtime.....	428
log.....	215
log10.....	215
log2.....	215
logical.....	44
logistic_cdf.....	322
logistic_inv.....	322
logistic_pdf.....	322
logistic_regression.....	317
logistic_rnd.....	327
logit.....	308
loglog.....	183
loglogerr.....	186

logm.....	240
logncdf.....	322
logninv.....	322
lognpdf.....	323
lognrnd.....	327
logspace.....	209
lookfor.....	18
lookup.....	345
lqe.....	399
lqg.....	399
lqr.....	400
ls.....	449
ls_command.....	449
lsim.....	400
lsode.....	283
lsode_options.....	284
lstat.....	436, 438
ltifr.....	390
lu.....	236
luinc.....	275
lyap.....	379

M

magic.....	210
mahalanobis.....	306
makeinfo_program.....	19
manova.....	313
mark_as_command.....	133
mark_as_rawcommand.....	134
mat2cell.....	71
mat2str.....	49
matrix_type.....	233
max.....	216
max_recursion_depth.....	91
mcnemar_test.....	314
md5sum.....	453
mean.....	303
meansq.....	304
median.....	303
menu.....	156
mesh.....	187
meshc.....	188
meshgrid.....	188
mex.....	486
mexext.....	486
mfilename.....	125
min.....	216
minfo.....	367
mislocked.....	129
mkdir.....	436
mkfifo.....	436
mkoctfile.....	461
mkpp.....	339
mkstemp.....	176
mktime.....	429
mlock.....	129
mod.....	216

moddemo..... 401
moment..... 306
more..... 154
movefile..... 440
mu2lin..... 423
munlock..... 129

N

NA..... 32
NaN..... 228
nargchk..... 121
nargin..... 119
nargout..... 120
native_float_format..... 160
nbincdf..... 323
nbininv..... 323
nbinpdf..... 323
nbinrnd..... 328
nchoosek..... 306, 517
ndgrid..... 188
ndims..... 33
nelem..... 465
newplot..... 181
newtroot..... 102
nextpow2..... 217
nichols..... 391
nnz..... 254
nonzeros..... 254
norm..... 234
normcdf..... 323
normest..... 266
norminv..... 323
normpdf..... 323
normrnd..... 328
now..... 427
nper..... 331
npv..... 331
nthroot..... 217
ntsc2rgb..... 421
null..... 234
num2cell..... 70
num2str..... 50
numel..... 33
nyquist..... 390
nzmax..... 254

O

obsv..... 381
ocean..... 419
octave_config_info..... 452
octave_core_file_limit..... 161
octave_core_file_name..... 161
octave_core_file_options..... 161
OCTAVE_HOME..... 452
OCTAVE_VERSION..... 452
ols..... 302

ones..... 204
operator ()..... 465
ord2..... 369
orderfields..... 66
orient..... 193
orth..... 234
output_max_field_width..... 37
output_precision..... 37

P

P_tmpdir..... 441
pack..... 442
page_output_immediately..... 155
page_screen_output..... 154
PAGER..... 154
PAGER_FLAGS..... 154
parallel..... 375
parseparams..... 122
pascal..... 211
path..... 126
pathdef..... 126
pathsep..... 126
pause..... 432
pcg..... 271
pchip..... 412
pclose..... 444
pcr..... 273
peaks..... 193, 194
periodogram..... 412
perms..... 307
permute..... 199
pi..... 228
pink..... 420
pinv..... 235
pipe..... 446
pkg..... 455
place..... 400
playaudio..... 424
plot..... 179
plot3..... 187
pmt..... 332
poisscdf..... 323
poissinv..... 324
poisspdf..... 324
poissrnd..... 328
pol2cart..... 227
polar..... 183
poly..... 336
polyarea..... 349
polyder..... 336
polyderiv..... 336
polyfit..... 336
polygcd..... 336
polyinteg..... 337
polyout..... 338
polyreduce..... 337
polyval..... 337

polyvalm	337
popen	444
popen2	444
postpad	203
pow2	217
ppplot	309
ppval	338
prepad	203
primes	217
print	192
print_answer_id_name	154
print_empty_dimensions	39
print_usage	138
printf	165
prism	420
probit	308
prod	222
program_name	15
prompt	401
prop_test_2	314
PS1	24
PS2	25
PS4	25
putenv	448
puts	164
pv	332
pvl	332
pwd	449
pzmap	382

Q

qconj	425
qcoordinate_plot	426
qderiv	425
qderivmat	425
qinv	425
qmult	426
qp	299
qqplot	309
qr	236
qtrans	426
qtransv	426
qtransvmat	426
quad	279
quad_options	279
quadl	280
quaternion	425
quit	16
qz	237
qzhess	238
qzval	379

R

rainbow	420
rand	205
rande	206

randg	207
randn	206
randp	206
randperm	208
range	308
rank	235
ranks	307
rat	161
rate	332
rats	162
read_readline_init_file	24
readdir	436
readlink	435
real	218
realmax	229
realmin	229
record	424
rectangle_lw	412
rectangle_sw	413
regexp	54
regexpi	55
regexprep	55
rehash	127
rem	217
rename	435
repmat	205
reshape	200
residue	337
resize	465
rethrow	141
return	124
rgb2hsv	421
rgb2ind	418
rgb2ntsc	421
rindex	52
rldemo	401
rlocus	401
rmdir	436
rmfield	66
rmpath	126
roots	338
rosser	211
rot90	198
rotdim	198
round	217
rows	33
rref	235
run_count	307
run_history	23
run_test	314

S

save	156
save_header_format_string	160
save_precision	159
saveaudio	423
saveimage	415

savepath	126	spectral_adf	413
saving_history	23	spectral_xdf	413
schur	238	speed	506
sec	219	spencer	413
secd	221	speye	248
sech	220	spfind	253
SEEK_CUR	178	spfun	248
SEEK_END	178	sph2cart	227
SEEK_SET	178	spinv	267
semilogx	183	spkron	267
semilogxerr	186	splchol	268
semilogy	184	spline	344
semilogyerr	186	split	53
set	190	split_long_rows	37
setaudio	424	splu	268
setdiff	334	spmmax	249
setfield	66	spmin	249
setgrent	451	spones	249
setpwent	450	spparms	269
setxor	334	spprod	250
shg	181	spqr	269
shift	201	sprand	250
shiftdim	200	sprandn	250
SIG	447	sprandsym	250
sighup_dumps_octave_core	161	sprank	270
sign	218	spring	420
sign_test	314	sprintf	165
sigterm_dumps_octave_core	161	spstats	254
silent_functions	119	spsum	250
sin	219	spsumsq	251
sinc	409	spy	256
sind	220	sq	300
sinetone	413	sqrt	218
sinewave	413	sqrtm	240
single	35	squeeze	34
sinh	219	ss	354, 356
size	33	ss2tf	402
size_equal	34	ss2zp	402
sizeof	34	sscanf	171
skewness	305	stairs	185
sleep	432	starp	402
sort	201	stat	436
sortcom	402	statistics	306
sortrows	202	std	304
source	131	stderr	162
spalloc	252	stdin	162
sparse	252	stdout	162
spatan2	247	stem	184
spchol	267	step	388
spchol2inv	267	stft	413
spcholinv	267	str2double	58
spconvert	253	str2func	131
spcumprod	247	str2mat	49
spcumsum	247	str2num	59
spdet	267	strcat	48
spdiag	247	strcmp	50
spdiags	248	strcmppi	51
spearman	307	strfind	52

strftime	429
string_fill_char	49
strjust	59
strmatch	53
strncmp	51
strncmpi	51
strptime	430
strrep	53
strtok	53
strtrunc	49
struct	65
struct_levels_to_print	64
struct2cell	67
strvcat	49
studentize	306
sub2ind	89
subplot	191
subsasgn	98
subsref	88
substr	53
substruct	67
sum	222
summer	420
sumsq	222
suppress_verbose_help_message	19
svd	239
swap	202
swapcols	202
swaprows	202
syl	241
syvester_matrix	211
symamd	264
symbfact	270
symlink	435
symrcm	265
synthesis	414
sys2fir	354
sys2ss	358
sys2tf	360
sys2zp	362
sysadd	370
sysappend	370
syschnames	362
syschtsam	362
sysconnect	371
syscont	372
sysdimensions	362
sysdisc	372
sysdup	372
sysgetsignals	363
sysgettsam	367
sysgettype	365
sysgroup	373
sysmin	375
sysmult	373
sysout	367
sysprune	373
sysreorder	374

sysrepdemo	352
syssscale	374
syssetsignals	365
syssub	374
system	443
sysupdate	366

T

t_test	314
t_test_2	315
t_test_regression	315
table	307
tan	219
tand	221
tanh	220
tar	441
tcdf	324
tempname	441
test	501
text	191
tf	359
tf2ss	403
tf2sys	360
tf2zp	403
tfout	367
tic	431
tilde_expand	440
time	427
tinvc	324
title	190
tmpfile	176
tmpnam	176
toascii	59
toc	431
toeplitz	211
tolower	59
toupper	59
tpdf	324
trace	235
trapz	281
treemap	257
triangle_lw	414
triangle_sw	414
tril	202
triu	202
trnd	328
true	45
type	82
typeinfo	31
tzero	392
tzero2	392

U

u_test	315
ugain	375
uint16	40

uint32.....	40
uint64.....	41
uint8.....	40
umask.....	436
uname.....	451
undo_string_escapes.....	59
unidirnd.....	328
unifcdf.....	324
unifinv.....	324
unifpdf.....	324
unifrnd.....	328
union.....	333
unique.....	333
unix.....	444
unlink.....	435
unmark_command.....	133
unmark_rawcommand.....	134
unmkpp.....	339
unpack.....	442
untar.....	441
unwrap.....	409
unzip.....	442
urlread.....	442
urlwrite.....	442
usage.....	139
usleep.....	432

V

values.....	307
vander.....	211
var.....	304
var_test.....	316
vec.....	203
vech.....	203
vectorize.....	133
ver.....	452
version.....	452
vertcat.....	199
view.....	188
vol.....	332

W

waitpid.....	446
warning.....	142
wavread.....	424
wavwrite.....	424

wblcdf.....	324
wblinv.....	324
wblpdf.....	325
wblrnd.....	328
weekday.....	434
welch_test.....	316
wgt10.....	375
which.....	82
white.....	420
who.....	79
whos.....	79, 80
whos_line_format.....	80
wienrnd.....	329
wilcoxon_test.....	316
wilkinson.....	212
winter.....	420

X

xlabel.....	191
xor.....	195

Y

ylabel.....	191
yulewalker.....	414

Z

z_test.....	316
z_test_2.....	317
zeros.....	205
zgfmul.....	379
zgfslv.....	379
zginit.....	379
zgreduce.....	379
zgrownorm.....	379
zgscal.....	380
zgsgiv.....	380
zgshsr.....	380
zip.....	441
zlabel.....	191
zp.....	361
zp2ss.....	403
zp2sys.....	361
zp2tf.....	404
zpout.....	367

Operator Index

!		\ 92
!..... 94		/
!=..... 93		/..... 92
"		/=..... 97
"..... 32, 47		;
&	 36
&..... 94		<
&&..... 95		<..... 93
,		<=..... 93
,..... 32, 47, 92		<>..... 93
(=
(..... 87		=..... 96
)		==..... 93
)..... 87		>
*		>..... 93
*..... 92		>=..... 93
**..... 92		[
*=..... 97		[..... 36
+]
+..... 91, 92]..... 36
++..... 99		^
+=..... 97		^..... 92
,		\
,..... 36		\..... 92
-		
-..... 91, 92	 94
--..... 99	 95
-=..... 97		~
.		~..... 94
,..... 92		~=..... 93
*..... 92		C
**..... 92		colon..... 39
+...... 91		
./..... 92		
.^..... 92		