Octave

A high-level interactive language for numerical computations
Edition 1.1 for Octave version 1.1.1
January 1995

John W. Eaton

Preface

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

Clearly, Octave is now much more than just another 'courseware' package with limited utiltiy 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 is also currently used in several other undergraduate and graduate courses in our department, 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 [Copying], page 5) 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.

Many people have already contributed to Octave's development. In addition to John W. Eaton, the following people have helped write parts of Octave or helped out in various other ways.

- Karl Berry (karl@cs.umb.edu) wrote the `kpathsea` library that allows Octave to recursively search directory paths for function and script files.
- Georg Beyerle (gbeyerle@awi-potsdam.de) contributed code to save values in MATLAB's '`.mat`'-file format, and has provided many useful bug reports and suggestions.
- John Campbell (jcc@che.utexas.edu) wrote most of the file and C-style input and output functions.
- Brian Fox (bfox@gnu.ai.mit.edu) wrote the `readline` library used for command history editing, and the portion of this manual that documents it.

- A. Scottedward Hodel (scotte@eng.auburn.edu) contributed a number of functions including `expm`, `qzval`, `qzhess`, `syl`, `lyap`, and `balance`.
- Kurt Hornik (Kurt.Hornik@ci.tuwien.ac.at) provided the `corrcoef`, `cov`, `kurtosis`, `pinv`, and `skewness` functions.
- Phil Johnson (johnsonp@nicco.sscnet.ucla.edu) has helped to make Linux releases available.
- Friedrich Leisch (leisch@ci.tuwien.ac.at) provided the `mahalanobis` function.
- Ken Neighbors (wkn@leland.stanford.edu) has provided many useful bug reports and comments on MATLAB compatibility.
- Rick Niles (niles@axp745.gsfc.nasa.gov) rewrote Octave's plotting functions to add line styles and the ability to specify an unlimited number of lines in a single call. He also continues to track down odd incompatibilities and bugs.
- Mark Odegard (meo@sugarland.unocal.com) provided the initial implementation of `fread`, `fwrite`, `feof`, and `ferror`.
- Tony Richardson (tony@guts.biomed.uakron.edu) wrote Octave's image processing functions as well as most of the original polynomial functions.
- R. Bruce Tenison (Bruce.Tenison@eng.auburn.edu) wrote the `hess` and `schur` functions.
- Teresa Twaroch (twaroch@ci.tuwien.ac.at) provided the functions `gls` and `ols`.
- Fook Fah Yap (ffy@eng.cam.ac.uk) provided the `fft` and `ifft` functions and valuable bug reports for early versions.

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

- Digital Equipment Corporation, for an equipment grant as part of their External Research Program.
- Sun Microsystems, Inc., for an Academic Equipment grant.
- 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.
- James B. Rawlings, Associate Professor, Department of Chemical Engineering, The University of Texas at Austin.
- Richard Stallman, for writing GNU.

Portions of this document have been adapted from the `gawk`, `readline`, `gcc`, and C library manuals, published by the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.

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

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

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 Library 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) 19yy  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., 675 Mass Ave, Cambridge, MA 02139, 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 Library General Public License instead of this License.

1 A Brief Introduction to Octave

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

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.

This document corresponds to Octave version 1.1.1.

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 the `CTRL` key 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.

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
```

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
```

To form the matrix product $a^T a$, type the command

```
octave:6> a' * a
```

Solving Linear Equations

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

```
octave:7> a \ b
```

This is conceptually equivalent to $\mathbf{A}^{-1}\mathbf{b}$, 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.

Integrating Differential Equations

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

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

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.

Producing Graphical Output

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

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

If you are using the X Window System, Octave will automatically create a separate window to display the plot. If you are using a terminal that supports some other graphics commands, you will need to tell Octave what kind of terminal you have. Type the command

```
set term
```

to see a list of the supported terminal types. Octave uses `gnuplot` to display graphics, and can display graphics on any terminal that is supported by `gnuplot`.

To capture the output of the plot command in a file rather than sending the output directly to your terminal, you can use a set of commands like this

```
set term postscript
set output "foo.ps"
replot
```

This will work for other types of output devices as well. Octave's '`set`' command is really just piped to the `gnuplot` subprocess, so that once you have a plot on the screen that you like, you should be able to do something like this to create an output file suitable for your graphics printer.

Or, you can eliminate the intermediate file by using commands like this

```
set term postscript
set output "|lpr -Pname_of_your_graphics_printer"
replot
```

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, type `Control-P` (usually written `C-p` for short). `C-p` gets its name from the fact that you type it by holding down the `CTRL` key and then pressing `p`. 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 Appendix Appendix C [Command Line Editing], page 179.

Getting Help

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. You can then get more help on anything that is listed 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 carriage return to advance one line, a space character to advance one page, and '`q`' to exit the pager.

Help via Info

The part of Octave's help facility that allows you to read the complete text of the printed manual from within Octave uses a 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 Appendix Appendix D [Using Info], page 189.

1.3 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 many Unix systems[1] (and someday on GNU).

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

```
#! /usr/local/bin/octave -qf

# a sample Octave program
printf ("Hello, world!\n");
```

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[2] as if you had typed:

```
octave hello
```

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.

1.4 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)
```

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

[2] 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 Chapter 2 [Invoking Octave], page 21.

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

  r = 0.25

  and so on...

endfunction
```

The help command (see Chapter 26 [Help], page 157) 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.

1.5 Errors

There are two classes of errors that Octave produces when it encounters input that it is unable to understand, or when it is unable to perform an action.

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

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

Octave will respond immediately with a message like this:

```
parse error:

  functon 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 the y because the first name by itself was accepted as valid input.

Another class of error message occurs 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 expression 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 numbered beginning with the line containing the '`function`' keyword. 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, say '`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 Invoking Octave

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 up 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 Command Line Options

`--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.

`--help`
`-h`
`-?`
: Print short help message and exit.

`--ignore-init-file`
`--norc`
`-f`
: Don't read any of the system or user initialization files at startup.

`--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 '`INFO_FILE = "filename"`' commands found in the system or user startup files.

`--interactive`
`-i`
: Force interactive behavior.

`--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 '`LOADPATH = "path"`' commands found in the system or user startup files.

`--silent`
`--quiet`
`-q`
: Don't print message at startup.

`--verbose`
`-V` Turn on verbose output.

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

`--echo-commands`
`-x` Echo commands as they are executed.

file Execute commands from *file*.

2.2 Startup Files

When Octave starts, it looks for commands to execute from the following files:

`OCTAVE_HOME/lib/octave/VERSION/startup/octaverc`
: Where `OCTAVE_HOME` is the directory in which all of Octave is installed (the default is '`/usr/local`'), 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. 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 after it reads '`~/.octaverc`', so 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, it will avoid executing commands from '`~/.octaverc`' twice.

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

Startup files may contain any valid Octave commands, including multiple function definitions.

3 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.

3.1 Constant Expressions

The simplest type of expression is the *constant*, which always has the same value. There are two types of constants: numeric constants and string constants.

3.1.1 Numeric Constants

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 all numeric values are represented within Octave in double-precision floating point format (complex constants are stored as pairs of double-precision floating point values). 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.

3.1.2 String Constants

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 3.10 [Arithmetic Ops], page 35) but double-quote marks have no other purpose in Octave, it is best to use double-quote marks to denote strings.

Some characters cannot be included literally in a string constant. You represent them instead with *escape sequences*, which are character sequences beginning with a backslash ('`\`').

One use of an escape sequence is to include a double-quote (single-quote) character in a string constant that has been defined using double-quote (single-quote) marks. Since a plain double-quote would end the string, you must use '`\"`' to represent a single double-quote character as a part of the string. The backslash character itself is another character that cannot be included normally. You must write '`\\`' to put one backslash in the string. Thus, the string whose contents are the two characters '`"\`' must be written `"\"\\"`.

Another use of backslash is to represent unprintable characters such as newline. While there is nothing to stop you from writing most of these characters directly in a string constant, they may look ugly.

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

`\\` Represents a literal backslash, '`\`'.

`\"` Represents a literal double-quote character, '`"`'.

`\'` Represents a literal single-quote character, '`'`'.

`\a` Represents the "alert" character, control-g, ASCII code 7.

`\b` Represents a backspace, control-h, ASCII code 8.

`\f` Represents a formfeed, control-l, ASCII code 12.

`\n` Represents a newline, control-j, ASCII code 10.

`\r` Represents a carriage return, control-m, ASCII code 13.

`\t` Represents a horizontal tab, control-i, ASCII code 9.

`\v` 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`'. The next section explains more about how to create matrices.

3.2 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 =

  1  2
  3  4
```

The commas which separate the elements on a row may be omitted, and the semicolon that marks the beginning of a new row may be replaced by one or more new lines. The expression

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

is equivalent to the one above.

Elements of a matrix may be arbitrary expressions, provided that the dimensions all agree. 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
```

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

```
[ linspace (1, 2) ]
```

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 ]`.

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 to not do so would make it impossible to correctly parse the valid expression

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

For clarity, it is probably best to always use commas and semicolons to separate matrix elements and rows. It is possible to enforce this style by setting the built-in variable `whitespace_in_literal_matrix` to `"ignore"`. See Chapter 6 [Built-in Variables], page 65.

3.2.1 Empty Matrices

A matrix may have one or both dimensions zero, and operations on empty matrices are handled as described by Carl de Boor 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`, and an m by n matrix `M(mxn)`, and an m by n empty matrix `[](mxn)` (with either one or both dimensions equal to zero), the following are true:

```
s * [](mxn) = [](mxn) * s = [](mxn)

    [](mxn) + [](mxn) = [](mxn)

    [](0xm) * M(mxn) = [](0xn)

    M(mxn) * [](nx0) = [](mx0)

    [](mx0) * [](0xn) = 0(mxn)
```

By default, dimensions of the empty matrix are now printed along with the empty matrix symbol, '`[]`'. For example:

```
octave:13> zeros (3, 0)
```

```
ans =

[](3x0)
```

The built-in variable `print_empty_dimensions` controls this behavior (see Section 6.2 [User Preferences], page 66).

Empty matrices may also be used in assignment statements as a convenient way to delete rows or columns of matrices. See Section 3.13 [Assignment Expressions], page 39.

3.3 Ranges

A *range* is a convenient way to write a row vector with evenly spaced elements. A range constant 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 ]`' (the increment has been omitted, so it is taken as 1), and the range

```
1 : 3 : 5
```

defines the set of values '`[ 1 4 ]`'. In this case, the base value is 1, the increment is 3, and the limit is 5.

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 up 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 Chapter 21 [Special Matrices], page 139).

3.4 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__
fucnrdthsucngtagdjb
```

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 3.13 [Assignment Expressions], page 39.

A number of variables have special built-in meanings. For example, `PWD` holds the current working directory, and `pi` names the ratio of the circumference of a circle to its diameter. See Chapter 6 [Built-in Variables], page 65, 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 can be assigned either numeric or string values. Variables may not be used before they have been given a value. Doing so results in an error.

3.5 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 expression. Matrices require two indices unless the value of the built-in variable `do_fortran_indexing` is `"true"`, in which case a matrix may also be indexed by a single expression (see Section 6.2 [User Preferences], page 66).

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.

A special form of indexing may be used to select elements of a matrix or vector. If the indices are vectors made up of only ones and zeros, the result is a new matrix whose elements correspond to the elements of the index vector that are equal to one. For example,

```
a = [1, 2; 3, 4];
a ([1, 0], :)
```

selects the first row of the matrix '`a`'.

This operation can be useful for selecting elements of a matrix based on some condition, since the comparison operators return matrices of ones and zeros.

Unfortunately, this special zero-one form of indexing leads to a conflict with the standard indexing operation. For example, should the following statements

```
a = [1, 2; 3, 4];
a ([1, 1], :)
```

return the original matrix, or the matrix formed by selecting the first row twice? Although this conflict is not likely to arise very often in practice, you may select the behavior you prefer by setting the built-in variable `prefer_zero_one_indexing` (see Section 6.2 [User Preferences], page 66).

Finally, indexing a scalar with a vector of ones can be used to create a vector the same size as the 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 Chapter 21 [Special Matrices], page 139.

3.6 Data Structures

Octave includes a limited amount of 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.

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

  1  2
  3  4
```

```
octave:4> x.c
x.c = string
```

Structures may be copied.

```
octave:1> y = x
y =

<structure: a b c>
```

Note that when the value of a structure is printed, Octave only displays the names of the elements. This prevents long and confusing output from large deeply nested structures, but makes it more difficult to view the values of simple structures, so this behavior may change in a future version of Octave.

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
x.b =

<structure: d>

octave:3> x.b.d
x.b.d = 3
```

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:1> f (rand (3) + rand (3) * I);
ans =

<structure: im re>

octave:3> ans.im
ans.im =

  0.093411  0.229690  0.627585
  0.415128  0.221706  0.850341
  0.894990  0.343265  0.384018
```

```
octave:4> ans.re
ans.re =

  0.56234  0.14797  0.26416
  0.72120  0.62691  0.20910
  0.89211  0.25175  0.21081
```

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

```
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

octave:8> x
x =

<structure: s u v>
```

You can also use the function `is_struct` to determine whether a given value is a data structure. For example

```
is_struct (x)
```

returns 1 if the value of the variable *x* is a data structure.

This feature should be considered experimental, but you should expect it to work. Suggestions for ways to improve it are welcome.

3.7 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 5 [Functions and Scripts], page 53, 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 3.5 [Index Expressions], page 28, and Section 3.13 [Assignment Ops], page 39.

3.7.1 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"`.

3.7.2 Recursion

Recursive function calls are allowed. A *recursive function* is one which calls itself, either directly or indirectly. For example, here is an inefficient[1] 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.

There is currently no limit on the recursion depth, so infinite recursion is possible. If this happens, Octave will consume more and more memory attempting to store intermediate values for each function call context until there are no more resources available. This is obviously undesirable, and will probably be fixed in some future version of Octave by allowing users to specify a maximum allowable recursion depth.

[1] 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.

3.8 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 b = 2
global c = 3, d, e = 5
```

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:

```
octave:1> function f (x)
> x = 3
> endfunction
octave:2> global x = 0
octave:3> x              # This is the value of the global variable.
x = 0
octave:4> f (x)
x = 3                    # The value of the local variable x is 3.
octave:5> x              # But it was a *copy* so the global variable
x = 0                    # remains unchanged.
```

3.9 Keywords

The following identifiers are keywords, and may not be used as variable or function names:

```
break      endfor       function   return
continue   endfunction  global     while
else       endif        gplot
elseif     endwhile     gsplot
end        for          if
```

The following command-like functions are also keywords, and may not be used as variable or function names:

`casesen`	`document`	`history`	`set`
`cd`	`edit_history`	`load`	`show`
`clear`	`help`	`ls`	`who`
`dir`	`format`	`run_history`	`save`

3.10 Arithmetic Operators

The following arithmetic operators are available, and work on scalars and matrices.

x `+` *y*
: 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.

x `.+` *y*
: Element by element addition. This operator is equivalent to `+`.

x `-` *y*
: Subtraction. If both operands are matrices, the number of rows and columns of both must agree.

x `.-` *y*
: Element by element subtraction. This operator is equivalent to `-`.

x `*` *y*
: Matrix multiplication. The number of columns of '`x`' must agree with the number of rows of '`y`'.

x `.*` *y*
: Element by element multiplication. If both operands are matrices, the number of rows and columns must both agree.

x `/` *y*
: Right division. This is conceptually equivalent to the expression

```
(inverse (y') * x')'
```

but it is computed without forming the inverse of '`y'`'.

If the system is not square, or if the coefficient matrix is singular, a minimum norm solution is computed.

x `./` *y*
: Element by element right division.

x `\` *y*
: Left division. This is conceptually equivalent to the expression

```
inverse (x) * y
```

but it is computed without forming the inverse of '`x`'.

If the system is not square, or if the coefficient matrix is singular, a minimum norm solution is computed.

x `.\` *y*
: Element by element left division. Each element of '`y`' is divided by each corresponding element of '`x`'.

x `^` *y*

x `**` *y*
: Power operator. If *x* and *y* are both scalars, this operator returns *x* raised to the power *y*. If *x* is a scalar and *y* is a square matrix, the result is computed using an eigenvalue expansion. If *x* is a square matrix. the result is computed by repeated multiplication if *y* is an integer, and by an eigenvalue expansion if *y* is not an integer. An error results if both *x* and *y* are matrices.

  The implementation of this operator needs to be improved.

x `.^` *y*

x `.**` *y*
: Element by element power operator. If both operands are matrices, the number of rows and columns must both agree.

`-`*x*
: Negation.

`+`*x*
: Unary plus. This operator has no effect on the operand.

x`'`
: 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

```
conj (x.')
```

x`.'`
: 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.

3.11 Comparison Operators

Comparison operators compare numeric values for relationships such as equality. They are written using *relational operators*, which are a superset of those in C.

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, evaluating the expression

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

returns the result

```
ans =

  1  0
  0  1
```

x `<` *y* True if *x* is less than *y*.

x `<=` *y* True if *x* is less than or equal to *y*.

x `==` *y* True if *x* is equal to *y*.

x `>=` *y* True if *x* is greater than or equal to *y*.

x `>` *y* True if *x* is greater than *y*.

x `!=` *y*
x `~=` *y*
x `<>` *y* True if *x* is not equal to *y*.

For matrix and vector arguments, the above table should be read as "an element of the result matrix (vector) is true if the corresponding elements of the argument matrices (vectors) satisfy the specified condition"

String comparisons should be performed with the `strcmp` function, not with the comparison operators listed above. See Section 3.7 [Calling Functions], page 31.

3.12 Boolean Expressions

3.12.1 Element-by-element Boolean Operators

An element-by-element *boolean expression* is a combination of comparison expressions or matching 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, before being used in the condition of an `if` or `while` statement, an implicit conversion from a matrix value to a scalar value occurs using the equivalent of `all (all (`*x*`))`. That is, a 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.

3.12.2 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 (all (`*boolean1*`))`. If it is false, the result of the expression is 0. If it is true, the expression *boolean2* is evaluated and converted to a scalar using the equivalent of the operation `all (all (`*boolean1*`))`. If it is true, the result of the expression is 1. Otherwise, the result of the expression is 0.

boolean1 `||` *boolean2*

The expression *boolean1* is evaluated and converted to a scalar using the equivalent of the operation `all (all (`*boolean1*`))`. If it is true, the result of the expression is 1. If it is false, the expression *boolean2* is evaluated and converted to a scalar using the equivalent of the operation `all (all (`*boolean1*`))`. If it is true, the result of the expression is 1. Otherwise, the result of the expression is 0.

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.

```
function f (a, b, c)
  if (nargin > 2)
    if (isstr (c))
      ...
```

3.13 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.

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.)

The '`=`' sign is called an *assignment operator*. It is the simplest assignment operator because the value of the right-hand operand is stored unchanged.

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 Section 3.4 [Variables], page 27). It can also be an element of a matrix (see Section 3.5 [Index Expressions], page 28) or a list of return values (see Section 3.7 [Calling Functions], page 31). 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.

Assigning an empty matrix '`[]`' works in most cases to allow you to delete rows or columns of matrices and vectors. See Section 3.2.1 [Empty Matrices], page 26. 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, c, d] = [u, s, v] = svd (a)
```

is equivalent to the expression above, except that the value of the variable `d` is left unchanged, and the expression

```
[a, b] = [u, s, v] = svd (a)
```

is equivalent to

```
[u, s, v] = svd (a)
a = u
b = s
```

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.

3.14 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.

`++`*x*
: This expression increments the variable *x*. The value of the expression is the *new* value of *x*. It is equivalent to the expression *x* `=` *x* `+ 1`.

`--`*x*
: This expression decrements the variable *x*. The value of the expression is the *new* value of *x*. It is equivalent to the expression *x* `=` *x* `- 1`.

x`++`
: This expression causes the variable *x* to be incremented. The value of the expression is the *old* value of *x*.

x`--`
: This expression causes the variable *x* to be decremented. The value of the expression is the *old* value of *x*.

It is not currently possible to increment index expressions. For example, you might expect that the expression *v*`(4)++` would increment the fourth element of the vector *v*, but instead it results in a parse error. This problem may be fixed in a future release of Octave.

3.15 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`
: '`^`', '`**`', '`.^`', '`.**`'.

4 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.

4.1 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 *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 satement.

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 `end` statements (`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
```

using the indentation to show how Octave groups the statements. See Chapter 5 [Functions and Scripts], page 53.

4.2 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 an `if` statement is a vector or a matrix, it is considered true only if *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 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.

4.3 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
```

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 either 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.

In the `for` statement, *body* stands for any statement or list of statements.

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.

4.4 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
```

4.5 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
```

4.6 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 built-in variable `do_fortran_indexing` even if an error occurs while performing the indexing operation.

```
save_do_fortran_indexing = do_fortran_indexing;
unwind_protect
```

```
  do_fortran_indexing = "true";
  elt = a (idx)
unwind_protect_cleanup
  do_fortran_indexing = save_do_fortran_indexing;
end_unwind_protect
```

Without unwind_protect, the value of *do_fortran_indexing* 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.

4.7 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 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 \   % comment two
    - 42                       % last comment
```

is equivalent to the one shown above.

In some cases, Octave will allow you to continue lines without having to specify continuation characters. For example, it is possible to write statements like

```
if (big_long_variable_name == other_long_variable_name
    || not_so_short_variable_name > 4
    && y > x)
  some (code, here);
endif
```

without having to clutter up the if statement with continuation characters.

5 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 ones.

5.1 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 underscores, 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 20 [Input and Output], page 123) simply tells Octave to print the string `"\a"`. The special character '`\a`' stands for the alert character (ASCII 7). See Section 3.1.2 [String Constants], page 24.

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 `beep` 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.

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 (is_vector (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 (is_vector (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)
    error ("usage: avg (vector)");
  endif
  if (is_vector (v))
    retval = sum (v) / length (v);
  else
    error ("avg: expecting vector argument");
  endif
endfunction
```

Although Octave does not consider it an error if you call a function with more arguments than were expected, doing so is probably an error, so we check for that possibility too, and issue the error message if either too few or too many arguments have been provided.

The body of a user-defined function can contain a `return` statement. This statement returns control to the rest of the Octave program. A `return` statement is assumed at the end of every function definition.

5.2 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
```

5.3 Variable-length Argument Lists

Octave has a real mechanism for handling functions that take an unspecified number of arguments, so it is not necessary to place an upper bound on the number of optional arguments that a function can accept.

Here is an example of a function that uses the new syntax to print a header followed by an unspecified number of values:

```
function foo (heading, ...)
  disp (heading);
  va_start ();
  while (--nargin)
    disp (va_arg ());
  endwhile
endfunction
```

The ellipsis that marks the variable argument list may only appear once and must be the last element in the list of arguments.

Calling `va_start()` positions an internal pointer to the first unnamed argument and allows you to cycle through the arguments more than once. It is not necessary to call `va_start()` if you do not plan to cycle through the arguments more than once.

The function `va_arg()` returns the value of the next available argument and moves the internal pointer to the next argument. It is an error to call `va_arg()` when there are no more arguments available.

Sometimes it is useful to be able to pass all unnamed arguments to another function. The keyword *all_va_args* makes this very easy to do. For example, given the functions

```
function f (...)
  while (nargin--)
    disp (va_arg ())
  endwhile
endfunction
function g (...)
  f ("begin", all_va_args, "end")
endfunction
```

the statement

```
g (1, 2, 3)
```

prints

```
begin
1
2
3
end
```

The keyword `all_va_args` always stands for the entire list of optional argument, so it is possible to use it more than once within the same function without having to call `var_start ()`. It can only be used within functions that take a variable number of arguments. It is an error to use it in other contexts.

5.4 Variable-length Return Lists

Octave also has a real mechanism for handling functions that return an unspecified number of values, so it is no longer necessary to place an upper bound on the number of outputs that a function can produce.

Here is an example of a function that uses the new syntax to produce N values:

```
function [...] = foo (n, x)
  for i = 1:n
    vr_val (i * x);
  endfor
endfunction
```

Each time `vr_val()` is called, it places the value of its argument at the end of the list of values to return from the function. Once `vr_val()` has been called, there is no way to go back to the beginning of the list and rewrite any of the return values.

As with variable argument lists, the ellipsis that marks the variable return list may only appear once and must be the last element in the list of returned values.

5.5 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.

5.6 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 the list of directories specified by the built-in variable `LOADPATH` for files ending in '`.m`' that have the same base name as the undefined identifier.[1] See Section 6.2 [User Preferences], page 66 for a description of `LOADPATH`. 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 and executed. See Section 5.7 [Script Files], page 59, 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. After that, it checks the time stamp on the file every time it needs the function. If the time stamp indicates that the file has changed since the last time it was read, Octave reads it again.

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. Checking the time stamp every time a function is used is rather inefficient, but it has to be done to ensure that the correct function definition is used.

Octave assumes that function files in the '`/usr/local/lib/octave/1.1.1`' directory tree 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 setting the variable `ignore_function_time_stamp` to `"all"`, so that Octave will ignore the time stamps for all function files. Setting it to `"system"` gives the default behavior. If you set it to anything else, Octave will check the time stamps on all function files.

5.7 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.

---

[1] The '`.m`' suffix was chosen for compatibility with MATLAB.

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 `LOADPATH`, 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 lookup 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 the 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 `LOADPATH`.

```
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 '`eval ()`' statement 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).

5.8 Dynamically Linked Functions

On some systems, Octave can dynamically load and execute functions written in C++ or other compiled languages. This currently only works on systems that have a working version of the GNU dynamic linker, `dld`. Unfortunately, `dld` does not work on very many systems, but someone is working on making `dld` use the GNU Binary File Descriptor library, `BFD`, so that may soon change. In any case, it should not be too hard to make Octave's dynamic linking features work on other systems using system-specific dynamic linking facilities.

Here is an example of how to write a C++ function that Octave can load.

```
#include <iostream.h>

#include "defun-dld.h"
#include "tree-const.h"

DEFUN_DLD ("hello", Fhello, Shello, -1, -1,
  "hello (...)\n\
\n\
Print greeting followed by the values of all the arguments passed.\n\
Returns all the arguments passed.")
{
  Octave_object retval;
  cerr << "Hello, world!\n";
  int nargin = args.length ();
  for (int i = 1; i < nargin; i++)
    retval (nargin-i-1) = args(i).eval (1);
  return retval;
}
```

Octave's dynamic linking features currently have the following limitations.

- Dynamic linking only works on systems that support the GNU dynamic linker, `dld`.
- Clearing dynamically linked functions doesn't work.

- Configuring Octave with `--enable-lite-kernel` seems to mostly work to make nonessential built-in functions dynamically loaded, but there also seem to be some problems. For example, fsolve seems to always return `info == 3`. This is difficult to debug since `gdb` won't seem to allow breakpoints to be set inside dynamically loaded functions.
- Octave uses a lot of memory if the dynamically linked functions are compiled to include debugging symbols. This appears to be a limitation with `dld`, and can be avoided by not using `-g` to compile functions that will be linked dynamically.

If you would like to volunteer to help improve Octave's ability to dynamically link externally compiled functions, please contact `bug-octave@che.utexas.edu`.

5.9 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.

'`control`' Functions for design and simulation of automatic control systems.

'`elfun`' Elementary functions.

'`general`' Miscellaneous matrix manipulations, like `flipud`, `rot90`, and `triu`, as well as other basic functions, like `is_matrix`, `nargchk`, etc.

'`image`' Image processing tools. These functions require the X Window System.

'`linear-algebra`'
Functions for linear algebra.

'`miscellaneous`'
Functions that don't really belong anywhere else.

'`plot`' A set of functions that implement the MATLAB-like plotting functions.

'`polynomial`'
Functions for manipulating polynomials.

'`set`' Functions for creating and manipulating sets of unique values.

'`signal`' Functions for signal processing applications.

'`specfun`' Special functions.

'`special-matrix`'
Functions that create special matrix forms.

'`startup`' Octave's system-wide startup file.

'`statistics`'
: Statistical functions.

'`strings`'
: Miscellaneous string-handling functions.

See Section 6.2 [User Preferences], page 66 for an explanation of the built-in variable `LOADPATH`, and Section 5.6 [Function Files], page 59 for a description of the way Octave resolves undefined variable and function names.

6 Built-in Variables

Most Octave variables are available for you to use for your own purposes; they never change except when your program assigns values to them, and never affect anything except when your program examines them.

A few variables have special built-in meanings. Some of them, like `pi` and `eps` provide useful predefined constant values. Others, like `do_fortran_indexing` and `page_screen_output` are examined automatically by Octave, so that you can to tell Octave how to do certain things. There are also two special variables, `ans` and `PWD`, that are set automatically by Octave and carry information from the internal workings of Octave to your program.

This chapter documents all the built-in variables of Octave. Most of them are also documented in the chapters that describe functions that use them, or are affected by their values.

6.1 Predefined Constants

`I`, `i`, `J`, `j`
: A pure imaginary number, defined as $\sqrt{-1}$. The `I` and `J` forms are true constants, and cannot be modified. The `i` and `j` forms are like ordinary variables, and may be used for other purposes. However, unlike other variables, they once again assume their special predefined values if they are cleared See Section 27.2 [Miscellaneous Utilities], page 160.

`Inf`, `inf`
: Infinity. This is the result of an operation like 1/0, or an operation that results in a floating point overflow.

`NaN`, `nan`
: Not a number. This is the result of an operation like '`0/0`', or '`Inf - Inf`', or any operation with a NaN.

`SEEK_SET`
`SEEK_CUR`
`SEEK_END`
: These variables may be used as the optional third argument for the function `fseek`.

`eps`
: The machine precision. More precisely, `eps` is the smallest value such that '`1+eps`' is not equal to 1. This number is system-dependent. On machines that support 64 bit IEEE floating point arithmetic, `eps` is approximately 2.2204×10^{-16}.

`pi`
: The ratio of the circumference of a circle to its diameter. Internally, `pi` is computed as '`4.0 * atan (1.0)`'.

`realmax`
: 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}.

`realmin`
: The smallest 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}.

`stdin`
`stdout`
`stderr` These variables are the file numbers corresponding to the standard input, standard output, and standard error streams. These streams are preconnected and available when Octave starts.

6.2 User Preferences

This section describes the variables that you can use to customize Octave's behavior.

Normally, preferences are set in the file '`~/.octaverc`', so that you can customize your environment in the same way each time you use Octave without having to remember and retype all the necessary commands. See Section 2.2 [Startup Files], page 22 for more information.

`EDITOR` A string naming 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 `"vi"`.

`IMAGEPATH`
A colon separated list of directories in which to search for image files. See Chapter 19 [Image Processing], page 121 for a description of Octave's image processing capabilities.

`INFO_FILE`
A string naming the location of the Octave info file.

The default value is `"/usr/local/info/octave.info"`.

`LOADPATH` A colon separated list of directories in which to search for function files. See Chapter 5 [Functions and Scripts], page 53. The value of `LOADPATH` overrides the environment variable `OCTAVE_PATH`. See Appendix A [Installation], page 167.

`LOADPATH` is now handled in the same way as TeX handles `TEXINPUTS`. If the path starts with '`:`', the standard path is prepended to the value of `LOADPATH`. If it ends with '`:`' the standard path is appended to the value of `LOADPATH`.

In addition, if any path element ends in '`//`', that directory and all subdirectories it contains are searched recursively for function files. This can result in a slight delay as Octave caches the lists of files found in the `LOADPATH` the first time Octave searches for a function. After that, searching is usually much faster because Octave normally only needs to search its internal cache for files.

To improve performance of recursive directory searching, it is best for each directory that is to be searched recursively to contain *either* additional subdirectories *or* function files, but not a mixture of both.

See Section 5.9 [Organization of Functions], page 62 for a description of the function file directories that are distributed with Octave.

`OCTAVE_VERSION`
: The version number of Octave, as a string.

`PAGER`
: The default value is `"less"`, or, if `less` is not available on your system, `"more"`. See Appendix A [Installation], page 167, and Chapter 20 [Input and Output], page 123.

`PS1`
: The primary prompt string. When executing interactively, Octave displays the primary prompt `PS1` when it is ready to read a command. Octave allows the prompt to be customized by inserting a number of backslash-escaped special characters that are decoded as follows:

  ‘`\t`’ The time.

  ‘`\d`’ The date.

  ‘`\n`’ Begins a new line by printing the equivalent of a carriage return followed by a line feed.

  ‘`\s`’ The name of the program (usually just `octave`).

  ‘`\w`’ The current working directory.

  ‘`\W`’ The basename of the current working directory.

  ‘`\u`’ The username of the current user.

  ‘`\h`’ The hostname.

  ‘`\#`’ The command number of this command, counting from when Octave starts.

  ‘`\!`’ The history number of this command. This differs from ‘`\#`’ by the number of commands in the history list when Octave starts.

  ‘`\$`’ If the effective UID is 0, a #, otherwise a $.

  ‘`\nnn`’ The character whose character code in octal is ‘`nnn`’.

  ‘`\\`’ A backslash.

  The default value of `PS1` 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`’. Note that two backslashes are required to enter a backslash into a string. See Section 3.1.2 [String Constants], page 24.

`PS2`
: The secondary prompt string, which is printed when Octave is expecting additional input to complete a command. For example, when defining a function over several lines, Octave will print the value of `PS1` at the beginning of each line after the first. Octave allows `PS2` to be customized in the same way as `PS1`. The default value of `PS2` is `"> "`.

PS4
: If Octave is invoked with the `--echo-input` option, the value of `PS4` is printed before each line of input that is echoed. Octave allows `PS4` to be customized in the same way as `PS1`. The default value of `PS4` is `"+ "`. See Chapter 2 [Invoking Octave], page 21, for a description of `--echo-input`.

automatic_replot
: If this variable is `"true"`, Octave will automatically send a `replot` command to `gnuplot` each time the plot changes. Since this is fairly inefficient, the default value is `"false"`.

default_return_value
: The value given to otherwise unitialized return values if `define_all_return_values` is `"true"`. The default value is `[]`.

default_save_format
: Specify the default format used by the `save` command. Options are `"ascii"`, `"binary"`, `"mat-binary"`, or `"float-binary"`. The default value is `ascii`.

define_all_return_values
: If the value of `define_all_return_values` is `"true"`, Octave will substitute the value specified by `default_return_value` for any return values that remain undefined when a function returns. The default value is `"false"`.

do_fortran_indexing
: If the value of `do_fortran_indexing` is `"true"`, Octave allows you to select elements of a two-dimensional matrix using a single index by treating the matrix as a single vector created from the columns of the matrix. The default value is `"false"`.

empty_list_elements_ok
: This variable controls whether Octave ignores empty matrices in a matrix list.

  For example, if the value of `empty_list_elements_ok` is `"true"`, Octave will ignore the empty matrices in the expression

```
a = [1, [], 3, [], 5]
```

  and the variable '`a`' will be assigned the value '`[ 1 3 5 ]`'.

  The default value is `"warn"`.

gnuplot_binary
: The name of the program invoked by the plot command. The default value is `"gnuplot"`. See Appendix A [Installation], page 167.

ignore_function_time_stamp
: This variable variable can be used to prevent Octave from making the system call `stat()` each time it looks up functions defined in function files. If `ignore_function_time_stamp` to `"system"`, Octave will not automatically recompile function files in subdirectories of `/usr/local/lib/1.1.1` if they have changed since they were last

compiled, but will recompile other function files in the LOADPATH if they change. If set to "all", Octave will not recompile any function files unless their definitions are removed with clear. For any other value of ignore_function_time_stamp, Octave will always check to see if functions defined in function files need to recompiled. The default value of ignore_function_time_stamp is "system".

implicit_str_to_num_ok
: If the value of implicit_str_to_num_ok is "true", implicit conversions of strings to their numeric ASCII equivalents are allowed. Otherwise, an error message is printed and control is returned to the top level. The default value is "false".

ok_to_lose_imaginary_part
: If the value of ok_to_lose_imaginary_part is "true", implicit conversions of complex numbers to real numbers are allowed (for example, by fsolve). If the value is "warn", the conversion is allowed, but a warning is printed. Otherwise, an error message is printed and control is returned to the top level. The default value is "warn".

output_max_field_width
: This variable specifies the maximum width of a numeric output field. The default value is 10.

  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 20.1 [Basic Input and Output], page 123.

output_precision
: This variable specifies the minimum number of significant figures to display for numeric output. The default value is 5.

  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 20.1 [Basic Input and Output], page 123.

page_screen_output
: If the value of page_screen_output is "true", all output intended for the screen 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 A [Installation], page 167) are also capable of moving backward on the output. The default value is "true". See Chapter 20 [Input and Output], page 123.

  You can choose the program to use as the pager by setting the variable PAGER.

prefer_column_vectors
: If prefer_column_vectors is "true", operations like

```
for i = 1:10
  a (i) = i;
endfor
```

(for 'a' previously undefined) produce column vectors. Otherwise, row vectors are preferred. The default value is "false".

If a variable is already defined to be a vector (a matrix with a single row or column), the original orientation is respected, regardless of the value of prefer_column_vectors.

prefer_zero_one_indexing

If the value of prefer_zero_one_indexing is "true", Octave will perform zero-one style indexing when there is a conflict with the normal indexing rules. See Section 3.5 [Index Expressions], page 28. For example, given a matrix

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

with prefer_zero_one_indexing is set to "true", the expression

```
a ([1, 1, 1, 1])
```

results in the matrix '[1 2 3 4]'. If the value of prefer_zero_one_indexing set to "false", the result would be the matrix '[1 1 1 1]'.

In the first case, Octave is selecting each element corresponding to a '1' in the index vector. In the second, Octave is selecting the first element multiple times.

The default value for prefer_zero_one_indexing is "false".

print_answer_id_name

If the value of print_answer_id_name is "true", variable names are printed along with the result. Otherwise, only the result values are printed. The default value is "true".

print_empty_dimensions

If the value of print_empty_dimensions is "true", the dimensions of empty matrices are printed along with the empty matrix symbol, '[]'. For example, the expression

```
zeros (3, 0)
```

will print

```
ans =

[](3x0)
```

propagate_empty_matrices

If the value of propagate_empty_matrices is "true", functions like inverse and svd will return an empty matrix if they are given one as an argument. The default value is "true". See Section 3.2.1 [Empty Matrices], page 26.

resize_on_range_error

If the value of resize_on_range_error is "true", expressions like

```
for i = 1:10
  a (i) = i;
endfor
```

(for 'a' previously undefined) result in the variable 'a' being resized to be just large enough to hold the new value. Otherwise uninitialized elements are set to zero. If the value of `resize_on_range_error` is `"false"`, an error message is printed and control is returned to the top level. The default value is `"true"`.

`return_last_computed_value`

If the value of `return_last_computed_value` is true, and a function is defined without explicitly specifying a return value, the function will return the value of the last expression. Otherwise, no value will be returned. The default value is `"false"`.

For example, the function

```
function f ()
  2 + 2;
endfunction
```

will either return nothing, if `return_last_computed_value` is `"false"`, or 4, if it is `"true"`.

`save_precision`

This variable specifies the number of digits to keep when saving data with the `save` command. The default value is 17.

`silent_functions`

If the value of `silent_functions` is `"true"`, internal output from a function is suppressed. Otherwise, the results of expressions within a function body that are not terminated with a semicolon will have their values printed. The default value is `"false"`.

For example, if the function

```
function f ()
  2 + 2
endfunction
```

is executed, Octave will either print '`ans = 4`' or nothing depending on the value of `silent_functions`.

`split_long_rows`

For large matrices, Octave may not be able to display all the columns of a given row on one line of your screen. This can result in missing information or output that is nearly impossible to decipher, depending on whether your terminal truncates or wraps long lines.

If the value of `split_long_rows` is `"true"`, Octave will display the matrix in a series of smaller pieces, each of which can fit within the limits of your terminal width. Each set of rows is labeled so that you can easily see which columns are currently being displayed. For example:

```
octave:13> rand (2, 9)
ans =

 Columns 1 through 7:

   0.92205  0.72628  0.99841  0.62590  0.82422  0.77486  0.30258
```

```
 0.15999  0.79484  0.75443  0.86995  0.91430  0.23980  0.64591

Columns 8 and 9:

 0.08894  0.13266
 0.28008  0.65575
```

The default value of `split_long_rows` is `"true"`.

`suppress_verbose_help_message`
: If the value of `suppress_verbose_help_message` is `"true"`, Octave will not add additional help information to the end of the output from the `help` command and usage messages for built-in commands.

`treat_neg_dim_as_zero`
: If the value of `treat_neg_dim_as_zero` is `"true"`, expressions like

```
eye (-1)
```

produce an empty matrix (i.e., row and column dimensions are zero). Otherwise, an error message is printed and control is returned to the top level. The default value is `"false"`.

`warn_assign_as_truth_value`
: If the value of `warn_assign_as_truth_value` is `"true"`, 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 setting `warn_assign_as_truth_value` to `"false"`, 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.

The default value of `warn_assign_as_truth_value` is `"true"`.

`warn_comma_in_global_decl`

If the value of `warn_comma_in_global_decl` is `"true"`, a warning is issued for statements like

```
global a = 1, b
```

which makes the variables '`a`' and '`b`' global and assigns the value 1 to the variable '`a`', because in this context, the comma is not interpreted as a statement separator.

The default value of `warn_comma_in_global_decl` is `"true"`.

`warn_divide_by_zero`

If the value of `warn_divide_by_zero` is `"true"`, a warning is issued when Octave encounters a division by zero. If the value is `"false"`, the warning is omitted. The default value is `"true"`.

`warn_function_name_clash`

If the value of `warn_function_name_clash` is `"true"`, 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 value is `"false"`, the warning is omitted. The default value is `"true"`.

`whitespace_in_literal_matrix`

This variable allows some control over how Octave decides to convert spaces to commas and semicolons in matrix expressions like '`[m (1)]`' or

```
[ 1, 2,
  3, 4 ]
```

If the value of `whitespace_in_literal_matrix` is `"ignore"`, Octave will never insert a comma or a semicolon in a literal matrix list. For example, the expression '`[1 2]`' will result in an error instead of being treated the same as '`[1, 2]`', and the expression

```
[ 1, 2,
  3, 4 ]
```

will result in the vector [1 2 3 4] instead of a matrix.

If the value of `whitespace_in_literal_matrix` is `"traditional"`, Octave will convert spaces to a comma between identifiers and '`(`'. For example, given the matrix

```
m = [3 2]
```

the expression

```
[m (1)]
```

will be parsed as

```
[m, (1)]
```

and will result in

```
[3 2 1]
```

and the expression

```
[ 1, 2,
  3, 4 ]
```

will result in a matrix because the newline character is converted to a semicolon (row separator) even though there is a comma at the end of the first line (trailing commas or semicolons are ignored). This is apparently how MATLAB behaves.

Any other value for `whitespace_in_literal_matrix` results in behavior that is the same as traditional, except that Octave does not convert spaces to a comma between identifiers and '`(`'. For example, the expression

```
[m (1)]
```

will produce '`3`'. This is the way Octave has always behaved.

6.3 Other Built-in Variables

In addition to these variables, there are two other special built-in variables whose values are automatically updated.

`ans`
: This variable holds 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 of `ans` is '`25`'.

`PWD`
: The current working directory. The value of `PWD` is updated each time the current working directory is changed with the '`cd`' command. See Chapter 24 [System Utilities], page 151.

6.4 Summary of Preference Variables

Here is a summary of all of Octave's preference variables and their default values. In the following table `OCTAVE_HOME` stands for the root directory where Octave is installed, and `VERSION` stands for the Octave version number.

```
EDITOR                  EDITOR environment variable, or "vi"
INFO_FILE               "OCTAVE_HOME/info/octave.info"
LOADPATH                OCTAVE_PATH environment variable, or
                        ".:OCTAVE_HOME/lib/VERSION"
PAGER                   "less", or "more"
PS1                     "\s:\#> "
PS2                     "> "
PS4                     "+ "
automatic_replot        "false"
default_return_value    []
do_fortran_indexing     "false"
```

```
define_all_return_values      "false"
empty_list_elements_ok        "warn"
gnuplot_binary                "gnuplot"
ignore_function_time_stamp    "system"
implicit_str_to_num_ok        "false"
ok_to_lose_imaginary_part     "warn"
output_max_field_width        10
output_precision              5
page_screen_output            "true"
prefer_column_vectors         "false"
prefer_zero_one_indexing      "false"
print_answer_id_name          "true"
print_empty_dimensions        "true"
resize_on_range_error         "true"
return_last_computed_value    "false"
save_precision                17
silent_functions              "false"
split_long_rows               "true"
suppress_verbose_help_message "true"
treat_neg_dim_as_zero         "false"
warn_assign_as_truth_value    "true"
warn_comma_in_global_decl     "true"
warn_divide_by_zero           "true"
warn_function_name_clash      "true"
whitespace_in_literal_matrix  ""
```

7 Arithmetic

Unless otherwise noted, all of the functions described in this chapter will work for real and complex scalar or matrix arguments.

7.1 Utility Functions

The following functions are available for working with complex numbers. Each expects a single argument, and given a matrix, they work on an element by element basis.

`ceil (x)` Return the smallest integer not less than *x*. If *x* is complex, return `ceil (real (x)) + ceil (imag (x)) * I`.

`floor (x)` Return the largest integer not greater than *x*. If *x* is complex, return `floor (real (x)) + floor (imag (x)) * I`.

`fix (x)` Truncate *x* toward zero. If *x* is complex, return `fix (real (x)) + fix (imag (x)) * I`.

`round (x)` Return the integer nearest to *x*. If *x* is complex, return `round (real (x)) + round (imag (x)) * I`.

`sign (x)` Compute the *signum* function, which is defined as

$$\operatorname{sign}(x) = \begin{cases} 1, & x > 0; \\ 0, & x = 0; \\ -1, & x < 0. \end{cases}$$

For complex arguments, `sign` returns `x ./ abs (x)`.

`exp (x)` Compute the exponential of *x*. To compute the matrix exponential, see Chapter 8 [Linear Algebra], page 81.

`gcd (x, ...)`

Compute the greatest common divisor of the elements of *x*, or the list of all the arguments. For example,

```
gcd (a1, ..., ak)
```

is the same as

```
gcd ([a1, ..., ak])
```

An optional second return value, *v* contains an integer vector such that

```
g = v(1) * a(k) + ... + v(k) * a(k)
```

`lcm (x, ...)`

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]).
```

`log (x)` Compute the natural logarithm of *x*. To compute the matrix logarithm, see Chapter 8 [Linear Algebra], page 81.

`log2 (x)` Compute the base-2 logarithm of *x*.

`log10 (x)` Compute the base-10 logarithm of *x*.

`sqrt (x)` Compute the square root of *x*. To compute the matrix square root, see Chapter 8 [Linear Algebra], page 81.

`max (x)` For a vector argument, return the maximum value. For a matrix argument, return the maximum value from each column, as a row vector. Thus,

```
max (max (x))
```

returns the largest element of *x*.

For complex arguments, the magnitude of the elements are used for comparison.

`min (x)` Like `max`, but return the minimum value.

`rem (x, y)`

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.

7.2 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.

`abs (x)` Compute the magnitude of *x*.

`angle (x)`
`arg (x)` Compute the argument of *x*.

`conj (x)` Return the complex conjugate of *x*.

`imag (x)` Return the imaginary part of *x*.

`real (x)` Return the real part of *x*.

7.3 Trigonometry

Octave provides the following trigonometric functions:

```
sin    asin    sinh    asinh
cos    acos    cosh    acosh
tan    atan    tanh    atanh

sec    asec    sech    asech
csc    acsc    csch    acsch
cot    acot    coth    acoth
```

Each of these functions expect a single argument. For matrix arguments, they work on an element by element basis. For example, the expression

```
sin ([1, 2; 3, 4])
```

produces

```
ans =

   0.84147   0.90930
   0.14112  -0.75680
```

`atan2` (*y*, *x*)

7.4 Sums and Products

`sum` (*x*) For a vector argument, return the sum of all the elements. For a matrix argument, return the sum of the elements in each column, as a row vector. The sum of an empty matrix is 0 if it has no columns, or a vector of zeros if it has no rows (see Section 3.2.1 [Empty Matrices], page 26).

`prod` (*x*) For a vector argument, return the product of all the elements. For a matrix argument, return the product of the elements in each column, as a row vector. The product of an empty matrix is 1 if it has no columns, or a vector of ones if it has no rows (see Section 3.2.1 [Empty Matrices], page 26).

`cumsum` (*x*)

Return the cumulative sum of each column of *x*. For example,

```
cumsum ([1, 2; 3, 4])
```

produces

```
ans =

  1  2
  4  6
```

`cumprod` (*x*)

Return the cumulative product of each column of *x*. For example,

```
cumprod ([1, 2; 3, 4])
```

produces

```
ans =

  1  2
  3  8
```

`sumsq (x)` For a vector argument, return the sum of the squares of all the elements. For a matrix argument, return the sum of the squares of the elements in each column, as a row vector.

7.5 Special Functions

`beta` Returns the beta function,

$$\beta(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

`betai (a, b, x)`

Returns the incomplete beta function,

$$\beta(a, b, x) = \beta(a, b)^{-1} \int_0^x t^{(a-z)}(1-t)^{(b-1)} dt.$$

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.

`erf` Computes the error function,

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

`erfc (z)` Computes the complementary error function, `1 - erf (z)`.

`erfinv` Computes the inverse of the error function.

`gamma (z)` Computes the gamma function,

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt.$$

`gammai (a, x)`

Computes the incomplete gamma function,

$$\Gamma(a, x) = \frac{\int_0^x e^{-t} t^{a-1} dt}{\Gamma(a)}$$

If a is scalar, then `gammai (a, x)` is returned for each element of x and vice versa.

If neither a nor x is scalar, the sizes of a and x must agree, and *gammai* is applied element-by-element.

`lgamma` Returns the natural logarithm of the gamma function.

8 Linear Algebra

This chapter documents the linear algebra functions of Octave. Reference material for many of these options may be found in Golub and Van Loan, *Matrix Computations, 2nd Ed.*, Johns Hopkins, 1989, and in *LAPACK Users' Guide*, SIAM, 1992.

8.1 Basic Matrix Functions

`balance`

```
aa = balance (a, opt)
[dd, aa] =  balance(a, opt)
[dd, aa] = balance (a, opt)
[cc, dd, aa, bb] = balance (a, b, opt)
```

`[dd, aa] = balance (a)` returns `aa = dd \ a * dd`. `aa` is a matrix whose row/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` (`bb`) `= cc*a*dd (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)` 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.

`det (a)` Compute the determinant of a using LINPACK.

`eig`

```
              = eig (a)
[v, lambda] = eig (a)
```

The eigenvalues (and eigenvectors) of a matrix are computed in a several step process which begins with a Hessenberg decomposition (see `hess`), followed by a Schur decomposition (see `schur`), from which the eigenvalues are apparent. The eigenvectors, when desired, are computed by further manipulations of the Schur decomposition.

See also: `hess`, `schur`.

`givens`

```
[c, s] = givens (x, y)
G = givens (x, y)
```

`G = givens(x, y)` returns a 2×2 orthogonal matrix `G = [c s; -s' c]` such that `G [x; y] = [*; 0]` (x, y scalars)

`inv` (*a*)

`inverse` (*a*)

Compute the inverse of the square matrix *a*.

`norm` (*a*, *p*)

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 *a*.

p = `2` Largest singular value of *a*.

p = `Inf` Infinity norm, the largest row sum 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)`.

`null` (*a*, *tol*)

Returns 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*)

Returns an orthonormal basis of the range 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*`)`
Returns 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*`)`
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` is the largest singular value of `a`.

`trace (`*a*`)` Compute the trace of *a*, `sum (diag (a))`.

8.2 Matrix Factorizations

`chol (`*a*`)` Compute the Cholesky factor, *r*, of the symmetric positive definite matrix *a*, where $R^T R = A$.

`hess (`*a*`)` Compute the Hessenberg decomposition of the matrix *a*.

```
     h = hess (a)
[p, h] = hess (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 `p * h * p' = a` where `p` is a square unitary matrix (`p' * p = I`, using complex-conjugate transposition) and `h` is upper Hessenberg (`i >= j+1 => h (i, j) = 0`).

`lu (`*a*`)` 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
```

`qr (a)` 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 `q * r = a` where `q` is an orthogonal matrix and `r` is upper triangular.

The permuted `qr` factorization `[q, r, pi] = 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, pi] = qr(a)
```

returns

```
q =

  -0.44721  -0.89443
  -0.89443   0.44721

r =

  -4.47214  -3.13050
   0.00000   0.44721

p =

   0  1
   1  0
```

The permuted qr factorization `[q, r, pi] = qr (a)` factorization allows the construction of an orthogonal basis of `span (a)`.

`schur`

```
[u, s] = schur (a, opt)   opt = "a", "d", or "u"
     s = schur (a)
```

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 $S = U^T AU$ where U is a unitary matrix ($U'U$ 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 blocks 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.

`svd (a)` Compute the singular value decomposition of a

$$A = U \Sigma V^H$$

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*.

8.3 Functions of a Matrix

`expm`

`expm (a)`

Returns the exponential of a matrix, defined as the infinite Taylor series

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots$$

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*`)` 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.

`sqrtm (`*a*`)` Compute the matrix square root 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.

`kron (`*a*`, `*b*`)`

Form the kronecker product of two matrices, defined block by block as

```
x = [a(i, j) b]
```

`qzhess` (*a*, *b*)

Compute the Hessenberg-triangular decomposition of the matrix pencil `(a, b)`. This function returns `aa = q * a * z`, `bb = q * b * z`, `q`, `z` orthogonal. For example,

```
[aa, bb, q, z] = qzhess (a, b)
```

The Hessenberg-triangular decomposition is the first step in Moler and Stewart's QZ decomposition algorithm. (The QZ decomposition will be included in a later release of Octave.)

Algorithm taken from Golub and Van Loan, *Matrix Computations, 2nd edition.*

`qzval` (*a*, *b*)

Compute generalized eigenvalues.

`syl` (*a*, *b*, *c*)

Solve the Sylvester equation

$$AX + XB + C = 0$$

using standard LAPACK subroutines.

9 Polynomial Manipulations

In Octave, a polynomial is represented by its coefficients (arranged in descending order). For example, a vector c of length $n+1$ corresponds to the following n-th order polynomial

$$p(x) = c_1 x^n + \ldots + c_n x + c_{n+1}.$$

`compan (c)`

Compute the companion matrix corresponding to polynomial coefficient vector c.

The companion matrix is

$$A = \begin{bmatrix} -c_2/c_1 & -c_3/c_1 & \cdots & -c_n/c_1 & -c_{n+1}/c_1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

The eigenvalues of the companion matrix are equal to the roots of the polynomial.

`conv (a, b)`

Convolve two vectors.

`y = conv (a, b)` returns a vector of length equal to `length (a) + length (b) - 1`. If a and b are polynomial coefficient vectors, `conv` returns the coefficients of the product polynomial.

`deconv (y, a)`

Deconvolve two vectors.

`[b, r] = deconv (y, a)` solves for b and r such that `y = 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.

`poly (a)`

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)`

Returns the coefficients of the derivative of the polynomial whose coefficients are given by vector c.

`polyinteg (c)`

Returns the coefficients of the integral the polynomial whose coefficients are represented by the vector c.

The constant of integration is set to zero.

`polyreduce` (*c*)

Reduces a polynomial coefficient vector to a minimum number of terms by stripping off any leading zeros.

`polyval` (*c*, *x*)

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*.

`polyvalm` (*c*, *x*)

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.

`residue` (*b*, *a*, *tol*)

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:

$$\frac{P(s)}{Q(s)} = \sum_{m=1}^{M} \frac{r_m}{(s - p_m)_m^e} + \sum_{n=1}^{N} k_n s^{N-n}.$$

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,

Example:

```
b = [1, 1, 1];
a = [1, -5, 8, -4];

[r, p, k, e] = residue (b, a)
```

returns

```
r = [-2, 7, 3]
```

```
p = [2, 2, 1]

k = [](0x0)

e = [1, 2, 1]
```

which implies the following partial fraction expansion

$$\frac{s^2+s+1}{s^3-5s^2+8s-4} = \frac{-2}{s-2} + \frac{7}{(s-2)^2} + \frac{3}{s-1}$$

`roots (v)`

For a vector v with n components, return the roots of the polynomial

$$v_1 z^{n-1} + \cdots + v_{n-1} z + v_n.$$

10 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`.

For example, to solve the set of equations

$$-2x^2 + 3xy + 4\sin(y) - 6 = 0$$
$$3x^2 - 2xy^2 + 3\cos(x) + 4 = 0$$

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)
```

prints

```
solution converged to requested tolerance
```

Tolerances and other options for `fsolve` may be specified using the function `fsolve_options`.

11 Differential Equations

Octave has two built-in functions for solving differential equations. Both are based on reliable ODE solvers written in Fortran.

11.1 Ordinary Differential Equations

The function `lsode` can be used Solve ODEs of the form

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

using Hindmarsh's ODE solver LSODE.

`lsode` (*fcn*, *x0*, *t_out*, *t_crit*)

The first argument is the name of the function to call to compute the vector of right hand sides. It must have the form

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

where *xdot* and *x* are vectors and *t* is a scalar.

The second argument specifies the initial condition, and the third specifies a vector of output times at which the solution is desired, including the time corresponding to the initial condition.

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.

Tolerances and other options for `lsode` may be specified using the function `lsode_options`.

Here is an example of solving a set of two differential equations using `lsode`. The function

```
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
```

is integrated with the command

```
x = lsode ("f", [1; 2], (t = linspace (0, 50, 200)'));
```

producing a set of 200 values stored in the variable *x*. Note that this example takes advantage of the fact that an assignment produces a value to store the values of the output times in the variable *t* directly in the function call The results can then be plotted using the command

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

See Alan C. Hindmarsh, *ODEPACK, A Systematized Collection of ODE Solvers*, in Scientific Computing, R. S. Stepleman, editor, (1983) for more information about this family of ODE solvers.

11.2 Differential-Algebraic Equations

The function `dassl` can be used Solve DAEs of the form

$$0 = f(\dot{x}, x, t), \qquad x(t = 0) = x_0, \dot{x}(t = 0) = \dot{x}_0$$

`dassl (`*fcn*`, `*x_0*`, `*xdot_0*`, `*t_out*`, `*t_crit*`)`

The first argument is the name of the function to call to compute the vector of residuals. It must have the form

res `= f (`*x*`, `*xdot*`, `*t*`)`

where *x*, *xdot*, and *res* are vectors, and *t* is a scalar.

The second and third arguments to `dassl` 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 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.

Tolerances and other options for `dassl` may be specified using the function `dassl_options`.

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.

12 Optimization

12.1 Quadratic Programming

`qpsol`

```
[x, obj, info, lambda]
   = qpsol (x, H, c, lb, ub, lb, A, ub)
```

Solve quadratic programs using Gill and Murray's QPSOL. Because QPSOL is not freely redistributable, this function is only available if you have obtained your own copy of QPSOL. See Appendix A [Installation], page 167.

Tolerances and other options for `qpsol` may be specified using the function `qpsol_options`.

12.2 Nonlinear Programming

`npsol`

```
[x, obj, info, lambda]
   = npsol (x, 'phi', lb, ub, lb, A, ub, lb, 'g', ub)
```

Solve nonlinear programs using Gill and Murray's NPSOL. Because NPSOL is not freely redistributable, this function is only available if you have obtained your own copy of NPSOL. See Appendix A [Installation], page 167.

The second argument is a string containing the name of the objective function to call. The objective function must be of the form

```
y = phi (x)
```

where x is a vector and y is a scalar.

Tolerances and other options for `npsol` may be specified using the function `npsol_options`.

12.3 Linear Least Squares

`gls` (*Y*, *X*, *O*)

Generalized least squares (GLS) estimation for the multivariate model

```
Y = X * B + E,  mean(E) = 0,  cov(vec(E)) = (s^2)*O
```

with

```
Y an T x p matrix
X an T x k matrix
B an k x p matrix
E an T x p matrix
O an Tp x Tp matrix
```

Each row of Y and X is an observation and each column a variable.

Returns BETA, v, and, R, where BETA is the GLS estimator for B, v is the GLS estimator for s^2, and R = Y - X*BETA is the matrix of GLS residuals.

`ols` (*Y*, *X*)

Ordinary Least Squares (OLS) estimation for the multivariate model

```
Y = X*B + E,  mean (E) = 0,  cov (vec (E)) = kron (S, I)
```

with

```
Y an T x p matrix
X an T x k matrix
B an k x p matrix
E an T x p matrix
```

Each row of Y and X is an observation and each column a variable.

Returns BETA, SIGMA, and R, where BETA is the OLS estimator for B, i.e.

```
BETA = pinv(X)*Y,
```

where pinv(X) denotes the pseudoinverse of X, SIGMA is the OLS estimator for the matrix S, i.e.

```
SIGMA = (Y - X*BETA)'*(Y - X*BETA) / (T - rank(X))
```

and R = Y - X*BETA is the matrix of OLS residuals.

13 Quadrature

13.1 Functions of one Variable

`quad`

```
[v, ier, nfun] = quad ("f", a, b)
[v, ier, nfun] = quad ("f", a, b, tol)
[v, ier, nfun] = quad ("f", a, b, tol, sing)
```

Integrate a nonlinear function of one variable using Quadpack.

Where the first argument is the name of the 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 specifies the desired accuracy of the result. The optional argument *sing* is a vector of values at which the integrand is singular.

Tolerances and other options for `quad` may be specified using the function `quad_options`.

13.2 Orthogonal Collocation

`colloc`

```
[r, A, B, q] = colloc (n)
[r, A, B, q] = colloc (n, "left")
[r, A, B, q] = colloc (n, "left", "right")
```

Compute derivative and integral weight matrices for orthogonal collocation using the subroutines given in J. Michelsen and M. L. Villadsen, *Solution of Differential Equation Models by Polynomial Approximation*.

14 Control Theory

`abcddim` (*a*, *b*, *c*, *d*)

Check for compatibility of the dimensions of the matrices defining the linear system [A, B, C, D] corresponding to

$$\frac{dx}{dt} = Ax + Bu$$

$$y = Cx + Du$$

or a similar discrete-time system.

If the matrices are compatibly dimensioned, then `abcddim` returns n = number of system states, m = number of system inputs, and p = number of system outputs. Otherwise `abcddim` returns $n = m = p = -1$.

`are` (*a*, *b*, *c*, *opt*)

Returns the solution, x, of the algebraic Riccati equation

```
a' x + x a - x b x + c = 0
```

for identically dimensioned square matrices a, b, c. If b (c) is not square, then the function attempts to use b`*`b`'` (c`'*`c) instead.

Solution method: apply Laub's Schur method (IEEE Transactions on Automatic Control, 1979) to the appropriate Hamiltonian matrix.

opt is an option passed to the eigenvalue balancing routine. Default is `"B"`.

`c2d` (*a*, *b*, *t*)

Converts the continuous time system described by:

$$\frac{dx}{dt} = Ax + Bu$$

into a discrete time equivalent model

$$x_{k+1} = A_d x_k + B_d u_k$$

via the matrix exponential assuming a zero-order hold on the input and sample time t.

`dare` (*a*, *b*, *c*, *r*, *opt*)

Returns the solution, x of the discrete-time algebraic Riccati equation

```
a' x a - x + a' x b (r + b' x b)^(-1) b' x a + c = 0
```

for matrices with dimensions:

a: n `by` n
b: n `by` m
c: n `by` n`, symmetric positive semidefinite`
r: m `by` m`, symmetric positive definite (invertible)`

If c is not square, then the function attempts to use `c'*c` instead.

Solution method: Laub's Schur method (IEEE Transactions on Automatic Control, 1979) is applied to the appropriate symplectic matrix.

See also: Ran and Rodman, *Stable Hermitian Solutions of Discrete Algebraic Riccati Equations*, Mathematics of Control, Signals and Systems, Volume 5, Number 2 (1992).

opt is an option passed to the eigenvalue balancing routine. The default is `"B"`.

`dgram (a, b)`

Returns the discrete controllability and observability gramian for the discrete time system described by

$$x_{k+1} = Ax_k + Bu_k$$

$$y_k = Cx_k + Du_k$$

`dgram (a, b)` returns the discrete controllability gramian and `dgram (a', c')` returns the observability gramian.

`dlqe (a, g, c, sigw, sigv [, z])`

Linear quadratic estimator (Kalman filter) design for the discrete time system

$$x_{k+1} = Ax_k + Bu_k + Gw_k$$

$$y_k = Cx_k + Du_k + w_k$$

where *w*, *v* are zero-mean gaussian noise processes with respective intensities *sigw* `= cov (w, w)` and *sigv* `= cov (v, v)`.

If specified, *z* is `cov (w, v)`. Otherwise `cov (w, v) = 0`.

The observer structure is

$$z_{k+1} = Az_k + Bu_k + k(y_k - Cz_k - Du_k)$$

Returns:

l is the observer gain, `(A - A L C)` is stable.

m is the Ricatti equation solution.

p is the estimate error covariance after the measurement update.

e are the closed loop poles of `(A - A L C)`.

`dlqr (a, b, q, r [, z])`

Linear quadratic regulator design for the discrete time system

$$x_{k+1} = Ax_k + Bu_k$$

to minimize the cost functional

```
J = Sum [ x' Q x + u' R u ],              Z omitted
```

or

```
J = Sum [ x' Q x + u' R u +2 x' Z u ],    Z included
```

Returns:

k is the state feedback gain, `(A - B K)` is stable.

p is the solution of algebraic Riccati equation.

e are the closed loop poles of *(A - B K)*.

`dlyap` (*a*, *b*)

Solve the discrete-time Lyapunov equation

```
a x a' - x + b = 0
```

for square matrices *a*, *b*. If *b* is not square, then the function attempts to solve either

```
a x a' - x + b b' = 0
```

or

```
a' x a - x + b' b = 0
```

whichever is appropriate.

Uses Schur decomposition method as in Kitagawa, International Journal of Control (1977); column-by-column solution method as suggested in Hammarling, IMA Journal of Numerical Analysis, (1982).

`is_controllable` (*a*, *b*, *tol*)

If the pair (a, b) is controllable, then return value 1. Otherwise, returns a value of 0.

tol is a roundoff parameter, set to `2*eps` if omitted.

Currently just constructs the controllability matrix and checks rank. A better method is as follows (Boley and Golub, Systems and Control Letters, 1984): Controllability is determined by applying Arnoldi iteration with complete re-orthogonalization to obtain an orthogonal basis of the Krylov subspace

$$\operatorname{span}\left([b\ ab\ \ldots\ a^{n-1}b]\right)$$

`is_observable` (*a*, *c*, *tol*)

Returns 1 if the pair `(a, c)` is observable. Otherwise, returns a value of 0.

`lqe` (*a*, *g*, *c*, *sigw*, *sigv*, *z*)

```
[k, p, e] = lqe (a, g, c, sigw, sigv, z)
```

Linear quadratic estimator (Kalman filter) design for the continuous time system

$$\frac{dx}{dt} = Ax + Bu$$

$$y = Cx + Du$$

where *w*, *v* are zero-mean gaussian noise processes with respective intensities

```
sigw = cov (w, w)
sigv = cov (v, v)
```

z (if specified) is the cross-covariance `cov (`*w*`, `*v*`)`; the default value is `cov (`*w*`, `*v*`) = 0`.

Observer structure is `dz/dt = A z + B u + k (y - C z - D u)`

returns:

k is observer gain: `(A - K C)` is stable.

p is solution of algebraic Riccati equation.

e is the vector of closed loop poles of `(A - K C)`.

`lqr (`*a*`, `*b*`, `*q*`, `*r*`, `*z*`)`
`[`*k*`, `*p*`, `*e*`] = lqr (`*a*`, `*b*`, `*q*`, `*r*`, `*z*`)`

Linear quadratic regulator design for the continuous time system

$$\frac{dx}{dt} = Ax + Bu$$

to minimize the cost functional

$$J = \int_0^\infty x'Qx + u'Ru$$

z omitted or

$$J = \int_0^\infty x'Qx + u'Ru + 2x'Zu$$

z included

Returns:

k is state feedback gain: `(A - B K)` is stable.

p is the stabilizing solution of appropriate algebraic Riccati equation.

e is the vector of the closed loop poles of `(A - B K)`.

`lyap (`*a*`, `*b*`, `*c*`)`

Solve the Lyapunov (or Sylvester) equation via the Bartels-Stewart algorithm (Communications of the ACM, 1972).

If `(a, b, c)` are specified, then `lyap` returns the solution of the Sylvester equation

```
a x + x b + c = 0
```

If only `(a, b)` are specified, then `lyap` returns the solution of the Lyapunov equation

```
a' x + x a + b = 0
```

If *b* is not square, then `lyap` returns the solution of either

```
a' x + x a + b' b = 0
```

or

```
a x + x a' + b b' = 0
```

whichever is appropriate.

Solves by using the Bartels-Stewart algorithm (1972).

`tzero` (*a*, *b*, *c*, *d*, *bal*)

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; see Hodel, *Computation of System Zeros with Balancing*, Linear Algebra and its Applications, July 1993.

15 Signal Processing

I hope that someday Octave will include more signal processing functions. If you would like to help improve Octave in this area, please contact `bug-octave@che.utexas.edu`.

`fft` (*a*) Compute the FFT of *a* using subroutines from FFTPACK.

`fft2` (*a*) Compute the two dimensional FFT of *a*.

`fftconv` (*a*, *b*, *N*)

This function returns the convolution of the vectors *a* and *b*, 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*)

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.

`filter` (*b*, *a*, *x*)

This function returns the solution to the following linear, time-invariant difference equation:

$$\sum_{k=0}^{N} a_{k+1} y_{n-k} + \sum_{k=0}^{M} b_{k+1} x_{n-k} = 0, \qquad 1 \le n \le P$$

where $a \in \Re^{N-1}$, $b \in \Re^{M-1}$, and $x \in \Re^{P}$. An equivalent form of this equation is:

$$y_n = \sum_{k=1}^{N} c_{k+1} y_{n-k} + \sum_{k=0}^{M} d_{k+1} x_{n-k}, \qquad 1 \le n \le P$$

where $c = a/a_1$ and $d = b/a_1$.

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:

$$H(z) = \frac{\sum_{k=0}^{M} d_{k+1} z^{-k}}{1 + \sum_{k+1}^{N} c_{k+1} z^{-k}}$$

When called as

```
[y, sf] = filter (b, a, x, si)
```

`filter` uses the argument *si* as the initial state of the system and and returns the final state in *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 set, the initial state vector is set to all zeros.

`freqz` Compute the frequency response of a filter.

`[`*h*`, `*w*`] = freqz (`*b*`)` returns the complex frequency response *h* of the FIR filter with coefficients *b*. The response is evaluated at 512 angular frequencies between 0 and π.

The output value *w* is a vector containing the 512 frequencies.

`[`*h*`, `*w*`] = freqz (`*b*`, `*a*`)` returns the complex frequency response of the rational IIR filter whose numerator has coefficients *b* and denominator coefficients *a*.

`[`*h*`, `*w*`] = freqz (`*b*`, `*a*`, `*n*`)` returns the response evaluated at *n* angular frequencies. For fastest computation n should factor into a small number of small primes.

`[`*h*`, `*w*`] = freqz (`*b*`, `*a*`, `*n*`, "whole")` evaluates the response at n frequencies between 0 and 2π.

`ifft (`*a*`)` Compute the inverse FFT of a using subroutines from FFTPACK.

`ifft2` Compute the two dimensional inverse FFT of a.

`sinc (`*x*`)` Returns $\sin(\pi x)/(\pi x)$.

16 Sets

Octave has a limited set of functions for managing sets of data, where a set is defined as a collection unique elements.

Given a matrix or vector of values, the function `create_set` returns a row vector containing unique values, sorted in ascending order. For example, '`create_set ([1, 2; 3, 4; 4, 2])`' returns the vector '`[1, 2, 3, 4]`'.

The functions `union` and `intersection` take two sets as arguments and return the union and interection, respectively. For example, '`union ([1, 2, 3], [2, 3, 5])`' returns the vector '`[1, 2, 5]`'.

The function `complement (`*a*`,` *b*`)` returns the elements of set *b* that are not in set *a*. For example, '`complement ([1, 2, 3], [2, 3, 5])`' returns the value '`5`'.

17 Statistics

I hope that someday Octave will include more statistics functions. If you would like to help improve Octave in this area, please contact `bug-octave@che.utexas.edu`.

`corrcoef (`*x* `[,` *y*`])`
: 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*. If invoked with one argument, compute `corrcoef (`*x*`,` *x*`)`.

`cov (`*x* `[,` *y*`])`
: 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*. If invoked with one argument, compute `cov (`*x*`,` *x*`)`.

`kurtosis (`*x*`)`
: If *x* is a vector of length N, return the kurtosis

```
kurtosis(x) = N^(-1) std(x)^(-4) SUM_i (x(i)-mean(x))^4 - 3
```

of *x*. If *x* is a matrix, return the row vector containing the kurtosis of each column.

`mahalanobis (`*x*`,` *y*`)`
: Returns 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).

`mean (`*a*`)`
: If *a* is a vector, compute the mean of the elements of *a*. If *a* is a matrix, compute the mean for each column and return them in a row vector.

`median (`*a*`)`
: If *a* is a vector, compute the median value of the elements of *a*. If *a* is a matrix, compute the median value for each column and return them in a row vector.

`skewness (`*x*`)`
: If *x* is a vector of length N, return the skewness

```
skewness (x) = N^(-1) std(x)^(-3) SUM_i (x(i)-mean(x))^3
```

of *x*. If *x* is a matrix, return the row vector containing the skewness of each column.

`std (`*a*`)`
: If *a* is a vector, compute the standard deviation of the elements of *a*. If *a* is a matrix, compute the standard deviation for each column and return them in a row vector.

18 Plotting

All of Octave's plotting functions use `gnuplot` to handle the actual graphics. There are two low-level functions, `gplot` and `gsplot`, that behave almost exactly like the corresponding `gnuplot` functions `plot` and '`splot`'. A number of other higher level plotting functions, patterned after the graphics functions found in MATLAB version 3.5, are also available. These higher level functions are all implemented in terms of the two low-level plotting functions.

18.1 Two-Dimensional Plotting

The syntax for Octave's two-dimensional plotting function, `gplot`, is

```
gplot ranges expression using title style
```

where the *ranges*, *using*, *title*, and *style* arguments are optional, and the *using*, *title* and *style* qualifiers may appear in any order after the expression. You may plot multiple expressions with a single command by separating them with commas. Each expression may have its own set of qualifiers.

The optional item *ranges* has the syntax

```
[ x_lo : x_up ] [ y_lo : y_up ]
```

and may be used to specify the ranges for the axes of the plot, independent of the actual range of the data. The range for the y axes and any of the individual limits may be omitted. A range `[:]` indicates that the default limits should be used. This normally means that a range just large enough to include all the data points will be used.

The expression to be plotted must not contain any literal matrices (e.g. `[ 1, 2; 3, 4 ]`) since it is nearly impossible to distinguish a plot range from a matrix of data.

See the help for `gnuplot` for a description of the syntax for the optional items.

By default, the `gplot` command plots the second column of a matrix versus the first. If the matrix only has one column, it is taken as a vector of y-coordinates and the x-coordinate is taken as the element index, starting with zero. For example,

```
gplot rand (100,1) with linespoints
```

will plot 100 random values and connect them with lines. When `gplot` is used to plot a column vector, the indices of the elements are taken as x values.

If there are more than two columns, you can choose which columns to plot with the *using* qualifier. For example, given the data

```
x = (-10:0.1:10)';
data = [x, sin(x), cos(x)];
```

the command

```
gplot [-11:11] [-1.1:1.1] data with lines, data using 1:3 with impulses
```

will plot two lines. The first line is generated by the command `data with lines`, and is a graph of the sine function over the range -10 to 10. The data is taken from the first two columns of the matrix because columns to plot were not specified with the *using* qualifier.

The clause `using 1:3` in the second part of this plot command specifies that the first and third columns of the matrix `data` should be taken as the values to plot.

In this example, the ranges have been explicitly specified to be a bit larger than the actual range of the data so that the curves do not touch the border of the plot.

In addition to the basic plotting commands, the whole range of `set` and `show` commands from `gnuplot` are available, as is `replot`.

The `set` and `show` commands allow you to set and show `gnuplot` parameters. For more information about the set and show commands, see the `gnuplot` user's guide (also available on line if you run `gnuplot` directly, instead of running it from Octave).

The `replot` command allows you to force the plot to be redisplayed. This is useful if you have changed something about the plot, such as the title or axis labels. The `replot` command also accepts the same arguments as `gplot` or `gsplot` (except for data ranges) so you can add additional lines to existing plots.

For example,

```
set term tek40
set output "/dev/plotter"
set title "sine with lines and cosine with impulses"
replot "sin (x) w l"
```

will change the terminal type for plotting, add a title to the current plot, add a graph of sin (x) to the plot, and force the new plot to be sent to the plot device. This last step is normally required in order to update the plot. This default is reasonable for slow terminals or hardcopy output devices because even when you are adding additional lines with a replot command, gnuplot always redraws the entire plot, and you probably don't want to have a completely new plot generated every time something as minor as an axis label changes.

Since this may not matter as much on faster terminals, you can tell Octave to redisplay the plot each time anything about it changes by setting the value of the builtin variable `automatic_replot` to the value `"true"`.

Note that NaN values in the plot data are automatically omitted, and Inf values are converted to a very large value before calling gnuplot.

The MATLAB-style two-dimensional plotting commands are:

`plot (`*args*`)`

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.

If more than one argument is given, they are interpreted as

```
plot (x [, y] [, fmt] ...)
```

where *y* and *fmt* 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 first argument is a vector and the second is a matrix, the the vector is plotted versus the columns (or rows) of the matrix. (using whichever combination matches, with columns tried first.)
- If the first argument is a matrix and the second is a vector, the the columns (or rows) of the matrix are plotted versus the vector. (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.

The *fmt* argument, if present is interpreted as follows. If *fmt* is missing, the default gnuplot line style is assumed.

‘`-`’ Set lines plot style (default).

‘`.`’ Set dots plot style.

‘`@`’ Set points plot style.

‘`-@`’ Set linespoints plot style.

‘`^`’ Set impulses plot style.

‘`L`’ Set steps plot style.

‘`#`’ Set boxes plot style.

‘`~`’ Set errorbars plot style.

‘`#~`’ Set boxerrorbars plot style.

‘n’ Interpreted as the plot color if *n* is an integer in the range 1 to 6.

‘nm’ If *nm* is a two digit integer and *m* is an integer in the range 1 to 6, *m* is interpreted as the point style. This is only valid in combination with the `@` or `-@` specifiers.

'c' If c is one of "r", "g", "b", "m", "c", or "w", it is interpreted as the plot color (red, green, blue, magenta, cyan, or white).

'+'

'*'

'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
```

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, "*")
```

This command will plot the data in *b* will be plotted with points displayed as `*`.

`hold` 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` Returns 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.

`loglog (`*args*`)`

Make a two-dimensional plot using log scales for both axes. See the description of `plot` above for a description of the arguments that `loglog` will accept.

`semilogx (`*args*`)`

Make a two-dimensional plot using a log scale for the *x* axis. See the description of `plot` above for a description of the arguments that `semilogx` will accept.

`semilogy (`*args*`)`

Make a two-dimensional plot using a log scale for the *y* axis. See the description of `plot` above for a description of the arguments that `semilogy` will accept.

`contour` (*z*, *n*, *x*, *y*)
: Make a contour plot of the three-dimensional surface described by *z*. Someone needs to improve `gnuplot`'s contour routines before this will be very useful.

`polar` (*theta*, *rho*)
: Make a two-dimensional plot given polar the coordinates *theta* and *rho*.

18.2 Three-Dimensional Plotting

The syntax for Octave's three-dimensional plotting function, `gsplot`, is

`gsplot` *ranges expression using title style*

where the *ranges*, *using*, *title*, and *style* arguments are optional, and the *using*, *title* and *style* qualifiers may appear in any order after the expression. You may plot multiple expressions with a single command by separating them with commas. Each expression may have its own set of qualifiers.

The optional item *ranges* has the syntax

```
[ x_lo : x_up ] [ y_lo : y_up ] [ z_lo : z_up ]
```

and may be used to specify the ranges for the axes of the plot, independent of the actual range of the data. The range for the y and z axes and any of the individual limits may be omitted. A range `[:]` indicates that the default limits should be used. This normally means that a range just large enough to include all the data points will be used.

The expression to be plotted must not contain any literal matrices (e.g. `[ 1, 2; 3, 4 ]`) since it is nearly impossible to distinguish a plot range from a matrix of data.

See the help for `gnuplot` for a description of the syntax for the optional items.

By default, the `gsplot` command plots each column of the expression as the z value, using the row index as the x value, and the column index as the y value. The indices are counted from zero, not one. For example,

```
gsplot rand (5, 2)
```

will plot a random surface, with the x and y values taken from the row and column indices of the matrix.

If parametric plotting mode is set (using the command '`set parametric`', then `gsplot` takes the columns of the matrix three at a time as the x, y and z values that define a line in three space. Any extra columns are ignored, and the x and y values are expected to be sorted. For example, with '`parametric`' set, it makes sense to plot a matrix like

$$\begin{bmatrix} 1 & 1 & 3 & 2 & 1 & 6 & 3 & 1 & 9 \\ 1 & 2 & 2 & 2 & 2 & 5 & 3 & 2 & 8 \\ 1 & 3 & 1 & 2 & 3 & 4 & 3 & 3 & 7 \end{bmatrix}$$

but not `rand (5, 30)`.

The MATLAB-style three-dimensional plotting commands are:

`mesh (x, y, z)`
: Plot a mesh given matrices `x`, and *y* from `meshdom` and a matrix *z* corresponding to the *x* and *y* coordinates of the mesh.

`meshdom (x, y)`
: Given vectors of *x* and *y* coordinates, return two matrices corresponding to the *x* and *y* coordinates of the mesh.

  See the file '`sombrero.m`' for an example of using `mesh` and `meshdom`.

18.3 Miscellaneous Plotting Functions

`bar (x, y)`
: 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 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.

`grid`
: For two-dimensional plotting, force the display of a grid on the plot.

`stairs (x, y)`
: Given two vectors of x-y data, bar 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.

`title (string)`
: Specify a title for the plot. If you already have a plot displayed, use the command `replot` to redisplay it with the new title.

`xlabel (`*string*`)`
`ylabel (`*string*`)`

Specify x and y axis labels for the plot. If you already have a plot displayed, use the command `replot` to redisplay it with the new labels.

`sombrero (`*n*`)`

Display a classic three-dimensional mesh plot. The parameter *n* allows you to increase the resolution.

`clearplot`
`clg`

Clear the plot window and any titles or axis labels. The name `clg` is aliased to `clearplot` for compatibility with MATLAB.

The commands '`gplot clear`', '`gsplot clear`', and '`replot clear`' are equivalent to '`clearplot`'. (Previously, commands like '`gplot clear`' would evaluate '`clear`' as an ordinary expression and clear all the visible variables.)

`closeplot`

Close stream to the `gnuplot` subprocess. If you are using X11, this will close the plot window.

`purge_tmp_files`

Delete the temporary files created by the plotting commands.

Octave creates temporary data files for `gnuplot` and then sends commands to `gnuplot` through a pipe. Octave will delete the temporary files on exit, but if you are doing a lot of plotting you may want to clean up in the middle of a session.

A future version of Octave will eliminate the need to use temporary files to hold the plot data.

`axis (`*limits*`)`

Sets the 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 second and third specify the limits for the y axis, and the fourth and fifth specify the limits for the z axis.

With no arguments, `axis` turns autoscaling on.

If your plot is already drawn, then you need to use `replot` before the new axis limits will take effect. You can get this to happen automatically by setting the built-in variable `automatic_replot` to `"true"`. See Section 6.2 [User Preferences], page 66.

`hist (`*y*`, `*x*`)`

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.

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.

19 Image Processing

To display images using these functions, you must be using Octave with the X Window System, and you must have either `xloadimage` or `xv` installed. You do not need to be running X in order to manipulate images, however, so some of these functions may be useful even if you are not able to view the results.

`colormap` 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.

`gray (`*n*`)` Create a gray colormap with values from 0 to *n*. The argument *n* should be a scalar. If it is omitted, 64 is assumed.

`gray2ind` Convert a gray scale intensity image to an Octave indexed image.

`image` Display an Octave image matrix.

`image (`*x*`)` displays 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.

`image (`*x*`, `*zoom*`)` changes the zoom factor. The default value is 4.

`imagesc` Scale and display a matrix as an image.

`imagesc (`*x*`)` displays a scaled version of the matrix *x*. The matrix is scaled so that its entries are indices into the current colormap. The scaled matrix is returned.

`imagesc (`*x*`, `*zoom*`)` sets the magnification, the default value is 4.

`imshow` Display images.

`imshow (`*x*`)` displays an indexed image using the current colormap.

`imshow (`*x*`, `*map*`)` displays an indexed image using the specified colormap.

`imshow (`*i*`, `*n*`)` displays a gray scale intensity image.

`imshow (`*r*`, `*g*`, `*b*`)` displays an RGB image.

`ind2gray` Convert an Octave indexed image to a gray scale intensity image.

y `= ind2gray (`*x*`)` converts an indexed image to a gray scale intensity image. The current colormap is used to determine the intensities. The intensity values lie between 0 and 1 inclusive.

y `= ind2gray (`*x*`, `*map*`)` uses the specified colormap instead of the current one in the conversion process.

`ind2rgb` Convert an indexed image to red, green, and blue color components.

`[`*r*`, `*g*`, `*b*`] = ind2rgb (`*x*`)` uses the current colormap for the conversion.

`[`*r*`, `*g*`, `*b*`] = ind2rgb (`*x*`, `*map*`)` uses the specified colormap.

`loadimage`

Load an image file.

`[`*x*`, `*map*`] = loadimage (`*file*`)` loads an image and it's associated color map from the specified *file*. The image must be stored in Octave's image format.

`ocean (`*n*`)` Create color colormap. The argument *n* should be a scalar. If it is omitted, 64 is assumed.

`rgb2ind` Convert and RGB image to an Octave indexed image.

`[`*x*`, `*map*`] = rgb2ind (`*r*`, `*g*`, `*b*`)`

`saveimage`

Save a matrix to disk in image format.

`saveimage (`*file*`, `*x*`)` saves matrix *x* to *file* in Octave's image format. The current colormap is also saved in the file.

`saveimage (`*file*`, `*x*`, "img")` saves the image in the default format and is the same as `saveimage (`*file*`, `*x*`)`.

`saveimage (`*file*`, `*x*`, "ppm")` saves the image in ppm format instead of the default Octave image format.

`saveimage (`*file*`, `*x*`, "ps")` saves the image in PostScript format instead of the default Octave image format. (Note: images saved in PostScript format can not be read back into Octave with loadimage.)

`saveimage (`*file*`, `*x*`, `*fmt*`, `*map*`)` saves the image along with the specified colormap in the specified format.

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.

20 Input and Output

There are two distinct classes of input and output functions. The first set are modeled after the functions available in MATLAB. The second set are modeled after the standard I/O library used by the C programming language. The C-style I/O functions offer more flexibility and control over the output, but are not quite as easy to use as the simpler MATLAB-style I/O functions.

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`) it also allows you to scan forward and backward, and search for specific items.

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 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 immediately. See Section 20.2 [C-Style I/O Functions], page 127.

You can select the program to run as the pager by setting the variable `PAGER`, and you can turn paging off by setting the value of the variable `page_screen_output` to the string '`"false"`'. See Section 6.2 [User Preferences], page 66.

20.1 Basic Input and 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

```
octave:13> 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`. For example, the following expression

```
disp ("The value of pi is:"), disp (pi)
```

will print

```
The value of pi is:
3.1416
```

Note that the output from `disp` always ends with a newline.

A simple way to control the output format is with the `format` statement. For example, to print more digits for pi you can use the command

```
format long
```

Then the expression above will print

```
The value of pi is:
3.14159265358979
```

Here is a summary of the options for `format`:

`short`
: This is the default format. 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.

`long`
: Octave will try to print numbers with at least 15 significant figures within a field that is a maximum of 24 characters wide.

  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
  ```

`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.

`+`
: 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 `input` function may be used for prompting the user for a value and storing the result in a variable. 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 Section 20.2 [C-Style I/O Functions], page 127.

The second input function, `keyboard`, is normally used for simple debugging. Using `keyboard`, it is possible to examine the values of variables within a function, and to assign newassign new variables Like `input`, it prompts the user for input, but no value is returned, 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.

For both of these functions, the normal command line history and editing functions are available at the prompt.

To save variables in a file, use the `save` command. For example, the command

```
save data a b c
```

saves the variables '`a`', '`b`', and '`c`' in the file '`data`'.

The *save* command can read files in Octave's text and binary formats as well as MATLAB's binary format. You can specify the default format with the built-in variable *default_save_format* using one of the following values: `"binary"` or `"mat-binary"`. The initial default save format is Octave's text format.

You can use the built-in variable `save_precision` to specify the number of digits to keep when saving data in text format.

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.

The following options may be specified for `save`.

`-ascii` Save the data in Octave's text data format. Using this flag overrides the value of the built-in variable `default_save_format`.

`-binary` Save the data in Octave's binary data format. Using this flag overrides the value of the built-in variable `default_save_format`.

`-float-binary`
Save the data in Octave's binary data format but only using single precision. Using this flag overrides the value of the built-in variable `default_save_precision`. You should use this format only if you know that all the values to be saved can be represented in single precision.

`-mat-binary`
Save the data in MATLAB's binary data format. Using this flag overrides the value of the built-in variable `default_save_format`.

`-save-builtins`
Force Octave to save the values of built-in variables too. By default, Octave does not save built-in variables.

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.

To restore the values from a file, use the `load` command. For example, to restore the variables saved in the file '`data`', use the command

```
load data
```

Octave will refuse to overwrite existing variables unless you use the option '`-force`'.

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.

As with `save`, you may specify a list of variables and `load` will only extract those variables with names that match.

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 added in the future).

The following options may be specified for `save`.

`-force` Force variables currently in memory to be overwritten by variables with the same name found in the file.

`-ascii` Force Octave to assume the file is in Octave's text format.

`-binary` Force Octave to assume the file is in Octave's binary format.

`-mat-binary`
: Force Octave to assume the file is in MATLAB's binary format.

20.2 C-Style I/O Functions

The 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 either to an integer file number (as returned by '`fopen`') or a file name.

There are three files that are always available:

`stdin`
: The standard input stream (file number 0). When Octave is used interactively, this is filtered through the command line editing functions.

`stdout`
: The standard output stream (file number 1). Data written to the standard output is normally filtered through the pager.

`stderr`
: The standard error stream (file number 2). Even if paging is turned on, the standard error is not sent to the pager. It is useful for error messages and prompts.

You should always use the symbolic names given in the table above, rather than referring to these files by number, since it will make your programs clearer.

20.2.1 Opening and Closing Files

To open a file, use the function `fopen (name, mode)`. It returns an integer value that may be used to refer to the file later. The second argument is a one or two character string that specifies whether the file is to be opened for reading, writing, or both.

For example,

```
myfile = fopen ("splat.dat", "r");
```

opens the file '`splat.dat`' for reading. Opening a file that is already open has no effect.

The possible values '`mode`' may have are

'`r`'
: Open a file for reading.

'`w`'
: Open a file for writing. The previous contents are discared.

'`a`'
: Open or create a file for writing at the end of the file.

'`r+`'
: Open an existing file for reading and writing.

'`w+`'
: Open a file for reading or writing. The previous contents are discared.

'`a+`'
: Open or create a file for reading or writing at the end of the file.

To close a file once you are finished with it, use the function `fclose (`*file*`)`. If an error is encountered while trying to close the file, an error message is printed and `fclose` returns 0. Otherwise, it returns 1.

20.2.2 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.

`printf (`*template*`, ...)`
: The `printf` function prints the optional arguments under the control of the template string *template* to the stream `stdout`.

`fprintf (`*file*`,` *template*`, ...)`
: This function is just like `printf`, except that the output is written to the stream *file* instead of `stdout`.

`sprintf (`*template*`, ...)`
: This is like `printf`, except that the output is written to 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.

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.

20.2.3 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.

20.2.4 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 20.2.5 [Integer Conversions], page 131, for details. '`%d`' and '`%i`' are synonymous for output, but are different when used with `scanf` for input (see Section 20.2.10 [Table of Input Conversions], page 134).

'`%o`'
: Print an integer as an unsigned octal number. See Section 20.2.5 [Integer Conversions], page 131, for details.

'`%u`'
: Print an integer as an unsigned decimal number. See Section 20.2.5 [Integer Conversions], page 131, for details.

'`%x`', '`%X`'
: Print an integer as an unsigned hexadecimal number. '`%x`' uses lower-case letters and '`%X`' uses upper-case. See Section 20.2.5 [Integer Conversions], page 131, for details.

'`%f`'
: Print a floating-point number in normal (fixed-point) notation. See Section 20.2.6 [Floating-Point Conversions], page 131, for details.

'`%e`', '`%E`'
: Print a floating-point number in exponential notation. '`%e`' uses lower-case letters and '`%E`' uses upper-case. See Section 20.2.6 [Floating-Point Conversions], page 131, for details.

'`%g`', '`%G`'
: Print a floating-point number in either normal or exponential notation, whichever is more appropriate for its magnitude. '`%g`' uses lower-case letters and '`%G`' uses upper-case. See Section 20.2.6 [Floating-Point Conversions], page 131, for details.

'`%c`'
: Print a single character. See Section 20.2.7 [Other Output Conversions], page 132.

'`%s`'
: Print a string. See Section 20.2.7 [Other Output Conversions], page 132.

'`%%`'
: Print a literal '`%`' character. See Section 20.2.7 [Other Output Conversions], page 132.

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.

20.2.5 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:

'-' Left-justify the result in the field (instead of the normal right-justification).

'+' For the signed '%d' and '%i' conversions, print a plus sign if the value is positive.

' ' For the signed '%d' and '%i' conversions, if the result doesn't start with a plus or minus sign, prefix it with a space character instead. Since the '+' 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.

20.2.6 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*.*ddd*e[+|-]*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.

20.2.7 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 '.

20.2.8 Formatted Input

Here are the descriptions of the functions for performing formatted input.

`scanf` (*template*)
: The `scanf` function reads formatted input from the stream `stdin` under the control of the template string *template*. The resulting values are returned.

`fscanf` (*file*, *template*)
: This function is just like `scanf`, except that the input is read from the stream *file* instead of `stdin`.

`sscanf` (*string*, *template*)
: This is like `scanf`, 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.

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.

20.2.9 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 use a pointer argument, 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 (those that don't are explicitly documented), and these discarded characters don't count towards the maximum field width.
- 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.

20.2.10 Table of Input Conversions

Here is a table that summarizes the various conversion specifications:

'%d' Matches an optionally signed integer written in decimal. See Section 20.2.11 [Numeric Input Conversions], page 135.

'%i' Matches an optionally signed integer in any of the formats that the C language defines for specifying an integer constant. See Section 20.2.11 [Numeric Input Conversions], page 135.

'%o' Matches an unsigned integer written in octal radix. See Section 20.2.11 [Numeric Input Conversions], page 135.

'%u' Matches an unsigned integer written in decimal radix. See Section 20.2.11 [Numeric Input Conversions], page 135.

'%x', '%X' Matches an unsigned integer written in hexadecimal radix. See Section 20.2.11 [Numeric Input Conversions], page 135.

'%e', '%f', '%g', '%E', '%G'
Matches an optionally signed floating-point number. See Section 20.2.11 [Numeric Input Conversions], page 135.

'%s' Matches a string containing only non-whitespace characters. See Section 20.2.12 [String Input Conversions], page 135.

'%c' 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 20.2.12 [String Input Conversions], page 135.

'%%' This matches a literal '%' 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.

20.2.11 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.

20.2.12 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 with 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,"`.

20.2.13 Binary I/O

Octave has to C-style functions for reading and writing binary data. They are `fread` and `fwrite` and are patterned after the standard C functions with the same names.

`fread` (*file*, *size*, *precision*)

: This function reads data in binary form of type *precision* from the specified *file*, which may be either a file name, or a file number as returned from `fopen`.

  The argument *size* specifies the size of the matrix to return. It may be a scalar or a two-element vector. If it is a scalar, `fread` returns a column vector of the specified length. If it is a two-element vector, it specifies the number of rows and columns of the result matrix, and `fread` fills the elements of the matrix in column-major order.

  The argument *precision* is a string specifying the type of data to read and may be one of `"char"`, `"schar"`, `"short"`, `"int"`, `"long"`, `"float"`, `"double"`, `"uchar"`, `"ushort"`, `"uint"`, or `"ulong"`. The default precision is `"uchar"`.

  The `fread` function returns two values, `data`, which is the data read from the file, and `count`, which is the number of elements read.

`fwrite` (*file*, *data*, *precision*)

: This function writes data in binary form of type *precision* to the specified *file*, which may be either a file name, or a file number as returned from `fopen`.

  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 argument *precision* is a string specifying the type of data to read and may be one of `"char"`, `"schar"`, `"short"`, `"int"`, `"long"`, `"float"`, `"double"`, `"uchar"`, `"ushort"`, `"uint"`, or `"ulong"`. The default precision is `"uchar"`.

  The `fwrite` function returns the number of elements written.

  The behavior of `fwrite` is undefined if the values in *data* are too large to fit in the specified precision.

20.2.14 Other I/O Functions

`fgets (`*file*`, len)`

Read '`len`' characters from a file.

To flush output to a stream, use the function `fflush (`*file*`)`. 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`.

Three functions are available for setting and determining the position of the file pointer for a given file.

The position of the file pointer (as the number of characters from the beginning of the file) can be obtained using the the function `ftell (`*file*`)`.

To set the file pointer to any location within the file, use the function `fseek (`*file*`, offset, origin)`. The pointer is placed `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). 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`. See Section 6.1 [Predefined Constants], page 65.

The function `frewind (`*file*`)` moves the file pointer to the beginning of a file, returning 1 for success, and 0 if an error was encountered. It is equivalent to `fseek (`*file*`, 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);
```

The function `feof (`*file*`)` allows you to find out if an end-of-file condition has been encountered for a given file. 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.

Similarly, the function `ferror (`*file*`)` allows you to find out if an error condition has been encountered for a given file. 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.

The function `kbhit` may be usd to read a single keystroke from the keyboard. For example,

```
x = kbhit ();
```

will set x to the next character typed at the keyboard, without requiring a carriage return to be typed.

Finally, it is often useful to know exactly which files have been opened, and whether they are open for reading, writing, or both. The command `freport` prints this information for all open files. For example,

```
octave:13> freport

 number  mode  name

      0     r  stdin
      1     w  stdout
      2     w  stderr
      3     r  myfile
```

21 Special Matrices

Octave provides a number of functions for creating special matrix forms. In nearly all cases, it is best to use the built-in functions for this purpose than to try to use other tricks to achieve the same effect.

21.1 Special Utility Matrices

The function `eye` returns 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 matrix or vector argument, `eye` returns an identity matrix with the same dimensions as the given argument.

For example,

```
eye (3)
```

creates an identity matrix with three rows and three columns,

```
eye (5, 8)
```

creates an identity matrix with five rows and eight columns, and

```
eye ([13, 21; 34, 55])
```

creates an identity matrix with two rows and two columns.

Normally, `eye` expects any scalar arguments you provide to be real and non-negative. The variables `ok_to_lose_imaginary_part` and `treat_neg_dim_as_zero` control the behavior of `eye` for complex and negative arguments. See Section 6.2 [User Preferences], page 66. Any non-integer arguments are rounded to the nearest integer value.

There is an ambiguity when these functions are called with a single argument. You may have intended to create a matrix with the same dimensions as another variable, but ended up with something quite different, because the variable that you used as an argument was a scalar instead of a matrix.

For example, if you need to create an identity matrix with the same dimensions as another variable in your program, it is best to use code like this

```
eye (rows (a), columns (a))
```

instead of just

```
eye (a)
```

unless you know that the variable *a* will *always* be a matrix.

The functions `ones`, `zeros`, and `rand` all work like `eye`, except that they fill the resulting matrix with all ones, all zeros, or a set of random values.

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 `rand` function also takes some additional arguments that allow you to control its behavior. For example, the function call

```
rand ("normal")
```

causes the sequence of numbers to be normally distributed. You may also use an argument of `"uniform"` to select a uniform distribution. To find out what the current distribution is, use an argument of `"dist"`.

Normally, `rand` obtains the seed 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. For example, the function call

```
rand ("seed", 13)
```

sets the seed to the number 13. To see what the current seed is, use the argument `"seed"`.

If it is invoked without arguments, `rand` returns a single element of a random sequence.

The `rand` function uses 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.

To create a diagonal matrix with vector v on diagonal k, use the function diag (v, 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,

```
octave:13> diag ([1, 2, 3], 1)
ans =

  0  1  0  0
  0  0  2  0
  0  0  0  3
  0  0  0  0
```

The functions `linspace` and `logspace` make it very easy to create vectors with evenly or logarithmically spaced elements. For example,

`linspace (`*base*`, `*limit*`, `*n*`)`

creates a vector with n (n greater than 2) linearly spaced elements between *base* and *limit*. 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 function `logspace` is similar to `linspace` except that the values are logarithmically spaced.

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.

21.2 Famous Matrices

The following functions return famous matrix forms.

`hadamard (k)`
: Return the Hadamard matrix of order n = 2^k.

`hankel (c, r)`
: 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, the last row is taken to be the same as the first column.

  A Hankel matrix formed from an m-vector c, and an n-vector r, has the elements

$$H(i,j) = \begin{cases} c_{i+j-1}, & i+j-1 \le m; \\ r_{i+j-m}, & \text{otherwise}. \end{cases}$$

`hilb (n)`
: Return the Hilbert matrix of order n. The i, j element of a Hilbert matrix is defined as

$$H(i,j) = \frac{1}{(i+j-1)}$$

`invhilb (n)`
: Return the inverse of a Hilbert matrix of order n. This is exact. Compare 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.

`toeplitz (c, r)`
: 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

$$\begin{bmatrix} c_0 & r_1 & r_2 & \ldots & r_n \\ c_1 & c_0 & r_1 & & c_{n-1} \\ c_2 & c_1 & c_0 & & c_{n-2} \\ \vdots & & & & \vdots \\ c_n & c_{n-1} & c_{n-2} & \ldots & c_0 \end{bmatrix}.$$

`vander (c)`
: Return the Vandermonde matrix whose next to last column is c.

  A Vandermonde matrix has the form

$$\begin{bmatrix} c_0^n & \ldots & c_0^2 & c_0 & 1 \\ c_1^n & \ldots & c_1^2 & c_1 & 1 \\ \vdots & & \vdots & \vdots & \vdots \\ c_n^n & \ldots & c_n^2 & c_n & 1 \end{bmatrix}.$$

22 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.

22.1 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.

Given a vector, the function `any` returns 1 if any element of the vector is nonzero.

For a matrix argument, `any` returns 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,

```
octave:13> any (eye (2, 4))
ans =

  1  1  0  0
```

To see if any of the elements of a matrix are nonzero, you can use a statement like

```
any (any (a))
```

For a matrix argument, `any` returns 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.

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 in a column of a matrix, are nonzero.

Since the comparison operators (see Section 3.11 [Comparison Ops], page 36) 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,

```
octave:13> all (all (rand (5) < 0.9))
ans = 0
```

tests a random 5 by 5 matrix to see if all of it's 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))`.

The functions `isinf`, `finite`, and `isnan` return 1 if their arguments are infinite, finite, or not a number, respectively, and return 0 otherwise. For matrix values, they all work on an element by element basis. For example, evaluating the expression

```
isinf ([1, 2; Inf, 4])
```

produces the matrix

```
ans =

  0  0
  1  0
```

The function `find` returns a vector of indices of nonzero elements of a matrix. 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,

```
octave:13> find (eye (2))
ans =

  1
  4
```

If two outputs are requested, `find` returns the row and column indices of nonzero elements of a matrix. For example,

```
octave:13> [i, j] = find (eye (2))
i =

  1
  2

j =

  1
  2
```

If three outputs are requested, `find` also returns the nonzero values in a vector.

22.2 Rearranging Matrices

The function `fliplr` reverses the order of the columns in a matrix, and `flipud` reverses the order of the rows. For example,

```
octave:13> fliplr ([1, 2; 3, 4])
ans =

  2  1
  4  3

octave:13> flipud ([1, 2; 3, 4])
ans =

  3  4
  1  2
```

The function `rot90 (a, n)` rotates a matrix 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)
ans =

  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)
```

The function `reshape (a, m, n)` returns a matrix with m rows and n columns whose elements are taken from the matrix a. To decide how to order the elements, Octave pretends that the elements of a matrix are stored in column-major order (like Fortran arrays are stored).

For example,

```
octave:13> reshape ([1, 2, 3, 4], 2, 2)
ans =

  1  3
  2  4
```

If the variable `do_fortran_indexing` is `"true"`, the `reshape` function is equivalent to

```
retval = zeros (m, n);
retval (:) = a;
```

but it is somewhat less cryptic to use `reshape` instead of the colon operator. Note that the total number of elements in the original matrix must match the total number of elements in the new matrix.

The function '`sort`' can be used to arrange the elements of a vector in increasing order. For matrices, `sort` orders the elements in each column.

For example,

```
octave:13> sort (rand (4))
ans =

  0.065359  0.039391  0.376076  0.384298
  0.111486  0.140872  0.418035  0.824459
  0.269991  0.274446  0.421374  0.938918
  0.580030  0.975784  0.562145  0.954964
```

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 =
```

```
  0.051724  0.485904  0.253614  0.348008
  0.391608  0.526686  0.536952  0.600317
  0.733534  0.545522  0.691719  0.636974
  0.986353  0.976130  0.868598  0.713884

i =

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

These values may be used to recover the original matrix from the sorted version. For example,

The `sort` function does not allow sort keys to be specified, so it can't be used to order the rows of a matrix according to the values of the elements in various columns[1] 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 third column.

```
octave:13> a = rand (4)
a =

  0.080606  0.453558  0.835597  0.437013
  0.277233  0.625541  0.447317  0.952203
  0.569785  0.528797  0.319433  0.747698
  0.385467  0.124427  0.883673  0.226632

octave:14> [s, i] = sort (a (:, 3));
octave:15> a (i, :)
ans =

  0.569785  0.528797  0.319433  0.747698
  0.277233  0.625541  0.447317  0.952203
  0.080606  0.453558  0.835597  0.437013
  0.385467  0.124427  0.883673  0.226632
```

The functions `triu (`*a*`, `*k*`)` and `tril (`*a*`, `*k*`)` extract the upper or lower triangular part of the matrix *a*, and set 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,

```
octave:13> tril (rand (4), 1)
ans =

  0.00000  0.00000  0.00000  0.00000
  0.09012  0.00000  0.00000  0.00000
  0.01215  0.34768  0.00000  0.00000
  0.00302  0.69518  0.91940  0.00000
```

forms a lower triangular matrix from a random 4 by 4 matrix, omitting the main diagonal, and

```
octave:13> tril (rand (4), -1)
ans =

  0.06170  0.51396  0.00000  0.00000
  0.96199  0.11986  0.35714  0.00000
  0.16185  0.61442  0.79343  0.52029
  0.68016  0.48835  0.63609  0.72113
```

forms a lower triangular matrix from a random 4 by 4 matrix, including the main diagonal and the first super-diagonal.

23 String Functions

Octave currently has a limited ability to work with strings.

The function `strcmp (s1, s2)` compares two strings, returning 1 if they are the same, and 0 otherwise.

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

The functions `int2str` and `num2str` convert a numeric argument to a string. These functions are not very flexible, but are provided for compatibility with MATLAB. For better control over the results, use `sprintf` (see Section 20.2.2 [Formatted Output], page 128).

The function `setstr` can be used to convert a vector to a string. Each element of the vector is converted to the corresponding ASCII character. For example,

```
setstr ([97, 98, 99])
```

creates the string

```
abc
```

The function `undo_string_escapes (string)` 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. See Section 3.1.2 [String Constants], page 24, for a description of string escapes.

24 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.

24.1 Timing Utilities

The function `clock` returns a vector containing the current year, month (1-12), day (1-31), hour (0-23), minute (0-59) and second (0-60). For example,

```
octave:13> clock
ans =

  1993     8    20     4    56     1
```

The function clock is more accurate on systems that have the `gettimeofday` function.

To get the date as a character string in the form DD-MMM-YY, use the command `date`. For example,

```
octave:13> date
ans = 20-Aug-93
```

Octave also has functions for computing time intervals and CPU time used. The functions `tic` and `toc` can be used to 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`.

The function `etime` provides another way to get elapsed wall-clock time by returning 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.

The function `cputime` allows you to obtain information about the amount of CPU time your Octave session is using. For example,

```
[total, user, system] = cputime ();
```

returns 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.

Finally, Octave's function `is_leap_year` returns 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,

```
octave:13> is_leap_year (2000)
ans = 1
```

Contrary to what many people who post misinformation to Usenet apparently believe, Octave knows that the year 2000 will be a leap year.

24.2 Interacting with the OS

You can execute any shell command using the function `system (`*cmd*`, `*flag*`)`. The second argument is optional. If it is present, the output of the command is returned by `system` as a string. If it is not supplied, any output from the command is printed, with the standard output filtered through the pager. For example,

```
users = system ("finger", 1)
```

places the output of the command `finger` in the variable `users`.

If you want to execute a shell command and have it behave as if it were typed directly from the shell prompt, you may need to specify extra arguments for the command. For example, to get `bash` to behave as an interactive shell, you can type

```
system ("bash -i >/dev/tty");
```

The first argument, '`-i`', tells `bash` to behave as an interactive shell, and the redirection of the standard output stream prevents any output produced by `bash` from being sent back to Octave, where it would be buffered until Octave displays another prompt.

The `system` function can return two values. The first is any output from the command that was written to the standard output stream, and the second is the output status of the command. For example,

```
[output, status] = system ("echo foo; exit 2");
```

will set the variable `output` to the string '`foo`', and the variable `status` to the integer '`2`'.

The name `shell_cmd` exists for compatibility with earlier versions of Octave.

You can find the values of environment variables using the function `getenv`. For example,

```
getenv ("PATH")
```

returns a string containing the value of your path.

The functions `clc`, and `home` clear your terminal screen and move the cursor to the upper left corner.

You can change the current working directory using the `cd` command. Tilde expansion is performed on the path. 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.

The name `chdir` is an alias for `cd`.

The command `pwd` prints the current working directory.

The functions `dir` and `ls` list directory contents. For example,

```
octave:13> 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.

24.3 System Information

If possible, `computer` prints a string of the form *cpu-vendor-os* that identifies the kind of computer Octave is running on. For example,

```
octave:13> computer
sparc-sun-sunos4.1.2
```

The function `isieee` returns 1 if your computer claims to conform to the IEEE standard for floating point calculations.

The function `version` returns Octave's version number as a string. This is also the value of the built-in variable `OCTAVE_VERSION`. See Chapter 6 [Built-in Variables], page 65.

24.4 Other Functions

The function `pause` allows you to suspend the execution of a 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...\n"), pause (5), clc
```

25 Command History Functions

Octave provides three functions for viewing, editing, and re-running chunks of commands from the history list.

The function `history` displays a list of commands that you have executed. It also allows you to write the current history to a file for safe keeping, and to replace the history list with the commands stored in a named file. Valid arguments are:

`-w file` Write the current history to the named file. If the name is omitted, use the default history file (normally '`~/.octave_hist`').

`-r file` Read the named 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`'.

The function `edit_history` allows you to edit a block of commands from the history list using the editor named by the environment variable `EDITOR`, or the default editor (normally `vi`). 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.

The command `run_history` is like `edit_history`, except that the editor is not invoked, and the commands are simply executed as they appear in the history list.

The `diary` command allows you to create a list of all commands *and* the output they produce, mixed together just as you see them on your terminal.

For example, the command

```
diary on
```

tells Octave to start recording your session in a file called '`diary`' in your current working directory. To give Octave the name of the file write to, use the a command like

```
diary my-diary.txt
```

Then Octave will write all of your commands to the file '`my-diary.txt`'.

To stop recording your session, use the command

```
diary off
```

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

26 Help

Octave's `help` command can be used to print brief usage-style messages, or to display information directly from an on-line version of the printed manual, using the GNU Info browser. If invoked without any arguments, `help` prints a list of all the available operators, functions, and built-in variables. If the first argument is `-i`, the `help` command searches the index of the on-line version of this manual for the given topics.

For example, the command

```
help help
```

prints a short message describing the `help` command, and

```
help -i help
```

starts the GNU Info browser at this node in the on-line version of the manual.

See Appendix D [Using Info], page 189, for complete details about how to use the GNU Info browser to read the on-line version of the manual.

27 Programming Utilities

27.1 Evaluating Strings as Commands

It is often useful to evaluate a string as if it were an Octave program, or use a string as the name of a function to call. These functions are necessary in order to evaluate commands that are not known until run time, or to write functions that will need to call user-supplied functions.

The function `eval (`*command*`)` parses *command* and evaluates it as if it were an Octave program, returning the last value computed. The *command* is evaluated in the current context, so any results remain available after `eval` returns. For example,

```
octave:13> a
error: ‘a’ undefined
octave:14> eval ("a = 13")
a = 13
ans = 13
octave:15> a
a = 13
```

In this case, two values are printed: one for the expression that was evaluated, and one for the value returned from `eval`. Just as with any other expression, you can turn printing off by ending the expression in a semicolon. For example,

```
octave:13> a
error: ‘a’ undefined
octave:14> eval ("a = 13;")
ans = 13
octave:15> a
a = 13
```

The function `feval (`*name*`, ...)` can be used to evaluate the function named *name*. Any arguments after the first are passed on to the named function. For example,

```
octave:12> feval ("acos", -1)
ans = 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.

For example, here is a simple-minded function for finding the root of a function of one variable:

```
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.

27.2 Miscellaneous Utilities

The following functions allow you to determine the size of a variable or expression, find out whether a variable exists, print error messages, or delete variable names from the symbol table.

`columns (a)`
: Return the number of columns of *a*.

`rows (a)`
: Return the number of rows of *a*.

`length (a)`
: Return the number of rows of *a* or the number of columns of *a*, whichever is larger.

`size (a [, n])`
: Return the number rows and columns of *a*.

With one input argument and one output argument, the result is returned in a 2 element row vector. If there are two output arguments, the number of rows is assigned to the first, and the number of columns to the second. For example,

```
octave:13> size ([1, 2; 3, 4; 5, 6])
ans =

  3  2

octave:14> [nr, nc] = size ([1, 2; 3, 4; 5, 6])
nr = 3

nc = 2
```

If given a second argument of either 1 or 2, `size` will return only the row or column dimension. For example

```
octave:15> size ([1, 2; 3, 4; 5, 6], 2)
ans = 2
```

returns the number of columns in the given matrix.

`is_global` (*a*)
: Return 1 if *a* is globally visible. Otherwise, return 0.

`is_matrix` (*a*)
: Return 1 if *a* is a matrix. Otherwise, return 0.

`is_vector` (*a*)
: Return 1 if *a* is a vector. Otherwise, return 0.

`is_scalar` (*a*)
: Return 1 if *a* is a scalar. Otherwise, return 0.

`is_square` (*x*)
: If *x* is a square matrix, then return the dimension of *x*. Otherwise, return 0.

`is_symmetric` (*x*, *tol*)
: 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.

`isstr` (*a*) Return 1 if *a* is a string. Otherwise, return 0.

`isempty` (*a*)
: 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.

`clear` *pattern* ...
: 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.

This command may not be used within a function body.

`who` *options pattern ...*

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 variables and functions. This includes all currently compiled function files, but does not include all function files that are in the `LOADPATH`.

`-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`.

`exist` (*name*)

Return 1 if the name exists as a variable, and 2 if the name (after appending '`.m`') is a function file in the path. Otherwise, return 0.

`error` (*msg*)

Print the message *msg*, prefixed by the string '`error: `', 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.

If *msg* does not end with a new line character, Octave will print a traceback of all the function calls leading to the error. For example,

```
function f () g () end
function g () h () end
function h () nargin == 1 || error ("nargin != 1"); end
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'
```

produces a list of messages that can help you to quickly locate the exact location of the error.

If *msg* ends in a new line character, Octave will only print *msg* and 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
```

`warning` (*msg*)

Print the message *msg* prefixed by the string '`warning: `'.

`usage` (*msg*)

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.

`perror` (*name*, *num*)

Print the error message for function *name* corresponding to the error number *num*. This function is intended to be used to print useful error messages for those functions that return numeric error codes.

`menu` (*title*, *opt1*, ...)

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.

`document` *symbol text*

Set the documentation string for *symbol* to *text*.

`file_in_path` (*path*, *file*)

Return the absolute name 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 the built-in variable `LOADPATH`.

If the file cannot be found in the path, an empty matrix is returned. For example,

```
octave:13> file_in_path (LOADPATH, "nargchk.m")
ans = /usr/local/lib/octave/1.1.0/m/general/nargchk.m
```

`nargchk` (*nargin˙min*, *nargin˙max*, *n*)

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.

`octave_tmp_file_name`

Return a unique temporary file name.

Since the named file is not opened, by `octave_tmp_file_name`, it is possible (though relatively unlikely) that it will not be available by the time your program attempts to open it.

`type` *name* ...

Display the definition of each *name* that refers to a function.

Currently, Octave can only display functions that can be compiled cleanly, because it uses its internal representation of the function to recreate the program text.

Comments are not displayed because Octave's currently discards them as it converts the text of a function file to its internal representation. This problem may be fixed in a future release.

`which` *name* ...

Display the type of each *name*. If *name* is defined from a function file, the full name of the file is also displayed.

28 Amusements

Octave cannot promise that you will actually win the lotto, but it can pick your numbers for you. The function `texas_lotto` will select six numbers between 1 and 50.

The function `list_primes (`*n*`)` uses a brute-force algorithm to compute the first *n* primes.

Other amusing functions include `casesen`, `flops`, `sombrero`, `exit`, and `quit`.

Appendix A Installing Octave

Here is the procedure for installing Octave from scratch on a Unix system. For instructions on how to install the binary distributions of Octave, see Section A.2 [Binary Distributions], page 171.

- 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 Fortran compiler is available.

  `--enable-dld`
  : Use DLD to make Octave capable of dynamically linking externally compiled functions. This only works on systems that have a working port of DLD.

  `--enable-lite-kernel`
  : Compile smaller kernel. This currently requires DLD so that Octave can load functions at run time that are not loaded at compile time.

  `--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.

  For version 1.1.1, you must have the GNU C++ compiler (gcc) version 2.6.3 or later to compile Octave. You will also need version 2.6.1 of the GNU C++ class library (libg++). If you plan to modify the parser you will also need GNU bison and fles. 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 libg++, gnuplot, bison, flex, and Texinfo are all available from many anonymous ftp archives, including ftp.che.utexas.edu, ftp.uu.net, prep.ai.mit.edu, and wuarchive.wustl.edu.

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 research.att.com.

On an otherwise idle SPARCstation II, it will take somewhere between 60 and 90 minutes to compile everything, depending on whether you are compiling the Fortran libraries with f2c or using the Fortran compiler directly. You will need about 50 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 B [Trouble], page 173, 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:

'*prefix*`/bin`'
: Octave and other binaries that people will want to run directly.

'*prefix*`/lib`'
: Libraries like libcruft.a and liboctave.a.

'*prefix*`/include/octave`'
: Include files distributed with Octave.

'*prefix*`/man/man1`'
: Unix-style man pages describing Octave.

'*prefix*`/info`'
: Info files describing Octave.

'*prefix*`/lib/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/`*host_type*'
: Executables to be run by Octave rather than the user.

'*prefix*`/lib/octave/`*version*`/oct/`*host_type*'
: Object files that will be dynamically loaded.

'*prefix*/lib/octave/*version*/imagelib'
    Image files that are distributed with Octave.

where *prefix* defaults to '/usr/local', *version* stands for the current version number of the interpreter, and *host_type* is the type of computer on which Octave is installed (for example, 'i486-unknown-gnu').

A.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 AIX (and possibly other) systems, GCC 2.6.2 generates invalid assembly code when compiling some parts of Octave. On AIX systems, it is possible to get a working binary by not using the compiler flag '-fno-implicit-templates'. You can specify this as an option to make by using a command like

  ```
  make NO_IMPLICIT_TEMPLATES=
  ```

- You may need to edit some files in the gcc include subdirectory to add prototypes for functions there. For example, Ultrix 4.2 needs proper declarations for the `signal()` 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 `signal()`.

  The gcc fixincludes/fixproto script should probably fix this when gcc installs its modified set of header files, but I don't think that's been done yet.

- There is a bug with the makeinfo program that is distributed with texinfo-3.1 that causes the indices in Octave's on-line manual to be generated incorrectly. If you need to recreate the on-line documentation, you should get the makeinfo program that is distributed with texinfo-3.1 and apply the patch for makeinfo that is distributed with Octave. See the file MAKEINFO.PATCH for more details.

- If you don't have NPSOL but you still want to be able to solve NLPs, or if you don't have QPSOL but you still want to solve QPs, you'll need to find replacements or order them from Stanford. If you know of a freely redistributable replacement, please let us know—we might be interested in distributing it with Octave.

  You can get more information about NPSOL and QPSOL from

  Stanford University
  Office of Technology Licensing
  857 Serra Street
  Stanford CA 94305-6225

Tel: (415) 723-0651
Fax: (415) 725-7295

Octave may soon support FSQP, an NLP solver from Andre Tits (andre@src.umd.edu) of the University of Maryland. FSQP is available free of charge to academic sites, but can not be redistributed to third parties.

- 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 kludge 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!

- With g++ 2.6.3 (and possibly other 2.6.x versions) on some Intel x86 systems, compiling 'Array-d.cc' fails with the messages like

  ```
  as: /tmp/cc005254.s:4057: Local symbol LBB103 never defined.
  as: /tmp/cc005254.s:4057: Local symbol LBE103 never defined.
  ```

  A possible workaround for this is to compile without -g.

- 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 'sysdep.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.

A.2 Binary Distributions

This section contains instructions for creating and installing a binary distribution.

A.2.1 Installing Octave from a Binary Distribution

- To install Octave from a binary distribution, execute the command

  ```
  sh ./doinstall.sh
  ```

  in the top level directory of the distribution.

  Binary distributions are normally compiled assuming that Octave will be installed in the following subdirectories of '/usr/local'.

  'bin' Octave and other binaries that people will want to run directly.

  'man/man1'
  : Unix-style man pages describing Octave.

  'info' Info files describing Octave.

  'lib/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.

  'lib/octave/*version*/exec/*host_type*'
  : Executables to be run by Octave rather than the user.

  'lib/octave/*version*/imagelib'
  : Image files that are distributed with Octave.

where *version* stands for the current version number of the interpreter, and *host_type* is the type of computer on which Octave is installed (for example, '`i486-unknown-gnu`').

If these directories don't exist, the script '`doinstall.sh`' will create them for you.

If this is possible for you to install Octave in '`/usr/local`', or if you would prefer to install it in a different directory, you can specify the name of the top level directory as an argument to the doinstall.sh script. For example:

```
sh ./doinstall.sh /some/other/directory
```

Octave will then be installed in subdirectories of the directory '`/some/other/directory`'

A.2.2 Creating a Binary Distribution

Here is how to build a binary distribution for others.

- Build Octave in the same directory as the source. This is required since the '`binary-dist`' targets in the Makefiles will not work if you compile outside the source tree.
- Use '`CFLAGS=-O CXXFLAGS=-O LDFLAGS=`' as arguments for Make because most people who get the binary distributions are probably not going to be interested in debugging Octave.
- Type '`make binary-dist`'. This will build everything and then pack it up for distribution.

Appendix B Known Causes of Trouble with Octave

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.

B.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.
- Control-C doesn't work properly in the pager on DEC Alpha systems running OSF/1 3.0.

  This appears to be a bug in the OSF/1 3.0 Bourne shell. The problem doesn't appear on systems running OSF/1 1.3.
- If you get messages like

  ```
  Input line too long
  ```

  when trying to plot many lines on one graph, you have probably generated a plot command that is too larger for `gnuplot`'s fixed-length buffer for commands. Splitting up the plot command doesn't help because replot is implemented in gnuplot by simply appending the new plotting commands to the old command line and then evaluating it again.

  You can demonstrate this 'feature' by running gnuplot and doing something like

  ```
  plot sin (x), sin (x), sin (x), ... lots more ..., sin (x)
  ```

  and then

  ```
  replot sin (x), sin (x), sin (x), ... lots more ..., sin (x)
  ```

  after repeating the replot command a few times, gnuplot will give you an error.

  Also, it doesn't help to use backslashes to enter a plot command over several lines, because the limit is on the overall command line length, once the backslashed lines are all pasted together.

  Because of this, Octave tries to use as little of the command-line length as possible by using the shortest possible abbreviations for all the plot commands and options. Unfortunately, the length of the temporary file names is probably what is taking up the most space on the command line.

You can buy a little bit of command line space by setting the environment variable TMPDIR to be "." before starting Octave, or you can increase the maximum command line length in gnuplot by changing the following limits in the file plot.h in the gnuplot distribution and recompiling gnuplot.

```
#define MAX_LINE_LEN 32768  /* originally 1024 */
#define MAX_TOKENS 8192     /* originally 400 */
```

Of course, this doesn't really fix the problem, but it does make it much less likely that you will run into trouble unless you are putting a very large number of lines on a given plot.

- String handling could use some work.

A list of ideas for future enhancements is distributed with Octave. See the file 'PROJECTS' in the top level directory in the source distribution.

B.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 B [Trouble], page 173. 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.

B.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.

B.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@che.utexas.edu
```

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 '`bug-octave`'.

As a last resort, send bug reports on paper to:

```
Octave Bugs c/o John W. Eaton
Department of Chemical Engineering
The University of Texas at Austin
Austin, Texas 78712
```

B.5 How to Report Bugs

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 probalby 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.

B.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.

B.7 How To Get Help with Octave

If you need help installing, using or changing Octave, the mailing list

```
help-octave@che.utexas.edu
```

exists for the discussion of Octave matters related to using, installing, and porting Octave. If you would like to join the discussion, please send a short note to

```
help-octave-request@che.utexas.edu
             ^^^^^^^
```

Please do not send requests to be added or removed from the the mailing list, or other administrative trivia to the list itself.

Appendix C Command Line Editing

This text describes GNU's command line editing interface. It is relatively old and may not be entirely correct now. Please send a message to `bug-octave@che.utexas.edu` if you find any errors. See Section B.2 [Reporting Bugs], page 174, for more information about how to report bugs.

C.1 Introduction to Line Editing

The following paragraphs describe the notation we use to represent keystrokes.

The text `C-K` is read as 'Control-K' and describes the character produced when the Control key is depressed and the `K` key is struck.

The text `M-K` is read as 'Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the `K` key is struck. If you do not have a meta key, the identical keystroke can be generated by typing `ESC` *first*, and then typing `K`. Either process is known as *metafying* the `K` key.

The text `M-C-K` is read as 'Meta-Control-k' and describes the character produced by *metafying* `C-K`.

In addition, several keys have their own names. Specifically, `DEL`, `ESC`, `LFD`, `SPC`, `RET`, and `TAB` all stand for themselves when seen in this text, or in an init file (see Section C.7 [Readline Init File], page 181, for more info).

C.2 Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press `RETURN`. You do not have to be at the end of the line to press `RETURN`; the entire line is accepted regardless of the location of the cursor within the line.

C.3 Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use `DEL` to back up, and delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can type `C-B` to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with `C-F`.

When you add text in the middle of a line, you will notice that characters to the right of the cursor get 'pushed over' to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor get 'pulled back' to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

`C-B` Move back one character.

`C-F` Move forward one character.

`DEL` Delete the character to the left of the cursor.

`C-D` Delete the character underneath the cursor.

Printing characters
Insert itself into the line at the cursor.

`C-_` Undo the last thing that you did. You can undo all the way back to an empty line.

C.4 Readline Movement Commands

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to `C-B`, `C-F`, `C-D`, and `DEL`. Here are some commands for moving more rapidly about the line.

`C-A` Move to the start of the line.

`C-E` Move to the end of the line.

`M-F` Move forward a word.

`M-B` Move backward a word.

`C-L` Clear the screen, reprinting the current line at the top.

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.

C.5 Readline Killing Commands

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-DEL` 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-DEL` because the word boundaries differ.

And, here is how to *yank* the text back into the line. Yanking is

`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.

C.6 Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type `M-- C-K`.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first 'digit' you type is a minus sign (`-`), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the `C-D` command an argument of 10, you could type `M-1 0 C-D`.

C.7 Readline Init File

Although the Readline library comes with a set of Emacs-like keybindings, it is possible that you would like to use a different set of keybindings. You can customize programs that use Readline by putting commands in an *init* file in your home directory. The name of this file is '`~/.inputrc`'.

When a program which uses the Readline library starts up, the '`~/.inputrc`' file is read, and the keybindings are set.

In addition, the `C-X C-R` command re-reads this init file, thus incorporating any changes that you might have made to it.

C.8 Readline Init Syntax

There are only four constructs allowed in the '`~/.inputrc`' file:

Variable Settings

You can change the state of a few variables in Readline. You do this by using the `set` command within the init file. Here is how you would specify that you wish to use Vi line editing commands:

```
set editing-mode vi
```

Right now, there are only a few variables which can be set; so few in fact, that we just iterate them here:

`editing-mode`

The `editing-mode` variable controls which editing mode you are using. By default, GNU Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can either be set to `emacs` or `vi`.

`horizontal-scroll-mode`

This variable can either be set to `On` or `Off`. Setting it to `On` means that the text of the lines that you edit will scroll horizontally on a single screen line when they are larger than the width of the screen, instead of wrapping onto a new screen line. By default, this variable is set to `Off`.

`mark-modified-lines`

This variable when set to `On`, says to display an asterisk ('`*`') at the starts of history lines which have been modified. This variable is off by default.

`prefer-visible-bell`

If this variable is set to `On` it means to use a visible bell if one is available, rather than simply ringing the terminal bell. By default, the value is `Off`.

Key Bindings

The syntax for controlling keybindings in the '`~/.inputrc`' file is simple. First you have to know the *name* of the command that you want to change. The following pages contain tables of the command name, the default keybinding, and a short description of what the command does.

Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the '`~/.inputrc`' file. The name of the key can be expressed in different ways, depending on which is most comfortable for you.

keyname: *function-name* or *macro*

keyname is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: ">&output"
```

In the above example, `C-U` is bound to the function `universal-argument`, and `C-O` is bound to run the macro expressed on the right hand side (that is, to insert the text '`>&output`' into the line).

"*keyseq*": *function-name* or *macro*

keyseq differs from *keyname* above in that strings denoting an entire key sequence can be specified. Simply place the key sequence in double quotes. GNU Emacs style key escapes can be used, as in the following example:

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In the above example, `C-U` is bound to the function `universal-argument` (just as it was in the first example), `C-X C-R` is bound to the function `re-read-init-file`, and `ESC [ 1 1 ~` is bound to insert the text '`Function Key 1`'.

C.8.1 Commands For Moving

`beginning-of-line (C-A)`
: Move to the start of the current line.

`end-of-line (C-E)`
: Move to the end of the line.

`forward-char (C-F)`
: Move forward a character.

`backward-char (C-B)`
: Move back a character.

`forward-word (M-F)`
: Move forward to the end of the next word.

`backward-word (M-B)`
: Move back to the start of this, or the previous, word.

`clear-screen (C-L)`
: Clear the screen leaving the current line at the top of the screen.

C.8.2 Commands For Manipulating The History

`accept-line (Newline, Return)`
: 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.

`previous-history (C-P)`
: Move 'up' through the history list.

`next-history (C-N)`
: Move 'down' through the history list.

`beginning-of-history (M-<)`
: Move to the first line in the history.

`end-of-history (M->)`
: Move to the end of the input history, i.e., the line you are entering!

`reverse-search-history (C-R)`
: Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

`forward-search-history (C-S)`
: Search forward starting at the current line and moving 'down' through the the history as necessary.

C.8.3 Commands For Changing Text

`delete-char (C-D)`
: Delete the character under the cursor. If the cursor is at the beginning of the line, and there are no characters in the line, and the last character typed was not `C-D`, then return EOF.

`backward-delete-char (Rubout)`
: Delete the character behind the cursor. A numeric arg says to kill the characters instead of deleting them.

`quoted-insert (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.

`tab-insert (M-TAB)`
: Insert a tab character.

`self-insert (a, b, A, 1, !, ...)`
: Insert yourself.

`transpose-chars (C-T)`
: Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative args don't work.

`transpose-words (M-T)`
: Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

`upcase-word (M-U)`
Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move point.

`downcase-word (M-L)`
Lowercase the current (or following) word. With a negative argument, do the previous word, but do not move point.

`capitalize-word (M-C)`
Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move point.

C.8.4 Killing And Yanking

`kill-line (C-K)`
Kill the text from the current cursor position to the end of the line.

`backward-kill-line ()`
Kill backward to the beginning of the line. This is normally unbound.

`kill-word (M-D)`
Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`backward-kill-word (M-DEL)`
Kill the word behind the cursor.

`unix-line-discard (C-U)`
Do what `C-U` used to do in Unix line input. We save the killed text on the kill-ring, though.

`unix-word-rubout (C-W)`
Do what `C-W` used to do in Unix line input. The killed text is saved on the kill-ring. This is different than backward-kill-word because the word boundaries differ.

`yank (C-Y)`
Yank the top of the kill ring into the buffer at point.

`yank-pop (M-Y)`
Rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

C.8.5 Specifying Numeric Arguments

`digit-argument (M-0, M-1, ... M--)`
Add this digit to the argument already accumulating, or start a new argument. `M--` starts a negative argument.

`universal-argument ()`
: Do what `C-U` does in emacs. By default, this is not bound.

C.8.6 Letting Readline Type For You

`complete (TAB)`
: Attempt to do completion on the text before point. This is implementation defined. Generally, if you are typing a file name argument, you can do file name completion; if you are typing a command, you can do command completion, if you are typing in a symbol to GDB, you can do symbol name completion, if you are typing in a variable to Bash, you can do variable name completion...

`possible-completions (M-?)`
: List the possible completions of the text before point.

C.8.7 Some Miscellaneous Commands

`re-read-init-file (C-X C-R)`
: Read in the contents of your '`~/.inputrc`' file, and incorporate any bindings found there.

`abort (C-G)`
: Ding! Stops things.

`do-uppercase-version (M-A, M-B, ...)`
: Run the command that is bound to your uppercase brother.

`prefix-meta (ESC)`
: Make the next character that you type be metafied. This is for people without a meta key. Typing `ESC F` is equivalent to typing `M-F`.

`undo (C-_)`
: Incremental undo, separately remembered for each line.

`revert-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.

C.9 Readline Vi Mode

While the Readline library does not have a full set of Vi editing functions, it does contain enough to allow simple editing of the line.

In order to switch interactively between Emacs and Vi editing modes, use the command `M-C-J` (toggle-editing-mode).

When you enter a line in Vi mode, you are already placed in 'insertion' mode, as if you had typed an 'i'. Pressing `ESC` switches you into 'edit' mode, where you can edit the text of the line with the standard Vi movement keys, move to previous history lines with 'k', and following lines with 'j', and so forth.

Appendix D Using Info

Info is a program which is used to view info files on an ASCII terminal. *info files* are the result of processing texinfo files with the program `makeinfo` or with the Emacs command `M-x texinfo-format-buffer`. Finally, *texinfo* is a documentation language which allows a printed manual and on-line documentation (an info file) to be produced from a single source file.

D.1 Moving the Cursor

Many people find that reading screens of text page by page is made easier when one is able to indicate particular pieces of text with some kind of pointing device. Since this is the case, GNU Info (both the Emacs and standalone versions) have several commands which allow you to move the cursor about the screen. The notation used in this manual to describe keystrokes is identical to the notation used within the Emacs manual, and the GNU Readline manual. See section "Character Conventions" in *the GNU Emacs Manual*, if you are unfamiliar with the notation.

The following table lists the basic cursor movement commands in Info. Each entry consists of the key sequence you should type to execute the cursor movement, the `M-x`[1] command name (displayed in parentheses), and a short description of what the command does. All of the cursor motion commands can take an *numeric* argument (see Section D.8 [Other Info Commands], page 199), to find out how to supply them. With a numeric argument, the motion commands are simply executed that many times; for example, a numeric argument of 4 given to `next-line` causes the cursor to move down 4 lines. With a negative numeric argument, the motion is reversed; an argument of -4 given to the `next-line` command would cause the cursor to move *up* 4 lines.

`C-n` (`next-line`)
: Moves the cursor down to the next line.

`C-p` (`prev-line`)
: Move the cursor up to the previous line.

`C-a` (`beginning-of-line`)
: Move the cursor to the start of the current line.

`C-e` (`end-of-line`)
: Moves the cursor to the end of the current line.

`C-f` (`forward-char`)
: Move the cursor forward a character.

`C-b` (`backward-char`)
: Move the cursor backward a character.

---

[1] `M-x` is also a command; it invokes `execute-extended-command`. See section "Executing an extended command" in *the GNU Emacs Manual*, for more detailed information.

`M-f` (`forward-word`)
: Moves the cursor forward a word.

`M-b` (`backward-word`)
: Moves the cursor backward a word.

`M-<` (`beginning-of-node`)
`b`
: Moves the cursor to the start of the current node.

`M->` (`end-of-node`)
: Moves the cursor to the end of the current node.

`M-r` (`move-to-window-line`)
: Moves the cursor to a specific line of the window. Without a numeric argument, `M-r` moves the cursor to the start of the line in the center of the window. With a numeric argument of *n*, `M-r` moves the cursor to the start of the *n*th line in the window.

D.2 Moving Text Within a Window

Sometimes you are looking at a screenful of text, and only part of the current paragraph you are reading is visible on the screen. The commands detailed in this section are used to shift which part of the current node is visible on the screen.

`SPC` (`scroll-forward`)
`C-v`
: Shift the text in this window up. That is, show more of the node which is currently below the bottom of the window. With a numeric argument, show that many more lines at the bottom of the window; a numeric argument of 4 would shift all of the text in the window up 4 lines (discarding the top 4 lines), and show you four new lines at the bottom of the window. Without a numeric argument, `SPC` takes the bottom two lines of the window and places them at the top of the window, redisplaying almost a completely new screenful of lines.

`DEL` (`scroll-backward`)
`M-v`
: Shift the text in this window down. The inverse of `scroll-forward`.

The `scroll-forward` and `scroll-backward` commands can also move forward and backward through the node structure of the file. If you press `SPC` while viewing the end of a node, or `DEL` while viewing the beginning of a node, what happens is controlled by the variable `scroll-behaviour`. See Section D.9 [Info Variables], page 201, for more information.

`C-l` (`redraw-display`)
: Redraw the display from scratch, or shift the line containing the cursor to a specified location. With no numeric argument, '`C-l`' clears the screen, and then redraws its entire contents. Given a numeric argument of *n*, the line containing the cursor is shifted so that it is on the *n*th line of the window.

`C-x w` (`toggle-wrap`)
: Toggles the state of line wrapping in the current window. Normally, lines which are longer than the screen width *wrap*, i.e., they are continued on the next line. Lines which wrap have a '`\`' appearing in the rightmost column of the screen. You can cause such lines to be terminated at the rightmost column by changing the state of line wrapping in the window with `C-x w`. When a line which needs more space than one screen width to display is displayed, a '`$`' appears in the rightmost column of the screen, and the remainder of the line is invisible.

D.3 Selecting a New Node

This section details the numerous Info commands which select a new node to view in the current window.

The most basic node commands are '`n`', '`p`', '`u`', and '`l`'.

When you are viewing a node, the top line of the node contains some Info *pointers* which describe where the next, previous, and up nodes are. Info uses this line to move about the node structure of the file when you use the following commands:

`n` (`next-node`)
: Selects the 'Next' node.

`p` (`prev-node`)
: Selects the 'Prev' node.

`u` (`up-node`)
: Selects the 'Up' node.

You can easily select a node that you have already viewed in this window by using the '`l`' command – this name stands for "last", and actually moves through the list of already visited nodes for this window. '`l`' with a negative numeric argument moves forward through the history of nodes for this window, so you can quickly step between two adjacent (in viewing history) nodes.

`l` (`history-node`)
: Selects the most recently selected node in this window.

Two additional commands make it easy to select the most commonly selected nodes; they are '`t`' and '`d`'.

`t` (`top-node`)
: Selects the node '`Top`' in the current info file.

`d` (`dir-node`)
: Selects the directory node (i.e., the node '`(dir)`').

Here are some other commands which immediately result in the selection of a different node in the current window:

`<` (`first-node`)
: Selects the first node which appears in this file. This node is most often '`Top`', but it doesn't have to be.

`>` (`last-node`)
: Selects the last node which appears in this file.

`]` (`global-next-node`)
: Moves forward or down through node structure. If the node that you are currently viewing has a '`Next`' pointer, that node is selected. Otherwise, if this node has a menu, the first menu item is selected. If there is no '`Next`' and no menu, the same process is tried with the '`Up`' node of this node.

`[` (`global-prev-node`)
: Moves backward or up through node structure. If the node that you are currently viewing has a '`Prev`' pointer, that node is selected. Otherwise, if the node has an '`Up`' pointer, that node is selected, and if it has a menu, the last item in the menu is selected.

You can get the same behavior as `global-next-node` and `global-prev-node` while simply scrolling through the file with `SPC` and `DEL`; See Section D.9 [Info Variables], page 201, for more information.

`g` (`goto-node`)
: Reads the name of a node and selects it. No completion is done while reading the node name, since the desired node may reside in a separate file. The node must be typed exactly as it appears in the info file. A file name may be included as with any node specification, for example

```
g(emacs)Buffers
```

finds the node '`Buffers`' in the info file '`emacs`'.

`C-x k` (`kill-node`)
: Kills a node. The node name is prompted for in the echo area, with a default of the current node. *Killing* a node means that Info tries hard to forget about it, removing it from the list of history nodes kept for the window where that node is found. Another node is selected in the window which contained the killed node.

`C-x C-f` (`view-file`)
: Reads the name of a file and selects the entire file. The command

`C-x C-f` *filename*

is equivalent to typing

`g(`*filename*`)*`

`C-x C-b` (`list-visited-nodes`)
: Makes a window containing a menu of all of the currently visited nodes. This window becomes the selected window, and you may use the standard Info commands within it.

`C-x b` (`select-visited-node`)
: Selects a node which has been previously visited in a visible window. This is similar to '`C-x C-b`' followed by '`m`', but no window is created.

D.4 Searching an Info File

GNU Info allows you to search for a sequence of characters throughout an entire info file, search through the indices of an info file, or find areas within an info file which discuss a particular topic.

`s` (`search`)
: Reads a string in the echo area and searches for it.

`C-s` (`isearch-forward`)
: Interactively searches forward through the info file for a string as you type it.

`C-r` (`isearch-backward`)
: Interactively searches backward through the info file for a string as you type it.

`i` (`index-search`)
: Looks up a string in the indices for this info file, and selects a node where the found index entry points to.

`,` (`next-index-match`)
: Moves to the node containing the next matching index item from the last '`i`' command.

The most basic searching command is '`s`' (`search`). The '`s`' command prompts you for a string in the echo area, and then searches the remainder of the info file for an occurrence of that string. If the string is found, the node containing it is selected, and the cursor is left positioned at the start of the found string. Subsequent '`s`' commands show you the default search string within '`[`' and '`]`'; pressing `RET` instead of typing a new string will use the default search string.

Incremental searching is similar to basic searching, but the string is looked up while you are typing it, instead of waiting until the entire search string has been specified.

D.5 Selecting Cross References

We have already discussed the '`Next`', '`Prev`', and '`Up`' pointers which appear at the top of a node. In addition to these pointers, a node may contain other pointers which refer you to a different node, perhaps in another info file. Such pointers are called *cross references*, or *xrefs* for short.

D.5.1 Parts of an Xref

Cross references have two major parts: the first part is called the *label*; it is the name that you can use to refer to the cross reference, and the second is the *target*; it is the full name of the node that the cross reference points to.

The target is separated from the label by a colon ':'; first the label appears, and then the target. For example, in the sample menu cross reference below, the single colon separates the label from the target.

```
* Foo Label: Foo Target.  More information about Foo.
```

Note the '.' which ends the name of the target. The '.' is not part of the target; it serves only to let Info know where the target name ends.

A shorthand way of specifying references allows two adjacent colons to stand for a target name which is the same as the label name:

```
* Foo Commands::  Commands pertaining to Foo.
```

In the above example, the name of the target is the same as the name of the label, in this case `Foo Commands`.

You will normally see two types of cross references while viewing nodes: *menu* references, and *note* references. Menu references appear within a node's menu; they begin with a '`*`' at the beginning of a line, and continue with a label, a target, and a comment which describes what the contents of the node pointed to contains.

Note references appear within the body of the node text; they begin with `*Note`, and continue with a label and a target.

Like '`Next`', '`Prev`' and '`Up`' pointers, cross references can point to any valid node. They are used to refer you to a place where more detailed information can be found on a particular subject. Here is a cross reference which points to a node within the Texinfo documentation: See section "Writing an Xref" in *the Texinfo Manual*, for more information on creating your own texinfo cross references.

D.5.2 Selecting Xrefs

The following table lists the Info commands which operate on menu items.

`1` (`menu-digit`)
`2` ... `9`
: Within an Info window, pressing a single digit, (such as '`1`'), selects that menu item, and places its node in the current window. For convenience, there is one exception; pressing '`0`' selects the *last* item in the node's menu.

`0` (`last-menu-item`)
: Select the last item in the current node's menu.

`m` (`menu-item`)
: Reads the name of a menu item in the echo area and selects its node. Completion is available while reading the menu label.

`M-x find-menu`
: Moves the cursor to the start of this node's menu.

This table lists the Info commands which operate on note cross references.

`f` (`xref-item`)
`r` Reads the name of a note cross reference in the echo area and selects its node. Completion is available while reading the cross reference label.

Finally, the next few commands operate on menu or note references alike:

`TAB` (`move-to-next-xref`)
: Moves the cursor to the start of the next nearest menu item or note reference in this node. You can then use `RET` (`select-reference-this-line` to select the menu or note reference.

`M-TAB` (`move-to-prev-xref`)
: Moves the cursor the start of the nearest previous menu item or note reference in this node.

`RET` (`select-reference-this-line`)
: Selects the menu item or note reference appearing on this line.

D.6 Manipulating Multiple Windows

A *window* is a place to show the text of a node. Windows have a view area where the text of the node is displayed, and an associated *mode line*, which briefly describes the node being viewed.

GNU Info supports multiple windows appearing in a single screen; each window is separated from the next by its modeline. At any time, there is only one *active* window, that is, the window in which the cursor appears. There are commands available for creating windows, changing the size of windows, selecting which window is active, and for deleting windows.

D.6.1 The Mode Line

A *mode line* is a line of inverse video which appears at the bottom of an info window. It describes the contents of the window just above it; this information includes the name of the file and node appearing in that window, the number of screen lines it takes to display the node, and the percentage of text that is above the top of the window. It can also tell you if the indirect tags table for this info file needs to be updated, and whether or not the info file was compressed when stored on disk.

Here is a sample mode line for a window containing an uncompressed file named '`dir`', showing the node '`Top`'.

```
-----Info: (dir)Top, 40 lines --Top---------------------------------------
            ^^   ^   ^^^        ^^
          (file)Node #lines    where
```

When a node comes from a file which is compressed on disk, this is indicated in the mode line with two small 'z''s. In addition, if the info file containing the node has been split into subfiles, the name of the subfile containing the node appears in the modeline as well:

```
--zz-Info: (emacs)Top, 291 lines --Top-- Subfile: emacs-1.Z---------------
```

When Info makes a node internally, such that there is no corresponding info file on disk, the name of the node is surrounded by asterisks ('*'). The name itself tells you what the contents of the window are; the sample mode line below shows an internally constructed node showing possible completions:

```
-----Info: *Completions*, 7 lines --All-----------------------------------
```

D.6.2 Window Commands

It can be convenient to view more than one node at a time. To allow this, Info can display more than one *window*. Each window has its own mode line (see Section D.6.1 [The Mode Line], page 195) and history of nodes viewed in that window (see Section D.3 [history-node], page 191).

`C-x o` (`next-window`)
: Selects the next window on the screen. Note that the echo area can only be selected if it is already in use, and you have left it temporarily. Normally, '`C-x o`' simply moves the cursor into the next window on the screen, or if you are already within the last window, into the first window on the screen. Given a numeric argument, '`C-x o`' moves over that many windows. A negative argument causes '`C-x o`' to select the previous window on the screen.

`M-x prev-window`
: Selects the previous window on the screen. This is identical to '`C-x o`' with a negative argument.

`C-x 2` (`split-window`)
: Splits the current window into two windows, both showing the same node. Each window is one half the size of the original window, and the cursor remains in the original window. The variable `automatic-tiling` can cause all of the windows on the screen to be resized for you automatically, please see Section D.9 [automatic-tiling], page 201 for more information.

`C-x 0` (`delete-window`)
: Deletes the current window from the screen. If you have made too many windows and your screen appears cluttered, this is the way to get rid of some of them.

`C-x 1` (`keep-one-window`)
: Deletes all of the windows excepting the current one.

`ESC C-v` (`scroll-other-window`)
: Scrolls the other window, in the same fashion that '`C-v`' might scroll the current window. Given a negative argument, the "other" window is scrolled backward.

`C-x ^` (`grow-window`)
: Grows (or shrinks) the current window. Given a numeric argument, grows the current window that many lines; with a negative numeric argument, the window is shrunk instead.

`C-x t` (`tile-windows`)
: Divides the available screen space among all of the visible windows. Each window is given an equal portion of the screen in which to display its contents. The variable `automatic-tiling` can cause `tile-windows` to be called when a window is created or deleted. See Section D.9 [`automatic-tiling`], page 201.

D.6.3 The Echo Area

The *echo area* is a one line window which appears at the bottom of the screen. It is used to display informative or error messages, and to read lines of input from you when that is necessary. Almost all of the commands available in the echo area are identical to their Emacs counterparts, so please refer to that documentation for greater depth of discussion on the concepts of editing a line of text. The following table briefly lists the commands that are available while input is being read in the echo area:

`C-f` (`echo-area-forward`)
: Moves forward a character.

`C-b` (`echo-area-backward`)
: Moves backward a character.

`C-a` (`echo-area-beg-of-line`)
: Moves to the start of the input line.

`C-e` (`echo-area-end-of-line`)
: Moves to the end of the input line.

`M-f` (`echo-area-forward-word`)
: Moves forward a word.

`M-b` (`echo-area-backward-word`)
: Moves backward a word.

`C-d` (`echo-area-delete`)
: Deletes the character under the cursor.

`DEL` (`echo-area-rubout`)
: Deletes the character behind the cursor.

`C-g` (`echo-area-abort`)
: Cancels or quits the current operation. If completion is being read, '`C-g`' discards the text of the input line which does not match any completion. If the input line is empty, '`C-g`' aborts the calling function.

`RET` (`echo-area-newline`)
: Accepts (or forces completion of) the current input line.

`C-q` (`echo-area-quoted-insert`)
: Inserts the next character verbatim. This is how you can insert control characters into a search string, for example.

printing character (`echo-area-insert`)
: Inserts the character.

`M-TAB` (`echo-area-tab-insert`)
: Inserts a TAB character.

`C-t` (`echo-area-transpose-chars`)
: Transposes the characters at the cursor.

The next group of commands deal with *killing*, and *yanking* text. For an in depth discussion of killing and yanking, see section "Killing and Deleting" in *the GNU Emacs Manual*

`M-d` (`echo-area-kill-word`)
: Kills the word following the cursor.

`M-DEL` (`echo-area-backward-kill-word`)
: Kills the word preceding the cursor.

`C-k` (`echo-area-kill-line`)
: Kills the text from the cursor to the end of the line.

`C-x DEL` (`echo-area-backward-kill-line`)
: Kills the text from the cursor to the beginning of the line.

`C-y` (`echo-area-yank`)
: Yanks back the contents of the last kill.

`M-y` (`echo-area-yank-pop`)
: Yanks back a previous kill, removing the last yanked text first.

Sometimes when reading input in the echo area, the command that needed input will only accept one of a list of several choices. The choices represent the *possible completions*, and you must respond with one of them. Since there are a limited number of responses you can make, Info allows you to abbreviate what you type, only typing as much of the response as is necessary to uniquely identify it. In addition, you can request Info to fill in as much of the response as is possible; this is called *completion*.

The following commands are available when completing in the echo area:

`TAB` (`echo-area-complete`)
`SPC` Inserts as much of a completion as is possible.

`?` (`echo-area-possible-completions`)

Displays a window containing a list of the possible completions of what you have typed so far. For example, if the available choices are:

```
bar
foliate
food
forget
```

and you have typed an '`f`', followed by '`?`', the possible completions would contain:

```
foliate
food
forget
```

i.e., all of the choices which begin with '`f`'. Pressing `SPC` or `TAB` would result in '`fo`' appearing in the echo area, since all of the choices which begin with '`f`' continue with '`o`'. Now, typing '`l`' followed by '`TAB`' results in '`foliate`' appearing in the echo area, since that is the only choice which begins with '`fol`'.

`ESC C-v` (`echo-area-scroll-completions-window`)

Scrolls the completions window, if that is visible, or the "other" window if not.

D.7 Printing Out Nodes

You may wish to print out the contents of a node as a quick reference document for later use. Info provides you with a command for doing this. In general, we recommend that you use TeX to format the document and print sections of it, by running `tex` on the texinfo source file.

`M-x print-node`

Pipes the contents of the current node through the command in the environment variable `INFO_PRINT_COMMAND`. If the variable doesn't exist, the node is simply piped to `lpr`.

D.8 Miscellaneous Info Commands

GNU Info contains several commands which self-document GNU Info:

`M-x describe-command`

Reads the name of an Info command in the echo area and then displays a brief description of what that command does.

`M-x describe-key`

Reads a key sequence in the echo area, and then displays the name and documentation of the Info command that the key sequence invokes.

`M-x describe-variable`

Reads the name of a variable in the echo area and then displays a brief description of what the variable affects.

`M-x where-is`
: Reads the name of an Info command in the echo area, and then displays a key sequence which can be typed in order to invoke that command.

`C-h` (`get-help-window`)
`?`
: Creates (or moves into) the window displaying `*Help*`, and places a node containing a quick reference card into it. This window displays the most concise information about GNU Info available.

`h` (`get-info-help-node`)
: Tries hard to visit the node `(info)Help`. The info file '`info.texi`' distributed with GNU Info contains this node. Of course, the file must first be processed with `makeinfo`, and then placed into the location of your info directory.

Here are the commands for creating a numeric argument:

`C-u` (`universal-argument`)
: Starts (or multiplies by 4) the current numeric argument. '`C-u`' is a good way to give a small numeric argument to cursor movement or scrolling commands; '`C-u C-v`' scrolls the screen 4 lines, while '`C-u C-u C-n`' moves the cursor down 16 lines.

`M-1` (`add-digit-to-numeric-arg`)
`M-2` ... `M-9`
: Adds the digit value of the invoking key to the current numeric argument. Once Info is reading a numeric argument, you may just type the digits of the argument, without the Meta prefix. For example, you might give '`C-l`' a numeric argument of 32 by typing:

```
C-u 3 2 C-l
```

or

```
M-3 2 C-l
```

'`C-g`' is used to abort the reading of a multi-character key sequence, to cancel lengthy operations (such as multi-file searches) and to cancel reading input in the echo area.

`C-g` (`abort-key`)
: Cancels current operation.

The '`q`' command of Info simply quits running Info.

`q` (`quit`)
: Exits GNU Info.

If the operating system tells GNU Info that the screen is 60 lines tall, and it is actually only 40 lines tall, here is a way to tell Info that the operating system is correct.

`M-x set-screen-height`
: Reads a height value in the echo area and sets the height of the displayed screen to that value.

Finally, Info provides a convenient way to display footnotes which might be associated with the current node that you are viewing:

`ESC C-f` (`show-footnotes`)
: Shows the footnotes (if any) associated with the current node in another window. You can have Info automatically display the footnotes associated with a node when the node is selected by setting the variable `automatic-footnotes`. See Section D.9 [`automatic-footnotes`], page 201.

D.9 Manipulating Variables

GNU Info contains several *variables* whose values are looked at by various Info commands. You can change the values of these variables, and thus change the behavior of Info to more closely match your environment and info file reading manner.

`M-x set-variable`
: Reads the name of a variable, and the value for it, in the echo area and then sets the variable to that value. Completion is available when reading the variable name; often, completion is available when reading the value to give to the variable, but that depends on the variable itself. If a variable does *not* supply multiple choices to complete over, it expects a numeric value.

`M-x describe-variable`
: Reads the name of a variable in the echo area and then displays a brief description of what the variable affects.

Here is a list of the variables that you can set in Info.

`automatic-footnotes`
: When set to `On`, footnotes appear and disappear automatically. This variable is `On` by default. When a node is selected, a window containing the footnotes which appear in that node is created, and the footnotes are displayed within the new window. The window that Info creates to contain the footnotes is called '`*Footnotes*`'. If a node is selected which contains no footnotes, and a '`*Footnotes*`' window is on the screen, the '`*Footnotes*`' window is deleted. Footnote windows created in this fashion are not automatically tiled so that they can use as little of the display as is possible.

`automatic-tiling`
: When set to `On`, creating or deleting a window resizes other windows. This variable is `Off` by default. Normally, typing '`C-x 2`' divides the current window into two equal parts. When `automatic-tiling` is set to `On`, all of the windows are resized automatically, keeping an equal number of lines visible in each window. There are exceptions to the automatic tiling; specifically, the windows '`*Completions*`' and '`*Footnotes*`' are *not* resized through automatic tiling; they remain their original size.

`visible-bell`
: When set to `On`, GNU Info attempts to flash the screen instead of ringing the bell. This variable is `Off` by default. Of course, Info can only flash the screen if the terminal allows it; in the case that the terminal does not allow it, the setting of this variable has no effect. However, you can make Info perform quietly by setting the `errors-ring-bell` variable to `Off`.

`errors-ring-bell`
: When set to `On`, errors cause the bell to ring. The default setting of this variable is `On`.

`gc-compressed-files`
: When set to `On`, Info garbage collects files which had to be uncompressed. The default value of this variable is `Off`. Whenever a node is visited in Info, the info file containing that node is read into core, and Info reads information about the tags and nodes contained in that file. Once the tags information is read by Info, it is never forgotten. However, the actual text of the nodes does not need to remain in core unless a particular info window needs it. For non-compressed files, the text of the nodes does not remain in core when it is no longer in use. But de-compressing a file can be a time consuming operation, and so Info tries hard not to do it twice. `gc-compressed-files` tells Info it is okay to garbage collect the text of the nodes of a file which was compressed on disk.

`show-index-match`
: When set to `On`, the portion of the matched search string is highlighted in the message which explains where the matched search string was found. The default value of this variable is `On`. When Info displays the location where an index match was found, (see Section D.4 [`next-index-match`], page 193), the portion of the string that you had typed is highlighted by displaying it in the inverse case from its surrounding characters.

`scroll-behaviour`
: Controls what happens when forward scrolling is requested at the end of a node, or when backward scrolling is requested at the beginning of a node. The default value for this variable is `Continuous`. There are three possible values for this variable:

  `Continuous`
  : Tries to get the first item in this node's menu, or failing that, the '`Next`' node, or failing that, the '`Next`' of the '`Up`'. This behavior is identical to using the '`]`' (`global-next-node`) and '`[`' (`global-prev-node`) commands.

  `Next Only`
  : Only tries to get the '`Next`' node.

  `Page Only`
  : Simply gives up, changing nothing. If `scroll-behaviour` is `Page Only`, no scrolling command can change the node that is being viewed.

`scroll-step`
: The number of lines to scroll when the cursor moves out of the window. Scrolling happens automatically if the cursor has moved out of the visible portion of the node

text when it is time to display. Usually the scrolling is done so as to put the cursor on the center line of the current window. However, if the variable `scroll-step` has a nonzero value, Info attempts to scroll the node text by that many lines; if that is enough to bring the cursor back into the window, that is what is done. The default value of this variable is 0, thus placing the cursor (and the text it is attached to) in the center of the window. Setting this variable to 1 causes a kind of "smooth scrolling" which some people prefer.

`ISO-Latin`

When set to `On`, Info accepts and displays ISO Latin characters. By default, Info assumes an ASCII character set. `ISO-Latin` tells Info that it is running in an environment where the European standard character set is in use, and allows you to input such characters to Info, as well as display them.

Concept Index

Variable Index

Function Index

O

P

Q

R

S

Operator Index

Readline Index

Y

Info Index

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W

X

Table of Contents