PRO-MATLAB

for Unix Computers

Users' Guide

by Cleve Moler, John Little and Steve Bangert

The MathWorks, Inc.

PRO-MATLAB

for Unix Computers

Version 3.2-Unix
August 17, 1987

by Cleve Moler, John Little and Steve Bangert

The MathWorks, Inc.
20 North Main St., Suite 250
Sherborn, MA 01770
(617) 653-1415

Telex: 9102405521
E-mail: na.mathworks@score.stanford.edu

PRO-MATLAB User's Guide
August 17, 1987

Preface

What is MATLAB?

MATLAB is an interactive program to help you with your scientific and engineering numeric calculations. The name MATLAB stands for *matrix laboratory*. Originally written in Fortran for mainframe computers, it provides easy access to matrix software developed by the LINPACK and EISPACK projects. Together, LINPACK and EISPACK represent the state of the art in software for matrix computation.

MATLAB is an interactive system whose basic data element is a matrix that does not require dimensioning. This allows you to solve many numerical problems in a fraction of the time it would take to write a program in a language like Fortran, Basic, or C. Furthermore, problem solutions are expressed in MATLAB almost exactly as they are written mathematically.

MATLAB has evolved over more than half a decade with input from many users. In university environments it has become the standard instructional tool used in introductory courses in applied linear algebra, as well as advanced courses in other areas. In industrial settings, MATLAB is used for research, and to solve practical engineering and mathematical problems. Typical uses include general purpose numeric computation, algorithm prototyping, and solving the special purpose problems with matrix formulations that arise in disciplines like automatic control theory, statistics, and digital signal processing (time-series analysis).

The highly optimized, second generation MATLAB that runs on IBM and other MS-DOS compatible personal computers is called PC-MATLAB. On larger computers, like Sun Workstations and VAX computers, the modern version of MATLAB is called PRO-MATLAB. On the Macintosh, it's MacMATLAB. Entirely written in the C language, MATLAB is a complete "integrated" system, including graphics, programmable macros, IEEE arithmetic, a fast interpreter, and many analytical commands. Experienced users of earlier versions of MATLAB will find the modern version of MATLAB to be a significantly more powerful version of an old friend. New users of MATLAB will benefit from the years of experience gained in developing the earlier versions. Both will find that MATLAB has evolved beyond the "matrix laboratory" to become a versatile scientific "spreadsheet" for numeric calculations.

About This Guide:

The MATLAB *User's Guide* is divided into three main parts:

COMPUTER The first section contains instructions that are specific to the particular computer that you are using. This includes *installation* instructions, explaining the mechanics of making backup copies, configuring the system, and invoking the program. Also explained in this section are *interactions* with MATLAB that are system dependent, like last-line editing and hardcopy operations. For users of Sun Workstation or Macintosh versions, the special added features that allow mouse and window system interaction are discussed.

TUTORIAL The *tutorial* is an introduction to the MATLAB system. The basic features including matrix manipulation, graphics, language features, and M-files are described. Many examples are given.

REFERENCE The *reference* section contains extensive details on all MATLAB functions, including information on how the algorithms work.

How To Use This Guide:

Once MATLAB is installed and running, this guide will not be needed very often. The on-line help and demo facilities provide most of what's needed in order to use the program.

New users should start by reading the *tutorial.* The most important things to learn are how to enter matrices, how to use the colon ":" operator, and how to invoke functions. After the basics are mastered, the on-line help facility is usually sufficient to learn the other commands.

Users of earlier versions of MATLAB should skim the *tutorial* to find out about graphics and M-files, which are the major differences between the old and the new MATLAB.

Users familiar with MATLAB on other machines should look at the computer-specific first section to find out about system dependent features.

Experienced users can rely on just the on-line help facility. Occasionally it may be useful to look up more detail on a particular function in the *reference section* of the manual, or to skim the *reference section* to discover new features. Take notice that there are *two* indices - one at the end of the computer-specific first section and one at the end of the guide.

About The Cover:

The covers of this guide and other *MathWorks, Inc.* publications depict various solutions to a problem which has played a small, but interesting, role in the history of numerical methods during the last 30 years. The problem involves finding the modes of vibration of a membrane supported by an L-shaped domain consisting of three unit squares. The nonconvex corner in the domain generates singularities in the solutions, thereby providing challenges for both the underlying mathematical theory and the computational algorithms. There are important applications, including wave guides, structures and semiconductors.

Two of the founders of modern numerical analysis, George Forsythe and J. H. Wilkinson, worked on the problem in the 1950's. (See G. E. Forsythe and W. R. Wasow, *Finite-Difference Methods for Partial Differential Equations*, Wiley, 1960.) One of the authors of this guide (Moler) used finite difference techniques to compute solutions in 1965. Typical computer runs took up to half an hour of dedicated computer time on what were then Stanford University's primary computers, an IBM 7090 and a Burroughs B5000.

The first version of the approach we now use was published in 1967 by L. Fox, P. Henrici and C. Moler (SIAM J. Numer. Anal. 4, 1967, pp.89-102.) It replaced finite differences by combinations of distinguished fundamental solutions to the underlying differential equation formed from Bessel and trigonometric functions. The idea is a generalization of the fact that the real and imaginary parts of complex analytic functions are solutions to Laplace's equation. In the early 1970's new matrix algorithms, particularly Gene Golub's orthogonalization techniques for least squares problems, provided further algorithmic improvements.

Today, MATLAB allows us to express the entire algorithm in a few dozen lines, to compute the solution with great accuracy in a few minutes on a computer at home, and to readily manipulate three dimensional displays of the results. Our MATLAB work was one step in the development of a demonstration program for the Intel iPSC hypercube multiprocessor which computes and displays moving pictures of the solution to the time-dependent wave equation extension of the membrane problem. We have included our MATLAB program, membrane.m, with the M-files in the *Utility Library*.

Who Wrote MATLAB?

The original MATLAB was written in Fortran by Cleve Moler, in an evolutionary process over several years. The underlying matrix algorithms are from the many people who worked on the LINPACK and EISPACK projects.

The new MATLAB program was written in C by *The MathWorks, Inc.* Steve Bangert wrote the parser/user interface, Steve Kleiman implemented the graphics, and John Little and Cleve Moler wrote the analytical - numerical routines and most of the M-files. Steve Herskovitz is responsible for the Macintosh version and Loren Shure did the conversion to Apollo. Marc Ullman figured out how to do all the nasty system things that make MATLAB fast and complete. John Little and Cleve Moler are the authors of this user's guide.

Probably the most important feature of MATLAB, and one that we took care to perfect, is the easy extensibility. This allows *you* to become a contributing author too, creating your own applications. We look forward to the promise of many scientists, mathematicians and engineers quickly developing new and interesting commands, all without writing a single line of Fortran or other "low-level" code.

January 14, 1985
Portola Valley, California

Unix MATLAB

System requirements

MATLAB requires the following hardware and software:

- Berkeley Unix version 4.3.
- A 1600 BPI ½" 9-track tape drive.
- Approximately 4 Mbytes of disk space.
- VT100/VT240 compatible terminals with Tektronix 4014 graphics capabilities.

The following are optional for Unix MATLAB systems:

- Apple LaserWriter or other PostScript compatible printer.
- HP 7475 plotter or other Hewlett-Packard Graphics Language (HPGL) compatible hardcopy device.
- Hardcopy devices that accept Tektronix 4014 format commands, for example, Imagen laser printers.
- Other hardcopy devices.

1. Overview

This section of the user's guide contains instructions that are specific to the Unix implementation of MATLAB. These include

- *Installation* instructions, for your system manager, explaining the mechanics of installing and configuring the software on your system, and
- *Installation* instructions, for you the user, describing how you should prepare your account for using MATLAB, and
- *Operational* instructions for using Unix-specific features of MATLAB like editing, command-line changes and hardcopy operations.

Unless you are the system manager, ready to install MATLAB, skip to section 4.

2. Installing PRO-MATLAB - for the system manager

The installation instructions assume a basic working knowledge of Unix and the C-shell. If you are not familiar with these, we recommend that you seek the assistance of someone who is. There are other possible ways in which to configure MATLAB, depending upon your preferences in organizing the Unix file system. We will cover only a suggested basic configuration.

MATLAB is distributed on a single tape in Unix tar format. The tape contains the MATLAB program, related utility files, and any program options you may have ordered, such as the CONTROL SYSTEM TOOLBOX or the SYSTEM IDENTIFICATION TOOLBOX.

The following are instructions for installing MATLAB on your Unix computer. We assume a tape drive with device name /rst0. All commands given are Unix commands; for more information consult the Unix manuals, or better, a knowledgeable Unix user.

[1] *Create a new disk directory*: Create a new subdirectory called /matlab in your file system for the MATLAB binary and approximately 150 other small utility files. Set this new directory to the current directory. For example, if the new directory is to be /usr/matlab, the commands are:

```
% cd /usr
% mkdir matlab
% cd matlab
```

[2] *Move the files from the tape to disk*: Put the tape in the tape drive and tar the files into the new disk directory:

```
% tar xvfb /dev/rst0 126
```

This will run for several minutes, transferring all files and directories into the designated directory. When finished, remove the tape cartridge and set aside in a safe place.

[3] *Modify the* matlab_def *script file*: One of the files now in the current directory is a shell script file called matlab_def. Edit matlab_def and set up the environmental variables correctly, based upon the directory name you've chosen for MATLAB. You can probably get away with just reading the instructions found in the comments in the script file, but if you need more information see section 8.

[4] *Modify the M-file* print.m: One of the files in the current directory is an M-file called print.m. This file implements the print command that MATLAB users invoke to spool graphics hardcopy. You can edit this file to set an appropriate site-wide default for device type and location. See section 6.2.

[5] *Optionally change* matlab.m: The M-file matlab.m is invoked automatically each time a user starts MATLAB. You may put here welcome messages, default definitions, or any MATLAB expressions that you decide should be foisted on the users. If most users have a certain graphics terminal type, it may be useful to invoke a Terminal Personality M-file so that your users won't have to. See the section 4 for more information.

This completes the system manager's part of the program installation.

3. Tape contents

The /matlab directory should now contain the MATLAB program, related utility files, and new subdirectories containing any program options you may have ordered, such as the CONTROL SYSTEM TOOLBOX and the SYSTEM IDENTIFICATION TOOLBOX. Here is a summary of the files you will find:

readme.m	A file you should read that lists undocumented features of MATLAB that were added after this guide was printed.
matlab	Main executable binary image.
matlab_def	A C-shell script file that sets up aliases that allow a user to invoke MATLAB.

matlab.hlp	The file used by the help facility.
xxx.m	There are more than 120 M-files that contain demos and support functions.
gpp	The Graphics Post-Processor program that translates MATLAB graphics metafiles (MET-files) into device-specific files for hardcopy output.
translate	A utility program that translates data files in other formats to the special MATLAB data file format. (The MAT-file format.)
xxx.c	Some subroutines to help interface C and Fortran programs to MATLAB data files. More information on these is found in the last section of the *Tutorial*.
testls.f	An example of using the interface subroutines to create a MAT-file from a Fortran program.
/signal	A subdirectory containing the M-files for the SIGNAL PROCESSING TOOLBOX.
/control	If you ordered this option, a subdirectory containing the M-files for the CONTROL SYSTEM TOOLBOX.
/ident	If you ordered this option, a subdirectory containing the M-files for the SYSTEM IDENTIFICATION TOOLBOX.
/kermit	A subdirectory containing the KERMIT communications program. You may delete the contents of this directory if you already have KERMIT on your system.

4. Installing PRO-MATLAB - for the user

If you have never used MATLAB before, or if you have a brand new Unix account, there are a few simple steps you need to take before you can start using MATLAB. First you need to ask your system manager where the MATLAB files are located. Specifically, ask him the full directory name of the file named matlab_def. He will tell you something like

```
/usr/matlab/matlab_def
```

The next step is to edit your .cshrc file in your home directory and to insert a new line anywhere in this file:

```
source /usr/matlab/matlab_def
```

This statement creates a set of aliases or definitions that allow you to run the

MATLAB system. Furthermore, by putting it in your .cshrc file, the definitions are created automatically each time you login. Specifically, matlab_def defines three new commands that you can use:

```
matlab
gpp
translate
```

The first allows you to invoke MATLAB. The second invokes the Graphics Post-Processor for converting graphics metafiles into hardcopy. The third invokes the data translation program that comes with MATLAB.

Next, we suggest that you create a new directory called /matlab off of your home directory. Suppose your username is /usr/witt. Use the commands:

```
% cd /usr/witt
% mkdir matlab
% cd matlab
```

We ask you to create this directory because MATLAB's search path for M-files includes looking in the directory by this name off of your home directory. As you become more experienced with MATLAB you will create your own library of personalized M-files. Putting them in this special location gives you access to them no matter where you are in your file system when you invoke MATLAB.

As a new user, you won't yet have any M-files in /usr/witt/matlab yet, but we will start your collection now. Invoke your editor to create a new file called startup.m (Make sure you are in the special /matlab directory you just created off of your home directory). Startup.m is a file that MATLAB looks for and tries to execute each time it is invoked. You may define constants, engineering conversion factors, or anything else you would like pre-defined in your workspace. We are going to use it to tell MATLAB the type of graphics terminal you have so you won't need to do this manually each time you invoke the program.

Here is a table of Terminal Personality M-files:

Terminal Personality M-files	
tek	Tektronix 4014
tek4100	Tektronix 4100 series
retro	RetroGraphics card
hds	Human Designed Systems
sg100	Selanar graphics 100
sg200	Selanar graphics 200
vt240	VT240 series
hp2647	Hewlett-Packard 2647
ergo	Ergo terminal
graphon	Graphon terminal
citoh	C.Itoh terminal
vt100	No graphics/VT100
vt52	No graphics/VT52
xterm	X-windows xterm utility
versa	VersaTerm emulator on Macintosh

Hopefully your terminal is listed here. Pick out the M-file name in the left column of this table and enter it into startup.m, the empty file that should now be in your editor. You can save the file and quit the editor.

If you do not find your terminal listed, talk to your system manager or call *The MathWorks*. We can assist you in creating a new Personality File for your terminal.

It is also possible to tell MATLAB your terminal type interactively. Execute the command terminal to be prompted for your terminal type. This is also a good way to find out if there are new terminal types available that didn't make it into the table shown above.

This completes your part of the program installation. If you logout and log back in, you will be able to invoke MATLAB from the Unix prompt by executing matlab. If this is your first time with MATLAB, you may wish to start by typing demo, which brings up a set of demonstrations for you to run.

5. Last-line editing and recall

The arrow keys on the keypad can be used to edit mistyped commands or to recall previous command lines. For example, suppose you enter

```
log(sqt(atan(2*(3+4))))
```

You have misspelled sqrt. The response from MATLAB is the error

message,

```
Undefined variable or function.
Symbol in question → sqt
```

Instead of retyping the entire line, simply hit the Up-Arrow key. The incorrect line will be displayed again and you can move the cursor over using the Left-Arrow key until you can insert the missing r.

```
log(sqrt(atan(2*(3+4))))

ans =
  0.2026
```

The Arrow-keys on the keypad work on copies of the previous input lines, which have been saved in a moderately sized input buffer. Here is a brief description of their function:

Last-line Editing and Recall Keys	
Up Arrow	Recall previous line.
Down Arrow	Recall next line.
Left Arrow	Move left one character.
Right Arrow	Move right one character.
Ctrl-L	Move left one word.
Ctrl-R	Move right one word.
Ctrl-B	Move to beginning of line.
Ctrl-E	Move to end of line.
Ctrl-U	Cancel current line.
Ctrl-A	Toggle between insert and overtype mode.
Delete	Delete character at cursor.
Backspace	Delete character left of cursor.

6. Graphics hardcopy

The Unix version of MATLAB provides graphics hardcopy through graphics *metafiles* and a graphics *post-processor program.* Two MATLAB commands, print and meta, are associated with hardcopy operations:

• Meta *file* opens a high-resolution graphics metafile, using the specified filename, and writes the current graph to it for later processing. Subsequent meta commands append to the previously specified file. The metafile may be processed later using the graphics post processor (GPP) program. Metafiles use a default filetype of .met.

• Print immediately spools a copy of the graph on the screen to the printer *if* you've configured print.m correctly. Print is an M-file that uses the metafile facility.

6.1. Using GPP

The graphics post-processor, GPP, operates on device-independent MET-files (files with a filetype of .met produced by the meta command) to produce device-specific plot files that can be spooled to a particular hard-copy device. GPP is invoked at the Unix level with a command of the form

% gpp *filename* -d*device* [-f -t -p -op -ol -cs -cc]

where spaces are used between arguments, the brackets indicate optional arguments, and *device* can be one of the following:

GPP Devices	
ps	PostScript (Apple LaserWriter)
epsd	Epson printer, draft quality
epsf	Epson printer, final quality
jet	HP Laser Jet Plus (300 dpi)
jet150	HP Laser Jet Plus (150 dpi)
hpgl	HPGL (HP compatible plotters)
pen	Simple ASCII "pen" movements
tek	Tektronix 4014 output
img	Imagen Laserprinter

GPP takes the MET-file and creates a new file containing device-specific control sequences for the requested device. The filename extension on the new file is the same as the device type in the table, without the leading "d". For example, the MATLAB commands

```
t = -50:.3:50;
y = sin(t)./t;
plot(t,y)
meta sincplot
plot(diff(y)./diff(t))
meta
```

create a metafile called sincplot.met containing two graphs, one of a *sinc* function and a second of its derivative. To get actual hardcopy from the metafile, the next step is to run GPP. For example, from Unix:

```
% gpp sincplot -dps
% lpr sincplot.ps
```

In this case we've selected PostScript output and sent the resulting file, sincplot.ps, to a LaserWriter connect to our computer. Note that we could have issued these commands without quitting from MATLAB using the "!" operator.

Optional Arguments:

-f The optional GPP argument -f can be used to select a name for the output file created by GPP.

-t Optional argument -t causes output to be sent to standard output (normally the terminal, if no redirection). If you set the device type to tek, Tektronix output is sent to your screen, which can be used to view the contents of a metafile on a terminal.

-p Optional argument -p tells GPP to pause between plots if there are several on the metafile. This can be useful if GPP is being used with an interactive display device to preview graphs. GPP will pause until any key is struck.

-op -ol These two arguments select whether hardcopy is made in *portrait* or in *landscape* orientation. Different hardcopy devices have different defaults, and some do not offer both options. In general, the higher-resolution devices default to portrait orientation, which is the best for inclusion into books and reports.

-cs -cc These two optional arguments select the text character quality used on devices whose characters are stroke-generated. -cs uses a simplex or single-line character set, while -cc uses a higher quality complex or multi-line character set. Devices that use stroke characters include the HP LaserJet, Epson printers, and the pen driver. By default, the complex font is used, except for draft-quality mode on Epson printers, on which the low resolution causes a poor appearance.

The PostScript, HPGL, and Pen output files, unlike the other output formats, are ASCII files. The PostScript file consists of human readable statements, which may be edited to scale picture sizes, change fonts, include into TEX documents, etc.

New output devices will be added to GPP over time. If you invoke GPP with no arguments, you'll see a usage summary and a list of the currently available devices.

MET-files, the metafiles produced by PC-MATLAB, can be sent directly to Sun, PC, and other implementations of MATLAB that you may have access to. This can be useful if the other machines have laser printers or other high quality hardcopy devices that you would like to use.

6.2. Configuring PRINT

The MATLAB print command immediately spools a copy of the graph on the screen to the printer. Print is really an M-file that saves the graph using meta and invokes the graphics post processor. Here is what print.m looks like as distributed:

```
meta metatmp
!csh -c "gpp metatmp -dps"
delete metatmp.met
!lpr -r metatmp.ps
```

Your system manager may have modified the print.m on your system to reflect your local equipment. To see what your print.m is, execute the MATLAB command type print. You may create your own print.m to select the target hardcopy device that is convenient for you. You should put it in the /matlab directory off of your home directory and use a different name, like myprint.m.

6.3. Pen file

The -dpen GPP device option creates a simple ASCII "pen" movement file. If you have an unusual hardcopy device that is not supported directly by GPP, it is possible to write your own translation program to convert the pen-file to the output commands for your own device. The pen-file is an ASCII file where each carriage-return separated line contains three fields: X-position, Y-position, and Pen-flag. Here is a short excerpt from a typical pen-file:

```
   0    0 0
 231  235 1
1543 5031 1
1873 1236 1
```

The X- and Y-positions are 4-digit integers where (0,0) corresponds to the lower-left corner of the plotting area and (9999,9999) is the upper right. The Pen-flag is 0 if the pen is up, 1 if the pen is down, 2 to signal end-of-picture, and 3 to signal the end-of-file.

7. Using editors and external programs

As you become more proficient with MATLAB, you will find yourself increasingly working with M-files. M-files are created and modified using an editor or word-processor. By design, MATLAB does *not* have a built-in editor. The choice of editors and word-processors is a matter of personal preference - instead of forcing you to learn a new editor, MATLAB lets you use whatever editor you are already accustomed to.

Working from a terminal, you can edit M-files using ! or CTRL-Z to invoke your editor. The exclamation point character ! is used within MATLAB to indicate that the rest of an input line should be issued as a command to the Unix operating system. This is useful for running more than just editors - you can invoke any Unix utility or other program, without quitting from MATLAB. When execution of the program completes, control is returned to MATLAB.

Under Unix, pressing CTRL-Z on the keyboard suspends the current process and brings up a new shell, from which an editor can be invoked. When editing is finished, the character %, followed by the appropriate job number, returns MATLAB to the foreground. The Unix command jobs lists current job numbers.

No matter which method you use to edit M-files, MATLAB has to take special steps to clear the old compiled M-files from memory. The first time an M-file is used, it is compiled and saved in memory for subsequent use. The M-file is not accessed after this time. But if you edit the M-file on disk, the old compiled version MUST be cleared from memory before MATLAB will respond to the changed version. So, under Unix, all compiled M-functions are automatically cleared from memory each time a ! or CTRL-Z is detected. See clear for more information on clearing compiled M-functions from memory.

8. Environmental parameters

MATLAB has a number of configuration parameters that it obtains from the Unix *environment*. The *environment* is a special global "message-board" area used by the operating system to hold customization information that various programs might want.

We have pre-defined the environmental variables for you in the script file matlab_def which you invoke in your .cshrc file. Under certain circumstances you may wish to change these variables, so the following sections provide descriptions.

8.1. MATLABPATH

Like Unix, MATLAB has a search path. The Unix search path is specified with PATH, MATLAB's with MATLABPATH. If you input the name of something to MATLAB, for example by typing joy, the MATLAB interpreter:

[1] Looks to see if joy is a variable.

[2] Checks if joy is a built-in function.

[3] Looks in the current directory for a file named joy.m.

[4] Searches the directories specified by the environment symbol MATLABPATH for joy.m.

MATLABPATH has been pre-defined for you in the script file matlab_def. If you examine matlab_def, you will see something like

```
setenv MATLABPATH /usr/matlab
```

which is the C-shell command for setting environmental symbols. If you use the Bourne shell, you could achieve the same effect with

```
MATLABPATH=/usr/matlab
export MATLABPATH
```

These commands tell MATLAB to search in /usr/matlab - this is where the MATLAB utility files reside. The utility files that come with MATLAB include the help file, the data files, and a large number of M-files.

The best part about MATLABPATH is that we can specify several such search directories. You can list as many search directories as you wish, separating the names by colons ':'. If you are using the CONTROL SYSTEM TOOLBOX, your system manager will have included the control directory in the search-path:

```
setenv MATLABPATH /usr/matlab:/usr/matlab/control
```

We've set up the matlab_def file so that, in addition to the directories your system manager puts in the search path, a directory called /matlab off of your own home directory is included automatically in the search path. This is why we had you create this special directory in section 4.

You can find out what MATLABPATH is set to by executing the C-shell command

```
% printenv
```

If you wish, you can set or append to MATLABPATH yourself. This allows you to organize your own libraries of M-files. Suppose Sarah, a geophysicist, has written her own set of M-files for analyzing geophysical data. If she puts them in a directory called /usr/sarah/geo and adds a line to her .cshrc file *after* the matlab_def statement that appends her new path,

```
setenv MATLABPATH $MATLABPATH\:/usr/sarah/geo
```

then she can maintain her own private library of M-files. MATLAB will respond to them in the normal way and there won't be a need to copy these files into every directory from which she might wish to run MATLAB.

9. Installing toolboxes

Optional *Toolboxes* are available that extend MATLAB, providing additional application-specific capabilities. These include:

- The CONTROL SYSTEM TOOLBOX, by Dr. Alan J. Laub and John N. Little, that includes more than forty additional commands for control engineering and systems theory, with an emphasis on state-space techniques.

- The SYSTEM IDENTIFICATION TOOLBOX, by Dr. Lennart Ljung, that adds twenty commands for parametric modelling and system identification. It specializes in estimating models of a system based upon input-output data, or on time-series. The model structures are sometimes known as ARMA, ARMAX, Box-Jenkins or other names.

Toolboxes consist of M-files provided in additional directories on the MATLAB distribution tape. For Unix MATLAB, Toolboxes come preinstalled and no special action is required beyond checking that MATLABPATH is set correctly.

10. Kermit

MATLAB users may have occasion to work with MATLAB implementations on several different computer systems, or have a need to transmit MATLAB applications to users on other systems. MATLAB applications are comprised of M-files, containing functions and scripts, MAT-files, containing data, and MET-files, containing graphics images. M-files are ASCII files, comprised of ordinary text, while MAT-files and MET-files contain binary data. All three types of files can be transported directly between different computers. M-files, since they are ASCII, are machine

independent (at least for the machines that MATLAB currently runs on). MAT-files and MET-files, although they contain binary data, can be transported between machines because they contain a machine signature in the file header. MATLAB is smart enough to check the signature when it loads files and, if a signature indicates that a file is foreign, perform the necessary conversion.

All that is required, in order to use MATLAB across different machines, is a facility for exchanging both binary and ASCII data between the two machines. If you have such a facility between your target machines, then you can stop reading right here. An example of such a facility is DECNETDOS, a software product from DEC for communicating between MS-DOS computers and VAX/VMS computers. If you don't have such a facility, then read on.

One of the more pleasant developments in the complex and often pitfall-laden world of inter-computer data communications is a public-domain program called KERMIT. KERMIT is a protocol for transferring sequential files between computers of all sizes over ordinary asynchronous telecommunication lines using packets, check-sums, and retransmission to promote data integrity. KERMIT is non-proprietary, thoroughly documented, and in wide use. The protocol and the original implementations were developed at Columbia University and have been shared with many other institutions, some of which have made significant contributions of their own. KERMIT is presently available for many different computer systems, including all those for which MATLAB is available:

Apple Macintosh
MS-DOS Personal Computers
Sun Workstations
Apollo Workstations
VAX/Unix
VAX/VMS

In the event that you do not have KERMIT already on your system, we have provided KERMIT on the MATLAB distribution tape. Unless your system manager deleted it, KERMIT resides in a subdirectory called /kermit off of the /matlab directory where the MATLAB system is located. There are a number of KERMIT related files in this directory, including the documentation KERMIT.DOC, which you can send to your printer. There is enough information here to get you started using KERMIT, but if you get really serious about using KERMIT, you may wish to obtain a full current release. The Apollo and other versions of KERMIT are available directly from Columbia University:

Kermit Distribution
Columbia University Center for Computing Activities
612 West 115th Street
New York, NY 10025
(212) 280-3703

There are a number of settings you can give to the various implementations of KERMIT. Here is a list of some settings you must use in order to transmit M-files and MAT-files:

[MS-DOS KERMIT]

To send or receive M-files, use set eof ctrl-z.
To send or receive MAT-files, use set eof noctrl-z.

[VAX/VMS KERMIT]

To send or receive M-files, use set file type ascii.
To receive MAT-files, use set file type binary.
To send MAT-files, use set file type fixed.

[SUN KERMIT]

To send or receive M-files, use set file type text.
To send or receive MAT-files, use set file type binary.

MET-files, the MATLAB graphics metafiles, are binary files and can be sent using the same settings used for MAT-files.

As a final note we must remind you that it breaks copyright laws, violates the license agreement, and is otherwise illegal to copy or transmit between machines any files that are supplied as part of the MATLAB system, part of the toolboxes, or copyrighted by any third parties.

Index

Tutorial

MATLAB: A Tutorial

1. Fundamentals

MATLAB works with essentially only one kind of object, a rectangular numerical matrix with possibly complex elements. In some situations, special meaning is attached to 1-by-1 matrices, which are scalars, and to matrices with only one row or column, which are vectors. Operations and commands in MATLAB are intended to be natural, in a matrix sense, not unlike how they might be indicated on paper.

Let's begin by looking at how matrices can be entered into MATLAB. If you're a "never-ever" user, you may find it useful to invoke MATLAB at this time and follow along.

1.1. Entering simple matrices

Matrices can be introduced into MATLAB in several different ways:

- Entered by an explicit list of elements,
- Generated by built in statements and functions,
- Created in M-files,
- Loaded from external data files.

There are no dimension statements or type declarations in the MATLAB language. Storage is allocated automatically, up to the amount available on any particular computer.

The easiest method of entering small matrices is to use an explicit list. The explicit list of elements is separated by blanks or commas, is surrounded by brackets, [and], and uses the semicolon ; to indicate the ends of the rows. For example, entering the statement

```
A = [1 2 3; 4 5 6; 7 8 9]
```

results in the output

```
A =
   1   2   3
   4   5   6
   7   8   9
```

The matrix A is saved for later use.

Large matrices can be spread across several input lines, with carriage returns replacing the semicolons. Although hardly necessary for a matrix of this size, the above matrix could also have been produced by three lines of input,

```
A = [ 1 2 3
      4 5 6
      7 8 9 ]
```

Matrices can be input from disk files with names ending in .m . Say a file named gena.m contains three lines of text,

```
A = [ 1 2 3
      4 5 6
      7 8 9 ]
```

then the statement gena reads the file and generates A.

The load command can be used to read matrices generated by earlier MATLAB sessions, or by other programs. More on this later.

1.2. Matrix elements

Matrix elements can be any MATLAB expressions, for example

```
x = [ -1.3  sqrt(3)  (1+2+3)*4/5 ]
```

results in

```
x =
  -1.3000    1.7321    4.8000
```

Individual matrix elements can be referenced with indices inside parentheses, (and). Continuing our example,

```
x(5) = abs(x(1))
```

produces

```
x =
  -1.3000    1.7321    4.8000    0.0000    1.3000
```

Notice that the size of x is automatically increased to accommodate the new element and that the undefined intervening elements are set to zero.

Big matrices can be constructed using little matrices as elements. For example, we could attach another row to our matrix A with

```
A = [A; [10 11 12] ]
```

which results in

```
A =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

Little matrices can be extracted from big matrices using :. For example,

```
A = A(1:3, :);
```

takes the first three rows and all the columns of the current A to give us back the original A. (More on : later.)

1.3. Statements and variables

MATLAB is an *expression* language. Expressions typed by the user are interpreted and evaluated by the MATLAB system. MATLAB statements are frequently of the form

variable = expression

or simply

expression

Expressions are composed from operators and other special characters, from functions, and from variable names. Evaluation of the expression produces a matrix, which is then displayed on the screen and assigned to the variable for future use. If the variable name and the = sign are omitted, a variable with the name ans, which stands for "answer", is automatically created. For example, typing the expression

```
1900/81
```

produces

```
ans =
  23.4568
```

A statement is normally terminated with the carriage return or enter key. However, if the last character of a statement is a semicolon, ;, the printing is suppressed, but the assignment is still carried out.

If the expression is so complicated that the statement will not fit on one line, an ellipsis consisting of two or more periods, .., followed by the carriage return, can be used to indicate that the statement continues on the next line. For example,

```
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7  ...
    - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

evaluates the partial sum of the series, assigns the sum to the variable s, but does not print anything. The blank spaces around the =, + and – signs are optional, but are included here to improve readability.

Variable and function names are formed by a letter, followed by any number of letters and digits (or underscores). Only the first 19 characters of a name are remembered.

MATLAB is case-sensitive; it normally distinguishes between upper and lower case letters, so a and A are NOT the same variable. All the function names must be in lower case; inv(A) would invert A, but INV(A) references an undefined function. However, the command casesen makes MATLAB insensitive to the case of letters. In this mode, a and A refer to the same matrix, and INV(a) would invert it.

1.4. Who and permanent variables

The example statements entered up to this point have created variables that are stored in the MATLAB *workspace*. Typing

```
who
```

causes the response:

```
Your variables are eps, pi, Inf, NaN ...

A      ans      s      x

leaving 291636 bytes of memory free.
```

This shows that four variables have been generated by our examples, including ans. It also lists four predefined or permanent variables: eps, pi, Inf and NaN.

The variable eps is used as a tolerance in determining such things as near singularity and rank. Its initial value is the distance from 1.0 to the next largest floating point number. For the IEEE arithmetic used on many personal computers and workstations,

$$\text{eps} = 2^{-52}$$

which is approximately 2.22×10^{-16}. The user may reset this to any other value, including zero.

The variable pi is π, precalculated by the program as 4*atan(1). A more provocative way to generate π is

```
imag(log(-1))
```

The variable Inf, which stands for infinity, is found in very few calculator systems or computer languages. On some computers, it is made possible by the IEEE arithmetic implemented in a math coprocessor. On other computers, floating point software is used to simulate a coprocessor. One way to generate an Inf is

```
s = 1/0
```

which results in

```
s =
   Inf

Warning: Divide by Zero.
```

On machines with IEEE arithmetic, division by zero does *not* lead to an error condition or termination of execution. It does produce a warning message and a special value which can behave in a sensible manner in subsequent computation.

The variable NaN is an IEEE number related to Inf, but has different properties. It stands for "Not a Number" and is produced by calculations such as Inf/Inf or 0/0.

More detailed information showing the size of each of the current variables is obtained by typing whos, which for our example so far, produces

```
Name        Size            Total    Complex

  A         3 by 3              9         No
 ans        1 by 1              1         No
  s         1 by 1              1         No
  x         1 by 5              5         No

Grand total is (16 * 8) = 128 bytes,

leaving 291636 bytes of memory free.
```

Each element of a real matrix requires 8 bytes of memory, so our 3-by-3 A uses 72 bytes and all our variables use a total of 128 bytes. The amount of remaining free memory depends upon the total amount available in the system and will vary from computer to computer. On computers with virtual memory, an unlimited amount may be available.

1.5. Numbers and arithmetic expressions

Conventional decimal notation, with optional decimal point and leading minus sign, is used for numbers. A power-of-ten scale factor can be included as a suffix. Here are some examples of legal numbers:

```
3                -99              0.0001
9.6397238        1.60210E-20      6.02252e23
```

On computers using IEEE floating point arithmetic, the relative accuracy of numbers is eps, which is about 16 significant decimal digits. The range is roughly 10^{-308} to 10^{308}.

Expressions can be built up using the usual arithmetic operators and precedence rules:

+	addition
–	subtraction
*	multiplication
/	right division
\	left division
^	power

The operations on matrices described later make it convenient to have the two symbols for division. The scalar expressions 1/4 and 4\1 have the same numerical value, namely 0.25. Parentheses are used in the standard way to alter the usual precedence of arithmetic operations.

Most ordinary elementary mathematical functions found on a good scientific calculator are built-in functions, for example, abs, sqrt, log, and sin, etc. More can be added easily with M-files. A later section has a complete list of functions.

1.6. Complex numbers and matrices

Complex numbers are allowed in MATLAB, but in order to enter them, it is first necessary to generate a complex unit. Some of us might use

```
i = sqrt(-1)
```

while others might prefer

```
j = sqrt(-1)
```

Once the complex unit has been defined, complex numbers can be generated with statements like

```
z = 3 + 4*i
```

or

```
w = r*exp(i*theta)
```

There are at least two convenient ways to enter complex matrices. They are illustrated by the statements

```
A = [1 2; 3 4] + i*[5 6; 7 8]
```

and

```
A = [1+5*i  2+6*i;  3+7*i  4+8*i]
```

which produce the same result. When complex numbers are being entered as matrix elements, it is important to avoid any blank spaces, because an expression like 1 + 5*i, with blanks surrounding the + sign, represents two separate numbers. (The same is true of real numbers; a blank before the exponent part in 1.23 e-4 causes an error.)

A statement generating the complex unit can be put in the file startup.m, in which case it is created automatically each time MATLAB is invoked. Other commonly used physical constants, or conversion factors, can also be placed in this file. See startup in the *reference section* for more information.

1.7. Output format

The result of any MATLAB assignment statement is displayed on the screen, as well as assigned to the specified variable, or to ans if no variable is given. The numeric display format can be controlled using the format command. Format affects only how matrices are printed, not how they are computed or saved (MATLAB performs all computation in double precision).

If all the elements of a matrix are exact integers, the matrix is printed with a format which does not have any decimal points. For example

```
x = [ -1 0 1 ]
```

always results in

```
x =
   -1    0    1
```

If at least one of the elements of a matrix is not an exact integer, there are several possible output formats. The default format, called the short format, shows about 5 significant decimal digits. The other formats show more significant digits, or use scientific notation. As an example, suppose

```
x = [ 4/3  1.2345e-6 ]
```

The formats, and the resulting output for this vector, are:

```
format short

   1.3333    0.0000

format short e

  1.3333E+000  1.2345E-006

format long

  1.333333333333338  0.000001234500000

format long e

  1.333333333333338E+000  1.234500000000003E-006
```

```
format hex

  3FF5555555555555   3EB4B6231ABFD271

format +

++
```

For the long formats, the last significant digit may appear to be incorrect, but the output is actually an accurate decimal representation of the binary number stored in the computer.

With the short and long formats, if the largest element of a matrix is larger than 1000 or smaller than 0.001, a common scale factor is applied to the entire matrix when it is printed. For example,

```
x = 1.e20*x
```

multiplies x by 10^{20} and results in the display

```
x =

 1.0E+020 *

   1.3333    0.0000
```

The + format is a compact way of displaying large matrices. The symbols +, – and blank are printed for positive, negative and zero elements.

One final command, format compact, suppresses many of the line-feeds that appear between matrix displays, and allows more information to be crowded on the screen.

1.8. The HELP facility

A HELP facility is available, providing on-line information on most MATLAB topics. To get a list of HELP topics, type

```
help
```

To get HELP on a specific topic, type help *topic*. For example,

```
help eig
```

provides HELP information on the use of the eigenvalue function,

```
help [
```

tells how to use brackets to enter matrices, and

```
help help
```

is self-referential, but works just fine.

1.9. Quitting and saving the workspace

To quit MATLAB, type quit or exit. Termination of a MATLAB session causes the variables in the workspace to be lost. Before quitting, the workspace may be saved for later use by typing

```
save
```

This saves all variables in a file on disk named matlab.mat. The next time MATLAB is invoked, the workspace may be restored from matlab.mat by typing

```
load
```

Save and load may be used with other file names, or to save only selected variables. The command save temp stores the current variables in the file named temp.mat. The command

```
save temp X
```

saves only variable X, while

```
save temp X Y Z
```

saves X, Y, and Z.

Load temp retrieves all the variables from the file named temp.mat. Load is also capable of reading ASCII data files; see the *reference section* for details.

2. Matrix operations

Matrix operations are fundamental to MATLAB; wherever possible they are indicated the way they would be in a textbook or on paper, subject only to the character set limitations of the computer.

2.1. Transpose

The special character prime ' denotes the transpose of a matrix with real entries. The statements

```
A = [ 1 2 3; 4 5 6; 7 8 0 ]
B = A'
```

result in

```
A =
   1   2   3
   4   5   6
   7   8   0

B =
   1   4   7
   2   5   8
   3   6   0
```

and

```
x = [ -1  0  2 ]'
```

produces

```
x =
    -1
     0
     2
```

The prime ' transposes in a formal matrix sense; if Z is a complex matrix, then Z' is its complex conjugate transpose. This can sometimes lead to unexpected results if used carelessly with complex data. To get an unconjugated transpose, use Z.' or conj(Z').

2.2. Addition and subtraction

Addition and subtraction of matrices are denoted by + and −. The operations are defined whenever the matrices have the same dimensions. For example, with the above matrices, A + x is not correct because A is 3-by-3 and x is 3-by-1. However,

```
C = A + B
```

is acceptable, and results in

```
C =
     2     6    10
     6    10    14
    10    14     0
```

Addition and subtraction are also defined if one of the operands is a scalar, that is a 1-by-1 matrix. In this case, the scalar is added to or subtracted from all the elements in the other operand. For example

```
y = x - 1
```

gives

```
y =
    -2
    -1
     1
```

2.3. Matrix multiplication

Multiplication of matrices is denoted by * . The operation is defined whenever the "inner" dimensions of the two operands are the same; that is X*Y is permitted if the second dimension of X is the same as the first dimension of Y. For example, the above x and y are both 3-by-1, so the expression x*y is NOT defined and results in an error message. However, several other vector products are defined, and are very useful. The most common is the inner product, also called the dot product or the scalar product. This is

```
x'*y
```

which results in

```
ans =
   4
```

Of course, `y'*x` would give the same result. There are two outer products, which are transposes of each other.

```
x*y'   =
     2     1    -1
     0     0     0
    -4    -2     2

y*x'   =
     2     0    -4
     1     0    -2
    -1     0     2
```

An element-by-element product is described in the next section. (There is no special provision in MATLAB for computing vector cross products. However, anyone needing cross products can easily write an M-file to compute them.)

Matrix-vector products are special cases of general matrix-matrix products. For our example `A` and `x`,

```
b = A*x
```

is allowed and results in the output

```
b =
     5
     8
    -7
```

Naturally, a scalar can multiply, or be multiplied by, any matrix.

```
pi*x

ans =
   -3.1416
    0.0000
    6.2832
```

2.4. Matrix division

There are two "matrix division" symbols in MATLAB, \ and /. If A is a nonsingular square matrix, then A\B and B/A correspond formally to left and right multiplication of B by the inverse of A , that is inv(A)*B and B*inv(A), but the result is obtained directly without the computation of the inverse. In general,

X = A\B is a solution to A*X = B

X = B/A is a solution to X*A = B

Left division, A\B, is defined whenever B has as many rows as A. If A is square, it is factored using Gaussian elimination. The factors are used to solve the equations A*X(:,j) = B(:,j) where B(:,j) denotes the *j*-th column of B. The result is a matrix X with the same dimensions as B. If A is nearly singular (according to the LINPACK condition estimator, RCOND), a warning message is printed.

If A is not square, it is factored using Householder orthogonalization with column pivoting. The factors are used to solve the under- or over-determined equations in a least squares sense. The result is an *m*-by-*n* matrix X where *m* is the number of columns of A and *n* is the number of columns of B . Each column of X has at most *k* non-zero components, where *k* is the effective rank of A.

Right division, B/A, can be defined in terms of left division by B/A = (A'\B')'.

For example, since our vector b was computed as A*x, the statement

```
z = A\b
```

results in

```
z =
    -1
     0
     2
```

Sometimes, the use of \ and / to compute least squares solutions to over- or under-determined systems of equations can cause surprises. It is possible to "divide" one vector by another. For example, with the above vectors x and y,

```
s = x\y
```

produces

```
s =
   0.8000
```

This is because $s = 0.8$ is the value of the scalar which solves the overdetermined equation $xs = y$ in a least squares sense. We invite the reader to explain why

```
S = y/x
```

gives

```
S =
    0.0000    0.0000   -1.0000
    0.0000    0.0000   -0.5000
    0.0000    0.0000    0.5000
```

2.5. Matrix powers

The expression A^p means A to the p-th power and is defined if A is a square matrix and p is a scalar. If p is an integer greater than one, the power is computed by repeated multiplication. For other values of p, the calculation involves eigenvalues and eigenvectors, such that if [V,D] = eig(A), then

```
A^p = V*D.^p/V
```

If P is a matrix, and a a scalar, a^P is a raised to the matrix power P using eigenvalues and eigenvectors. X^P, where both X and P are matrices, is an error.

2.6. Transcendental functions of matrices

In MATLAB, expressions like exp(A) and sqrt(A) are regarded as array operations, defined on the individual elements of A. MATLAB can also calculate matrix transcendental functions, such as the matrix exponential and matrix logarithm. These special functions are defined only for square matrices, are rather difficult and expensive to compute, and sometimes have subtle mathematical properties.

A transcendental mathematical function will be interpreted as a matrix function, if an "m" is appended to the function name, as in expm(A) and sqrtm(A). As MATLAB is distributed, only three of these functions are defined:

expm	matrix exponential
logm	matrix logarithm
sqrtm	matrix square root

However, the list can be extended by adding more M-files, or using funm. See the files sqrtm.m, logm.m, and funm.m in the *Utility Library*, and expm, funm in the *reference section.*

3. Array operations

We use the term "array operations" to refer to element-by-element arithmetic operations, instead of the usual linear algebraic matrix operations denoted by the symbols * / \ ^ '. Preceding an operator with a period "." indicates an array or element-by-element operation.

3.1. Array addition and subtraction

For addition and subtraction, the array operations and the matrix operations are the same, so + and – can be regarded as either matrix or array operations.

3.2. Array multiplication and division

Array, or element-by-element, multiplication is denoted by .* . If A and B have the same dimensions, then A .* B denotes the array whose elements are simply the products of the individual elements of A and B. For example, if

```
x = [ 1  2  3 ];  y = [ 4  5  6 ];
```

then

```
z = x .* y
```

results in

```
z =
   4   10   18
```

The expressions A ./ B and A .\ B give the quotients of the individual elements. So,

```
z = x .\ y
```

results in

```
z =
   4.0000   2.5000   2.0000
```

3.3. Array powers

Element-by-element powers are denoted by .^. Here are several examples, using the above vectors x and y. Typing

```
z = x .^ y
```

results in

```
z =
   1   32  729
```

The exponent can be a scalar.

```
z = x .^ 2

z =
   1   4   9
```

Or, the base can be a scalar.

```
z = 2 .^ [x y]

z =
   2   4   8   16   32   64
```

This last example illustrates one of MATLAB's syntactic subtleties. Although it is difficult to see, the space between the digit 2 and the period "." is important. If it was not there, the period would be interpreted as a decimal point associated with the 2. MATLAB would then see only the isolated "^" and would attempt to calculate a matrix power, which in this case would result in an error message, because the exponent matrix is non-square. An alternative to the space is to use parentheses, forcing the correct precedence.

3.4. Relational operations

There are six relational operators which can be used to compare two matrices of equal dimensions.

RELATIONAL OPERATORS	
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
~=	not equal

The comparison is done between the pairs of corresponding elements and the result is a matrix of 1's and 0's, with 1 representing TRUE and 0 FALSE. For example,

```
2+2 ~= 4
```

is simply 0.

Relational operators can show the patterns of matrix elements which satisfy various conditions. For example, here is the magic square of order 6.

```
A = magic(6)

A =
    35     1     6    26    19    24
     3    32     7    21    23    25
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
     4    36    29    13    18    11
```

A magic square of order n is an n-by-n matrix constructed from the integers from 1 through n^2 with equal row and column sums. If you stare at this particular matrix long enough, you might notice that the elements which are divisible by 3 occur on every third diagonal. To display this curiosity, we type

```
P = (rem(A,3) == 0)
```

The double = is the test-for-equality operator, rem(A,3) is a matrix of remainders, 0 is expanded to a matrix of zeros, and P becomes a matrix of 1's and 0's.

```
P =
   0   0   1   0   0   1
   1   0   0   1   0   0
   0   1   0   0   1   0
   0   0   1   0   0   1
   1   0   0   1   0   0
   0   1   0   0   1   0
```

To see the pattern a little more clearly, format + prints matrices in a compact form, with a + where there is a positive element, a – where there is a negative element, and a blank space for zeros.

```
format +
P

  + +
+ +
 + +
  + +
+ +
 + +
```

The function find is helpful with relational operators, finding nonzero elements in a 0-1 matrix, and hence the data elements that satisfy some relational condition. For example, if Y is a vector, find(Y < 3.0) returns a vector containing the indices of the elements in Y that are less than 3.0.

The relation X==NaN returns NaNs everywhere, since, according to the IEEE arithmetic specifications, any operation with a NaN results in NaN. But it is sometimes necessary to test for NaNs. So a function isnan(X) is provided that returns 1's where the elements of X are NaNs and 0's elsewhere. Also useful is finite(x), which returns 1's for $\infty < x < -\infty$.

3.5. Logical operations

There are three logical operators which work element-wise and are usually used on 0-1 matrices.

&	AND
|	OR
~	NOT

The & and | operators compare two scalars, or two matrices of equal dimensions. For matrices they work element-wise; if A and B are 0-1

matrices, then A & B is another 0-1 matrix representing the logical AND of the corresponding elements of A and B. The logical operators regard anything non-zero as TRUE. They return 1's where TRUE and 0's where FALSE.

NOT, or logical complement, is a unary operator. The expression ~A returns 0's where A is non-zero and 1's where A is zero. Thus the two expressions

```
P | (~P)
P & (~P)
```

return all 1's and all 0's, respectively.

The functions any and all are useful in conjuction with logical operators. If x is a 0-1 vector, any(x) returns 1 if *any* of the elements of x are non-zero, and returns 0 otherwise. The function all(x) returns a 1 only if *all* of the elements of x are non-zero. These functions are particulary useful in if statements,

```
if all(A < .5)
      do something
end
```

because an if wants to respond to a single condition, not a vector of possibly conflicting suggestions.

For matrix arguments, any and all work column-wise to return a row vector with the result for each column. Applying the function twice, as in any(any(A)), always reduces the matrix to a scalar condition.

3.6. Elementary math functions

A set of elementary mathematical functions are applied on an element-by-element basis to arrays. For example,

```
A = [1 2 3; 4 5 6]
B = fix(pi*A)
C = cos(pi*B)
```

produces

```
A =
   1   2   3
   4   5   6
```

```
B =
     3     6     9
    12    15    18

C =
    -1     1    -1
     1    -1     1
```

Here is a list of the elementary functions available. More can be added easily using M-files.

ELEMENTARY MATH FUNCTIONS	
abs	absolute value or complex magnitude
sqrt	square root
real	real part
imag	imaginary part
conj	complex conjugate
round	round to nearest integer
fix	round towards zero
floor	round towards $-\infty$
ceil	round towards ∞
sign	signum function
rem	remainder
sin	sine
cos	cosine
tan	tangent
asin	arcsine
acos	arccosine
atan	arctangent
atan2	four quadrant arctangent
sinh	hyperbolic sine
cosh	hyperbolic cosine
tanh	hyperbolic tangent
exp	exponential base e
log	natural logarithm
log10	log base 10
bessel	Bessel functions
gamma	gamma function
rat	rational approximation

4. Vectors and subscripts

The *subscripting* abilities of MATLAB allow manipulation of rows, columns, individual elements, and subportions of matrices. Central to subscripting are vectors, which are generated using "Colon Notation". Vectors and subscripting are used often and make it possible to achieve fairly complex data manipulation effects.

4.1. Generating vectors

The colon, :, is an important character in MATLAB. The statement

```
x = 1:5
```

generates a row vector containing the numbers from 1 to 5 with unit increment. It produces

```
x =
   1   2   3   4   5
```

Increments other than one can be used.

```
y = 0: pi/4: pi
```

results in

```
y =
   0.0000   0.7854   1.5708   2.3562   3.1416
```

Negative increments are possible.

```
z = 6:-1:1
```

gives

```
z =
   6   5   4   3   2   1
```

The colon notation allows the easy generation of tables. To get a vertical tabular form, transpose the row vector obtained from the colon notation, compute a column of function values, then form a matrix from the

two columns. For example

```
x = (0.0: 0.2: 3.0)';
y = exp(–x) .* sin(x);
[x y]
```

produces

```
ans =
   0.0000    0.0000
   0.2000    0.1627
   0.4000    0.2610
   0.6000    0.3099
   0.8000    0.3223
   1.0000    0.3096
   1.2000    0.2807
   1.4000    0.2430
   1.6000    0.2018
   1.8000    0.1610
   2.0000    0.1231
   2.2000    0.0896
   2.4000    0.0613
   2.6000    0.0383
   2.8000    0.0204
   3.0000    0.0070
```

4.2. Subscripting

Individual matrix elements may be referenced by enclosing their subscripts in parentheses. An expression used as a subscript is rounded to the nearest integer. For example, given a matrix A:

```
A =
     1     2     3
     4     5     6
     7     8     9
```

the statement

```
a(3,3) = a(1,3) + a(3,1)
```

results in

```
A =
     1     2     3
     4     5     6
     7     8    10
```

A subscript can be a vector. If X and V are vectors, then X(V) is [X(V(1)), X(V(2)), ..., X(V(n))]. For matrices, vector subscripts allow access to contiguous and noncontiguous submatrices. For example, suppose that A is a 10-by-10 matrix. Then

```
A(1:5,3)
```

specifies the 5-by-1 submatrix, or column vector, that consists of the first five elements in the third column of A. Similarly,

```
A(1:5,7:10)
```

is the 5-by-4 submatrix of elements from the first five rows and the last four columns.

Using the colon by itself in place of a subscript denotes *all* of the corresponding row or column. For example,

```
A(:,3)
```

is the third column and

```
A(1:5,:)
```

is the first five rows.

Fairly sophisticated effects are obtained using submatrix referencing on both sides of an assignment statement. For example,

```
A(:,[3 5 10]) = B(:,1:3)
```

replaces the third, fifth, and tenth columns of A with the first three columns of B.

In general, if v and w are vectors with integer components, then

```
A(v,w)
```

is the matrix obtained by taking the elements of A with row subscripts from

v and column subscripts from w. So

```
A(:,n:-1:1)
```

reverses the columns of A and

```
v = 2:2:n;
w = [3 1 4 1 6];
A(v,w)
```

is legal, but probably of questionable utility.

One more feature in this vein is useful in advanced fiddling, A(:). On the right hand side of an assignment statement, A(:) denotes all the elements of A strung out in a long column vector. So

```
A = [1 2; 3 4; 5 6]
b = A(:)
```

results in

```
A =
   1   2
   3   4
   5   6

b =
   1
   3
   5
   2
   4
   6
```

On the left hand side of an assignment statement, A(:) can be used to *reshape* or *resize* a matrix. To do this, A must already exist. Then A(:) denotes a matrix with the same dimensions as A, but with new contents from the right hand side. For example, the above A has three rows and two columns, so

```
A(:) = 11:16
```

reshapes the six element row vector into a 3-by-2 matrix,

```
A =
   11   14
   12   15
   13   16
```

4.3. Subscripting with 0-1 vectors

The 0-1 vectors usually created from relational operations can be used to reference submatrices. Suppose A is an *m*-by-*n* matrix and L is a length *m* vector of 0's and 1's. Then

```
A(L,:)
```

specifies the rows of A where the elements of L are non-zero.

Here is how "outliers" could be removed from a vector, those elements greater than 3 standard deviations:

```
x = x(x <= 3*std(x));
```

Similarly,

```
L = X(:,3) > 100;
X = X(L,:);
```

replaces X with those rows of X whose third column is greater than 100.

4.4. Empty matrices

The statement

```
x = [ ]
```

assigns a "matrix of dimension zero-by-zero" to x. Subsequent use of this matrix will NOT lead to an error condition; it will propagate empty matrices. This is different from the statement

```
clear x
```

which removes x from the list of current variables. Empty matrices *do* exist in the workspace; they just have zero size. The function exist can be used to test for the existence of a matrix (or a file for that matter), while size indicates if a matrix is empty.

It is possible to generate empty vectors. If n is less than 1, then 1:n contains no elements and so

```
x = 1:n
```

is a complicated way of creating an empty x.

More importantly, another way of removing rows and columns of a matrix is to assign them to an empty matrix. So

```
A(:,[2 4]) = [ ]
```

deletes columns 2 and 4 of X.

Certain matrix functions will return mathematically plausible values if given empty matrices. These include det, cond, prod, sum, and possibly others. For example, prod, det, and sum return 1, 1, and 0, respectively, when given null matrix arguments.

As far as we know, the literature on the algebra of empty matrices is itself empty. We're not sure we've done it correctly, or even consistently, but we have found the idea useful.

5. More fundamentals

Much of MATLAB's power is derived from its extensive set of functions. MATLAB has a large number of functions, well over 150 at the time of this writing. Some of the functions are intrinsic, or "built-in" to the MATLAB processor itself. Others are available in the library of external M-files distributed with MATLAB. It is transparent to the user whether a function is intrinsic or contained in an M-file. This is, of course, an important feature of MATLAB; a user can create his own new functions, and they act just like the intrinsic functions built into MATLAB. More on M-files, in a later section.

We have already seen some elementary mathematical and matrix functions in the previous sections. Here is a list of some general categories of functions that are available in MATLAB:

Elementary mathematical
Elementary matrix
Data analysis
Matrix building and manipulation
Matrix decompositions and factorizations
Polynomial
Signal processing

Subsequent sections will introduce these different categories of analytic functions. We will not, in the *Tutorial*, go into details on all the individual functions; this is done online by the help facility and in the *reference section.*

5.1. Functions with multiple arguments

MATLAB functions can be used with multiple arguments. Up until now, we have seen only functions with one input argument and returning one value. For example,

```
x = sqrt(log(z))
```

shows the nested use of two simple functions. There are MATLAB functions that use two or more input arguments. For example,

```
theta = atan2(y, x)
```

Of course, each one of the arguments could have been an expression.

Some functions *return* two or more output values. The output values are surrounded by brackets, [and], and separated by commas:

```
[V,D] = eig(A)
[y,i] = max(X)
```

The first function here returns two matrices, V and D, the eigenvectors and eigenvalues, respectively, of matrix A. The second example, using max, returns the maximum value y, and the index i of the maximum value in vector X.

Functions with multiple output arguments can be used with fewer output arguments. For example, max used with one output argument,

```
max(X)
```

returns just the maximum value.

5.2. Matrix building functions

Several matrix building functions create some handy utility matrices. These functions include eye(A), which returns an identity matrix of the same size as A. The catchy name is used because I and i are often used as subscripts or as sqrt(−1).

The group also includes zeros and ones, which generate constant matrices of various sizes, and rand, which generates matrices of uniformly or normally distributed random elements. For example, to generate a random 4-by-3 matrix

```
A = rand(4,3)
```

results in

```
A =
    0.2113    0.8096    0.4832
    0.0824    0.8474    0.6135
    0.7599    0.4524    0.2749
    0.0087    0.8075    0.8807
```

The three functions diag, triu and tril provide access to diagonal, upper triangular and lower triangular portions of matrices. For example,

```
tril(A)
```

produces

```
ans =
    0.2113         0         0
    0.0824    0.8474         0
    0.7599    0.4524    0.2749
    0.0087    0.8075    0.8807
```

Also very useful are size and length. The function size returns a two element vector containing the row and column dimensions of a matrix. If a variable is known to be a vector, length returns the length of the vector, or max(size(V)).

5.3. Building larger matrices

Larger matrices can be formed from smaller matrices by surrounding the smaller matrices with brackets, [and]. For example

```
C = [A  eye(4); ones(A)  A^2]
```

creates one such larger matrix, assuming that A has four rows. The smaller matrices in this type of construction must be dimensionally consistent, or an error message results.

5.4. Disk files

The commands dir, type, delete, and chdir implement a set of generic operating system commands for manipulating files. Here is a table that indicates how these commands map to other operating systems, perhaps one you are familiar with:

MATLAB	MS-DOS	UNIX	VAX/VMS
dir	dir	ls	dir
type	type	cat	type
delete	del	rm	delete
chdir	chdir	cd	set default

For most of these commands, pathnames, wildcards, and drive designators may be used in the usual way.

The type command differs from the usual type commands in an important way; if no file type is given, .m is used by default. This makes it convenient for the most frequent use of type, to list M-files on the screen.

The diary command can be used to save a diary of your MATLAB session in a disk file (graphics are not saved, however). The resulting ASCII file is suitable for inclusion into reports and other documents using any word-processor.

For more details on these commands, see the *reference section* or use the on-line help facility.

5.5. Running external programs

The exclaimation point character ! is a *shell escape*, used to indicate that the rest of the input line should be issued as a command to the operating system. This is quite useful for invoking utilities or running other programs without quitting from MATLAB. For example

```
!f77 simpleprog
```

invokes a Fortran compiler called F77

```
!translate
```

runs the translate program provided with MATLAB, and

```
!edt darwin.m
```

invokes an editor called edt on a file named darwin.m. After these programs complete, control is returned to MATLAB.

The exact behavior of ! depends upon the machine you are using. See the machine specific section 1 for more information.

6. Data analysis

This section presents an introduction to data analysis using MATLAB and describes some elementary statistical tools. More powerful techniques are available using the linear algebra and signal processing functions covered in later sections.

6.1. Column-oriented analysis

Matrices are, of course, used to hold all data, but this leaves a choice of orientation for multivariate data. By convention, the different variables in a set of data are put in columns, allowing observations to vary down through the rows. A data set consisting of 50 samples of 13 variables would be stored in a matrix of size 50-by-13.

Starting an example, the ever-present Longley econometric data consist of the variables

1) GNP deflator
2) GNP
3) Unemployment
4) Armed Forces
5) Population
6) Year
7) Employment

In general, there are several methods for getting data into MATLAB; these are covered in detail in a later section. Assuming that the data are not already in a machine readable form, the easiest way to enter the data is using a text editor or word processor. If we create a file called longley.m that contains the assignment statement

```
ldata = [
 83.0   234.289   235.6   159.0   107.608   1947   60.323
 88.5   259.426   232.5   145.6   108.632   1948   61.122
 88.2   258.054   368.2   161.6   109.773   1949   60.171
 89.5   284.599   335.1   165.0   110.929   1950   61.187
 96.2   328.975   209.9   309.9   112.075   1951   63.221
 98.1   346.999   193.2   359.4   113.270   1952   63.639
 99.0   365.385   187.0   354.7   115.094   1953   64.989
100.0   363.112   357.8   335.0   116.219   1954   63.761
101.2   397.469   290.4   304.8   117.388   1955   66.019
104.6   419.180   282.2   285.7   118.734   1956   67.857
```

```
108.4   442.769   293.6   279.8   120.445   1957   68.169
110.8   444.546   468.1   263.7   121.950   1958   66.513
112.6   482.704   381.3   255.2   123.366   1959   68.655
114.2   502.601   393.1   251.4   125.368   1960   69.564
115.7   518.173   480.6   257.2   127.852   1961   69.331
116.9   554.894   400.7   282.7   130.081   1962   70.551 ];
```

we can type the command longley. This accesses longley.m and creates the matrix called ldata (or any other name of your choice) in the workspace. You could try entering this matrix interactively, but the chances are remote that you would get it correct the first time. If you were to make a mistake resulting in an error message, there would be no way to correct it without starting over. (Unless your version of MATLAB has provisions for editing previous lines - see the computer specific section).

If there are more observations than will fit across the screen, rows can be continued to the next line using the ellipsis .. consisting of two periods. The matrix could also be entered in blocks of columns and the whole thing pieced together at the end.

For the Longley data there are 16 observations of 7 variables. This is revealed by

```
[n,p] = size(ldata)

n =
   16
p =
    7
```

For data entered in this column-wise fashion, a group of functions provide basic data analysis capabilities:

DATA ANALYSIS	
max	maximum value
min	minimum value
mean	mean value
std	standard deviation
median	median value
sort	sorting
sum	sum of elements
prod	product of elements
cumsum	cumulative sum of elements
cumprod	cumulative product of elements
diff	difference functions
hist	histograms
table1	table look-up
corr	correlation matrix
cov	covariance matrix
any	logical conditions
all	logical conditions
find	find array indices of logical values

For vector arguments, these functions don't care whether the vectors are oriented in a row or column direction. For array arguments, the functions operate in a *column oriented* fashion on the data in the arrays. This means, for example, that if max is applied to an array, the result is a row vector containing the maximum values over each column.

Thus if

```
A =
   9   8   4
   1   6   5
   3   2   7
```

then

```
x = max(A)
mv = mean(A)
s = sort(A)
```

results in

```
m =
   9   8   7

sv =
   4.3333   5.3333   5.3333

s =
   1   2   4
   3   6   5
   9   8   7
```

Or for the Longley data

```
m = median(ldata)

m =
 1.0E+003 *

0.1012  0.3975  0.3351  0.2798  0.1174  1.9550  0.0660
```

More functions can be added to this list using M-files, but when doing so, *care must be exercised to handle the row vector case*. If you are writing your own column-oriented M-files, you should look at how this is accomplished in other M-files, for example mean.m and diff.m.

6.2. Missing values

The special value, NaN, stands for *Not-a-Number* in MATLAB. Normally, it is produced by undefined expressions like 0/0, instead of an error message, thanks to conventions established by the IEEE floating point arithmetic standard. For statistics purposes, NaNs can be used to represent missing values or NAs, data that are *not available*.

The "correct" handling of NAs is a difficult problem and often varies in different situations. MATLAB, however, is uniform and rigorous in its treatment of NaNs; they propagate naturally through to the final result in any calculation. Thus if a NaN is used in any intermediate computation, the final result will be a NaN, unless the final result does not depend on the value of the NaN, were it to have one.

What this means, practically, is that NaNs should be removed from the data before statistical computions are performed. The NaNs in a vector x can be found with

```
i = find(isnan(x));
```

so

```
x = x(find(~isnan(x)))
```

returns the data in x with the NaNs removed. Two other ways of doing the same thing are

```
x = x(~isnan(x));
x(isnan(x)) = [ ];
```

of which the second is perhaps the most straightforward. We have been using the special function isnan to find NaNs because we can *not* use

```
x(x == NaN) = [ ];
```

NaNs return NaN as the result of all operations, including relational ones.

If, instead of a vector, the data are in the columns of a matrix, and any *rows* of the matrix with NaNs are to be removed, we could use instead,

```
X(any(isnan(X)'),:) = [ ];
```

which is a particularly nasty, but effective statement. If you object that you will have a hard time remembering this, you are completely justified. If you frequently need to remove NaNs, the solution is to write a short M-file, for example

```
function X = excise(X)
X(any(isnan(X)'),:) = [ ];
```

Now, typing

```
X = excise(X);
```

accomplishes the same thing. More on M-files later.

6.3. Removing outliers

Outliers in a data set are removed in much the same manner as NaNs. For the Longley data, the mean and standard deviations of each column of data are:

```
mv   = mean(ldata)
sigma = std(ldata)

mv =
  1.0E+003 *

 0.101  0.387  0.319  0.260  0.117  1.954  0.065

sigma =

 10.448  96.238  90.479  67.382  6.735  4.609  3.400
```

The number of rows who have outliers greater than 3 standard deviations is

```
[n,p] = size(ldata);
e = ones(n,1);
dist = abs(ldata – e*mv);
outliers = dist > 3*e*sigma;
nout = sum(any(outliers'))

nout =
   0
```

There are none. If there were, they could be removed with

```
X(any(outliers'),:) = [ ];
```

6.4. Regression and curve fitting

Before attempting to fit a curve to data, the data should be normalized. Normalization can improve the accuracy of the final results. Still working with our Longley data, one way to normalize is to remove the mean

```
X = X – e*mean(X);
```

and to normalize to unit standard-deviation

```
X = X ./ (e*std(X));
```

We can regress unemployment (the last column) on the earlier columns, using in this case our raw data,

```
y = ldata(:,7);
A = [ldata(:,1:6) ones(y)];
coef = A\y
```

which results in

```
coef =

  1.0E+003 *

    0.00001506187227
   -0.00003581917929
   -0.00002020229804
   -0.00001033226867
   -0.00005110410565
    0.00182915146461
   -3.48225863459802
```

The Longley data are known to be highly correlated, which we see by looking at the correlation coefficients.

```
corr(X)

ans =

1.0000  0.9916  0.6206  0.4647  0.9792  0.9911  0.9709
0.9916  1.0000  0.6043  0.4464  0.9911  0.9953  0.9836
0.6206  0.6043  1.0000 -0.1774  0.6866  0.6683  0.5025
0.4647  0.4464 -0.1774  1.0000  0.3644  0.4172  0.4573
0.9792  0.9911  0.6866  0.3644  1.0000  0.9940  0.9604
0.9911  0.9953  0.6683  0.4172  0.9940  1.0000  0.9713
0.9709  0.9836  0.5025  0.4573  0.9604  0.9713  1.0000
```

Often it is useful to find a polynomial fit to data. In general, a polynomial fit to data in vectors x and y is a function, p, of the form:

$$p(x) = c_1 x^d + c_2 x^{d-1} + \cdots + c_n$$

The degree is d and the number of coefficients is $n = d+1$. The coefficients $c_1, c_2, \cdots c_n$ are determined by solving a system of simultaneous linear equations:

$$Ac = y$$

The columns of A are successive powers of the x vector. Here is one way to create A

```
for j=1:n
      A(:,j) = x.^(n-j);
end
```

The solution to the system of simultaneous linear equations $Ac = y$ is obtained using MATLABs matrix division operator:

```
c = A\y
```

The function polyfit.m in the *Utility Library* automates this procedure.

In the *regression* problem, other functions, usually multivariate functions of the columns of the data matrix, are fit to the data by forming the appropriate A matrix. For example, using the Longley data,

```
A = [ldata(:,1) ldata(:,2).^2 sin(ldata(:,3)) ones(n,1)];
coef = A\y;
```

finds the regression coefficients for a more complicated function.

7. Matrix functions

Much of MATLAB's mathematical power is derived from its matrix functions. Some of the functions are "built-in" to the MATLAB processor itself. Others are available in the library of external M-files distributed with MATLAB. And some have been added by individual users, or groups of users, for more specialized applications. We will not go into details about the individual functions here; that is done in the help facility and the *reference section.* Further information is also available in the User's Guides for the LINPACK and EISPACK software packages, which provide the algorithmic foundation for MATLAB. In this section we will just give an overview of the available functions by grouping according to the underlying factorization and decompositions. There are four such groups:

Triangular factorization
Orthogonal factorization
Eigenvalue decomposition
Singular value decomposition

7.1. Triangular factorization

The most basic factorization expresses any square matrix as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the "LU", or sometimes the "LR", factorization. Most of the algorithms for computing it are variants of Gaussian elimination.

The factors themselves are available from the lu function. The factorization is used to obtain the inverse with inv and the determinant with det. It is also the basis for the linear equation solution or "matrix division" obtained with \ and / for square matrices. For example, start with

```
A =
   1   2   3
   4   5   6
   7   8   0
```

To see the LU factorization, we use MATLAB's double assignment statement.

```
[L,U] = lu(A)
```

which gives

```
L =
    0.1429    1.0000    0
    0.5714    0.5000    1.0000
    1.0000    0         0

U =
    7.0000    8.0000    0.0000
    0         0.8571    3.0000
    0         0         4.5000
```

Notice that L is a permutation of a lower triangular matrix that has ones on the permuted diagonal, and that U is upper triangular. To check that the factorization does its job, we can compute the product

```
L*U
```

which should give us back the original A. It does.

```
ans =
   1   2   3
   4   5   6
   7   8   0
```

The inverse of the example matrix can be obtained with

```
X = inv(A)
```

The inverse is actually computed from the inverses of the triangular factors

```
X = inv(U)*inv(L)
```

The determinant of the example matrix can be obtained with

```
d = det(A)
```

which gives

```
d =
   27
```

It is computed from the determinants of the triangular factors

```
d = det(L)*det(U)
```

which produce

```
d =
  27.0000
```

Why do the two d's print in different formats? When MATLAB is asked to compute det(A) it makes note of the fact that all the elements of A are integers, so it forces the determinant to be an integer. But in calculating the second d, the elements of U are not integers, so MATLAB does not produce an exact integer result.

As an example of a system of simultaneous linear equations, take

```
b =
   1
   3
   5
```

The solution to the equation $Ax = b$ is obtained with the MATLAB matrix division operation

```
x = A\b
```

which produces

```
x =
   0.3333
   0.3333
   0.0000
```

The solution is actually computed by solving two triangular systems,

```
y = L\b, x = U\y
```

The intermediate solution is

```
y =
   5.0000
   0.2857
   0.0000
```

Triangular factorization is also used by a specialized function, rcond. This is a quantity produced by several of the LINPACK subroutines that is an estimate of the reciprocal condition number of a square matrix.

Two other functions, chol and rref, can be included in this group because the underlying algorithms are closely related to LU factorization. The function chol produces the Cholesky factorization of a symmetric, positive definite matrix. The reduced row echelon form of a rectangular matrix, rref, is of some interest in theoretical linear algebra, although it has little computational value. It is included in MATLAB for pedagogical reasons.

7.2. Orthogonal factorization

The "QR" factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of an orthonormal matrix and an upper triangular matrix. For example, take

```
A =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

We have chosen a rank-deficient matrix; the middle column is the average of the other two columns. The rank deficiency is revealed by the factorization.

```
[Q,R] = qr(A)
```

gives

```
Q =
   -0.0776   -0.8331    0.5444    0.0605
   -0.3105   -0.4512   -0.7709    0.3251
   -0.5433   -0.0694   -0.0913   -0.8317
   -0.7762    0.3124    0.3178    0.4461
```

```
R =
  -12.8841   -14.5916   -16.2992
    0         -1.0413    -2.0826
    0          0          0.0000
    0          0          0
```

We could check that the product Q*R produces the original A, but let's not bother. The triangular structure of R gives it zeros below the diagonal; the zero on the diagonal in R(3,3) implies that R and consequently A do not have full rank.

The QR factorization is used in solving linear systems with more equations than unknowns. For example

```
b =
   1
   3
   5
   7
```

The linear system $Ax = b$ is four equations in only three unknowns. The best solution in a least squares sense is computed by

```
x = A\b
```

which produces

```
Warning: Rank deficient, rank = 2  tol = 1.4594E-014

x =
   0.5000
   0.0000
   0.1667
```

We are warned about the rank deficiency. The quantity tol is a tolerance used in deciding that a diagonal element of R is negligible. The solution x was computed using the factorization and the two steps

```
y = Q'*b;
x = R\y
```

If we were to check the computed solution by forming A*x, we would find that it equals b to within roundoff error. This tells us that even though the

simultaneous equations $Ax = b$ are overdetermined and rank deficient, they happen to be consistent. There are infinitely many solution vectors x; the QR factorization has found just one of them.

This factorization is also the basis for the functions null and orth, which generate orthonormal bases for the null space and range of a given rectangular matrix.

7.3. Singular value decomposition

We will not attempt to explain the singular value decomposition here; we have to be content with claiming that it is a powerful tool for analysis of problems involving matrices. See the *LINPACK User's Guide* or the book by Golub and VanLoan for some justification of this claim. In MATLAB, the triple assignment

```
[U,S,V] = svd(A)
```

produces the three factors in the singular value decomposition,

```
A = U*S*V'
```

The matrices U and V are orthogonal and the matrix S is diagonal. By itself, the function svd(A) returns just the diagonal elements of S, which are the singular values of A.

The singular value decomposition is used by several other functions, including the pseudo-inverse, pinv(A); the rank, rank(A); the Euclidean matrix norm, norm(A,2); and the condition number, cond(A).

7.4. Eigenvalues

If A is an n-by-n matrix, the n numbers λ that satisfy $Ax = \lambda x$ are the *eigenvalues* of A. They are found using

```
eig(A)
```

which returns the eigenvalues in a column vector. If A is real and symmetric, the eigenvalues will be real. But if A is not symmetric, the eigenvalues are frequently complex numbers. For example, with

```
A =
     0     1
    -1     0
```

The statement eig(A) produces

```
ans =
   0.0000 + 1.0000 i
   0.0000 - 1.0000 i
```

Eigenvalues and eigenvectors can be obtained with a double assignment statement,

```
[X,D] = eig(A)
```

in which case the diagonal elements of D are the eigenvalues and the columns of X are the corresponding eigenvectors such that A*X = X*D.

Two intermediate results used in computing eigenvalues are the Hessenberg form, hess(A), and the Schur form, schur(A). The Schur form is used to compute transcendental mathematical functions of matrices, such as sqrtm(A) and logm(A).

If A and B are square matrices, the function eig(A,B) returns a vector containing the *generalized eigenvalues* solving the equation

$$Ax = \lambda Bx$$

The double assignment can be used to obtain eigenvectors

```
[X,D] = eig(A,B)
```

producing a diagonal matrix D of generalized eigenvalues and a full matrix X whose columns are the corresponding eigenvectors so that A*X = B*X*D. The intermediate results involved in the solution of this generalized eigenvalue problem are available from qz(A,B).

7.5. Rank

There are several different places in MATLAB where the rank of a matrix is implicitly computed: in rref(A), in A\B for non-square A, in orth(A) and null(A), and in the pseudoinverse pinv(A). Three different algorithms with three different criteria for negligibility are used and so it is possible that three different values could be produced for the same matrix.

With rref(A), the rank of A is the number of nonzero rows. The elimination algorithm used for rref is the fastest of the three rank-determining algorithms, but it is the least sophisticated numerically and the least reliable.

With A\B, orth(A), and null(A), the QR factorization is used as described in chapter 9 of the LINPACK guide.

With pinv(A) the algorithm is based on the singular value decomposition and is described in chapter 11 of the LINPACK guide. The pinv algorithm is the most time-consuming, but the most reliable and is therefore also used for the explicit rank computation, rank(A).

8. Signal processing and polynomials

MATLAB has functions for working with polynomials, and functions for digital signal processing. These functions operate primarily on vectors.

8.1. Polynomials

Polynomials are represented in MATLAB as row vectors containing the coefficients ordered by descending powers. For example, the characteristic equation of the matrix

```
A =
   1   2   3
   4   5   6
   7   8   0
```

is found with poly,

```
p = poly(A)
```

resulting in

```
p =
   1   -6   -72   -27
```

The roots of this equation are found using roots,

```
r = roots(p)

r   =
   12.1229
   -5.7345
   -0.3884
```

which are, of course, the same as the eigenvalues of matrix A. These may be reassembled into a polynomial with poly,

```
p2 = poly(r)

p2 =
   1   -6   -72   -27
```

In addition to roots and poly, other polynomial functions include polyval and polyvalm, for evaluating polynomials, conv, for multiplying polynomials, and residue for partial-fraction expansion.

8.2. Signal processing

Vectors are used to hold sampled-data signals, or sequences, for signal processing. For multi-input systems, each row corresponds to a sample point, with the observations spread across the columns of the matrix. Here is a list of some functions for signal processing:

SIGNAL PROCESSING	
fft	radix-2 discrete Fourier transform
ifft	inverse transform
fftshift	rearrange FFT results
filter	direct filter implementation
freqz	*z*-transform frequency response
freqs	Laplace-transform frequency response
conv	convolution
deconv	deconvolution
xcorr	cross-correlation function
cov	covariance

Some of these have 2-dimensional counterparts, in which case the "signal" is actually a matrix:

2-D SIGNAL PROCESSING	
fft2	two-dimensional FFT
ifft2	inverse 2-d FFT
fftshift	rearrange FFT results
conv2	2-D convolution
xcorr2	2-D cross-correlation function

There are more signal processing functions available, but they will not be discussed here. This section is intended to be a brief introduction to the signal processing capabilities of MATLAB; a comprehensive description is found in the separate SIGNAL PROCESSING TOOLBOX user's guide.

8.2.1. Filtering

The function

```
y = filter(b, a, x)
```

filters the data in vector x with the filter described by vectors a and b,

creating filtered data y. The filter structure is the general tapped delay-line filter described by the difference equation:

$$y(n) = b(1)x(n) + b(2)x(n-1) + \cdots + b(nb)x(n-nb+1) - a(2)y(n-1) - \cdots - a(na)y(n-na+1)$$

or equivalently, the Z-transform:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb)z^{-(nb-1)}}{1 + a(2)z^{-1} + \cdots + a(na)z^{-(na-1)}}$$

For example, to find and plot the n-point unit impulse response of a digital filter:

```
x = [1 zeros(1,n-1)];
y = filter(b,a,x);
plot(y,'o')
```

The function freqz returns the complex frequency response of digital filters. The frequency response is $H(z)$ evaluated around the unit circle in the complex plane, $z = e^{j\omega}$. Here is how freqz is used to find and plot an n-point frequency response:

```
[h,w] = freqz(b,a,n);
mag = abs(h);
phase = angle(h);
semilogy(w,mag), plot(w,phase)
```

The function freqs can be used to calculate the frequency response of analog filters.

Several functions are available in the SIGNAL PROCESSING TOOLBOX for designing digital filters. We will have to be content here to claim that with some knowledge of filter design techniques, many methods are possible. For example, MATLAB's built-in complex arithmetic allows techniques like bilinear transformation and pole-zero mapping to convert *s-domain* prototypes into the *z-domain*. Also, FIR filters are designed easily using windowing techniques.

8.2.2. FFT

Fft(x) is the discrete Fourier transform (DFT) of vector x, computed with a radix-2 fast-Fourier transform algorithm. If the length of x is not an exact power of two then it is padded with trailing zeros. Ifft(x) is the inverse discrete Fourier transform of vector x.

The two functions implement the transform - inverse transform pair given by:

$$X(k+1) = \sum_{n=0}^{N-1} x(n+1)W_N^{kn}$$

$$x(n+1) = 1/N \sum_{k=0}^{N-1} X(k+1)W_N^{-kn}$$

where $W_N = e^{-j(2\pi/N)}$ and $N = length(x)$. Note that the series is written in an unorthodox way, with the subscripts from $n+1$ and $k+1$ instead of the usual n and k because subscripts on MATLAB vectors run from 1 to N instead of from 0 to $N-1$.

In order to plot FFT results on a meaningful frequency axis, the relationship between the FFT bins and the actual frequency points needs to be known. Suppose a sequence of N points is obtained at a sample frequency of f_s. Then, for up to the Nyquist frequency, or point $n = N/2+1$, the relationship between the bin number and the actual frequency is:

$$f = (bin\_number - 1)*f_s/N$$

Here's a simple example that plots the spectrum of some data stored in vector X:

```
Y = fft(X);
N = length(Y);
f = Fs * (0:N-1) / N;
plot(f,abs(Y))
```

Another example involves the algorithm employed by the function freqz. When n is a power of two, freqz calculates the frequency response using an FFT algorithm. The frequency reponse is the ratio of the FFT's of the numerator and denominator coefficients padded with zeros:

```
Hb = fft([b zeros(1,n-length(b))]);
Ha = fft([a zeros(1,n-length(a))]);
H = Hb ./ Ha;
```

Suppose we have two statistically related sampled data sequences x and y. Here is how MATLAB can be used to calculate some spectral analysis functions:

SPECTRAL FUNCTIONS	
X = fft(x)	X Discrete Fourier transform
Y = fft(y)	Y Discrete Fourier transform
Pxx = abs(X/N).^2	X Power spectral density
Pyy = abs(Y/N).^2	Y Power spectral density
Pxy = Y.*conj(X)/N^2	Cross spectral density
Tyx = Pxy ./ Pxx	Transfer function
Cxy = (abs(Pxy).^2)./(Pxx.*Pyy)	Coherence function

Of course these functions are often averaged across adjacent records or "chunks" of, say, N=512 points. It is also possible to *window* the data. Windows are used to truncate long duration signals into groups of N points, often for FFT processing. The simplest window, the boxcar or rectangular window, is equivalent to truncating a vector. More complicated windows are possible. The *Hanning window* defined as

$$w(n) = .5(1 - cos(2\pi n/(N-1)), \; 0 \leq n \leq N-1$$

is expressed in MATLAB, for a 512-point window, as

```
n = 0:511;
w = .5 * (1 - cos(2*pi*n/511));
```

and used to window a sequence with

```
xw = x(1:512) .* w;
```

In the SIGNAL PROCESSING TOOLBOX a function called spectrum assembles these capabilities into an automated procedure for spectral analysis. Spectrum uses FFTs and windowing to calculate the power spectral densities, transfer function and coherence function of a pair of statistically related signals. The function specplot shows these results graphically.

For more information on functions used here, see the *reference section*. If you plan on doing lots of signal processing, see the SIGNAL PROCESSING TOOLBOX user's guide.

9. Graphing

Scientific and engineering data are examined graphically in MATLAB using "graph paper" commands to create plots on the screen. There are six different types of "graph paper" from which to choose:

GRAPH PAPER	
plot	linear X-Y plot
loglog	loglog X-Y plot
semilogx	semi-log X-Y plot (x-axis logarithmic)
semilogy	semi-log X-Y plot (y-axis logarithmic)
polar	polar plot
mesh	3-dimensional mesh surface
bar	bar chart

Once a graph is on the screen, the graph may be labeled, titled, or have grid lines drawn in:

title	graph title
xlabel	x-axis label
ylabel	y-axis label
text	label data points
grid	grid lines

Finally, there are commands for screen control, manual axis scaling, and hardcopy on a printer:

shg	show graph screen
clc	clear command screen
clg	clear graph screen
home	home command cursor
axis	manual axis scaling
hold	hold the plot on the screen
subplot	break graph screen into subwindows
prtsc	screen dump hardcopy
print	send graph to printer
meta	graphics metafile

9.1. X-Y plots

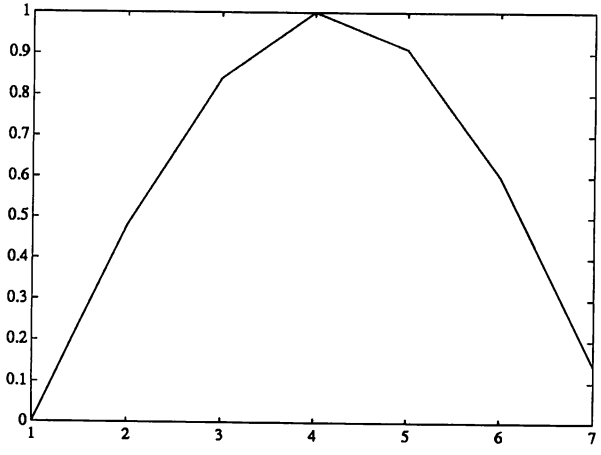
The plot command creates linear X-Y plots. Once the plot command is understood, logarithmic and polar plots are created by substituting the commands loglog, semilogx, semilogy, or polar for plot. All five commands are used the same way; they only affect how the axis is scaled and how the data are displayed.

9.2. Basic form

If Y is a vector, plot(Y) produces a linear plot of the elements of Y versus the index of the elements of Y. For example, suppose we wish to plot the numbers {0., .48, .84, 1., .91, .6, .14}. This is accomplished with

```
Y = [0. .48 .84 1. .91 .6 .14];
plot(Y)
```

which results in a graph on your screen:



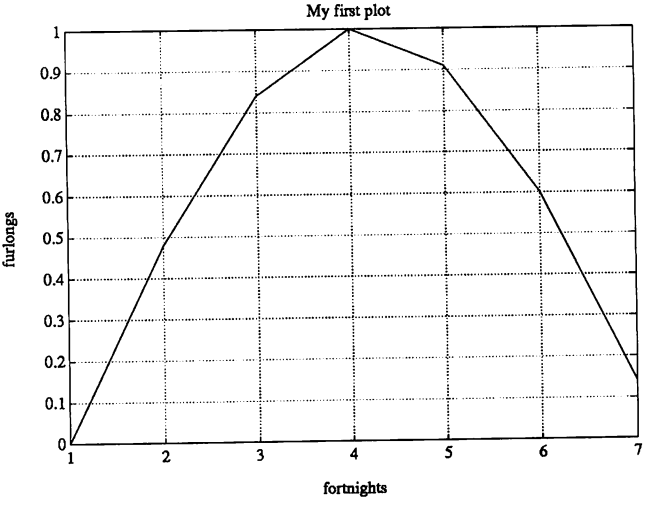
Notice that the data are auto-scaled and that X and Y axes are drawn.

At this point, depending upon the exact hardware you are using, the screen full of commands that you have typed in may have vanished to make way for the graph display. MATLAB has two displays, a graph display and a command display. Some hardware configurations allow both to be seen simultaneously, while others allow only one to be seen at a time. If the command display is no longer there, it can be brought back by striking any key on the keyboard.

Once the command display has been brought back, a graph title, X and Y labels, and grid lines can be put on the plot by successively entering the commands

```
title('My first plot')
xlabel('fortnights')
ylabel('furlongs')
grid
```

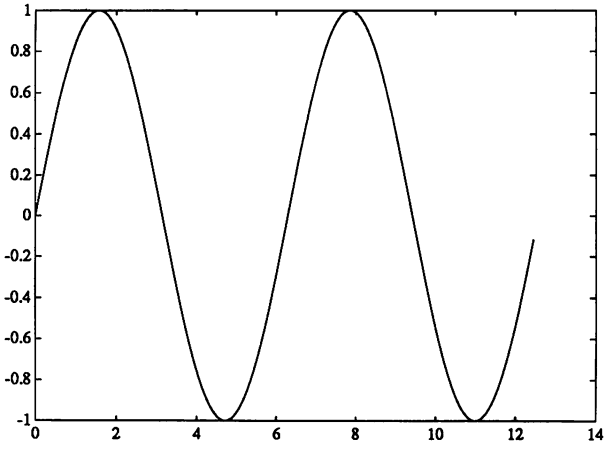
This results in the display:



If X and Y are vectors of the same length, the command plot(X,Y) draws an X-Y plot of the elements of X versus the elements of Y. For example,

```
t = 0:.05:4*pi;
y = sin(t);
plot(t,y)
```

results in



9.3. Multiple lines

There are two methods for plotting multiple lines on a single graph. First, if plot is used with two arguments, plot(X,Y), and either X, Y, or both, are matrices, then:

[1] If Y is a matrix, and X a vector, plot(X,Y) successively plots the rows or columns of Y versus the vector X, using a different linetype for each. The row or column "direction" of Y is selected that has the same number of elements as the X vector. If Y is a square matrix, the column direction is arbitrarily used.

[2] If X is a matrix, and Y a vector, then the above rules are applied except the family of lines from X are plotted versus vector Y.

[3] If X and Y are both matrices of the same size, plot(X,Y) plots the columns of X versus the columns of Y.

[4] If no X is specified, as in plot(Y), where Y is a matrix, then lines are plotted for each column of Y versus the row index.

The second, and easier, method is to use plot with multiple arguments:

```
plot(X1,Y1,X2,Y2,...,Xn,Yn)
```

where X1,Y1 X2,Y2 etc. are pairs of vectors. Each X-Y pair is plotted, generating multiple lines on the plot. Multiple arguments have the benefit of allowing vectors of differing lengths to be displayed on the same graph. As before, different linetypes are used for each pair.

9.4. Line and mark styles

9.4.1. Type

The line-types used on a graph may be controlled if the defaults are not satisfactory. Point plots using various symbols may also be selected. For example,

```
plot(X,Y,'x')
```

draws a point plot using x-mark symbols while

```
plot(X1,Y1,':',X2,Y2,'+')
```

uses a dotted line for the first curve and the plus symbol + for the second curve. Other line and point types are:

LINE-TYPES		POINT-TYPES	
solid	-	point	.
dashed	--	plus	+
dotted	:	star	*
dashdot	-.	circle	o
		x-mark	x

9.4.2. Color

On systems that support color, line- and mark-colors may be specified in a manner similar to line- and mark-types. For example, the statements

```
plot(X,Y,'r')
plot(X,Y,'+g')
```

use a red line on the first graph and green +-marks on the second. Other colors are:

COLORS	
red	r
green	g
blue	b
white	w
invisible	i

If your hardcopy device does not support color, the various colors on the interactive display are mapped to different line-types for output.

9.5. Imaginary and complex data

When plot is used with data that are complex (have non-zero imaginary parts), normally the imaginary part is ignored. There is a special case, however, in which the result is a shortcut to a plot of the real part versus the imaginary part. The special case arises when plot is used with a single argument, as in plot(Z), and matrix or vector Z is complex. In this case, plot(Z) is equivalent to plot(real(Z),imag(Z)).

To plot multiple lines in the complex plane, there is no shortcut, and the real and imaginary parts must be taken explicitly.

9.6. Logarithmic, polar, and bar plots

The use of loglog, semilogx, semilogy, and polar is identical to the use of plot. These commands allow data to be plotted on different types of "graph paper", i.e. in different coordinate systems:

- Polar(theta, rho) makes a plot using polar coordinates of the angle theta, in radians, versus the radius rho. Subsequent use of the grid command draws polar grid lines.
- Loglog makes a plot using $\log_{10}$-$\log_{10}$ scales.
- Semilogx makes a plot using semi-log scales. The X-axis is $\log_{10}$ while the Y axis is linear.
- Semilogy makes a plot using semi-log scales. The Y-axis is $\log_{10}$ while the X axis is linear.

Bar(x) displays a bar chart of the elements of vector x. Bar does not accept multiple arguments.

9.7. Three dimensional mesh surface plots

The statement mesh(Z) creates a three dimensional perspective plot of the elements in matrix Z. A mesh surface is defined by the Z coordinates of points above a rectangular grid in the X-Y plane. The plot is formed by joining adjacent points with straight lines.

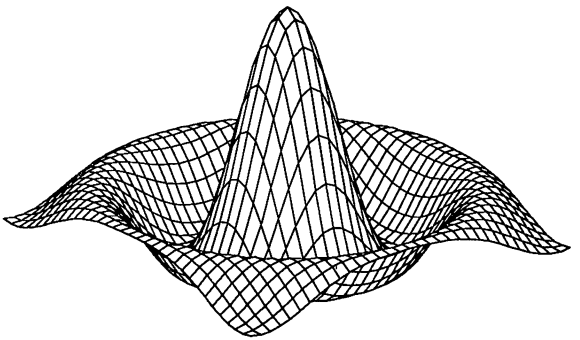
Mesh can be used to visualize large matrices that are otherwise too large to print out in numerical form. It can also be used to graph functions of two variables.

The first step towards displaying a function $z = f(x,y)$ of two variables is to generate special X and Y matrices that consist of repeated rows and columns, respectively, over the domain of the function. The function can then be evaluated directly and graphed.

Consider the *sin(r)/r* or *sinc* function that results in the "sombrero" surface that everybody likes to show. One way to create this is:

```
x = -8: .5: 8;
y = x';
X = ones(y)*x;
Y = y*ones(x);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(Z)
```

The first command defines the x-domain over which the function is to be evaluated. The third statement creates a matrix X of repeated rows. After generating a corresponding Y, a matrix R is created containing the distance from the center of the matrix, which is the origin. Forming the *sinc* function and applying mesh results in



What does an identity matrix look like as a mesh surface? Try mesh(eye(14)). For an easier method of generating the special X and Y matrices used for evaluating functions of two variables, see the M-file meshdom.m or look in the *reference section.* Also handy is rot90, which rotates matrices in increments of 90 degrees, and flipx and flipy.

9.8. Screen control

MATLAB conceptually has two displays, a *graph window* and a *command window.* Your particular hardware configuration may allow both to be seen simultaneously in different windows, or it may allow only one to be seen at a time. Several commands are available to switch back and forth

between the windows, and/or erase the windows, as required:

shg	show graph window
any key	bring back command window
clc	clear command window
clg	clear graph window
home	home command cursor

For example, if during your MATLAB session only the command display is on the screen, typing shg will recall the last plot that was drawn on the graphing display.

By default, hardware configurations that cannot display both the command screen and the graph screen simultaneously will pause in graphics mode after a plot is drawn and wait for a key to be hit.

It is possible to split the graph window into multiple partitions, in order to show several plots at the same time. The statement subplot(mnp) breaks the graph window into an *m*-by-*n* grid and uses the *p*'th box for the subsequent plot. For example,

```
subplot(211), plot(abs(y))
subplot(212), plot(angle(y))
```

breaks the screen in two, plots the magnitude of a complex vector in the top half, and plots the phase in the bottom half. The command subplot(111), or just subplot, returns to the default single whole-screen window.

9.9. Manual axis scaling

In certain situations, it may be desirable to override the automatic axis scaling feature of the plot command and to manually select the plotting limits. Typing axis by itself freezes the current axis scaling for subsequent plots. Typing axis again resumes auto-scaling. Axis returns a 4 element row vector containing the [x_min, x_max, y_min, y_max] used on the last plot. Axis(V) where V is a 4 element vector sets the axis scaling to the prescribed limits.

A second use of axis is to control the aspect ratio of the plot on the screen. Axis('square') sets the plot region on the screen to be square. With a square aspect ratio, a line with slope 1 is at a true 45 degrees, not skewed by the irregular shape of the screen. Also, circles, like plot(sin(t),cos(t)), look like circles instead of an ovals. Axis('normal') sets the aspect ratio back to normal.

The hold command holds the current graph on the screen. Subsequent plot commands will add to the plot, using the already established axis

limits, and retaining the previously plotted curves. Hold remains in effect until issued again.

9.10. Hardcopy

The easiest way to obtain graphics hardcopy, on many personal computers, is to hold the Shift key down and to press the PrtSc key. This sends a screen dump of the picture in the graph window to the printer.

Three commands, prtsc, print and meta, provide more general hardcopy capability:

prtscr — Prtsc initiates a graph window screen dump, like Shift-PrtSc, and allows it to be done in an M-file or in a for loop. In general, this results in a low resolution plot, because the pixels on the screen are transferred to pixels on a printer.

meta — Meta *file* opens a graphics metafile, using the specified filename, and writes the current graph to it for later processing. Subsequent meta commands append to the previously specified filename. The metafile may be processed later, using the graphics post processor (GPP) program.

print — Print sends a high resolution copy of the current plot to the printer. On some machines with limited memory, this feature may not be available.

See the machine specific section for more information on how hardcopy is obtained on your machine.

10. Control flow

MATLAB has control flow statements, like those found in most computer languages. The control flow statements carry MATLAB beyond the level of a simple desk calculator, allowing it to be used as a complete high-level matrix language.

10.1. FOR loops

MATLAB has its own version of the "do" or "for" loop found in computer languages. It allows a statement, or group of statements, to be repeated a fixed, predetermined number of times. For example

```
for i = 1:n, x(i) = 0, end
```

assigns 0 to the first n elements of x. If n is less than 1, the construction is still legal, but the inner statement will not be executed. If x does not yet exist, or has fewer than n elements, then additional space will be allocated automatically.

The loops can be nested and are usually indented for readability.

```
for i = 1:m
   for j = 1:n
      A(i,j) = 1/(i+j-1);
   end
end
A
```

The semicolon terminating the inner statement suppresses repeated printing and the A after the loops displays the final result.

An important point: each for *must* be matched with an end. If you simply type

```
for i = 1:n, x(i) = 0
```

the system will patiently wait for you to enter the remaining statements in the body of the loop. Nothing will happen until you type the end. (Users of the older versions of MATLAB where the end was not required will find themselves especially vulnerable to this pitfall.)

For another example, assume

```
t =
    -1
     0
     1
     3
     5
```

and that we want to generate the following matrix whose columns are powers of elements of t.

```
A =
     1    -1     1    -1     1
     0     0     0     0     1
     1     1     1     1     1
    81    27     9     3     1
   625   125    25     5     1
```

Here is the most obvious double loop.

```
n = max(size(t));
for j = 1:n
   for i = 1:n
      A(i,j) = t(i)^(n-j);
   end
end
```

But the following single loop with vector operations is significantly faster and also illustrates the fact that for loops can go backwards.

```
A(:,n) = ones(n,1);
for j = n-1:-1:1
   A(:,j) = t .* A(:,j+1);
end
```

The general form of a for loop is

```
for v = expression
  statements
end
```

The *expression* is actually a matrix, because that's all there is in MATLAB. The columns of the matrix are assigned one by one to the variable v and then the *statements* are executed. Another, clearer, way of accomplishing almost the same thing is

```
E = expression;
[m,n] = size(E);
for j = 1:n
  v = E(:,j);
  statements
end
```

Usually, the *expression* is something like m:n, or m:i:n, which is a matrix with only one row, and so its columns are simply scalars. In this special case, the for loop is like the "for" or "do" loops of other languages.

10.2. WHILE loops

MATLAB also has its version of the "while" loop, which allows a statement, or group of statements, to be repeated an indefinite number of times, under control of a logical condition. Here is a simple problem to illustrate a while loop. What is the first integer *n* for which *n!* (that is *n* factorial) is a 100 digit number? The following while loop will find it. If you don't already know the answer, you can run this yourself.

```
n = 1;
while prod(1:n) < 1.e100, n = n+1; end
n
```

A more practical computation illustrating while is the calculation of the exponential of a matrix, called expm(A) in MATLAB. One possible definition of the exponential function is the power series,

```
expm(A) = I + A + A^2/2! + A^3/3! + ...
```

It is reasonable to use this for the actual computation as long as the elements of A are not too large. The idea is to sum as many terms of this series as are needed to produce a result which would not change if more terms are added in finite precision machine arithmetic. In the following loop, A is the given matrix, E will become the desired exponential, F is an individual term in the series, and k is the index of that term. The indented statements will be repeated until F is so small that adding it to E doesn't change E.

```
E = zeros(A);
F = eye(A);
k = 1;
while norm(E+F-E,1) > 0
  E = E + F;
  F = A*F/k;
  k = k+1;
end
```

If we want to compute the array or element-by-element exponential exp(A) instead, we just have to change the initialization of F from eye(A) to ones(A) and change the matrix multiplication A*F to array multiplication A.*F.

The general form of a while loop is

```
while expression
  statements
end
```

The *statements* are executed repeatedly as long as all of the elements in the *expression matrix* are nonzero. The expression matrix is almost always a 1-by-1 relational expression, so nonzero corresponds to TRUE. When the expression matrix is not a scalar, it can be reduced using the functions any and all.

10.3. IF and BREAK statements

Here are a couple of examples illustrating MATLAB's if statements. The first shows how a computation might be broken down into three cases, depending upon the sign and parity of n.

```
if n < 0
  A = negative(n)
elseif mod(n,2) == 0
  A = even(n)
else
  A = odd(n)
end
```

The second example involves a fascinating problem from number theory. Take any positive integer. If it is even, divide it by 2; if it is odd, multiply it by 3 and add 1. Repeat this process until your integer becomes a one. The fascinating unsolved problem is: Is there any integer for which

the process does not terminate? Our MATLAB program illustrates while and if statements. It also shows the input function, which prompts for keyboard input, and the break statement, which provides an exit jump out of loops.

```
% Classic "3n+1" problem from number theory.
while 1
  n = input('Enter n, negative quits. ');
  if n <= 0, break, end
  while n > 1
    if rem(n,2) == 0
      n = n/2
    else
      n = 3*n+1
    end;
  end
end
```

Can you make this run forever?



11. M-files: Scripts and Functions

MATLAB is usually used in a command driven mode; when single-line commands are entered, MATLAB processes them immediately and displays the results. MATLAB is also capable of executing sequences of commands that are stored in files. Together, these two modes form an interpretive environment.

Disk files that contain MATLAB statements are called *M-files* because they have a file type of ''.m'' as the last part of the filename. For example, a file named bessel.m might contain MATLAB statements that evaluate Bessel functions.

An M-file consists of a sequence of normal MATLAB statements, possibly including references to other M-files. An M-file can call itself recursively.

One use of M-files is to automate long sequences of commands. Such files are called *script files* or just *scripts*. A second type of M-file provides extensibility to MATLAB. Called *function files*, they allow new functions to be added to the existing functions. Much of the power of MATLAB derives from this ability to create new functions that solve user-specific problems.

Both types of M-files, *scripts* and *functions*, are normal ASCII text files, and are created using an editor or word-processor of your choice.

11.1. Script files

When a *script* is invoked, MATLAB simply executes the commands found in the file, instead of waiting for input from the keyboard. The statements in a *script file* operate globally on the data in the workspace. *Scripts* are useful for performing analyses, solving problems, or doing designs that require such long sequences of commands that they become cumbersome to do interactively.

As an example, suppose the MATLAB commands

```
% An M-file to calculate Fibonnaci numbers
f = [1 1];  i = 1;
while f(i) + f(i+1) < 1000
      f(i+2) = f(i) + f(i+1);
      i = i + 1;
end
plot(f)
```

are contained in a file called fibno.m. Typing the statement fibno causes MATLAB to execute the commands, calculate the first 1000 Fibonnaci numbers, and create a plot. After execution of the file is complete, the variables f and i remain in the workspace.

The demos supplied with MATLAB are good examples of using *scripts* to perform more complicated tasks.

11.2. Function files

If the first line of an M-file contains the word "function", the file is a *function file*. A *function file* differs from a *script* in that arguments may be passed, and that variables defined and manipulated inside the file are local to the function and do not operate globally on the workspace. *Function files* are useful for extending MATLAB, that is, creating new MATLAB functions using the MATLAB language itself.

Here is a simple example. The file mean.m on your disk contains the statements:

```
function y = mean(x)
% MEAN  Average or mean value. For vectors,
%       MEAN(x) returns the mean value. For
%       matrices, MEAN(x) is a row vector
%       containing the mean value of each column.
[m,n] = size(x);
if m == 1
     m = n;       % Handle isolated row vector.
end
y = sum(x) / m;
```

The existence of this disk file defines a new function called mean. The new function mean is used just like any other MATLAB function. For example, if Z is a vector of the integers from 1 to 99,

```
Z = 1:99;
```

the mean value is found by typing

```
mean(Z)
```

which results in

```
ans =
   50
```

Let's examine some of the details of mean.m:

- The first line declares the function name, the input arguments, and the output arguments. Without this line, the file would be a *script file*, instead of a *function file*.
- The % symbol is used to indicate that the rest of a line is a comment and should be ignored.
- The first few lines document the M-file and will be displayed if help mean is typed.
- The variables m, n, and y are local to mean and will not exist in the workspace after mean has finished. (Or, if they previously existed, they will be unchanged.)
- It was not necessary to put our integers from 1 to 99 in a variable with the name x. In fact, we used mean with a variable called Z. The vector Z that contained the integers from 1 to 99 was passed or copied into mean where it became a local variable named x.

Here is a slightly more complicated version of mean called stat that calculates standard deviation too:

```
function [mean,stdev] = stat(x)
[m,n] = size(x);
if m == 1
    m = n;      % Handle isolated row vector.
end
mean = sum(x) / m;
stdev = sqrt(sum(x.^2) / m - mean.^2);
```

Stat illustrates that it is possible to return multiple output arguments.

A final function that calculates the rank of a matrix uses multiple input arguments:

```
function r = rank(x,tol)
% rank of a matrix
s = svd(x);
if (nargin == 1)
    tol = max(size(x)) * s(1) * eps;
end
r = sum(s > tol);
```

This example demonstrates the use of the permanent variable nargin to find the number of input arguments. The variable nargout, although not used here, contains the number of output arguments.

Some helpful hints:

When a *M-function file* is used for the first time during a MATLAB session, it is compiled and placed into memory. It is then available for subsequent use without recompilation. It remains in memory for the duration of the session, unless you run low on free memory, in which case it may be cleared automatically.

The what command shows a directory listing of the M-files that are available in the current directory on your disk, type lists M-files, and ! is used to invoke your editor, allowing you to create or modify M-files.

In general, if you input the name of something to MATLAB, for example by typing whoopie, the MATLAB interpreter goes through the following steps:

[1] Looks to see if whoopie is a variable.

[2] Checks if whoopie is a built-in function.

[3] Looks in the current directory for a file named whoopie.m.

[4] Looks in the directories specified by the environment symbol MATLABPATH for a file named whoopie.m. (See the installation instructions to learn how to set MATLABPATH)

Thus MATLAB first tries to use whoopie as a variable, if it exists, before trying to use whoopie as a function.

11.3. Echo, input, pause, keyboard

Normally, while an M-file is executing, the commands in the file are not displayed on the screen. A command called echo allows M-files to be viewed as they execute, which can be useful for debugging, or for demonstrations. See the *reference section* for further details.

The function input obtains input from the user. Input('How many apples') gives the user the prompt in the text string, waits, and then returns the number or expression input from the keyboard. One use of input is to build menu driven M-files. The demo facility is an example of this.

Similar to input, but more powerful, is keyboard. This function invokes your computer keyboard as a script. Placed in M-files, this feature is useful for debugging, or for modifying variables during execution.

A command called pause causes a procedure to stop and wait for the user to strike any key before continuing. Pause(n) pauses for n seconds before continuing.

It is also possible to define global variables, although we don't recommend it. See the *reference section* if you insist.

11.4. Strings and string macros

Text strings are entered into MATLAB surrounded by single quotes. For example,

```
s = 'Hello'
```

results in

```
s =

Hello
```

The text is stored in a vector, one character per element. In this case,

```
size(s)

ans =
   1    5
```

indicates that s has 5 elements. The characters are stored as their ASCII values and abs shows these values,

```
abs(s)

ans =
  72  101  108  108  111
```

The function setstr sets the vector to display as text instead of showing

ASCII values. Also useful are disp, which simply displays the text in the variable, and sprintf, num2str, and int2str which convert numbers to strings.

Text variables can be concatenated into larger strings using brackets, for example,

```
s = [s, ' World']

s =

Hello World
```

Eval is a function that works with text variables to implement a simple text macro facility. Eval(t) causes the text contained in t to be evaluated. If STRING is the source text for any MATLAB expression or statement, then

```
t = 'STRING';
```

encodes the text in t. Typing t prints the text and eval(t) causes the text to be interpreted, either as a statement or as a factor in an expression. For example

```
t = '1/(i+j-1)';
for i = 1:n
    for j = 1:n
        a(i,j) = eval(t);
    end
end
```

generates the Hilbert matrix of order n. Another example showing indexed text,

```
S = ['x = 3                    '
     'y = 4                    '
     'z = sqrt(x*x+y*y)'];
for k=1:3
  eval(S(k,:));
end
```

It is necessary that the strings making up the "rows" of the "matrix" S have the same lengths. Here is a final example showing how eval can be

used with the load command to load ten sequentially numbered data files:

```
fname = 'mydata';
for i=1:10
      eval(['load ',fname,int2str(i)])
end
```

The text macro facility is particularly useful for passing function names to M-function files. For an example, see the file funm.m in the *Utility Library*.

11.5. External programs

It is possible, and often useful, to make your own external stand-alone programs act like new MATLAB functions. This can be done by writing M-files that

[1] Save the variables on disk,

[2] Run the external programs (which read the data files, process them, and write the results back out to disk), and

[3] Load the processed files back into the workspace.

For example, here is a hypothetical M-function that finds the solution to Garfield's equation using an external program called GAREQN

```
function y = garfield(a,b,q,r)
save gardata a b q r
!gareqn
load gardata
```

It requires that you have written a program (in Fortran or some other language) called GAREQN that reads a file called gardata.mat, processes it, and puts the results back out to the same file. The utility subroutines described in the next section can be used to read and write MAT-files.

This facility is the preferred mechanism for "linking your own code" to MATLAB. On many systems, it is really quite fast. It is certainly more convenient than physically linking new object code into the program, an option that, if it is available for your machine, is described in the machine specific section 1.



12. Importing and exporting data

Data from other programs and the outside world can be introduced into MATLAB using several different methods. Similarly, MATLAB data can be exported to the outside world. It is also possible to have your programs manipulate data directly in the file format used by MATLAB.

The best method depends upon how much data there are, whether the data are already in machine readable form, what the form is, etc. Here are some choices; select the one that looks appropriate.

[1] Enter it as an explicit list of elements. If you have a small amount of data, say less than 10-15 elements, it is easy to type in the data explicitly using brackets, [and]. This method is awkward for larger amounts of data because you can't edit your input if you make a mistake. See *section 1.1*.

[2] Create it in an M-file. Use your text editor to create a script M-file that enters your data as an explicit list of elements. This method is good when your data are not already in computer-readable form and you have to type them in anyway. Essentially the same as method 1, it has the advantage of allowing you to use your editor to change the data or to fix mistakes. You can then just re-run the M-file to re-enter the data.

[3] Load it from an ASCII flat file. If the data are stored in ASCII form, with fixed length rows terminated with newlines (carriage returns), and with spaces separating the numbers, then the file is a so called *flat file*. (ASCII flat files can be edited using a normal text editor.) Flat files can be read directly into MATLAB using the load command. The result is put into a variable whose name is the filename.

[4] Translate your data file to the MATLAB file format and use the load command. A program called translate is supplied with MATLAB in the *Utility Library*. The translate program converts ASCII flat files, binary files, Fortran unformatted files, and DIF files (Data Interchange Format from spreadsheets, etc.) to the special MAT-files used by MATLAB. This method is best if you have a large amount of data and they are already in a disk file on your computer.

Here are some methods of getting MATLAB data back to the outside world:

[1] For small matrices, use the diary command to create a diary file, and then list the variables on this file. You can use your text editor to manipulate the diary file at a later time. The output of diary includes

the MATLAB commands used during the session, which is useful for inclusion into documents and reports.

[2] Save your variables using the save command, quit MATLAB, and use the translate program to convert the MAT-file to one of the other available file formats.

You may also choose to have your programs manipulate the data directly in the MAT-files used by the MATLAB load and save commands. The format of MAT-files is documented under load in the *reference section* of this guide.

If you program in Fortran, there are some routines provided in the *Utility Library* to help you interface your Fortran programs to MAT-files:

FORMAT.FOR	An example Fortran program that writes a MAT-data file.
SAVEMAT.FOR	A subroutine call that writes MAT-files.
LOADMAT.FOR	A subroutine call that reads MAT-files.
TESTLS.FOR	An example of using SAVEMAT and LOADMAT.

These routines should be easy to convert to other programming languages.

13. References

[1] J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[2] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, C. B. Moler, *Matrix Eigensystem Routines -- EISPACK Guide*, Lecture Notes in Computer Science, volume 6, second edition, Springer-Verlag, 1976.

[3] B. S. Garbow, J. M. Boyle, J. J. Dongarra, C. B. Moler, *Matrix Eigensystem Routines -- EISPACK Guide Extension*, Lecture Notes in Computer Science, volume 51, Springer-Verlag, 1977.

[4] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, 1983.

Reference

Reference

This section contains detailed descriptions of all MATLAB functions. It begins with a list of functions grouped by subject area and continues with the reference entries in alphabetical order. Information is also available through the on-line help facility.

GENERAL	
help	help facility
demo	run demonstrations
who	list current variables in memory
what	list M-files on disk
size	row and column dimensions
length	vector length
clear	clear workspace
^C	local abort
quit	terminate program
exit	same as quit

MATRIX OPERATORS		ARRAY OPERATORS	
+	addition	+	addition
-	subtraction	-	subtraction
*	multiplication	.*	multiplication
/	right division	./	right division
\	left division	.\	left division
^	power	.^	power
'	conjugate transpose	.'	transpose

RELATIONAL AND LOGICAL OPERATORS			
<	less than	&	AND
<=	less than or equal	|	OR
>	greater than	~	NOT
>=	greater than or equal		
==	equal		
~=	not equal		

SPECIAL CHARACTERS	
=	assignment statement
[	used to form vectors and matrices
]	see [
(	arithmetic expression precedence
)	see (
.	decimal point
..	continue statement to the next line
,	separate subscripts and function arguments
;	end rows, suppress printing
%	comments
:	subscripting, vector generation
!	execute operating system command

SPECIAL VALUES	
ans	answer when expression is not assigned
eps	floating point precision
pi	π
inf	∞
NaN	Not-a-Number
clock	wall clock
nargin	number of function input arguments
nargout	number of function output arguments

DISK FILES	
chdir	change current directory
delete	delete file
diary	diary of the session
dir	directory of files on disk
load	load variables from file
save	save variables on file
translate	data translation
type	list function or file
what	show M-files on disk

COMMAND WINDOW	
format	set output display format
disp	display matrix or text
fprintf	print formatted number
clc	clear command screen
home	home cursor
echo	enable command echoing

GRAPHING WINDOW	
plot	linear X-Y plot
loglog	loglog X-Y plot
semilogx	semi-log X-Y plot
semilogy	semi-log X-Y plot
polar	polar plot
mesh	3-dimensional mesh surface
meshdom	domain for mesh plots
bar	bar charts
grid	draw grid lines
title	plot title
xlabel	x-axis label
ylabel	y-axis label
text	label data points
axis	manual axis scaling
hold	hold plot on screen
shg	show graph screen
clg	clear graph screen
subplot	split graph window
print	send graph to printer
prtsc	screen dump
meta	graphics metafile

CONTROL FLOW	
if	conditionally execute statements
elseif	used with **if**
else	used with **if**
end	terminate **if, for, while**
for	repeat statements a number of times
while	do while
break	break out of **for** and **while** loops
return	return from functions
pause	pause until key hit

PROGRAMMING AND M-FILES	
input	get numbers from keyboard
keyboard	call keyboard as M-file
error	display error message
function	define function
eval	interpret text in variables
echo	enable command echoing
exist	check if variables exist
casesen	set case sensitivity
global	define global variables
startup	startup M-file

TEXT AND STRINGS	
abs	convert string to ASCII values
eval	evaluate text macro
num2str	convert number to string
int2str	convert integer to string
setstr	set flag indicating matrix is a string
sprintf	convert number to string

ELEMENTARY MATH FUNCTIONS	
abs	absolute value or complex magnitude
angle	phase angle
sqrt	square root
real	real part
imag	imaginary part
conj	complex conjugate
round	round to nearest integer
fix	round towards zero
floor	round towards $-\infty$
ceil	round towards ∞
sign	signum function
rem	remainder
sin	sine
cos	cosine
tan	tangent
asin	arcsine
acos	arccosine
atan	arctangent
atan2	four quadrant arctangent
sinh	hyperbolic sine
cosh	hyperbolic cosine
tanh	hyperbolic tangent
exp	exponential base e
log	natural logarithm
log10	log base 10
bessel	Bessel function
gamma	gamma function
rat	rational approximation

RELATIONAL AND LOGICAL FUNCTIONS	
any	logical conditions
all	logical conditions
find	find array indices of logical values
isnan	detect NaNs
finite	detect infinities
isempty	detect empty matrices

UTILITY MATRICES	
diag	diagonal matrices
eye	identity matrices
ones	constant matrices
zeros	zero matrices
rand	random matrices
logspace	logarithmically spaced vectors
magic	magic square
tril	lower triangular part
triu	upper triangular part
toeplitz	Toeplitz matrices
rsf2csf	convert real-schur to complex

MATRIX CONDITIONING	
cond	condition number in 2-norm
det	determinant
norm	1-norm, 2-norm, F-norm, ∞-norm
rank	rank
rcond	condition estimate

DECOMPOSITIONS AND FACTORIZATIONS	
chol	Cholesky factorization
eig	eigenvalues and eigenvectors
hess	Hessenberg form
inv	inverse
lu	factors from Gaussian elimination
null	null space
orth	orthogonalization
pinv	pseudoinverse
qr	orthogonal-triangular decomposition
qz	QZ algorithm
rref	reduced row echelon form
schur	Schur decomposition
svd	singular value decomposition

OTHER MATRIX FUNCTIONS	
expm	matrix exponential
logm	matrix logarithm
sqrtm	matrix square root
funm	arbitrary matrix functions
poly	characteristic polynomial
kron	Kronecker tensor product

POLYNOMIALS	
poly	characteristic polynomial
roots	polynomial roots
polyval	polynomial evaluation
polyvalm	matrix polynomial evaluation
conv	multiplication
deconv	division
residue	partial-fraction expansion
polyfit	polynomial curve fitting

COLUMN-WISE DATA ANALYSIS	
max	maximum value
min	minimum value
mean	mean value
std	standard deviation
median	median value
sort	sorting
sum	sum of elements
prod	product of elements
cumsum	cumulative sum of elements
cumprod	cumulative product of elements
diff	approximate derivatives
hist	histograms
table1	table look-up
corr	correlation matrix
cov	covariance matrix
spline	cubic spline interpolation

SIGNAL PROCESSING	
abs	complex magnitude
conv	convolution
conv2	2-D convolution
cov	covariance
deconv	deconvolution
dft	discrete Fourier transform
fft	radix-2 fast Fourier transform
fft2	two-dimensional FFT
idft	inverse transform
ifft	inverse fast Fourier transform
ifft2	inverse 2-D FFT
fftshift	FFT rearrangement
filter	direct filter implementation
freqz	digital frequency response
freqs	analog frequency response
xcorr	cross-correlation function
xcorr2	2-D cross-correlation

Purpose:

Absolute value.
Phase angle.

Synopsis:

```
abs(x)
angle(x)
```

Description:

Abs(Z) is the absolute value of the elements of Z. If Z is complex, abs returns the complex modulus (magnitude):

```
abs(Z) = sqrt(real(Z).^2 + imag(Z).^2)
```

Angle(Z) returns the phase angles, in radians, of the elements of complex matrix Z.

For complex $z = x + iy = re^{i\theta}$, the magnitude and phase are given by

```
r = abs(z)
theta = angle(z)
```

and the statements

```
i = sqrt(-1)
z = r.*exp(i*theta)
```

convert back to the original complex z.

Algorithm:

Angle is an M-file in the *Utility Library*. Angle can be expressed as:

```
angle(x) = imag(log(x)) = atan2(imag(x),real(x))
```

Purpose:

Test arrays for logical conditions.

Synopsis:

```
any(x)
all(x)
```

Description:

For vectors, any(v) returns 1 if *any* of the elements of the vector are non-zero. Otherwise it returns 0. For matrices, any(X) operates on the columns of X, returning a row vector of 1's and 0's.

For vectors, all(v) returns 1 if *all* of the elements of the vector are non-zero. Otherwise it returns 0. For matrices, all(X) operates on the columns of X, returning a row vector of 1's and 0's.

Examples:

These functions are particulary useful in if statements,

```
if all(A < .5)
    do something
end
```

because an if wants to respond to a single condition, not a vector of possibly conflicting suggestions. Applying the function twice, as in any(any(A)), always reduces the matrix to a scalar condition.

See also:

& | ~

Purpose:

Manual axis scaling on plots.

Synopsis:

axis
axis([xmin xmax ymin ymax])
axis('*aspect format*')

The *aspect format* may be square or normal.

Description:

Axis is used to set manual axis scaling on plots. Typing axis by itself freezes the current axis scaling for subsequent plots. Typing axis again resumes auto-scaling. Axis returns a 4 element row vector containing the [xmin, xmax, ymin, ymax] used on the last plot.

Axis(V), where V is a 4 element vector, sets the axis scaling to the prescribed limits.

For logarithmic plots, the elements of V are $\log_{10}$ of the minimums and maximums.

Axis('square') sets the plot region on the screen to be square. With a square aspect ratio, a line with slope 1 is at a true 45 degrees, not skewed by the irregular shape of the screen. Also, circles, like plot(sin(t),cos(t)), look like circles instead of an ovals. Axis('normal') sets the aspect ratio back to normal.

See also:

plot, hold

Purpose:

Bessel and gamma functions.

Synopsis:

bessel(alpha,X)
gamma(p)

Description:

The differential equation

$$x^2\ddot{y} + x\dot{y} + (x^2 - \alpha^2)y = 0$$

where α is a non-negative constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*. Bessel evaluates Bessel functions of the first kind. Bessel(alpha,X) evaluates the Bessel function of order alpha, $J_\alpha(x)$, at every element in array X.

Gamma(p) returns the gamma function evaluated at scalar p.

$$\Gamma(p) = \int t^{p-1}e^{-t}dt, \; p > 0$$

Examples:

Plot Bessel functions of order 0 and 1,

```
x=0:.25:10;
plot(x,[bessel(0,x);bessel(1,x)])
```

Algorithm:

Both bessel and gamma are M-files in the *Utility Library*. Bessel is computed using a power series expansion. The computation of gamma is based on an algorithm outlined in [1].

References:

[1] W. J. Cody, *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, 1975, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

Purpose:

Break out of control structures.
Return from M-functions.
Display error messages.

Synopsis:

break
return
error('*text*')

Description:

Break terminates the execution of for and while loops. For nested loops, break exits from the innermost loop only.

Return causes a normal return to the invoking function or to the keyboard.

Error(s) displays the text in s and causes an error return to the keyboard.

Examples:

Here is a contrived procedure to calculate machine epsilon that is really a while loop in disguise:

```
eps = 1;
for i=1:1000
      eps = eps/2; if (eps+1 <= 1), break, end
end
eps = eps*2
```

Error is used for error returns from M-files,

```
function foo(x,y)
if nargin ~= 2
      error('Wrong number of input arguments')
end
do_normal_things
```

See also:

while, if, for, end, disp

Purpose:

Select/deselect case-sensitivity.

Synopsis:

casesen

Description:

Casesen controls the case-sensitivity of MATLAB. Initially MATLAB is case-sensitive; it is able to distinguish between uppercase and lowercase variable and function names.

Typing casesen toggles the case-sensitivity.

Examples:

When MATLAB is case-sensitive, the variable VAR is different from the variable var. When MATLAB is not case-sensitive, the two names refer to the same variable.

Purpose:

Cholesky factorization.

Synopsis:

chol(X)

Description:

Chol computes the Cholesky factorization of a matrix. If X is positive definite, then

```
R = chol(X)
```

produces an upper triangular R so that R'*R = X. It uses only the diagonal and upper triangle of X; the lower triangular is assumed to be the (complex conjugate) transpose of the upper.

Algorithm:

Chol uses the algorithm from the LINPACK subroutine ZPOFA. For a detailed write up on the use of the Cholesky decomposition, see chapter 8 of the *LINPACK User's Guide*.

Diagnostics:

If X is not positive definite,

```
Matrix must be positive definite.
```

References:

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose:

Command window control.
Graph window control.

Synopsis:

clc
clg
shg
home

Description:

MATLAB conceptually has two displays, a *graph window* and a *command window*. Your particular hardware configuration may allow both to be seen simultaneously in different windows, or it may allow only one to be seen at a time. Several commands are available to switch back and forth between the windows, and/or erase the windows, as required:

SCREEN CONTROL	
shg	show graph window
any key	bring back command window
clc	clear command window
clg	clear graph window
home	home command cursor

By default, hardware configurations that cannot display both the command screen and the graph screen simultaneously will pause in graph mode after a plot is drawn and wait for any key to be hit.

Examples:

Show a "movie" of a random matrix:

```
clc, for i=1:25, home, a = rand(5), end
```

See also:

plot, hold, subplot

Purpose:

Remove items from memory.

Synopsis:

clear
clear *name*
clear *name1 name2 name 3 ...*
clear functions

The *name* can be either a variable-name or a function-name.

Description:

Typing clear by itself clears all variables and compiled functions from the workspace. It leaves the workspace empty, as if MATLAB were just invoked.

Clear X removes just variable or function X from the workspace.

Clear X Y Z removes X, Y, and Z from the workspace.

Assigning a variable to null does NOT clear it (This is a change from earlier versions of MATLAB). For example, X=[] frees the space occupied by the data, but leaves behind a matrix of dimension zero named X.

The statement clear functions clears all the currently compiled M-functions from memory.

See also:

pack

Purpose:

Wall clock and elapsed-time.

Synopsis:

```
clock
etime(t2,t1)
```

Description:

Clock returns a six element row vector containing the current time and date in decimal form:

```
clock = [year month day hour minute seconds]
```

The first five elements are integers. The *seconds* element is accurate to several digits beyond the decimal point.

Etime calculates the elapsed-time between clock output vectors. Etime(t2,t1) returns the time in seconds between t2 and t1.

Examples:

Rounding to the nearest second results in an integer display:

```
fix(clock)

ans =
  1985   10   28   13    5   52
```

Calculate the smallest time interval clock and etime are capable of resolving:

```
t1 = clock; etime(clock,t1)
```

Algorithm:

Etime is an M-file in the *Utility Library*.

Limitations:

Etime will fail across month and year boundaries. It could be fixed with some effort; see etime.m.

Purpose:

Condition number of a matrix.

Synopsis:

cond(X)
rcond(X)

Description:

The *condition number* of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results obtained from matrix inversion and linear equation solution.

Cond(X) returns the 2-norm condition number, the ratio of the largest singular value of X to the smallest.

The usual rule of thumb is that the exponent on the condition number, log10(cond(X)), indicates the number of decimal places that the computer can lose to roundoff errors from Gaussian elimination.

Rcond is a more efficient, but less reliable, method of estimating the condition of a matrix. Rcond(X) is an estimate for the reciprocal of the condition of X in 1-norm using the LINPACK condition estimator. If X is well conditioned, rcond(X) is near 1.0. If X is badly conditioned, rcond(X) is near 0.0.

Algorithm:

Cond uses the singular value decomposition (svd) and is implemented in an M-file in the *Utility Library*.

Rcond uses the condition estimator from the LINPACK routine ZGECO.

See also:

norm, svd

References:

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose:

Convolution and deconvolution.
Correlation.
Polynomial multiplication and division.

Synopsis:

conv(a,b), conv2(a,b)
xcorr(a,b), xcorr2(a,b)
xcorr(a), xcorr2(a)
[q,r] = deconv(b,a)

Description:

Conv(a,b) convolves vectors a and b. The convolution sum is:

$$c(n+1) = \sum_{k=0}^{N-1} a(k+1)b(n-k)$$

where N is the maximum sequence length. The series is written in an unorthodox way, indexed from $n+1$ and $k+1$ instead of the usual n and k because MATLAB vectors run from 1 to n instead of from 0 to $n-1$.

Xcorr(a,b) computes the cross-correlation function between vectors a and b. Xcorr(a) is the auto-correlation function xcorr(a,a).

Conv2(a,b) and xcorr2(a,b) perform the 2-dimensional operations.

[q,r] = deconv(b,a) deconvolves vector a out of vector b, using long division. The result is returned in vector q and the remainder in vector r such that b = conv(q,a) + r.

If a and b are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials, and deconvolution is polynomial division. The result of dividing b by a is quotient q and remainder r.

Examples:

If a = [1 2 3], b = [4 5 6], the convolution is

```
conv(a,b)

ans =
   4   13   28   27   18
```

Algorithm:

These functions are all M-files in the *Utility Library*. They use the filter primitive.

See also:

filter, residue

Purpose:

Covariance and correlation matrices.

Synopsis:

cov(X)
corr(X)

Description:

Cov forms the covariance matrix. For vectors, cov(x) returns the variance. For matrices, where each row is an observation, and each column a variable, cov(X) is the covariance matrix. Diag(cov(X)) is a vector of variances for each column, and sqrt(diag(cov(X))) is a vector of standard deviations.

Corr forms the correlation matrix. For a matrix X where each row is an observation, and each column a variable, if C is the covariance matrix, then corr(X) is the matrix whose $(i,j)'th$ element is

$$corr(i,j) = \frac{C(i,j)}{\sqrt{C(i,i)C(j,j)}}$$

Algorithm:

Both functions are M-files in the *Utility Library*. Here is the algorithm for cov:

```
[n,p] = size(x);
x = x - ones(n,1)*(sum(x)/n);
y = x'*x/n;
```

See also:

mean, median, std

Purpose:

Diagonal matrices.
Upper and lower triangular parts.

Synopsis:

diag(X), diag(X,k)
triu(X), triu(X,k)
tril(X), tril(X,k)

The optional argument k indicates the *k*-th diagonal. k = 0 is the main diagonal, k > 0 is above the main diagonal and k < 0 is below the main diagonal.

Description:

If V is a vector with *n* components, diag(V,k) is a square matrix of order *n+abs(k)* with the elements of V on the *k*-th diagonal. k = 0 is the main diagonal, k > 0 is above the main diagonal and k < 0 is below the main diagonal. Diag(V) simply puts V on the main diagonal.

If X is a matrix, diag(X,k) is a column vector formed from the elements of the *k*-th diagonal of X. Diag(X) is the main diagonal of X.

Triu(X) is the upper triangular part of X. Triu(X,k) is the elements on and above the *k*-th diagonal of X, with k defined as before.

Tril(X) is the lower triangular part of X. Tril(X,k) is the elements on and below the *k*-th diagonal of X.

Examples:

Diag(diag(X)) is a diagonal matrix.

sum(diag(X)) is the trace of X.

The statement

```
diag(-m:m) + diag(ones(2*m,1),1) + diag(ones(2*m,1),-1)
```

produces a tridiagonal matrix of order *2m+1*.

Algorithm:

Triu and tril are M-files in the *Utility Library*.

Purpose:

Save the session in a disk file, possibly for later printing.

Synopsis:

diary *file*
diary
diary on
diary off

Description:

The command diary *file*, where *file* is a filename, causes a copy of all subsequent keyboard input and most of the resulting output (but not graphs) to be written on the named file.

Diary off suspends the diary.

Diary on turns it back on again, using the current filename, or the default filename diary if none had yet been specified.

Diary by itself toggles diary on and off.

The output of diary is an ASCII file, suitable for printing, or for inclusion in reports and other documents.

Limitations:

No, you can't put a diary in the files called off and on.

Purpose:

Difference functions, approximate derivatives.

Synopsis:

diff(x)
diff(x,n)

The optional argument n indicates the *n*-th difference function.

Description:

Diff calculates differences. If X is a row or column vector

$$X = [x(1)\ \ x(2)\ \ \ldots\ \ x(n)]$$

then diff(X) returns a vector of differences between adjacent elements

$$[x(2)-x(1)\ \ \ x(3)-x(2)\ \ \ldots\ \ x(n)-x(n-1)]$$

The output vector will be one element shorter than the input vector.

If X is a matrix, the differences are calculated down each column:

$$diff(X) = X(2:m,:) - X(1:m-1,:)$$

Diff(X,n) is the *n*'th difference function.

Examples:

Diff(y)./diff(x) is an approximate derivative.

Algorithm:

Diff is an M-file in the *Utility Library*.

See also:

sum, prod

dir,type,delete,chdir

Purpose:

File manipulation.

Synopsis:

dir
type *file*
delete *file*
chdir *dir*

Description:

Dir, type, delete, and chdir implement a set of generic operating system commands for manipulating files. Here is a table that indicates how these commands map to other operating systems, perhaps one you are familiar with:

MATLAB	MS-DOS	UNIX	VAX/VMS
dir	dir	ls	dir
type	type	cat	type
delete	del	rm	delete
chdir	chdir	cd	set default

Pathnames, wildcards, and drive designators may be used in the usual way for your operating system.

The type command differs from the usual type commands in an important way; if no file type is given, .m is used by default. This makes it convenient for the most frequent use of type; to list M-files on the screen.

Other operating system commands can be issued using the exclamation point character !.

Examples:

type foo.bar lists the ASCII file foo.bar.
type foo lists the ASCII file foo.m.

See also:

who, what, !

References:

The manual for your operating system.

Purpose:

Display text or matrix.

Synopsis:

disp(X)

Description:

Disp(X) displays a matrix, without printing the matrix name. If X contains a text string, the string is displayed.

Another way to display a matrix on the screen is to type its name, but this prints a leading "A = ", which is not always desirable.

Examples:

One use of disp is to display a matrix with column labels,

```
disp('   Corn    oats    hay')
disp(rand(5,3))
```

which results in

Corn	oats	hay
0.2113	0.8474	0.2749
0.0824	0.4524	0.8807
0.7599	0.8075	0.6538
0.0087	0.4832	0.4899
0.8096	0.6135	0.7741

It shows the use of disp on both data and text strings.

See also:

setstr, num2str, fprintf, return

Purpose:

Echo M-files during execution.

Synopsis:

echo on, echo off
echo *fun*
echo on all, echo off all

Description:

Echo controls the echoing of M-files during execution. The commands in M-files are not normally displayed on the screen during execution. Command echoing can be enabled for debugging, or for demonstrations, allowing the commands to be viewed as they execute.

Echo behaves slightly differently, depending upon which of the two types of M-files, *script files*, or *function files*, is being considered. For *script files*, the use of echo is simple; echoing can be either on or off, in which case any *script* used is affected:

- echo on turns on the echoing of commands, in all *script files*.
- echo off turns off the echoing of all *script files*.
- echo, by itself, toggles the echo state.

This use of echo does not affect *function files*.

With *function files*, the use of echo is more complicated. If echo is enabled on a *function file*, the file is interpreted, instead of compiled, so that each input line can be viewed as it is executed. Since this results in inefficient execution, and should only be used for debugging, provision is made to use echo on just a single *function file*, instead of operating globally on all files:

- echo *file* on, where *file* is a function name, causes the named *function file* to be echoed when it is used.
- echo *file* off turns off the echoing of the named *function file*.
- echo *file* toggles the echo state of the named file.
- echo on all and echo off all set echoing for all *function files*.

See also:

M-files

Purpose:

Eigenvalues and eigenvectors.

Synopsis:

```
eig(X)
[V,D] = eig(X)
eig(A,B)
[V,D] = eig(A,B)
```

Description:

The eigenvalue problem is to determine the non-trivial solutions of the equation

$$Ax = \lambda x$$

where A is an n-by-n matrix, x is a length n column vector, and λ is a scalar. The n values of λ that satisfy the equation are the *eigenvalues* and the corresponding values of x are the *right eigenvectors*.

In MATLAB the function eig solves for the eigenvalues λ and optionally the eigenvectors x:

- Eig(A) is a vector containing the eigenvalues of matrix A.
- [X,D] = eig(A) produces a diagonal matrix D of eigenvalues and a full matrix X whose columns are the corresponding eigenvectors so that A*X = X*D.

Eigenvectors are normalized so that the largest element of each eigenvector is 1.0.

The *generalized* eigenvalue problem is to determine the non-trivial solutions of the equation

$$Ax = \lambda Bx$$

where both A and B are n-by-n matrices and λ is a scalar. The values of λ that satisfy the equation are the *generalized eigenvalues* and the corresponding values of x are the *generalized right eigenvectors*.

If B is non-singular, the problem could be solved by reducing it to a standard eigenvalue problem,

$$B^{-1}Ax = \lambda x$$

with A replaced by $B^{-1}A$. But since B could be singular, an alternative algorithm, called the QZ method, is used.

In MATLAB the function eig solves for the generalized eigenvalues and eigenvectors when used with two input arguments:

- Eig(A,B), if A and B are square matrices, returns a vector containing the generalized eigenvalues.
- [X,D] = eig(A,B) produces a diagonal matrix D of generalized eigenvalues and a full matrix X whose columns are the corresponding eigenvectors so that A*X = B*X*D.

Algorithm:

For real matrices, eig uses the EISPACK routines ORTRAN, ORTHES, and HQR2. ORTHES converts a real general matrix to Hessenberg form using orthogonal similarity transformations. ORTRAN accumulates the transformations used by ORTHES. HQR2 finds the eigenvalues and eigenvectors of a real upper Hessenberg matrix by the QR method. The EISPACK subroutine HQR2 was modified to to make the computation of eigenvectors optional.

When eig is used with two arguments, the EISPACK routines QZHES, QZIT, QZVAL, and QZVEC are used to solve for the generalized eigenvalues via the QZ algorithm. They have been modified for the complex case.

When eig is used with one complex argument, the solution is computed using the QZ algorithm as eig(X,eye(X)). Modifications to the QZ routines handle the special case $B = I$.

For detailed write ups on these algorithms, see the *EISPACK User's Guide*.

Diagnostics:

From eig if the limit of *30n* iterations is exhausted while seeking an eigenvalue:

```
Solution will not converge.
```

See also:

qz,hess,schur

References:

[1] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, C. B. Moler, *Matrix Eigensystem Routines -- EISPACK Guide*, Lecture Notes in Computer Science, volume 6, second edition, Springer-Verlag, 1976.

[2] B. S. Garbow, J. M. Boyle, J. J. Dongarra, C. B. Moler, *Matrix Eigensystem Routines -- EISPACK Guide Extension*, Lecture Notes in Computer Science, volume 51, Springer-Verlag, 1977.

[3] C.B Moler and G.W. Stewart, *An Algorithm for Generalized Matrix Eigenvalue Problems*, SIAM J. Numer. Anal., Vol. 10, No. 2, April 1973.

Purpose:

Terminate for, while, and if statements.

Synopsis:

while *s*
statements
end

Description:

End is used to terminate the scope of for, while, and if statements. Without end's, for, while, and if wait for further input. Each end is paired with the closest previous unpaired for, while, or if and serves to terminate its scope.

Examples:

Here is an example showing end used with for and if.

```
for i=1:n
    for j=1:n
        if i == j
            a(i,j) = 2;
        elseif abs(i-j) == 1
            a(i,j) = -1;
        else
            a(i,j) = 0;
        end
    end
end
```

The indentation is important only to human readers.

See also:

while, if, for, break, return

Purpose:

Text macro facility.

Synopsis:

eval(t)

Description:

Eval(t) executes the text contained in t. If STRING is the source text for any MATLAB expression or statement, then

```
t = 'STRING';
```

encodes the text in t. Typing t prints the text and

```
eval(t)
```

causes the text to be interpreted, either as a statement or as a factor in an expression. If the text is an expression, eval returns the result, which can be assigned to a variable. For example, these statements assign π to p:

```
t = '4*atan(1)';
p = eval(t);
```

The text macro facility is particularly useful as a mechanism for passing function names to *M-function files*. For example, see the file funm.m in the *Utility Library*.

Examples:

Generate the Hilbert matrix of order *n*:

```
t = '1/(i+j-1)';
for i = 1:n
   for j = 1:n
      a(i,j) = eval(t);
   end
end
```

Here is an example showing indexed text:

```
S = ['x = 3              '
     'y = 4              '
     'z = sqrt(x*x+y*y)'];
for k=1:3
  eval(S(k,:))
end
```

It is necessary that the strings making up the "rows" of the "matrix" S have the same lengths.

See also:

M-files, setstr

Purpose:

Exponential, logarithmic and square-root functions.

Synopsis:

```
exp(x)
log(x)
log10(x)
sqrt(x)
```

Description:

Exp, log, and log10 are elementary functions that operate element-wise on matrices. Their domain includes complex numbers which can lead to unexpected results if used unintentionally.

Exp(X) returns e^x for each of the elements of X. For complex $z = x + iy$, the complex exponential is returned:

$$e^z = e^x(cos(y) + i\ sin(y))$$

Log(X) is the natural logarithm of the elements of X. For complex or negative z, the complex logarithm is returned:

$$log(z) = log(abs(z)) + i\ atan2(y,x)$$

Log10(X) is $\log_{10}$ of the elements of X.

Sqrt(X) is the square-root of the elements of X. Complex results are produced for the elements of X that are negative or complex.

Examples:

The statement log(–1) is a clever way of generating π:

```
ans =
  0.0000 + 3.1416i
```

See also:

trig, expm, funm, abs, angle

Purpose:

Matrix exponential and other matrix functions.

Synopsis:

expm(X)
logm(X)
sqrtm(X)
funm(X,'*function*')

For funm, the *function* can be log, sqrt, sin, cos or any other elementary function.

Description:

Expm(X) is the matrix exponential of X. Complex results are produced if X has nonpositive eigenvalues. Expm uses a Padé expansion after scaling X for more reliable computation. Funm can also be used, but expm gives a much faster and sometimes more accurate result.

Funm can be used to evaluate more general matrix functions. For matrix arguments X, funm(X,'*fun*') evaluates the matrix function specified by *fun* using Parlett's method [1]. For example, funm(X,'exp') calculates the matrix exponential, funm(X,'log') the matrix logarithm, and funm(X,'sqrt') the matrix square root.

Sqrtm(X) and logm(X) are equivalent to funm(X,'sqrt') and funm(X,'log').

Algorithm:

Expm is a built-in function, but it uses the Padé approximation algorithm expressed in the file expm1.m in the *Utility Library*.

A second method of calculating the matrix exponential is via a Taylor series approximation. This method can be found in the file expm2.m.

A third way of calculating the matrix exponential, found in the file expm3.m, is to diagonalize the matrix, apply the function to the individual eigenvalues and then transform back.

Many algorithms for computing expm(x) are described and compared in references [1] and [2]. Our built-in method, expm1, is essentially method 3 of [2].

Funm is used to evaluate other matrix functions. Funm uses Parlett's method [1], and is an M-file in the *Utility Library*.

References:

[1] G. H. Golub and C. F. Van Loan, *Matrix Computation*, Johns Hopkins University Press, 1983, p. 384.

[2] C. B. Moler and C. F. Van Loan, *Nineteen Dubious Ways to Compute the Exponential of a Matrix*, SIAM Review 20, 801-836, 1979.

Purpose:

1-D and 2-D fast Fourier transforms.

Synopsis:

fft(x), ifft(x)
fft2(X), ifft2(x)
dft(x), idft(x)
fftshift(x)

Description:

Fft(x) is the discrete Fourier transform of vector x, computed with a radix-2 fast-Fourier transform algorithm. If the length of x is not an exact power of two, it is padded with trailing zeros.

Ifft(x) is the inverse discrete Fourier transform of vector x.

The two functions implement the transform - inverse transform pair given by:

$$X(k+1) = \sum_{n=0}^{N-1} x(n+1) W_N^{kn}$$

$$x(n+1) = 1/N \sum_{k=0}^{N-1} X(k+1) W_N^{-kn}$$

where $W_N = e^{-j(2\pi/N)}$ and $N = length(x)$. Note that the series is written in an unorthodox way, running over $n+1$ and $k+1$ instead of the usual n and k because MATLAB vectors run from 1 to N instead of from 0 to $N-1$.

Suppose a sequence of N points is obtained at a sample frequency of f_s. Then, for up to the Nyquist frequency, or point $n = N/2+1$, the relationship between the bin number and the actual frequency is:

$$f = (bin\_number - 1) * f_s / N$$

Fft2 and ifft2 perform two-dimensional FFT's.

Dft and idft calculate the discrete Fourier transform directly, allowing sequences that are not exact powers of two. They are substantially slower than fft, especially for large n.

Fftshift(x) rearranges the outputs of fft and fft2 by moving the zeroth lag to the center of the spectrum, which is sometimes a more convenient form. For vectors fftshift(x) returns a vector with the left and right halves swapped. For matrices, fftshift(X) swaps quadrants one and three and quadrants two and four.

Examples:

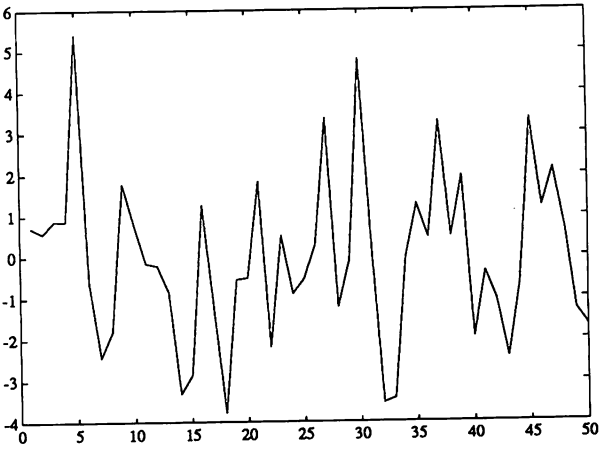
This example shows a simple use of the FFT function for spectral analysis. It is available as a demo in the *Utility Library* and can be run by typing fftdemo.

A common use of FFT's is to find the frequency components of a signal buried in a noisy time domain signal. Consider data sampled at 1000 *Hz*. Form a signal containing 50 *Hz* and 120 *Hz* and corrupt it with some zero-mean random noise,

```
t = 0:.001:.5;
x = sin(2*pi*50*t) + sin(2*pi*120*t);
rand('normal')
y = x + 2*rand(t);
```

The noisy signal y looks like

```
plot(y(1:50))
```



It is difficult to identify the frequency components from looking at the original signal. Converting to the frequency domain, the discrete Fourier transform of the noisy signal y is found by taking the fast-Fourier transform (FFT):

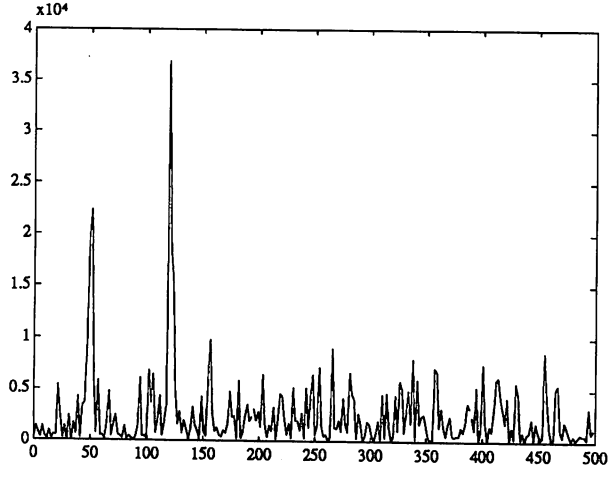
```
Y = fft(y);
```

The power spectral density, a measurement of the energy at various frequencies, is:

```
Pyy = Y .* conj(Y);
```

Plot the power spectral density, first forming a frequency axis for the first 256 points. (The other 256 points are symmetric.)

```
f = 1000*(0:255)/512;
plot(f,Pyy(1:256))
```



Algorithm:

All of these functions, except fft, are M-functions in the *Utility Library*.

See also:

filter, freqz

Purpose:

Filtering data.

Synopsis:

filter(b, a, x)
[y,xf] = filter(b, a, x, xi)

Description:

Filter filters data using a digital filter. The filter structure is the direct implementation, sometimes called a tapped delay-line filter.

y = filter(b, a, x) filters the data in vector x with the filter described by vectors a and b to create filtered data vector y. The operation performed by filter is described in the *time-domain* by the difference equation:

$$y(n) = b(1)x(n) + b(2)x(n-1) + ... + b(nb)x(n-nb+1) - a(2)y(n-1) - ... - a(na)y(n-na+1)$$

An equivalent representation is the z-transform or *frequency-domain* description:

$$Y(z) = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb)z^{-(nb-1)}}{1 + a(2)z^{-1} + \cdots + a(na)z^{-(na-1)}} X(z)$$

If $a(1) \neq 1$, the filter coefficients are normalized by $a(1)$.

When used with two left hand arguments, filter returns the final values of the delays (states):

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

When used with an extra right hand argument, initial conditions on the delays are specified:

```
y = filter(b, a, x, zi)
```

The size of the initial/final condition vector is $max(na, nb)$.

Examples:

Find and graph the n-point unit impulse response of a digital filter:

```
x = [1 zeros(1,n–1)];
y = filter(b,a,x);
plot(y,'o')
```

Filter two sequences, the second of which starts with initial conditions left over from the first:

```
[y1,z] = filter(b,a,x1);
y2 = filter(b,a,x2,z);
```

Find initial conditions to reduce the startup transient before filtering data:

```
[y,z] = filter(b,a,x(1)*ones(1,1000));
y = filter(b,a,x,z);
```

Eliminate all phase changes by re-filtering the data, in reverse:

```
y = filter(b,a,x);
y = filter(b,a,y(n:-1:1));
```

Diagnostics:

If $a(1) = 0$, filter produces the error message:

First denominator coefficient must be non-zero.

If the initial condition vector is not length $max(na,nb)$,

Initial condition vector has incorrect dimensions.

See also:

fft, ifft, freqz

Purpose:

Find indices of the non-zero elements in a vector.

Synopsis:

find(K)

Description:

Find(K) returns the indices of vector K that are non-zero. If none are found, find returns an empty matrix (a matrix of size 0-by-0).

Examples:

The statement

```
I = find(X>100)
```

returns the indices of X where X is greater than 100.

See also:

< <= > >= == ~=, isnan, finite, isempty

Purpose:

Count of floating point operations.

Synopsis:

flops
flops(0)

Description:

Flops returns the cumulative number of floating point operations.

Flops(0) resets the count to zero.

It is not feasible to count absolutely all floating point operations, but most of the important ones are counted. Additions and subtractions are one flop if real and two if complex. Multiplications and divisions count one flop each if the result is real and six flops if it is not. Elementary functions count one if real and more if complex.

Examples:

If A and B are real n-by-n matrices, here is a table of some typical flop counts:

OPERATION	FLOP COUNT
A + B	n^2
A * B	$2n^3$
A ^ 100	$99*2n^3$
lu(A)	$(2/3)n^3$

Purpose:

Repeat statements a specific number of times.

Synopsis:

```
for v = e
  statements
end
```

Description:

For allows statements to be repeated a specific number of times. The general format is:

```
for variable = expression
  statement
     ...
  statement
end
```

The *columns* of the *expression* are stored one at a time in the variable and then the following statements, up to the end, are executed.

In practice, the *expression* is almost always of the form x:y, in which case its columns are simply scalars.

Examples:

Assume n has already been assigned a value:

1) The Hilbert matrix:

```
for i=1:n,
     for j=1:n,
          a(i,j) = 1/(i+j-1);
     end
end
```

2) Step s with increments of –0.1:

```
for s = 1.0: -0.1: 0.0,... end
```

3) Successively set e to the unit n-vectors:

```
for e = eye(n), ... end
```

4) The line:

```
for V = A, ... end
```

has the same effect as

```
for j = 1:n, V = A(:,j); ... end
```

except j is also set here.

See also:

while, if, end, break, return

Purpose:

Output display format.

Description:

Format controls the output format. By default, MATLAB displays numbers in a "short" format with 5 decimal digits. Format switches between different display formats:

COMMAND	RESULT	EXAMPLE
format short	5 digit scaled fixed point	1.3333
format long	15 digit scaled fixed point	1.33333333333333
format short e	5 digit floating point	1.3333E+000
format long e	15 digit floating point	1.33333333333333E+000
format hex	Hexadecimal	3FF5555555555555
format +	+,– format	+
format bank	Fixed dollars and cents	1.33

Format compact suppresses excess line-feeds and results in a slightly more compact display. Format loose reverts to the more airy display. These two commands do not affect the numeric format.

Format, by itself, returns to the default formats.

Examples:

The + format displays +, –, and blank symbols for positive, negative and zero real elements. It is especially useful for displaying large matrices. Here is a simple example on a small matrix:

```
format +
E = ones(5);  P = triu(E) – tril(E)
P =
     ++++
    – +++
    –– ++
    ––– +
    ––––
```

See also:

num2str, sprintf, fprintf

Purpose:

Frequency response of analog filters.

Synopsis:

```
h = freqs(b,a,w)
```

Description:

h = freqs(b,a,w) returns the complex frequency response, $H(j\omega)$, of the analog filter (Laplace transform),

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^{(nb-1)} + b(2)s^{(nb-2)} + \cdots + b(nb)}{a(1)s^{(na-1)} + a(2)s^{(na-2)} + \cdots + a(na)}$$

given the numerator and denominator coefficients in vectors b and a. The frequency response is evaluated along the imaginary axis in the complex plane at the frequencies specified in real vector w.

Examples:

Find and graph the frequency response of a simple system:

```
a = [1 .4 1]; b = [.2 .3 1];
w = logspace(-1,1);
h = freqs(b,a,w);
mag = abs(h); phase = angle(h);
loglog(f,mag), semilogx(f,phase)
```

To convert to Hz, degrees, and decibels,

```
f = w/(2*pi);
phase = phase*180/pi;
mag = 20*log10(mag);
```

Algorithm:

Freqs is an M-file in the *Utility Library*. It evaluates the polynomials at each frequency point, and then divides the numerator response by the denominator response. Here is the complete algorithm:

```
s = sqrt(-1)*w;
h = polyval(b,s) ./ polyval(a,s);
```

See also:

logspace, freqz, abs, angle

Purpose:

Frequency response of digital filters.

Synopsis:

```
[h,w] = freqz(b,a,n)
[h,w] = freqz(b,a,n,'whole')
h = freqz(b,a,w)
```

Description:

[h,w] = freqz(b,a,n) returns the n-point complex frequency response, $H(e^{j\omega})$, of the digital filter,

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb)z^{-(nb-1)}}{a(1) + a(2)z^{-1} + \cdots + a(na)z^{-(na-1)}}$$

given the numerator and denominator coefficients in vectors b and a. Freqz returns both h, the complex frequency response, and w, a vector containing the n frequency points. The frequency response is evaluated at n points equally spaced around the upper half of the unit circle, so w contains n points between 0 and π.

It is best, although not neccessary, to choose an n that is an even power of two, because this allows fast computation using an FFT algorithm.

Freqz(b,a,n,'whole') uses n-points around the *whole* unit circle (from 0 to 2π).

Freqz(b,a,w) returns the frequency response at the arbitrary frequencies designated in vector w (they should still be between 0 and 2π, however).

Examples:

The output of freqz is a complex vector. The magnitude and phase can be plotted with:

```
[h,w] = freqz(b,a,n);
mag = abs(h);
phase = angle(h);
semilogy(w,mag), plot(w,phase)
```

Algorithm:

Freqz is an M-file in the *Utility Library*. It uses an FFT algorithm when n is a power of two. When n not a power of two, freqz evaluates the polynomials at each frequency point, using Horner's method of polynomial evaluation, and then divides the numerator response by the denominator response.

See also:

filter, fft, logspace, freqs, abs, angle

Purpose:

Define global variables.

Synopsis:

global *variable-name*
global *name-1 name-2 ...*

Description:

The global declaration makes variables global in scope, allowing them to be referenced inside *function M-files* without passing them through the argument list. The statement

```
global x y z
```

makes variables x, y, and z global, immediately making them available inside the bodies of *function M-files*. The variable names are separated by spaces, and need not have previously existed in the workspace.

The global statement can *not* be issued in the body of a *function M-file*, nor inside a for or while loop. It must be issued interactively, or in an *script M-file*, and is normally used at the beginning of the session. Variables remain global unless the entire workspace is cleared.

Limitations:

Variable name clash can occur between local function variable names and global variable names. If you make common variable names like X or A global, you are virtually guaranteed to get unpredictable effects when using functions out of the *Utility Library*. We recommend that you maximize the chance of getting unique names by using long names or by using an underscore "_" in the variable name.

When clash occurs, the global variable name wins, unless the local variable name is defined in the function argument list, or the variable was defined as global *after* the function was compiled into memory.

Global variables are, in general, bad programming practice. We hope you never have to use this facility, and as a matter of principle, we discourage its use.

Purpose:

Help facility.
Demonstrations.

Description:

Demo brings up a menu of the available demonstrations.

Typing help, by itself, gives a list of HELP topics, including the M-files in the various libraries on disk.

Help *topic* gives HELP on the specified *topic*. If the topic is not found in the help file, the help facility looks on the disk for an M-file with the filename *topic.m*. If it finds it, the first comment lines in the file are displayed. This allows you to get HELP on your own M-files.

See also:

who, what, MATLABPATH

Purpose:

Hessenberg form.
Schur decomposition.

Synopsis:

```
[P,H] = hess(X), hess(X)
[U,T] = schur(X), schur(X)
```

Description:

Hess finds the Hessenberg form of a matrix. A Hessenberg matrix is all zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal.

[P,H] = hess(A) produces a Hessenberg matrix H and a unitary matrix P so that A = P*H*P' and P'*P = eye(A). By itself, hess(A) returns H.

Schur computes the Schur form of a matrix. The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

[U,T] = schur(A) produces a Schur matrix T, and a unitary matrix U so that A = U*T*U' and U'*U = eye(A). By itself, schur(A) returns T.

If the matrix A is pure real, schur returns the *real Schur form.* If imag(A) is not zero, the *complex Schur form* is returned. The function rsf2csf.m in the *Utility Library* converts the real form to the complex form.

Algorithm:

For real matrices, hess and schur use the EISPACK routines ORTRAN, ORTHES, and HQR2. ORTHES converts a real general matrix to Hessenberg form using orthogonal similarity transformations. ORTRAN accumulates the transformations used by ORTHES. HQR2 finds the eigenvalues of a real upper Hessenberg matrix by the QR method.

The EISPACK subroutine HQR2 was modified to allow access to the Schur form, ordinarily just an intermediate result, and to make the computation of eigenvectors optional.

When hess and schur are used with a complex argument, the solution is computed using the QZ algorithm by the EISPACK routines QZHES, QZIT, QZVAL, and QZVEC. They have been modified for complex problems and to handle the special case $B = I$.

For detailed write ups on these algorithms, see the EISPACK User's Guide.

Diagnostics:

From schur if the limit of *30n* iterations is exhausted while seeking an eigenvalue:

```
Solution will not converge.
```

See also:

eig, qz, rsf2csf

References:

[1] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, C. B. Moler, *Matrix Eigensystem Routines -- EISPACK Guide*, Lecture Notes in Computer Science, volume 6, second edition, Springer-Verlag, 1976.

[2] B. S. Garbow, J. M. Boyle, J. J. Dongarra, C. B. Moler, *Matrix Eigensystem Routines -- EISPACK Guide Extension*, Lecture Notes in Computer Science, volume 51, Springer-Verlag, 1977.

[3] C.B Moler and G.W. Stewart, *An Algorithm for Generalized Matrix Eigenvalue Problems*, SIAM J. Numer. Anal., Vol. 10, No. 2, April 1973.

Purpose:

Histograms.

Synopsis:

```
[n,x] = hist(y)
[n,x] = hist(y,nb)
n = hist(y,x)
histogram(x), histogram(x,n)
```

Description:

Hist calculates histograms.

[n,x] = hist(y) calculates a 10 bin histogram for the data in vector y. Hist returns vector x with 10 equally spaced bins between the minimum and maximum values in y, and vector n with the frequency counts in each of the 10 bins. The histogram may be plotted with plot(x,n,'.').

[n,x] = hist(y,nb) uses nb bins.

n = hist(y,x), if x is a vector, calculates a histogram using the bins specified by the user in x.

Histogram(y) calculates *and* plots a bar chart histogram with 10 bins. Histogram(y,n) uses *n* bins to do the same.

Examples:

Generate bell-curve histograms from Gaussian data:

```
rand('normal')
y = rand(750,1);
[n,x] = hist(y);
plot(x,n,'x')  % This plots x-marks.

histogram(y,20) % This does a bar chart.
```

Algorithm:

Hist and histogram are M-files in the *Utility Library*.

Purpose:

Hold plot on screen.

Synopsis:

hold
hold on
hold off

Description:

Hold holds the current graph on the screen. Subsequent plot commands will add to the plot, using the already established axis limits, and retaining the previously plotted curves.

If hold is not in effect, each plot command starts by clearing the screen and finding fresh auto-ranging limits.

hold on sets holding on.

hold off turns holding off.

hold, by itself, toggles the hold state.

See also:

axis

Purpose:

Conditional control flow statements.

Synopsis:

if *expression*
  statements
elseif *expression*
  statements
else
  statements
end

Description:

If is used to conditionally execute statements. The simple form is:

```
if expression
  statements
end
```

The statements are executed if the *expression* has all non-zero elements. The expression is usually the result of

expression *rop* expression

where *rop* is ==, <, >, <=, >=, or ~=.

More complicated forms use else or elseif in the normal way. (But be careful not to put a space between the else and the if when you really mean elseif.)

Examples:

Here is an example showing if, else, and elseif.

```
for i=1:n,
      for j=1:n,
            if i == j,
                  a(i,j) = 2;
            elseif abs(i–j) == 1,
                  a(i,j) = –1;
            else
                  a(i,j) = 0;
            end
      end
end
```

See also:

while, for, end, break, return

Purpose:

Manipulation of complex numbers.

Synopsis:

```
imag(X)
real(X)
conj(X)
```

Description:

The functions real, imag, and conj operate on the individual elements of matrices:

- Real(X) is the real part of X.
- Imag(X) is the imaginary part of X.
- Conj(X) is the complex conjugate of X.

Imaginary numbers are not entered into MATLAB using the letters i or j as might be expected. This is because i and j are often used as indices, or variable names, and have no special meaning to MATLAB. To enter imaginary numbers, use sqrt(–1). For example, $3+2i$ could be entered as

```
3+2*sqrt(-1)
```

or you might prefer,

```
i = sqrt(-1);
3+2*i
```

Examples:

If X is a complex vector or matrix, then

```
conj(X) = real(X) - sqrt(-1)*imag(X)
```

Purpose:

User input.

Synopsis:

input('*prompt*')
input('*prompt*','s')

Description:

Input('How many apples') displays the text string as a prompt on the screen and waits for a number to be input from the keyboard. The number is returned by input.

It is legal to feed an *expression* like [1 2 3] or rand(3) to the prompt. Input can also be used with multiple output arguments if the expression typed by the user returns more than one output.

Input('What"s your name','s') returns the *string* typed by the user as a text variable.

See also:

keyboard

Purpose:

Matrix inverse.
Determinant.
LU decomposition.

Synopsis:

```
inv(X)
det(X)
[L,U] = lu(X)
```

Description:

Inv(X) is the inverse of the square matrix X. A warning message is printed if X is badly scaled or nearly singular.

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of inv arises when solving the system of linear equations $Ax = b$. One way to solve this is with x = inv(A)*b. A better way, from a numerical accuracy standpoint, is to use the "matrix division" operator, x = A\b. This produces the solution using Gaussian elimination, without forming the inverse. See \ and / for further information.

Det(X) is the determinant of the square matrix X. If X contains only integers, the result will be an integer.

The most basic factorization expresses any square matrix as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the "LU", or sometimes the "LR", factorization. Most of the algorithms for computing it are variants of Gaussian elimination.

The factors themselves are available from the lu function. The factorization is used to obtain the inverse with inv and the determinant with det. It is also the basis for the linear equation solution or "matrix division" obtained with \ and /.

[L,U] = lu(X) stores an upper triangular matrix in U and a "psychologically lower triangular matrix", i.e. a product of lower triangular and permutation matrices, in L, so that X = L*U.

By itself, LU(X) returns the output from the LINPACK routine ZGEFA.

Examples:

Start with

```
A =
   1   2   3
   4   5   6
   7   8   0
```

To see the LU factorization, we use MATLAB's double assignment statement.

```
[L,U] = lu(A)
```

which gives

```
L =
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
    1.0000         0         0

U =
    7.0000    8.0000    0.0000
         0    0.8571    3.0000
         0         0    4.5000
```

Notice that L is a permutation of a lower triangular matrix that has ones on the permuted diagonal, and that U is upper triangular. To check that the factorization does its job, we can compute the product

```
L*U
```

which gives us back the original A,

```
ans =
   1   2   3
   4   5   6
   7   8   0
```

The inverse of the example matrix can be obtained with

```
X = inv(A)
```

The inverse is actually computed from the inverses of the triangular factors

```
X = inv(U)*inv(L)
```

The determinant of the example matrix can be obtained with

```
d = det(A)
```

which gives

```
d =
   27
```

It is computed from the determinants of the triangular factors

```
d = det(L)*det(U)
```

The solution to the equation $Ax = b$ is obtained with the MATLAB matrix division operation

```
x = A\b
```

The solution is actually computed by solving two triangular systems,

```
y = L\b, x = U\y
```

Triangular factorization is also used by a specialized function, rcond. This is a quantity produced by several of the LINPACK subroutines that is an estimate of the reciprocal condition number of a square matrix.

Algorithm:

Inv and det use the subroutines ZGEDI and ZGEFA from LINPACK. For more information, see the *LINPACK User's Guide*.

Diagnostics:

From inv, if the matrix is singular:

```
Matrix is singular to working precision.
```

If the inverse was found, but is not reliable:

Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = *xxx*

See also:

rcond, rref, \, /

References:

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose:

Detect NaNs, empty matrices, and infinities.

Synopsis:

```
isnan(x)
isempty(x)
finite(x)
```

Description:

Isnan(X) returns 1's where the elements of X are NaNs and 0's where they are not.

Isempty(X) returns 1 if X is an empty matrix and 0 otherwise. An empty matrix has size 0-by-0.

Finite(X) returns 1's where the elements of X are finite and 0's where they are infinite.

Examples:

You might think that the statement

```
K = (A == NaN)
```

would return a matrix with 1's where there are NaNs, and 0's elsewhere, but it does not - it returns NaNs everywhere. Here is the correct way to detect NaNs:

```
K = isnan(A)
```

Algorithm:

Isempty and finite are M-files.

See also:

find, any, all, < <= > >= == ~=

Purpose:

Invoke keyboard as an M-file.

Synopsis:

keyboard

Description:

Keyboard invokes the keyboard as if it were a Script M-file. When placed in an M-file, keyboard stops execution of the file and gives control to the user's keyboard. The special status is indicated by a K appearing before the prompt. Variables may be examined or changed; all MATLAB commands are valid. The keyboard mode is terminated by typing CTRL-Z, the end-of-file control character (CTRL-D on Unix systems). Control returns to the invoking M-file.

Keyboard is quite useful as a tool for debugging M-files.

Limitations:

If you accidently type CTRL-Z twice, the second one will terminate MATLAB. (Actually this is feature; CTRL-Z is one way to terminate a session.)

See also:

input, quit

Purpose:

Kronecker product.

Synopsis:

kron(X,Y)

Description:

Kron(X,Y) is the Kronecker tensor product of X and Y. The result is a large matrix formed by taking all possible products between the elements of X and those of Y. For example, if X is 2-by-3, then kron(X,Y) is

```
[ X(1,1)*Y  X(1,2)*Y  X(1,3)*Y
  X(2,1)*Y  X(2,2)*Y  X(2,3)*Y ]
```

Algorithm:

Kron is an M-function in the *Utility Library*.

Purpose:

Saving and loading variables on disk.

Synopsis:

load *file*
save *file*
save *file* A B C

Description:

Save and load are used to store and retrieve variables on disk.

Save temp stores the current variables in a MAT-file named temp.mat. The command save temp X saves only variable X, while save temp X Y Z saves X, Y, and Z. Save, by itself, saves everything in a file named matlab.mat.

The load command is the inverse of save. Load temp retrieves all the variables from the MAT-file named temp.mat. Load, by itself, loads the variables from the file matlab.mat.

Load can also read ASCII flat files. If the data are stored in ASCII form, with fixed length rows terminated with newlines (carriage returns), and with spaces separating the numbers, then the file is a so-called *flat file*. (ASCII flat files can be edited using a normal text editor.) Flat files can be read directly into MATLAB using the load command. The result is put into a variable with the same name as the file.

MAT-files can be transferred between PC, VAX, Sun, Macintosh, and other implementations of MATLAB. MAT-files contain a signature that indicates the machine that wrote the file. The load command checks the signature and performs conversion, if neccessary. Full numeric precision is retained when moving between IEEE arithmetic machines, like the PC, Sun and Macintosh. There is a precision loss when exchanging files with the VAX, however.

We recommend a public-domain program called *Kermit* for transferring M-files and MAT-files between computers.

Algorithm:

The save command saves MATLAB variables on disk in a specially structured file we call a MAT-file, so-called because the filename ends with ".mat". It is possible to read and write MAT-files from your own programs, provided you use the special file structure.

A MAT-file may contain one or more variables. The variables are written sequentially on disk, with the bytes conceptually forming a continuous stream. Each variable starts with a fixed length 20 byte header which contains information describing certain attributes of the variable. The 20 byte header consists of 5 four-byte long-integers (words):

type Type flag. Word 1 contains an integer whose decimal digits encode the variable type. If the integer is represented as *MOPT* where *M* is the thousands digit, *O* is the hundreds digit, *P* is the tens digit, and *T* is the ones digit, then:

M indicates the numeric format of binary numbers on the machine that wrote the file. Use this table to determine the number to use for your machine:

MACHINE ID	
PC	0
Sun	1
Macintosh	1
VAX D-float	2
VAX G-float	3

O is normally 0, which means the data are stored in a *column-wise* orientation (varies fastest down a column). If $O = 1$, the data are transposed, and stored in a *row-wise* orientation (varies fastest across a row).

P is normally 0, which means the data are stored on disk in double precision (8 bytes/element). If $P = 1$, the data are stored in single precision (4 bytes/element). $P = 2$ is signed 32 bit integer data, $P = 3$ is 16 bit signed integers, and $P = 4$ is unsigned 16 bit integers.

T is normally 0, indicating that the data that follow describe a matrix. If $T = 1$, the variable is a text variable. This means that the numbers in the variable are floating point numbers between 0 and 255 representing the ASCII code of characters.

For PC's type is usually 0000, or 0, which indicates PC double precision matrix data stored by columns. Note that $P \neq 0$ and $O = 1$ are not produced by the save command, but they could be generated outside of MATLAB (to save file space) and accepted by load.

mrows Row dimension. Word 2 contains an integer with the row dimension of the variable.

ncols Column dimension. Word 3 contains an integer with the column dimension of the variable.

imagf Imaginary flag. Word 4 is an integer that is either 0 or 1. If 1, then the variable has an imaginary part. If 0, there is only real data.

namlen Name length. Word 5 contains an integer with the length of the variable name plus 1.

Immediately following the fixed length header is data that has a length dependent on the variables in the fixed length header:

name Variable name. The name consists of namlen ASCII bytes, the last one of which must be a NUL character (encoded as 0).

real Real part of the matrix. The real data consists of mrow * ncol double precision (8-byte) floating point numbers. Matrices are stored column-wise, first the first column, then the second column, etc., unless otherwise specified by type in the fixed header.

imag Imaginary part of the matrix, if any. If the imaginary flag, imagf is nonzero, the imaginary part of a matrix is here. It is stored in the same way as real data.

The structure is repeated for as many variables as there are stored on the file.

Here is some C language code that writes a single matrix on disk:

```
typedef struct {
   long type;  /* type */
   long mrows; /* row dimension */
   long ncols; /* column dimension */
   long imagf; /* flag indicating imag part */
   long namlen; /* name length (including NULL) */
} Fmatrix;

char *pname;     /* pointer to matrix name */
double *pr;      /* pointer to real data */
double *pi;      /* pointer to imag data */
FILE *fp;
Fmatrix x;
int mn;
```

```
fwrite(&x, sizeof(Fmatrix), 1, fp);
fwrite(pname, sizeof(char), x.namlen, fp);
mn = x.mrows * x.ncols;
fwrite(pr, sizeof(double), mn, fp);
if (x.imagf) {
   fwrite(pi, sizeof(double), mn, fp);
}
```

The Fortran equivalent of this code is available in the *Utility Library*. Perhaps you'll find code for other languages there too.

See also:

translate

Purpose:

Generate logarithmically spaced vectors.

Synopsis:

```
y = logspace(d1,d2)
y = logspace(d1,d2,n)
y = logspace(d1,pi)
```

Description:

Logspace generates logarithmically spaced vectors. Often used to create frequency vectors, it is a logarithmic equivalent of the ':' or colon operator.

Logspace(d1,d2) generates a vector of 50 points logarithmically equally spaced between decades 10^{d1} and 10^{d2}.

Logspace(d1,d2,n) generates n points.

If d2 is pi, the points are between 10^{d1} and π, which is useful for digital signal processing where frequencies go over this interval around the unit circle.

Examples:

Generate 50 points between .01 and 10,

```
w = logspace(-2,1);
```

Algorithm:

Logspace is an M-function in the *Utility Library.*

See also:

:

Purpose:

MATLAB search path.

Description:

MATLAB has a *search path.* If you input the name of something to MATLAB, for example by typing FOX, the MATLAB interpreter:

[1] Looks to see if FOX is a variable.

[2] Checks if FOX is a built-in function.

[3] Looks in the current directory for a file named FOX.M.

[4] Searches the directories specified by the environment symbol MATLABPATH for FOX.M.

MATLABPATH is an operating system environmental variable that has been pre-defined for you, but it may be changed if you wish to add your own directories to the search path. Consult the machine specific first section for more information.

Examples:

Here is how to set MATLABPATH on some different operating systems:

MS-DOS:

```
set MATLABPATH=c:\matlab;c:\matlab\control
```

Unix C-shell:

```
setenv MATLABPATH "/usr/matlab:/usr/matlab/control"
```

VAX/VMS:

```
MATLABPATH :== [matlab],[matlab.control]
```

Purpose:

Maximum and minimum.

Synopsis:

```
max(X)
[Y,I] = max(X)
max(A,B)
```

Description:

For vectors, max(X) is the largest element in X. For matrices, max(X) is a row vector containing the maximum element from each column.

[Y,I] = max(X) stores the indices of the maximum values in vector I.

Max(A,B) returns a matrix the same size as A and B with the largest elements taken from A or B.

When complex, the magnitude max(abs(x)) is used.

Min works the same as max, with the obvious difference.

See also:

sort

Purpose:

Basic statistical functions.

Synopsis:

```
mean(X)
median(X)
std(X)
```

Description:

Mean calculates the average or *mean* value. For vectors, mean(X) is the mean value of the elements in vector X. For matrices, mean(X) is a row vector containing the mean value of each column.

Median calculates the *median* value. For vectors, median(X) is the median value of the elements in vector X. For matrices, median(X) is a row vector containing the median value of each column. Since median is implemented using sort, it can be costly for large variables.

Std calculates *standard deviation*. For vectors, std(X) is the standard deviation of the elements in vector X. For matrices, std(X) is a row vector containing the standard deviation of each column.

Algorithm:

All three functions are M-files in the *Utility Library*.

See also:

cov, corr

Purpose:

Three dimensional mesh surface.
Evaluate functions of two variables.

Synopsis:

```
mesh(Z)
[X,Y] = meshdom(x,y)
```

Description:

Mesh creates mesh surfaces like the one on the cover of this guide. Mesh(Z) produces a 3-dimensional perspective plot of the values in matrix Z as heights above a plane. For example, mesh(eye(14)) shows an identity matrix.

Meshdom generates X and Y arrays for use with mesh.

[XX,YY] = meshdom(x,y) transforms the domains specified by vectors x and y into arrays XX and YY for evaluating functions of two variables with 3-d mesh plots.

Examples:

To evaluate the function

$$z = x\, e^{(-x^2 - y^2)}$$

over the range

$$-2 \le x \le 2, \quad -2 \le y \le 3$$

use the statements

```
[x,y] = meshdom(-2:.2:2, -2:.2:3);
z = x .* exp(-x.^2 - y.^2);
mesh(z)
```

Algorithm:

Meshdom is an M-file in the *Utility Library*.

See also:

format +, plot, title

Purpose:

Extensibility, programming, functions, and procedures.

Description:

Script files:

A *script file* is an external file that contains a sequence of MATLAB statements. By typing the filename, subsequent MATLAB input is obtained from the file. *Script files* must have a filetype of .m . To make a *script file* into a function, see below.

Function files:

New functions may be added to MATLAB's vocabulary if they are expressed in terms of other existing functions. The commands and functions that comprise the new function are put in a file whose name defines the name of the new function, with a file type of .m appended. A line is put at the top of the file that contains the syntax definition for the new function. For example, the existence of a file on disk called stat.m with:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = sum(x) ./ n;
stdev = sqrt(sum(x^2) / n - mean^2);
```

defines a new function called stat that calculates the mean and standard deviation of a vector. The variables within the body of the function are all local variables.

When a function is used that MATLAB does not recognize, it will search for a file by the same name on disk. If it is found, the function is compiled into memory for subsequent use.

If echo is enabled, the file will be interpreted instead of compiled so that each input line can be viewed as it is executed. Clear can be used to remove the function from memory.

In general, if you input the name of something to MATLAB, for example by typing foo, the MATLAB interpreter:

[1] Looks to see if foo is a variable.

[2] Checks if foo is a built-in function.

[3] Looks in the current directory for a file named foo.m.

[4] Looks in the directories specified by the environment symbol MATLABPATH for a file named foo.m.

Examples:

Here is a function that implements pinv, a pseudo-inverse function:

```
function X = pinv(A,tol)
% Pseudo-inverse,
% Ignore singular values <= tol.
% Default tol = max(size(A)) * s(1) * eps.
[U,S,V] = svd(A);
S = diag(S);
if (nargin == 1)
  tol = max(size(A)) * S(1) * eps;
end
r = sum(S > tol);
if (r == 0)
  X = zeros(A');
else
  S = diag(ones(r,1)./S(1:r));
  X = V(:,1:r)*S*U(:,1:r)';
end
```

See also:

type, echo, nargin, nargout, what.

Purpose:

Matrix norms.

Synopsis:

```
norm(X)
norm(X,p)
norm(X,'fro')
```

Description:

The norm of a matrix is a scalar that gives some measure of the "bigness" of the numbers in the matrix. Several different types of norms may be calculated:

When the argument X is a matrix:

```
norm(X) is the largest singular value of X.
norm(X,1) is the 1-norm, or largest column sum of X,
        max(sum(abs(real(X))+abs(imag(X))),
norm(X,2) is the same as norm(X).
norm(X,inf) is the infinity norm, or largest row sum of X
         max(sum(abs(real(X'))+abs(imag(X'))),
norm(X,'fro') is the F-norm, sqrt(sum(diag(X'*X))).
```

When the argument is a vector, slightly different rules apply:

```
norm(V,p) = sum(abs(V).^p)^(1/p)
norm(V) = norm(V,2)
Norm(V)/sqrt(n) is the root-mean-square (RMS) value.
norm(V,inf) = max(abs(V))
norm(V,-inf) = min(abs(V))
```

See also:

cond, rcond, svd, min, max

Purpose:

Number to string conversion.
Formatted output to screen or file.

Synopsis:

num2str(x), int2str(x)
sprintf('*format*',x), sprintf('*format*',x,y,z)
fprintf('*format*',x), fprintf('*format*',x,y,z)
fprintf('*filename*','*format*',x,...)

Legal in *format* specifiers are %e, %f, and %g.

Description:

Num2str, int2str, and sprintf convert numbers to their MATLAB string representations and are useful for labeling and titling plots with numeric values. Fprintf converts numbers and writes them to the screen or to a file.

s = num2str(x) converts the scalar number x into a string representation t with about 4 digits of precision and an exponent if required.

s = int2str(x) converts an integer to a string with integer format.

s = sprintf('*format*',x) gives more control over the format by converting the value of x to string representation s, according to control string *format*. The control string is comprised of ordinary characters, which are simply copied to the output string, and conversion specifiers, which each cause conversion of the next successive argument to sprintf. There can be from zero to three arguments after the control string. Legal conversion specifiers are %e, for exponential notation, %f, for fixed point notation, and %g which uses %e or %f, whichever is shorter. Between the % and the conversion character there may be [1]:

A minus sign, which specifies left adjustment of the converted argument in its field.

A digit string specifying a minimum field width.

A period, which separates the field width from the next digit string.

A digit string specifying the precision (number of digits to the right of the decimal point).

The character string \n signifies *newline* within a control string. For more information see the C language *stdio* routine *sprintf*.

Fprintf does output conversion like sprintf except that it sends the results to the screen, or to a file if a *filename* is given. If a file already exists, the output is appended.

On MS-DOS computers, COM1, PRN and other devices can be used as *filenames*, so fprintf can be used for some fairly sophisticated effects like sending characters to modems.

Examples:

Here are some examples of fprintf:

```
fprintf('A unit circle has circumference %g\n\n',2*pi)

A unit circle has circumference 6.283186

fprintf('X is %6.2f meters\n or %8.3g millimeters\n',9.9,9900)

X is  9.900 meters
 or 9900.000 millimeters
```

Algorithm:

Num2str and int2str are M-files in the *Utility Library* that use sprintf.

Sprintf and fprintf access low level C routines directly, with no input checking. It may be possible to wreak havoc if they are used incorrectly.

See also:

setstr

References:

Any C language reference. The original is:

[1] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice Hall, Inc., 1978.

Purpose:

Handy utility matrices.

Synopsis:

```
ones(n)
ones(m,n)
ones(A)
zeros, eye, magic
```

Description:

These functions generate special matrices:

ones — All ones. Ones(n) is an n-by-n matrix of ones. Ones(m,n) is an m-by-n matrix of ones. Ones(A) is the same size as A and all ones.

zeros — All zeros. Zeros(n) is an n-by-n matrix of zeros. Zeros(m,n) is an m-by-n matrix of zeros. Zeros(A) is the same size as A and all zeros.

eye — Identity matrix. Eye(n) is the n-by-n identity matrix. Eye(m,n) is an m-by-n matrix with 1's on the diagonal and zeros elsewhere. Eye(A) is the same size as A.

magic — Magic square. Magic(n) is an n-by-n matrix constructed from the integers 1 through n^2 with equal row and column sums. A magic square, scaled by its magic sum, is *doubly stochastic*.

Examples:

Try this:

```
for n = 3:20
   A = magic(n);
   plot(A,'-')
   r(n) = rank(A);
end
r
```

Limitations:

There is a syntactic difficulty with these functions that can bite you. Suppose A is a scalar with value 4. What size, then, is ones(A)? Using the definitions above, it could be either 4-by-4 or 1-by-1. The

answer is 4-by-4, of course, but the point is that ones(A) is the same size as A, *except* when A is a scalar. The implication is that if you are writing a general matrix algorithm, and you want it to work for the scalar case as well, use ones(n) preceded by a call to size(A) to find n explicitly. The functions zeros and eye should be treated similarly.

See also:

rand

Purpose:

Orthogonalization and null space.

Synopsis:

```
orth(X)
null(X)
```

Description:

Orth finds the range space of a matrix. The statement

```
Q = orth(X)
```

returns an orthonormal basis for the range of X. The columns of Q span the same space as the columns of X, matrix Q is orthogonal,

```
Q'*Q = eye(X)
```

and the number of columns of Q is the rank of X.

Null finds the null space of a matrix. The statement

```
Q = null(X)
```

returns an orthonormal basis for the null space of X, so that

```
Q'*Q = eye(X)
X*Q = 0
```

and the number of columns of Q is the nullity of X.

Algorithm:

Orth and null are M-files in the *Utility Library*.

See also:

qr

Purpose:

Memory garbage collection and compaction.

Synopsis:

pack

Description:

If you get the Out of memory message from MATLAB, the pack command may find you some free memory, without forcing you to delete variables.

Pack performs memory garbage collection. Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store that large variable you want to create.

The pack command:

[1] Saves all variables on disk in a temporary file called pack.tmp,

[2] Clears all variables and functions from memory,

[3] Reloads the variables back from pack.tmp, and

[4] Deletes the temporary file pack.tmp.

This results in your workspace having the variables packed or compressed into the minimum memory required, with no wasted space.

If you do a pack and there is still not enough free memory to proceed, you must clear some variables. If you are running on a computer that can hold more memory, and you run out of memory often, it may be time to add more.

See also:

clear

Purpose:

Pause until key is hit.

Synopsis:

```
pause
pause(n)
```

Description:

Pause causes M-files to stop and wait for the user to strike any key before continuing.

Pause(n) pauses for n seconds before continuing.

Examples:

An important use of pause is to halt M-files temporarily when graphics commands are encountered. If pause is not used, the graphics will be visible only momentarily. Here is an example of this:

```
t = -300:300;
plot(t.*sin(t),t.*cos(t)), pause
plot(rand(1,2000),rand(1,2000),'.'), ..
 title('Starry night'), pause
```

Try these plots; they are quite pleasant.

Purpose:

Pseudoinverse.

Synopsis:

```
pinv(X)
pinv(X,tol)
```

Description:

X = pinv(A) produces the Moore-Penrose pseudoinverse, which is a matrix X of the same dimensions as A' so that,

```
A*X*A = A
X*A*X = X
```

and A*X and X*A are Hermitian. The computation is based on svd(A) and any singular values less than a tolerance are treated as zero. The default tolerance is

```
tol = max(size(A)) * norm(A) * eps
```

This tolerance may be overridden with X = pinv(A,tol).

Algorithm:

Pinv is an M-file in the *Utility Library*.

See also:

rank, inv, svd, qr

Purpose:

Linear, logarithmic, and polar engineering graph paper.

Synopsis:

plot(y)
plot(x,y)
plot(x,y,'*line-type*')
plot(x1,y1,x2,y2, ...)
loglog(x,y), semilogx(x,y), semilogy(x,y)
polar(theta,rho)
bar(y)

Description:

Plot makes linear plots of vectors and matrices.

Plot(X,Y) plots vector X versus vector Y. If X or Y is a matrix, then the vector is plotted versus the rows or columns of the matrix, whichever lines up. If both X and Y are matrices of the same size, the columns of X are plotted versus the columns of Y.

Plot(X1,Y1,X2,Y2) is another way of producing multiple lines on the plot.

Plot(X1,Y1,':',X2,Y2,'+') uses a dotted line for the first curve and the point symbol + for the second curve.

Plot(X1,Y1,'r',X2,Y2,'+g') uses a red line for the first curve and a green + symbol for the second curve, if the system supports color. Other line, point, and color types are:

LINE-TYPES		POINT-TYPES		COLORS	
solid	-	point	.	red	r
dashed	--	plus	+	green	g
dotted	:	star	*	blue	b
dashdot	-.	circle	o	white	w
		x-mark	x	invisible	i

Plot(Y) plots the columns of Y versus their index. If Y has non-zero imaginary part, plot(Y) is equivalent to plot(real(Y),imag(Y)). For other uses of plot, the imaginary part is ignored.

The commands loglog, semilogx, semilogy, and polar are used exactly the same as plot, but they result in graphs on different scales.

Polar(theta, rho) makes a polar coordinate plot of the angle theta, in radians, versus the radius rho. Grid will draw polar grid lines.

plot,loglog,semilog,polar,bar

Loglog(x,y) makes a plot using log-log scales.

Semilogx(x,y) makes a plot using semi-log scales. The x-axis is $\log_{10}$ while the y-axis is linear.

Semilogy(x,y) makes a plot using semi-log scales. The y-axis is $\log_{10}$ while the x-axis is linear.

Bar(y) draws a bar chart of the elements of vector y.

See also:

title, xlabel, ylabel, text, grid, shg, clg, hold, axis, mesh, subplot.

Purpose:

Polynomial fitting.

Synopsis:

```
p = polyfit(x,y,n)
```

Description:

Given data in a vector x, polyfit finds a polynomial p such that $p(x)$ fits the data in a vector y in a least-squares sense.

Polyfit(x,y,n) returns the coefficients, in descending powers of x, of the n-th order polynomial that fits vector y to x.

Examples:

Select a polynomial and generate some noisy data:

```
p = [1, -6, 11, -6];
x = 0:.25:4; rand('normal')
y = polyval(p,x) + rand(x);
```

Now fit a polynomial and plot the actual data and fitted result:

```
c = polyfit(x,y,3);
fit = polyval(c,x);
plot(x,fit,x,y,'o')
```

Algorithm:

In general, a polynomial fit to data in vectors x and y is a function, p, of the form:

$$p(x) = c_1 x^d + c_2 x^{d-1} + \cdots + c_n$$

The degree is d and the number of coefficients is $n = d+1$. The coefficients $c_1, c_2, \cdots c_n$ are determined by solving a system of simultaneous linear equations, $Ac = y$, where the columns of A are successive powers of the x vector. The solution to the equations $Ac = y$ is obtained using the MATLAB "matrix division" operator, c = A\y.

This algorithm is implemented in the M-file polyfit in the *Utility Library*.

In the *regression* problem, other functions, usually multivariate functions of the columns of the data matrix, are fit to the data by forming

the appropriate *A* matrix. For example, if X is a matrix whose rows are observations,

```
A = [X(:,1), X(:,2).^2, sin(X(:,3)), ones(m,1)];
coef = Ay;
```

finds the regression coefficients for the indicated function.

See also:

poly, roots, conv, polyval

Purpose:

Polynomial evaluation.

Synopsis:

```
polyval(V,s)
polyvalm(X,S)
```

Description:

If V is a vector whose elements are the coefficients of a polynomial in descending powers, then polyval(V,s) is the value of the polynomial evaluated at s. If s is a matrix or vector, the polynomial is evaluated at each of the elements.

Polyvalm(X,S), with S a matrix, evaluates the polynomial in a matrix sense.

Examples:

The polynomial $p(s) = 3s^2 + 2s + 1$ is evaluated at s=5 with

```
p = [3 2 1];
polyval(p,5)
```

which results in

```
ans =
   86
```

The *Cayley-Hamilton theorem* states that every square matrix satisfies its own characteristic equation. We can show this by forming the quantity polyvalm(poly(A),A), which is zero (within roundoff error), for any matrix A.

Algorithm:

Polyval and polyvalm use Horner's method and are M-files in the *Utility Library*.

See also:

conv, roots, poly, residue

Purpose:

Graphics hardcopy.

Synopsis:

prtsc
print
meta *file*
meta

Description:

Three commands, prtsc, print and meta, provide general hardcopy capabilities.

• Prtsc initiates a *print screen*. The graph window screen is dumped to the printer on a pixel-by-pixel basis, resulting in hardcopy with the same resolution as the computer screen. Prtsc('ff') is the same as prtsc, but sends an extra form-feed to the printer after the plot is finished. On most personal computers, holding the Shift key down and pressing the PrtSc key also dumps the screen to a printer.

• Meta *file* opens a high resolution graphics metafile, using the specified filename, and writes the current graph to it for later processing. Subsequent meta commands append to the previously specified filename. The metafile may be processed later using the graphics post processor (GPP) program.

• Print sends a high resolution copy of the current plot to the printer. On some machines with limited memory, this feature may not be available.

Algorithm:

Print is an M-file that saves the graph using meta and invokes the graphics post processor. You may modify print.m to select your target hardcopy device.

See also:

These three functions vary from machine to machine. See the machine specific section 1 of this guide.

Purpose:

QR decomposition.

Synopsis:

```
[Q,R] = qr(X)
[Q,R,E] = qr(X)
qr(X)
```

Description:

Qr is used for the orthogonal-triangular decomposition of a matrix. The "QR" factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of a real orthonormal matrix or a complex unitary matrix and an upper triangular matrix.

[Q,R] = qr(X) produces an upper triangular matrix R of the same dimension as X and a unitary matrix Q so that X = Q*R.

[Q,R,E] = qr(X) produces a permutation matrix E, an upper triangular R with decreasing diagonal elements and a unitary Q so that X*E = Q*R.

By itself, qr(X) returns the output of ZQRDC. Triu(qr(X)) is R.

Examples:

Start with

```
A =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

We have chosen a rank-deficient matrix; the middle column is the average of the other two columns. The rank deficiency is revealed by the factorization.

```
[Q,R] = qr(A)
```

gives

```
Q =
   -0.0776   -0.8331    0.5444    0.0605
   -0.3105   -0.4512   -0.7709    0.3251
   -0.5433   -0.0694   -0.0913   -0.8317
   -0.7762    0.3124    0.3178    0.4461

R =
  -12.8841  -14.5916  -16.2992
         0   -1.0413   -2.0826
         0         0    0.0000
         0         0         0
```

The triangular structure of R gives it zeros below the diagonal; the zero on the diagonal in R(3,3) implies that R and consequently A do not have full rank.

The QR factorization is used in solving linear systems with more equations than unknowns. For example

```
b =
   1
   3
   5
   7
```

The linear system $Ax = b$ is four equations in only three unknowns. The best solution in a least squares sense is computed by

```
x = A\b
```

which produces

```
Warning: Rank deficient, rank = 2, tol = 1.4594E-014

x =
   0.5000
   0.0000
   0.1667
```

We are warned about the rank deficiency. The quantity tol is a tolerance used in deciding that a diagonal element of R is negligible. The solution x was computed using the factorization and the two steps

```
y = Q'*b;
x = R\y
```

If we were to check the computed solution by forming A*x, we would find that it equals b to within roundoff error. This tells us that even though the simultaneous equations $Ax = b$ are overdetermined and rank deficient, they happen to be consistent. There are infinitely many solution vectors x; the QR factorization has found just one of them.

Algorithm:

Qr uses the LINPACK routines ZQRDC and ZQRSL. ZQRDC computes the QR decomposition, while ZQRSL applies the decomposition.

See also:

orth, null

References:

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose:

QZ algorithm.

Synopsis:

```
[AA, BB, Q, Z, V] = qz(A,B)
```

Description:

The qz function gives access to what are normally only intermediate results in the computation of generalized eigenvalues. For square matrices A and B, the function

```
[AA, BB, Q, Z, V] = qz(A,B)
```

produces upper triangular matrices AA and BB, and matrices Q and Z containing the products of the left and right transformations, such that

```
Q*A*Z = AA
Q*B*Z = BB
```

Qz also returns the generalized eigenvector matrix V.

The α's and β's comprising the generalized eigenvalues are the diagonal elements of AA and BB so that

```
A*V*diag(BB) = B*V*diag(AA)
```

Algorithm:

Complex generalizations of the EISPACK routines QZHES, QZIT, QZVAL, and QZVEC implement the QZ algorithm.

See also:

eig

References:

C.B Moler and G.W. Stewart, *An Algorithm for Generalized Matrix Eigenvalue Problems*, SIAM J. Numer. Anal., Vol. 10, No. 2, April 1973.

Purpose:

Random numbers and matrices.

Synopsis:

rand(n)
rand(m,n)
rand('*distribution*')
rand('seed'), rand('seed',n)

The *distribution* can be either uniform or normal.

Description:

Rand generates random numbers and matrices. Rand(n) is an n-by-n matrix with random entries. Rand(m,n) is an m-by-n matrix with random entries. Rand(A) is the same size as A. Rand with no arguments is a scalar whose value changes each time it is referenced.

Ordinarily, random numbers are uniformly distributed in the interval (0.0,1.0). Rand('normal') switches to a normal distribution with mean 0.0 and variance 1.0. Rand('uniform') switches back to the uniform distribution.

Rand('seed') returns the current value of the seed for the generator. Rand('seed',n) sets the seed to n. Rand('seed',0) resets the seed to 0, its value when MATLAB is first entered.

Algorithm:

The uniformly distributed random numbers are obtained from a C version of the machine-independent random number generator URAND described in [1]. The normally distributed numbers are obtained using a transformation also described in [1].

See also:

ones,zeros,eye

References

[1] G.E. Forsythe, M.A. Malcolm and C.B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1977.

Purpose:

Rank of a matrix.

Synopsis:

```
rank(X)
rank(X,tol)
```

Description:

Rank calculates the rank of a matrix.

k = rank(X) is the number of singular values of X that are larger than max(size(X)) * norm(X) * eps.

k = rank(X,tol) is the number of singular values of X that are larger than tol.

Algorithm:

There are a number of ways to compute the rank of a matrix. MATLAB uses the method based on the singular value decomposition described in chapter 11 of the LINPACK guide. The SVD algorithm is the most time-consuming but also the most reliable.

Here is the rank algorithm, as implemented in an M-file in the *Utility Library*:

```
s = svd(x);
tol = max(size(x)) * s(1) * eps;
r = sum(s > tol);
```

References:

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose:

Rational approximation.

Synopsis:

```
rat(x)
[a,b] = rat(x)
rat(len,max)
```

Description:

Rat is an experimental function which attempts to remove the roundoff error from results that should be "simple" rational numbers.

Rat(X) approximates each element of X by a continued fraction of the form

$$a/b = d_1 + 1/(d_2 + 1/(d_3 + \cdots + 1/d_k))$$

The expansion is carried out while $k \leq$ len and $abs(d_i) \leq$ max. The default values of the parameters are

len = 5, max = 100

Rat(len,max) changes the default values. Increasing either len or max increases the number of possible fractions.

[A,B] = rat(X) produces integer matrices A and B so that

A ./ B = rat(X)

Examples:

A rational fraction approximation to π:

```
[a,b]=rat(pi)

a =
  355

b =
  113
```

Here are some other interesting examples to try:

```
T = hilb(6), X = inv(T)
[A,B] = rat(X)
H = A ./ B, S = inv(H)

d = 1:8,  e = ones(d),  A = abs(d'*e – e'*d)
X = inv(A)
Y = rat(X)
disp(Y)
```

Purpose:

Partial-fraction expansion or residue computation.

Synopsis:

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

Description:

[r,p,k] = residue(b,a) finds the residues, poles and direct term of a partial-fraction expansion of the ratio of two polynomials B and A:

$$\frac{B(s)}{A(s)} = \frac{r(1)}{s-p(1)} + \frac{r(2)}{s-p(2)} + \cdots + \frac{r(n)}{s-p(n)} + k(s)$$

Vectors b and a specify the coefficients of the polynomials in descending powers of s. The residues are returned in column vector r, the pole locations in column vector p, and the direct terms in row vector k.

[b,a] = residue(r,p,k) converts the partial-fraction expansion back to the polynomials B/A.

Algorithm:

Residue is an M-file in the *Utility Library*. First the poles are found using roots. Next, if the fraction is non-proper, the direct term k is found using deconv which performs polynomial long division. Finally, the residues are determined by evaluating the polynomial with individual roots removed.

Limitations:

It won't yet handle repeated roots of A.

See also:

deconv, poly, roots

References:

A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975., pg. 56.

Purpose:

Polynomial roots.
Characteristic polynomial.

Synopsis:

poly(X)
roots(p)

Description:

If A is an n-by-n matrix, poly(A) is an $n+1$ element row vector whose elements are the coefficients of the characteristic polynomial, $det(\lambda I - A)$. The coefficients are ordered in descending powers: if a vector c has $n+1$ components, the polynomial it represents is:

$$c_1\lambda^n + \cdots + c_n\lambda + c_{n+1}$$

If r is a column vector containing the *roots* of a polynomial, poly(r) returns a row vector whose elements are the *coefficients* of the polynomial.

If c is a row vector containing the *coefficients* of a polynomial, roots(c) is a column vector whose elements are the *roots* of the polynomial.

For vectors, roots and poly are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples:

Polynomials are represented in MATLAB as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix

```
A =
   1   2   3
   4   5   6
   7   8   0
```

is returned in a row vector by poly:

```
p = poly(A)

p =
   1   -6   -72   -27
```

The roots of this equation (eigenvalues of matrix A) are returned in a column vector by roots:

```
r = roots(p)

r   =
   12.1229
   -5.7345
   -0.3884
```

These may be reassembled into a polynomial with poly:

```
p2 = poly(r)

p2 =
   1   -6   -72   -27
```

Roots(poly(1:17)) generates the largest instance of Wilkinson's famous example that can be handled with IEEE precision arithmetic.

Algorithm:

Roots and poly are M-files in the *Utility Library*. The algorithms employed for poly and roots illustrate an interesting aspect of the modern approach to eigenvalue computation. Poly(A) generates the characteristic polynomial of A and roots(poly(A)) finds the roots of that polynomial, which are, of course, the eigenvalues of A. But both poly and roots use EISPACK eigenvalue subroutines, which are based on similarity transformations. So the classical approach which characterizes eigenvalues as roots of the characteristic polynomial is actually reversed.

If A is an n-by-n matrix, poly(A) produces the coefficients $c(1)$ through $c(n+1)$, with $c(1) = 1$, in

$$det(\lambda I - A) = c_1\lambda^n + \cdots + c_n\lambda + c_{n+1}$$

The algorithm is expressed compactly in an M-file:

```
z = eig(A);
c = zeros(n+1,1); c(1) = 1;
for j = 1:n
    c(2:j+1) = c(2:j+1) – z(j)*c(1:j);
end
```

This recursion is easily derived by expanding the product

$$(\lambda - \lambda_1)(\lambda - \lambda_2)...(\lambda - \lambda_n)$$

It is possible to prove that poly(A) produces the coefficients in the characteristic polynomial of a matrix within roundoff error of A. This is true even if the eigenvalues of A are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial which do not use the eigenvalues do not have such satisfactory numerical properties.

If c is a vector with $n+1$ components, roots(c) finds the roots of the polynomial of degree n,

$$p(\lambda) = c_1\lambda^n + \cdots + c_n\lambda + c_{n+1}$$

The algorithm simply involves computing the eigenvalues of the companion matrix:

```
A = diag(ones(n–1,1)),–1);
A(1,:) = –c(2:n–1)/c(1);
eig(A)
```

It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix A, but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in C.

See also:

polyval, conv, residue

Purpose:

Rounding, signum, and remainder elementary functions.

Synopsis:

round(X), fix(X), ceil(X), floor(X)
sign(X)
rem(X,Y)

Description:

These are elementary functions that operate element-wise on matrices.

Round(X) rounds the elements of X to the nearest integers.

Fix(X) rounds the elements of X to the nearest integers towards zero.

Ceil(X) rounds the elements of X to the nearest integers towards $+\infty$.

Floor(X) rounds the elements of X to the nearest integers towards $-\infty$.

Sign is the *signum* function. For each element of X, sign(X) returns 1 if the element is greater than zero, 0 if it equals zero and -1 if it is less than zero. For complex X, sign(X) = X ./ abs(X).

Rem calculates remainders. If n = fix(x/y) is the integer nearest the exact value x/y, the remainder, rem(x,y), is x – y * n.

See also:

trig, exp, log

Purpose:

Reduced row echelon form.

Synopsis:

```
rref(A)
rref(A,tol)
rrefmovie(A)
```

Description:

Rref(A) produces the reduced row echelon form of A using Gauss-Jordan elimination with partial pivoting. A default tolerance of

```
tol = max(size(A))*eps*norm(A,inf)
```

is used to test for negligible column elements.

Rref(A,tol) uses tol for the tolerance.

Rrefmovie(A) shows a "movie" of the algorithm working.

Examples:

Use rref on a rank-deficient magic square,

```
A = magic(4), R = rref(A)
A =
     16     2     3    13
      5    11    10     8
      9     7     6    12
      4    14    15     1
R =
      1     0     0     1
      0     1     0     3
      0     0     1    -3
      0     0     0     0
```

Algorithm:

Rref is an M-file in the *Utility Library*.

See also:

rank, lu, inv

Purpose:

Set to string.

Synopsis:

setstr(t)

Description:

The statement

```
t = 'Hello, World.'
```

creates a vector whose components are the ASCII codes for the characters. The column dimension of t is the number of characters. It is no different than other MATLAB vectors, except that when displayed, you see text and not the decimal ASCII codes:

```
t

t =
  Hello, World.
```

Associated with each MATLAB variable is a flag that, if set, tells the MATLAB output routines that the variable should be displayed as text.

t = abs(t) is one way of clearing the flag so the vector is displayed in its decimal ASCII representation.

t = setstr(t) sets the flag back to text display.

Strange behavior may be encountered if setstr is used on vectors that contain numbers outside of 0 to 255, or non-integers.

A quote within a string is indicated by two quotes.

Purpose:

Size of variables, length of vectors.

Synopsis:

```
size(X)
[m,n] = size(X)
length(X)
```

Description:

If X is an *m*-by-*n* matrix, then size(X) returns the two element row vector [m n].

Size can also be used with a multiple assignment,

```
[m,n] = size(X)
```

in which case m and n are returned separately.

Length(X) returns the length of vector X. It is equivalent to max(size(X)).

Purpose:

Sorting.

Synopsis:

```
sort(X)
[Y,I] = sort(X)
```

Description:

Sort(X) sorts each column of X in ascending order.

[Y,I] = sort(X) also returns matrix I containing the indexes used in the sort. If X is a vector, Y = X(I).

When X is complex, the elements are sorted by abs(X).

Examples:

To sort the eigenvalues and eigenvectors of a real symmetric matrix:

```
[V,D] = eig(A);
[lambda,k] = sort(diag(D));
V = V(:,k);
```

Purpose:

Cubic spline data interpolation.

Synopsis:

```
yi = spline(x,y,xi)
pp = spline(x,y)
```

Description:

Spline interpolates between data points using cubic spline fits.

If x and y are vectors containing coarsely spaced data, and xi contains a new, more finely spaced abscissa vector, then

```
yi = spline(x,y,xi)
```

uses cubic spline interpolation to find a vector yi corresponding to xi.

pp = spline(x,y) returns the *pp*-form of the cubic spline interpolant, for later use with ppval, etc.

Examples:

Here's an example that generates a coarse sine curve, then interpolates over a finer abscissa:

```
x = 0:10; y = sin(x);
xi = 0:.25:10;
yi = spline(x,y,xi);
plot(x,y,'o',xi,yi)
```

Algorithm:

Spline is an M-file in the *Utility Library*. It uses the M-files ppval, mkpp, and unmkpp. These routines form a small suite of functions for working with piece-wise polynomials. Spline uses them in a fairly simple fashion to perform cubic spline interpolation. For access to the more advanced features, see the M-files.

See also:

table1, polyfit

References:

C. de Boor, *A Practical Guide to Splines*, Springer-Verlag, 1978.

Purpose:

Starting and ending.

Synopsis:

matlab.m
startup.m
quit, exit

Description:

The M-scripts matlab.m and startup.m are executed automatically when MATLAB is invoked. Physical constants, engineering conversion factors, i=sqrt(−1), or anything else you would like pre-defined in your workspace may be put in these files. On multi-user or networked systems, matlab.m is reserved for use by the system manager. It can be used to implement system wide definitions and messages.

Quit terminates MATLAB. The workspace is not saved. See save.

Exit is a synonym for quit.

Typing CTRL-Z, the end-of-file character, will also terminate MATLAB.

Algorithm:

Only matlab.m is actually invoked by MATLAB at startup. However, matlab.m contains the statements

```
if exist('startup')==2
    startup
end
```

that invoke startup.m. This process may be extended to create additional startup M-files, if required.

See also:

!

Purpose:

Split the graph window into sub-windows.

Synopsis:

subplot(*mnp*)
subplot

Description:

Subplot(*mnp*), where *mnp* is a three digit number, breaks the graph window into an *m*-by-*n* grid of small sub-windows, and selects the *p*-th window for the current plot. Windows are numbered from left to right, top to bottom.

Subplot(111), or just subplot, returns to the default single-window configuration.

Examples:

The statements

```
t = -20:.2:20; y = sin(t)./t;
subplot(211), plot(y)
subplot(212), plot(diff(y)/.2)
```

plot the *sinc* function on the top half of the screen and its derivative on the bottom half.

Subplot(121) and subplot(122) split the display into left and right halves. Window(221), 222, 223, and 224 break the display into four small plotting boxes.

Limitations:

The graph window can be split into four windows, at most. The limits are $m \leq 2$ and $n \leq 2$. If $m > 2$ or $n > 2$ or $p > mn$, the value is out of range and subplot will print the error message:

```
Command option is unknown.
```

See also:

plot, clg, hold

Purpose:

Sums and products.

Synopsis:

sum(X), prod(X), cumsum(X), cumprod(X)

Description:

sum	For vectors, sum(X) is the sum of the elements of X. For matrices, sum(X) is a row vector with the sum over each column.
prod	For vectors, prod(X) is the product of the elements of X. For matrices, prod(X) is a row vector with the product over each column.
cumsum	For vectors, cumsum(X) is the cumulative sum of the elements of X. For matrices, cumsum(X) is a matrix containing the cumulative sums over each column.
cumprod	For vectors, cumprod(X) is the cumulative product of the elements of X. For matrices, cumprod(X) is a matrix containing the cumulative products over each column.

Examples:

Sum(diag(X)) is the *trace* of X.

See also:

diff

Purpose:

Singular value decomposition.

Synopsis:

```
svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
```

Description:

Svd computes the matrix singular value decomposition.

[U,S,V] = svd(X) produces a diagonal matrix S of the same dimension as X, with nonnegative diagonal elements in decreasing order, and unitary matrices U and V so that

```
X = U*S*V'
```

By itself, svd(X) returns a vector containing the singular values.

[U,S,V] = svd(X,0) produces the "economy size" decomposition. If X is *m*-by-*n* with $m > n$, then only the first *n* columns of U are computed and S is *n*-by-*n*.

Algorithm:

Svd uses the LINPACK routine ZSVDC.

Diagnostics:

If the limit of 75 QR step iterations is exhausted while seeking a singular value:

```
Solution will not converge.
```

References:

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose:

One-dimensional table look-up.

Synopsis:

table1(tab,x)

Description:

Table1 performs a one-dimensional table look-up using a linear interpolation.

If TAB is a table of numbers, with each column containing a different variable, the function

```
y = table1(TAB,x)
```

returns a linearly interpolated row from TAB, looking up x in the first column of TAB. The row vector is returned with a length one less than the width of the table. (For example, if TAB has dimensions *n*-by-2, table1 returns a scalar.)

Algorithm:

Table1 is an M-file in the *Utility Library*.

Limitations:

The table's first column must increase monotonically.

See also:

spline, polyfit

Purpose:

Graph labels, titles, and grid lines.

Synopsis:

```
title('text')
xlabel('text')
ylabel('text')
text(x,y,'text')
grid
```

Description:

Title('*text*') writes the text as a title at the top of the current plot.

Xlabel('*text*') writes the text on the current plot beneath the x-axis.

Ylabel('*text*') writes the text on the current plot beside the y-axis.

Text(X,Y,'*text*') writes *text* at (X,Y) on the graphics screen, where (X,Y) are in units from the last plot. If X and Y are vectors, text writes the text at all locations given. If *text* is an array the same length as X and Y, text marks each point with the corresponding row of the *text* array.

Text(X,Y,'*text*','sc') interprets the (X,Y) points in *screen coordinates* where (0.0,0.0) is the lower left corner of the screen, and (1.0,1.0) is the upper right.

Grid draws grid lines on the current plot.

Examples:

The statements

```
plot([1 5 10],[1 10 20],'x')
text(5,10,' Action point')
```

annotate the point at (5,10) with the text, while

```
plot(x1,y1,x2,y2)
text(x1,y1,'1'), text(x2,y2,'2')
```

marks two curves so they can be distinguished easily.

See also:

plot, clg, subplot, axis, hold, num2str, int2str

Purpose:

Form Toeplitz matrices.

Synopsis:

toeplitz(c,r)
toeplitz(c)

Description:

A *Toeplitz* matrix is defined by one row and one column. A *symmetric Toeplitz* matrix is defined by just one vector. Toeplitz generates Toeplitz matrices given just the row or column description.

Toeplitz(c,r) is a non-symmetric Toeplitz matrix having c as its first column and r as its first row. If the first elements of c and r are different, a message is printed and the column wins the disagreement.

Toeplitz(c) is the symmetric or Hermitian Toeplitz matrix formed from vector c.

Examples:

Here is a Toeplitz matrix with diagonal disagreement.

```
c = [1   2   3   4   5];
r = [1.5  2.5  3.5  4.5  5.5];
toeplitz(c,r)

Column wins diagonal conflict.

ans =
 1.000   2.500   3.500   4.500   5.500
 2.000   1.000   2.500   3.500   4.500
 3.000   2.000   1.000   2.500   3.500
 4.000   3.000   2.000   1.000   2.500
 5.000   4.000   3.000   2.000   1.000
```

Algorithm:

Toeplitz is an M-file in the *Utility Library*.

Purpose:

Translate data to and from MAT-files.

Synopsis:

translate
!translate

Description:

Translate is a separate program supplied with MATLAB in the *Utility library*. Translate is a tool for getting data in and out of MATLAB from other file formats. It can convert ASCII flat files, binary files, Fortran unformatted files, and DIF files (Data Interchange Format from spreadsheets, etc.) into the special MAT-files used by MATLAB. MAT-files can be loaded directly into MATLAB using the load command.

Translate can be invoked at the operating system level by typing translate. Typing translate from inside MATLAB invokes a short M-file that simply executes !translate.

Translate will prompt you for filenames and other information.

It is possible to build a "response-file" that automates the use of translate.

Examples:

Under MS-DOS and Unix, if a response file is called foo, use

```
translate <foo
```

See also:

load,save

Purpose:

Trigonometric functions.

Synopsis:

sin(x), cos(x), tan(x)
asin(x), acos(x), atan(x), atan2(y,x)
sinh(x), cosh(x), tanh(x)

Description:

The trigonometric functions operate element-wise on matrices. Their domains and ranges include complex numbers, which can lead to unexpected results if used unintentionally.

Sin, cos, and tan return trigonometric functions of radian arguments:

Sin(X) is the *sine* of the elements of X. For complex $z = x + iy$, the complex *sine* is returned:

$$sin(z) = sin(x)cosh(y) + i\ cos(x)sinh(y)$$

Cos(X) is the *cosine* of the elements of X. For complex $z = x + iy$, the complex *sine* is returned:

$$cos(z) = cos(x)cosh(y) - i\ sin(x)sinh(y)$$

Tan(X) is the *tangent* of the elements of X. For complex z, the complex *tangent* sin(z)/cos(z) is returned.

Acos, asin, atan, and atan2 return inverse trigonometric functions in radians:

Acos(X) is the *arccosine* of the elements of X. For real x, such that $abs(x) \leq 1.0$, the result is in the range 0 to π. Complex results are obtained if $abs(x) > 1.0$ for some element, or if x is complex. The complex *arccosine* is defined as:

$$\cos^{-1}(z) = -i\ log(z + i\ sqrt(1-z^2))$$

Asin(X) is the *arcsine* of the elements of X. For real x, such that $abs(x) \leq 1.0$, the result is in the range $-\pi/2$ to $+\pi/2$. Complex results are obtained if $abs(x) > 1.0$ for some element, or if x is complex. The complex *arcsine* is defined as:

$$\sin^{-1}(z) = -i\ log(iz + sqrt(1-z^2))$$

Atan(X) is the *arctangent* of the elements of X. For real x, the result is in the range $-\pi/2$ to $+\pi/2$. If x is complex, the complex

arctangent is returned:

$$\tan^{-1}(z) = \frac{i}{2}\log\frac{(i+z)}{(i-z)}$$

Atan2(Y,X) is the four-quadrant *arctangent* of the real elements of Y./X. The imaginary part is ignored. The result is in the range $-\pi$ to $+\pi$.

The hyperbolic functions sinh, cosh, and tanh are available too, but implemented as M-files in the *Utility Library*.

See also:

exp,log,log10,expm,funm

References:

Churchhill, R.V., Brown, J.W., and Verhey, R.F., *Complex Variables and Applications*, McGraw Hill, 1974.

ans,eps,pi,Inf,NaN,narg

Purpose:

Permanent variables.

Synopsis:

ans - answer when expression is not assigned
eps - machine epsilon
pi - π
Inf - ∞
NaN - Not-a-Number
nargin - number of function input arguments
nargout - number of function output arguments

Description:

Permanent variables have special meaning in MATLAB. They cannot be cleared and are globally accessible inside M-files. Otherwise the normal rules for using variables apply.

ans — Variable created automatically when expressions are not assigned to anything else.

eps — Floating point relative accuracy. A permanent variable whose value is initially the distance from 1.0 to the next largest floating point number. It may be assigned any value, including 0. Eps is used as a default tolerance by pinv and rank. One way to calculate it is:

```
eps = 1;
while (1+eps) > 1
      eps = eps/2;
end
eps = 2*eps
```

pi — pi = 4 *atan* (1) = π

Inf — A permanent variable representing IEEE arithmetic positive infinity. Infinity is obtained from operations like division by zero, 1.0/0.0.

NaN — The IEEE arithmetic representation for Not-a-Number (NaN). A NaN is obtained as a result of undefined operations like 0.0/0.0.

nargin — Inside the body of a *function M-file*, the permanent variable nargin indicates the number of input arguments that were used to call the function. Nargout indicates the number of output arguments.

Algorithm:

Actually, pi, Inf, and NaN are functions, not variables, so that their values can't be damaged, but you can think of them as variables.

See Also:

isnan, finite

Purpose:

Repeat statements an indefinite number of times.

Synopsis:

```
while s
  statements
end
```

Description:

While is used to repeat statements an indefinite number of times. The general format is:

```
while expression
  statements
end
```

The statements are executed while the *expression* has all non-zero elements. The *expression* is usually of the form

expression *rop* expression

where *rop* is ==, <, >, <=, >=, or ~= .

Examples:

The variable eps is a tolerance used to determine such things as near singularity and rank. Its initial value is the *machine epsilon*, the distance from 1.0 to the next largest floating point number on your machine. Its calculation demonstrates while loops:

```
eps = 1;
while (1+eps) > 1
      eps = eps/2;
end
eps = eps*2
```

See also:

for, if, end, break, return, any, all

Purpose:

Directory of the variables in memory.
Directory of the M-files in the current disk directory.
Check if a variable or file *exists*.

Synopsis:

who, whos
what
exist('*item*')

The *item* can be either a variable or a file.

Description:

Who lists the variables currently in memory.

Whos lists the current variables, their sizes, and whether they have non-zero imaginary parts.

What shows a directory listing of the M-files and MAT-files on the disk in the current directory. Files with other file-types are not shown.

Exist('A') returns 1 if A exists as a variable in the workspace, 2 if A.m is a file on disk, and 0 if A doesn't exist. Note that the variable name must be in quotes.

See also:

dir, help

Purpose:

Matrix and array arithmetic.

Synopsis:

```
+ - * / \ ^ '
+ -.* ./ .\ .^ .'
```

Description:

Matrix arithmetic operations are obtained using the symbols +, −, *, /, \, ^. Element-by-element array arithmetic operations are indicated by preceding the operator with a period '.' resulting in .*, .^, ./, and .\ . Here is a description of each of the six operators:

+ Addition, X + Y. X and Y must have the same dimensions unless one is a scalar. A scalar (1-by-1 matrix) can be added to anything.

− Subtraction, X − Y. X and Y must have the same dimensions unless one is a scalar. A scalar can be subtracted from anything.

* Matrix multiplication, X * Y. A scalar may multiply anything. Otherwise, the number of columns of X must equal the number of rows of Y. Element-by-element multiplication is indicated with X .* Y.

\ Backslash or matrix left division. If A is a square matrix, A\B is roughly the same as inv(A)*B, except it is computed in a different way. If A is an n-by-n matrix and B is a column vector with n components, or a matrix with several such columns, then X = A\B is the solution to the equation $AX = B$ computed by Gaussian elimination. A warning message is printed if A is badly scaled or nearly singular.

If A is an m-by-n matrix with $m \neq n$ and B is a column vector with m components, or a matrix with several such columns, then X = A\B is the solution in the least squares sense to the under- or overdetermined system of equations $AX = B$. The effective rank, k, of A, is determined from the QR decomposition with pivoting. A solution X is computed which has at most k nonzero components per column. If $k < n$ this will usually not be the same solution as pinv(A)*B.

If A and B have the same dimensions, then A.\B has elements $a(i,j)\backslash b(i,j)$ denoting element-by-element division. For example, C = A .\ B is the matrix with elements $c(i,j) = b(i,j)/a(i,j)$.

/ Slash or matrix right division. B/A is roughly the same as B*inv(A). More precisely, B/A = (A'\B')'. See \. C = A ./ B is the matrix with elements $c(i,j) = a(i,j)/b(i,j)$.

^ Powers. X^p is X to the power p if p is a scalar. If p is an integer greater than one, the power is computed by repeated multiplication. For other values of p, the calculation involves eigenvalues and eigenvectors, such that if [V,D] = eig(X), then X^p = V*D.^p/V.

If P is a matrix, x^P is x raised to the matrix power P using eigenvalues and eigenvectors. X^P, where X and P are matrices, is an error.

Element-by-element powers are obtained with X .^ Y.

The quote symbol, ', indicates transposition. X' is the matrix, or complex conjugate transpose. X.' is the array, or non-conjugate transpose.

Examples:

A\eye(A) is one way to find the inverse of A.

For the non-square case, A\eye(A) produces a generalized inverse of A.

Algorithm:

If A is square, the LINPACK routines ZGECO and ZGESL are used to solve the general nonsymmetric simultaneous linear equation problems B/A, and A\B. ZGECO uses Gaussian elimination with partial pivoting to compute the LU factorization of A and then estimates its condition. ZGESL uses the LU factorization of matrix A to solve the linear system $AX = B$, or $A'X = B$.

If A is not square, the LINPACK routines ZQRDC and ZQRSL are used to solve the over- or under-constrained least squares problem. ZQRDC computes the QR decomposition of matrix A, while ZQRSL applies the decomposition to B.

For more information, see the *LINPACK User's Guide*.

+ − * / \ ^ '

Diagnostics:

From matrix division, if a square A is singular:

```
Matrix is singular to working precision.
```

If the inverse was found, but is not reliable:

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = xxx
```

From matrix division, if a non-square A is rank deficient:

```
Warning: Rank deficient, rank = xxx  tol = xxx
```

See also:

inv, qr, det, lu, rcond, orth, rref
< <= > >= == ~=
& | ~

References:

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose:

Relational operations.

Synopsis:

```
< <= > >= == ~=
```

Description:

The relational operators are <, <=, >, >=, ==, and ~=. Relational operators do element-by-element comparisons between two matrices. They return a matrix of the same size, with elements set to one where the relation is true, and elements set to zero where it is not.

The relational operators have precedence midway between the logical operators and the arithmetic operators.

See also:

find, any, all, & | ~

Purpose:

Logical operations.

Synopsis:

& | ~

Description:

The symbols & , |, and ~ are the logical operators AND, OR, and NOT. They work element-wise on matrices, with 0 representing FALSE and anything non-zero regarded as TRUE. A & B logically ANDs the elements, A | B does a logical OR, and ~A complements the elements of A.

The logical operators have the lowest precedence, with relational operators and arithmetic operators being higher.

Examples:

Here are two scalar expressions that illustrate precedence relationships:

```
1 & 0 + 3
3 > 4 & 1
```

They evaluate to 1 and 0 respectively, and are equivalent to

```
1 & (0 + 3)
(3 > 4) & 1
```

See also:

find, any, all, < <= > >= == ~=

Purpose:

Special characters.

Synopsis:

[ ] ( ) , . ' ; % !

Description:

[] Brackets are used in forming vectors and matrices.
[6.9 9.64 sqrt(–1)] is a vector with three elements separated by blanks. [6.9, 9.64, sqrt(–1)] is the same thing.
[1+J 2–J 3] and [1 +J 2 –J 3] are not the same. The first has three elements, the second has five.
[11 12 13; 21 22 23] is a 2-by-3 matrix. The semicolon ends the first row.

Vectors and matrices can be used inside [ ] brackets. [A B;C] is allowed if the number of rows of A equals the number of rows of B and the number of columns of A plus the number of columns of B equals the number of columns of C. This rule generalizes in a hopefully obvious way to allow fairly complicated constructions.

A = [ ] stores an empty matrix in A.

For the use of [and] on the left of an "=" in multiple assignment statements, see lu, eig, svd and so on.

() Parentheses are used to indicate precedence in arithmetic expressions in the usual way. They are used to enclose arguments of functions in the usual way. They are also used to enclose subscripts of vectors and matrices in a manner somewhat more general than the usual way. If X and V are vectors, then X(V) is [X(V(1)), X(V(2)), ..., X(V(n))]. The components of V are rounded to nearest integers and used as subscripts. An error occurs if any such subscript is less than 1 or greater than the dimension of X. Some examples:

X(3) is the third element of X.
X([1 2 3]) is the first three elements of X. So is
X([sqrt(2), sqrt(3), 4*atan(1)]).

If X has n components, X(n:–1:1) reverses them. The same indirect subscripting is used in matrices. If V has m components and W has n components, then A(V,W) is the m by n matrix formed from the elements of A whose subscripts are the

elements of V and W. For example A([1,5],:) = A([5,1],:) interchanges rows 1 and 5 of A.

= Used in assignment statements. == is the relational EQUALS operator. See < <= > >= == ~=.

' Matrix transpose. X' is the complex conjugate transpose of X. X.' is the non-conjugate transpose.

Quote. 'any text' is a vector whose components are the ASCII codes for the characters. A quote within the text is indicated by two quotes.

. Decimal point. 314/100, 3.14 and .314e1 are all the same.

Element-by-element operations are obtained using .* , .^ , ./ , or .\. See + - * / \ ^ '.

Two or more points at the end of a line indicate continuation.

, Comma. Used to separate matrix subscripts and function arguments. Used to separate statements in multi-statement lines. For multi-statement lines, it may be replaced by a semicolon to suppress printing.

; Semicolon. Used inside brackets to end rows. Used after an expression or statement to suppress printing or separate statements.

% Percent. The percent symbol is used for comments; it indicates a logical end of line. Any following text is ignored.

! Exclamation point. Indicates that the rest of the input line should be issued as a command to the operating system.

See also:

Arithmetic, relational, and logical operators.

Purpose:

Creating vectors, matrix subscripting, and for iterations.

Description:

The colon is one of the most useful operators in MATLAB. It is used for creating vectors, in subscripts, and in for iterations.

Here are definitions that apply when the colon is used to create regularly spaced vectors:

```
j:k   is the same as [j, j+1, ..., k]
j:k   is empty if j > k.
j:i:k is the same as [j, j+i, j+2i, ..., k]
j:i:k is empty if i > 0 and j > k or if i < 0 and j < k.
```

Here are definitions that govern the use of the colon to pick out selected rows, columns, and elements of vectors and matrices:

```
A(:,j) is the j-th column of A
A(i,:) is the i-th row of A
A(:,:) is the same as A
A(j:k) is A(j),A(j+1),...,A(k)
A(:,j:k) is A(:,j),A(:,j+1),...,A(:,k) and so on.
A(:) is all the elements of A, regarded as a single column. On the
left side of an assignment statement, A(:) fills A, preserving its
shape from before.
```

If you are new to MATLAB, spend some time learning about the colon because once you master the colon, you've mastered MATLAB.

Examples:

Using the colon with integers:

```
D = 1:4
```

results in

```
D =
   1   2   3   4
```

Using two colons to create a vector with arbitrary real increments between the elements:

```
E = 0:.1:.5
```

results in

```
E =
  0.000  0.100  0.200  0.300  0.400  0.500
```

See also:

for, subscripts, logspace

Index