**Matlab Course**

[Manual Title page](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat_title.html)

[References](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/matlabrefs.htm)

[**Chapter 1:   Introduction to MATLAB**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat1.html)

|  |  |
| --- | --- |
| 1.1 | [Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat1.html#discuss) |
| 1.2 | [Links to other Web Sites](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat1.html#links) |
| 1.3 | [Use of help](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat1.html#help) |

[**Chapter 2:   Basic Procedures in Using MATLAB**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html)

|  |  |
| --- | --- |
| 2.1 | [Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor752627) |
| 2.2 | [Entering MATLAB and Using help and demo](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor752828) |
| 2.3 | [Defining Arrays and Arithmetic Operations with them](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor753435) |
| 2.4 | [Saving and Printing your Work](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor755297) |
| 2.5 | [Matrix Operations with Arrays](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor756143) |
| 2.6 | [Element by Element Operations on Arrays](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor756821) |
| 2.7 | [Plotting Results](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor757335) |
| 2.8 | [Find Roots of Equations: Using Polynomial Approximations](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor757772) |

[**Chapter 3:   Example Problems Solved with MATLAB**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html)

|  |  |
| --- | --- |
| 3.1 | [Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor815415) |
| 3.2 | [Controlling Your Screen Output with disp and format](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor815512) |
| 3.3 | [Solution of Simultaneous Equations](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor816798) |
| 3.4 | [Data Fitting](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor817409) |
| 3.5 | [Using Polynomials](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor817928) |
| 3.6 | [Trial and Error Solutions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor818840) |
| 3.7 | [Contour Plots and a Simple Function](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor820044) |
| 3.8 | [Easy Editing During a MATLAB Session](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor820692) |
| 3.9 | [Communicating with Other Systems and Languages](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor821141) |

[**Chapter 4:   Arithmetic Operations on the Elements in Arrays**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html)

|  |  |
| --- | --- |
| 4.1 | [Operations on Scalars](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html#anchor852722) |
| 4.2 | [Operations on Vectors of the Same Length](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html#anchor853518) |
| 4.3 | [Operations of a Vector with a Scalar](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html#anchor854123) |
| 4.4 | [Matrices Used with the Operators](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html#anchor854896) |

[**Chapter 5:   Logical Operators and Functions**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat5.html)

|  |  |
| --- | --- |
| 5.1 | [The Logical Operators: not, and, or, exclusive or](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat5.html#anchor971946) |
| 5.2 | [Associated Logical Functions: any, all, find](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat5.html#anchor972526) |
| 5.3 | [Looking for Special Numbers: NaN and Inf](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat5.html#anchor972894) |

[**Chapter 6:   Using MATLAB to Display Results**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html)

|  |  |
| --- | --- |
| 6.1 | [Two-Dimensional Plotting Functions in MATLAB](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#twodplot)  [6.1.1 Using fplot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#fplot)  [6.1.2 Annotating fplot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#annotate.fplot)  [6.1.3 Making a bar graph](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#bar)  [6.1.4 Making a pie chart](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#pie)  [6.1.5 Using plot and yxplot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#plot) |
| 6.2 | [Miscellaneous Graphics Commands](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6b.html) |
| 6.3 | [Three-Dimensional Plotting Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6c.html) |
| 6.4 | [Advanced Features of Plotting](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html)  [6.4.1 Using Plot Handles](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html#handles)  [6.4.2 Printing and Saving Graphics Files](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html#print)  [6.4.3 Images and Graphs](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html#images) |

[**Chapter 7:   More Useful MATLAB Arithmetic Functions**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html)

|  |  |
| --- | --- |
| 7.1 | [Find and Evaluate Character Strings: grep and eval](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1028533) |
| 7.2 | [The Exponential and Logarithm Functions: exp and log](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1029255) |
| 7.3 | [General Powers and Roots: x.^a, and, sqrt(x)](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1032101) |
| 7.4 | [Hyperbolic Functions & Their Inverses: cosh, sinh, tanh, etc.](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1033113) |
| 7.5 | [Trigonometric Functions & Their Inverses: cos, acos, sin](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1034129) |
| 7.6 | [Setting Nothing, Zeros and Random Values in Arrays](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1035004) |

[**Chapter 8:   Functions that Operate on Arrays**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html)

|  |  |
| --- | --- |
| 8.1 | [The sum and cumsum Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1080909) |
| 8.2 | [The prod and cumprod Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1082635) |
| 8.3 | [The max and min Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1084173) |
| 8.4 | [The sort Function](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1084917) |
| 8.5 | [The mean and median Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1086464) |
| 8.6 | [Moments and the Standard Deviation Function: std](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1087347) |
| 8.7 | [The diff Function and an Approximate Derivative](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1088143) |

[**Chapter 9:   The Main Group of Matrix Operations**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat9.html)

|  |  |
| --- | --- |
| 9.1 | [Matrix Multiply](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat9.html#anchor1057455) |
| 9.2 | [Determinants and Matrix Division](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat9.html#anchor1058758) |

[**Chapter 10:   MATLAB m-Files**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html)

|  |  |
| --- | --- |
| 10.1 | [Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor905182) |
| 10.2 | [Loops with for and while](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor905563) |
| 10.3 | [Tests with if Statements](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor908058) |
| 10.4 | [Global and Local Variables Used in Functions and m-files](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor910424) |
| 10.5 | [The Heading of a MATLAB Function](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor912374) |

[**Chapter 11:   Finding Errors and Testing Programs**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html)

|  |  |
| --- | --- |
| 11.1 | [Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html#anchor934791) |
| 11.2 | [Diagnostic Messages](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html#anchor934940) |
| 11.3 | [Using echo](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html#anchor935686) |
| 11.4 | [The keyboard Command](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html#anchor936537) |

[**Chapter 12:   New Features of Matlab**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html)

[**Chapter 13:   Structures, Cells and Other Arays**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html)

**Matlab List of Figures**

[Figure 2.1 Sin(t) and Cos(t)](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#figure1)

[Figure 2.2 Looking for a root](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor759228)

[Figure 3.1 Approximations to Experimental Data](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor818697)

[Figure 3.2 A Contour Plot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor820557)

[Figure 3.3 Steam Enthalpy Plot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor823392)

[Figure 6.1 Use of fplot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#anchor999824)

[Figure 6.2 Annotated plot (legend)](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#anchor1000591)

[Figure 6.3 Use of bar](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#anchor998958)

[Figure 6.4 Use of pie](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#anchor999368)

[Figure 6.5 Use of yxplot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#yxplot)

[Figure 6.6 Use of yxploti and ginput](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6b.html#yxploti)

[Figure 6.7 Use of meshgrid and mesh](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6c.html#meshgrid)

[Figure 6.8 Another View of a Three Dimension Object](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6c.html#anotherview3dobj)

[Figure 6.9 Use of surf](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6c.html#surf)

[Figure 6.10 Use of Handles](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html#handlespict)

[Figure 8.1 Numerical Approximation to a Derivative](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1089184)

**CHAPTER 1**

**Introduction to MATLAB**

* [1.1 Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat1.html#discuss)
* [1.2 Links to other Web Sites](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat1.html#links)
* [1.3 Use of help](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat1.html#help)

**1.1 Discussion**  
  
Before you can start using MATLAB on Owlnet, you must learn how to use the [UNIX](http://www.owlnet.rice.edu/%7Eceng303/otherinf/unix.html) operating system and the X-windows environment now used on all of the workstations on our network. These topics are discussed in some detail in the Notes on the Owlnet System prepared for you by Information System's at Rice. You will also need to find how to use the *aXe* (or another) editor before you get very far into the course. The *aXe* editor is very simple, but it requires some initial help in understanding the use of the multitude of menus used to control editing. Several other editors are available, but the *GNUEMACS* and *vi* are the ones that you will find most supported at Rice.  
  
You should be familiar with enough Unix to use a workstation reasonably well. MATLAB can be executed directly from a Unix prompt such as you might see when you login through a dumb terminal. Its graphics features can only be utilized on a workstation with multiple windows (such as X-windows) available for you to work in. Thus you are advised to read carefully topics on workstation use and the X window system in the Owlnet notes unless you already are experienced in the use of a Sun workstation.  
  
A description of most of the commands and techniques for programming will be found in the *PRO-MATLAB User's Guide*. A Copy of the *User's Guide* should be found in each Owlnet Laboratory. These notes cover only a small number of these commands so if you wish to expand your use of MATLAB you should obtain or have access to one of the User's Guides. The main purpose of these notes is to describe techniques found to be useful in a variety of problems in introductory engineering courses. We have also collected a large number of problems that we hope will challenge you to learn even more techniques.

1.3 Links to Other Sites

There are many web sites that now provide tutorials and information about the use of Matlab. Here are a couple that lead to many more.

* The vendor: [MathWorks Inc.](http://www.mathworks.com/) has sites that include FAQs, a Software Library and Technical Notes
* The [Stat/Math Center at Indiana University](http://www.indiana.edu/%7Estatmath/math/matlab/) has extensive notes about Matlab and a list of other web sites.

**1.3 Use of** *help*  
There may be commands available on the Owlnet version of MATLAB that have been added since the PRO-MATLAB User's Guide was last updated. Some of these may be described and used in these notes. Other commands may be found by listing the currently available function names with HELP while you are in a MATLAB session. This may be done by use of the *help* in the Matlab Command Window or by us of the *helpdesk* or *helpwin* options. The *helpdesk* command should bring up a "Web browser" window where you can examine a number of Matlab topics. In 2005, the version of Matlab supported on Owlnet was 6.5. The notes for this cousre were prepared using Matlab 5.2. The Unix version of the result given when you typed helpdesk for Matlab 5.2 looked like:



The *helpwin* command brings up the MATLAB Help Window that will enable you to get help on any Matlab function in your path. This window essentially does the same thing as *help*, thus you may find its use to parallel the following discussion. The command *help* by itself gives a list of HELP topics. If you type:

>> **help** *<-- asking for help*

you will then see a list of different primary help topics, which are mostly just directories of MATLAB functions. By typing *help topic name, you* get a listing of the specific functions in that topic. For example:

>> **help ceng301/matlab**

Directories in /home/ceng301/matlab

Directory Contains

ceng303 Functions originally developed for Masc223 and

now used in Ceng303

chemeq Chemical Equilibrium Functions

datbas Data Base programs

engmods Mass and Energy balance Modules

humid Humidity programs

massmods Mass balance modules.

misc Special programs used as examples in the notes

pichelps Programs that show pictures of the units

specmods Special modular functions

utilities Functions used by many of the modular programs

vle Vapor liquid equilibria functions

You should type:

>> **help dirname**

to get a listing of the files in the directory by the name: *dirname* You can then get a listing of the programs in each directory by following the  
suggestion:

>> **help pichelps**

Visual aid for matlab functions

picfraco.m - Goes to section in Chapter 5, cengnotes, that shows fraco.m

picmix.m - Goes to section in Chapter 5, cengnotes, that shows mix.m

picmixe.m - Goes to section in Chapter 5, cengnotes, that shows mixe.m

picphaseg.m - Goes to section in Chapter 5, cengnotes, that shows phaseg.m

picreact.m - Goes to section in Chapter 5, cengnotes, that shows react.m

picreacte.m - Goes to section in Chapter 5, cengnotes, that shows reactee.m

picreacten.m - Goes to section in Chapter 5, cengnotes, that shows reacten.m

picreactr.m - Goes to section in Chapter 5, cengnotes, that shows reactr.m

picsep.m - Goes to section in Chapter 5, cengnotes, that shows sep.m

picsepe.m - Goes to section in Chapter 5, cengnotes, that shows sepe.m

picsept.m - Goes to section in Chapter 5, cengnotes, that shows sept.m

picsplit.m - Goes to section in Chapter 5, cengnotes, that shows split.m

picsplite.m - Goes to section in Chapter 5, cengnotes, that shows splite.m

These are all specific functions which can be called in MATLAB. By typing:

>> **help function name**

you can get a detailed description of a function in MATLAB if it includes such a description in its heading. For example:

>> **help picsep**

function picsep

This displays a picture illustrating use of the command SEP.

Note that this will only run in Matlab versions that support the WEB

command (Matlab 5.0 and above). Otherwise, visit the following web site:

http://www.owlnet.rice.edu/~ceng301/5.html#5.1

Example: >> sep([0.2 0.3 0.8 0.5],1,[2 3])

This example separates stream 1 (which contains 4 compounds: a,b,c,d)

into exit streams 2 and 3.

Stream 2 is the first outlet stream because it is the first stream

specified in the vector: out (3rd argument of sep), and will get:

20% of cmpd a in Stream 1

30% of cmpd b in Stream 1

80% of cmpd c in Stream 1

50% of cmpd d in Stream 1

Stream 3 is the other outlet stream and would get the remaining

amounts of the cmpds.

NOTE: the sum of the t vector does NOT necessarily equal one!

Comments/Picture: TYLC, ACP

Comments/Picture/Program: ADD

Last updated: August, 1997

Several problems assigned in this course will ask you to investigate functions that may be found in several directories. You should use help on the functions first to see what instructions are given about the functions and then you should list the function if you need more information.  
  
The functions that have been developed for Ceng 301 have comments in them that should give you a lot of information about how to use them. If these comments are not clear please let the instructor know so that they can be further explained. When you type *help*, followed by a command name, you get a listing of comments at the beginning of the function.  
  
In addition to giving you a way to find out about functions that are not described in these notes or in the User's Guide, the *help*, command will frequently show you new ways to use functions. Both the developers of MATLAB and the authors of functions for your courses frequently expand the scope of functions to make them more useful. The comments in the functions are nearly always updated to reflect these changes. It may take a semester for these notes to be updated and it may take over a year for the User's Guide. Even then some of the new features may escape the updating of the texts.

[Continue on to Chapter 2](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 2**

**Basic Procedures in Using MATLAB**

**2.1 Discussion**  
  
MATLAB is a powerful set of programs that make a lot of operations that would be tedious in FORTRAN very straight forward. The programs run in an interpretive mode (like Basic and APL,) so much of the effort required to determine exactly what size to make arrays in FORTRAN is avoided. In addition, debugging is very simple since you can test programs and instructions line by line to see exactly what they do. MATLAB plots function in a very direct way. We will use this to look at most of our results from calculations in MATLAB. Another feature of using the language on a Unix workstation is the fact that you can open as many windows as needed to:

1) issue MATLAB commands in one,  
2) see the results of graphics in another,  
3) create and change MATLAB programs in others.

The User's Guide for MATLAB is quite well written and contains both a Tutorial and a Reference section. Both are quite brief so that until you learn the basic procedures and the names of a lot of functions, you will find that you have to spend considerable time looking for ways to use the language. These notes were written to help students get the background to understand how to do many of the basic operations, but you will find many other features in the Study Guide and a description of most of the functions available to help you  
  
This chapter will demonstrate the following MATLAB procedures.

[2.1 Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor752627)   
[2.2 Entering MATLAB and Using **help** and **demo**](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor752828)   
[2.3 Defining Arrays and Arithmetic Operations with them](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor753435)   
[2.4 Saving and Printing your Work](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor755297)   
[2.5 Matrix Operations with Arrays](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor756143)   
[2.6 Element by Element Operations on Arrays](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor756821)   
[2.7 Plotting Results](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor757335)   
[2.8 Find Roots of Equations: Using Polynomial Approximations](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat2.html#anchor757772)

**2.2 Entering MATLAB and Using help (again) and *demo***

To get into MATLAB you simply open the main menu in the gray part of the screen with the right mouse button. Darken the MATLAB bar in the menu by pointing to it with the right button held down and let go of the button. If you want your MATLAB session to access a directory of yours other than your home directory, you can change directories while in MATLAB by using the *cd*, command. (See *help cd*). Files can then be saved to this directory and be loaded or referenced from it . Entering MATLAB is much simpler to do than to describe and should become automatic after a few sessions. Immediately after opening MATLAB, a MATLAB Command Window, should appear for you to work in.

You will get a message that includes information like:

To get started, type one of these commands: helpwin, helpdesk, or demo.

For information on all of the MathWorks products, type tour.

The *demo* command shows how to use a variety of Matlab tools with emphasis on array operations and graphics.  
  
There is one inconsistency in the way a number of things are presented in MATLAB. Most messages about MATLAB functions (such as those that you get from HELP) capitalize function names, but MATLAB is case sensitive in its default state. The computer will only respond to function names that are typed in lower case. Thus to get HELP from the helpdesk or to run the DEMO packages, you must type these as *helpdesk*, and *demo*.

>> **demo**

will allow you to choose maong many MATLAB 5.1 demonstration programs which construct many interesting and detailed graphs. They also give a good description of how MATLAB works. You are urged to try several of these to get some feeling for what can be done with MATLAB. In particular note how very simple programs can produce rather sophisticated results.  
  
To get out of MATLAB, type:

>> **quit**

and all MATLAB windows should disappear.  
  
  
  
**2.3 Defining Arrays and Arithmetic Operations with them**  
  
  
MATLAB allows you to define three types of arrays:

1) scalars with a single element,  
2) vectors containing ordered lists of elements,  
3) matrices with ordered elements in a rectangular arrangement.

The elements stored in the arrays may be numeric or character types. No distinction is made between integer and real numbers. Numeric and character scalars may be defined using commands similar to FORTRAN. The following session will show that MATLAB automatically echoes the values you set unless you end your variable definition with a semicolon.

>> **s=1.2**

s =

1.2000  *<-- s is a numberic vector with one element in it.*

>> **s=1.2;**

>> **s**

s =

1.2000

>> **a='qwerty'**

a =

qwerty  *<-- a is a character string with the characters: qwerty in it*

Since the form used to show the value is rather verbose, you will usually want to end array definitions with a semicolon. If you forget this and start listing a very long array, simply press CONTROL C to terminate the listing. You will also find that MATLAB inserts blank lines to separate commands unless you instruct it to eliminate them with the command:

>> **format compact**

As in the second example defining s, you can always see what a variable's value is by typing its name.  
  
Let's define a few vectors:

>> **v=[1 2 3]**

v =

1 2 3  *<-- v is a numeric vector with three elements*

>> **vc=['asdf' '123']**

vc =

asdf123  *<-- vc is a character string with the characters: asdf123 in it*

The square brackets must be used in defining numeric vectors and may be useful in constructing character arrays. A character vector (or string) consists of any linear arrangement of characters, so adding more characters takes place by direct concatenation.  
  
Now finally, some matrices:

>> **m=[1 2 3**

**4 5 6]**

m =

1 2 3

4 5 6

>> **mc=['asdf'**

**'123 ']**

mc =

asdf

123

Suppose you want to see the names of the variables that you have created:

>> **who**

Your variables are:

a m n v

demos mc s vc

Some of these names are familiar: *a, m, mc, v,* and *vc*, were created in the sessions shown here. The variables *demos*, and *n*, must have been created when the *demo* command was tried.  
  
We will show some of the more important properties of arithmetic operations with arrays, in the following session: First, a short way to get a vector of numbers of the form:

a, a + 1, a + 2, . . . b - a, b

then some operations on that vector to show how vector functions make life easy in MATLAB. Set *t* as a vector with elements at a regular spacing. Choosing:

a=0, interval=0.5, b=2

>>  **t=0:.5:2**

t =

0 0.5000 1.0000 1.5000 2.0000

>>  **2\*t**

ans =

0 1 2 3 4

>>  **f=sin(t)**

f =

0 0.4794 0.8415 0.9975 0.9093

**2.4 Saving and Printing your Work**   
  
  
If you run out of time to finish a problem in MATLAB, you should save the workspace by typing:

>> **save**

which produces a file called matlab.mat., This may then be retrieved by:

>> **load**

when you can continue work on the problem. If you have several problems that you want separate workspaces for, simply give them different names as in:

>> **save prob1**

which can be retrieved by:

>> **load prob1**

If you want to start over with a fresh workspace, type:

>> **clear**

If you want to get rid of only a few variables in your active workspace, give the names of the variables to be deleted after the clear, command. You can also save only a few of the variables in a named workspace by listing the names of those variables after the workspace name in the *save* command.  
  
Note that when you use the save, command, you create a binary file that can not be listed or edited or otherwise used in another environment. If you have a long numeric array that you want to save so that it can be used in another environment, save that variable with the command:

>> **save name.dat name -ascii**

This will create a file called name.dat, with the data that was in the MATLAB variable name, stored so that it can be listed, edited or used elsewhere as any other ASCII file. It can also be loaded back into MATLAB to create the numeric array with the same data in it or with new data if the file has been edited.  
  
Here is an example where the variable xm, is saved, then the resulting file is edited to add more data. In MATLAB the array listed as:

xm =

12 15 17 19

2 -3 -5 -15

100 10000 1000000 100000000

It was saved by:

>> **save xm.dat xm -ascii**

to produce the file xm.dat, that lists as:

wsname% **cat xm.dat**

1.2000000e+01 1.5000000e+01 1.7000000e+01 1.9000000e+01

2.0000000e+00 -3.0000000e+00 -5.0000000e+00 -1.5000000e+01

1.0000000e+02 1.0000000e+04 1.0000000e+06 1.0000000e+08

Suppose we edit it to add another row:

wsname% **cat xm.dat**

1.2000000e+01 1.5000000e+01 1.7000000e+01 1.9000000e+01

2.0000000e+00 -3.0000000e+00 -5.0000000e+00 -1.5000000e+01

1.0000000e+02 1.0000000e+04 1.0000000e+06 1.0000000e+08

400 10.5 -18 0

Then in MATLAB we can get this new version of the variable by:

>> **clear**

>> **load xm.dat**

>> **format short e**

>> **xm**

xm =

1.2000e+01 1.5000e+01 1.7000e+01 1.9000e+01

2.0000e+00 -3.0000e+00 -5.0000e+00 -1.5000e+01

1.0000e+02 1.0000e+04 1.0000e+06 1.0000e+08

4.0000e+02 1.0500e+01 -1.8000e+01 0

If you want to save a copy of your session in a file for someone to study or to print, you can do so by starting the session with the diary, command. This is shown in the following example session:

>> **diary mat1.t**

>> **t=0:.1:6;**

>> **plot(t,sin(t)^2)**

??? Error using ==> ^

Matrix must be square.

>> **plot(t,sin(t).^2)**

>> **quit**

183 flops

Note the error in the squaring operation. The omitted period is one of the most common errors in MATLAB. We wanted an element by element operation on an array and must specify that.  
  
The file mat1.t, will then list exactly like the session just shown with only the line involving diary missing. The graphics window will show your curve with axes marked at integer or simple fractions. Try the commands shown in the example session to see how simple the use of diary, and plot, is. If you save several sessions in the same file, new ones are appended. Of course, the graphics output is not saved. The recording in the file may be turned on and off with the commands:

>> **diary off**

*any commands that you do not want saved*

>> **diary on**

The diary command rarely produces a file that is suitable for submission as a solution to assignments. In nearly all cases such files include rough output with numerous errors and statements out of order so that a grader can not follow what was done in completing the assignment. You should always plan to edit such files to make it clear by adding comments, reordering the output and deleting errors so that the results are easier to follow. In particular, you should show in the file the problem that is being worked and clearly designate the final answers to the problem. You may find it easier to make assignment solutions by copying the results in a Matlab or Maple session to a file that you are editing as you complete the assignment. When assignments are ready for submission, they should be stored in a file in your chbe301/chbe303 directory and clearly labeled with a name that tells what is in the file. For example, assignment 1 solution might be called solution1. When the file is ready to be graded, your TA should be sent an e-mail message telling her/him that the file is now ready for grading.  
  
  
  
  
**2.5 Matrix Operations with Arrays**   
  
  
There are two kinds of operations with arrays called: Matrix operations and Array operations. The first set includes taking the transpose of an array, ``matrix" multiply, adding and subtracting arrays of the same shape, etc. The transpose of a matrix is found with the single quote symbol . Matrix multiply is done with a \* and addition and subtraction use their normal symbols as seen in the following:

>> **m'**

ans =

1 4

2 5

3 6

>> **m\*v'**

ans =

14

32

>> **m1=[6 5 4**

**7 4 1];**

>> **m+m1**

ans =

7 7 7

11 9 7

Finally matrix division is used to solve sets of linear equations. To solve the set of equations:

5x1 + 3x2 - x3 = 5

2x1 - 7x2 - 3x3 = 0

x1 + 5x2 + 6x3 = -7

we need to define a 3 by 3 matrix with the coefficients of the unknowns in it and divide a column vector with the right hand side coefficients by this matrix. Unfortunately you have to get used to the fact that there are two divide symbols in MATLAB.

**A\B** gives the solution to: **A\*X = B  
A/B** gives the solution to: **X\*A = B**

In our case we need to use the first form so:

>> **a=[5 3 -1**

**2 -7 -3**

**1 5 6];**

>> **a\[5 0 -7]'** *<-- Note the quote or prime*

*to transpose the vector.*

ans =

0.1168

0.8426

-1.8883

We can confirm our solution by using matrix multiply.

>> **a\*ans**

ans =

5.0000

0.0000

-7.0000

If we forget how many elements there are in a vector, the command *length*, will tell us. The *size*, command will tell how many rows and columns there are in a defined matrix. These are shown in the following session:

>> **v=[1 3 5];**

>> **size(v)**

ans =

1 3

>> **length(v)**

ans =

3

**2.6 Element by Element Operations on Arrays**  
  
  
Array operations in MATLAB require some care. Essentially there are two types of operations involving arrays.

1) Element by element operations  
2) Vector-Matrix operations.

Confusing these can lead to real difficulties. The next session shows the formation and addition of two vectors:

>> **a=[1 2 3];**

>> **b=[2 5 8];**

>> **a+b**

ans =

3 7 11

Now suppose we try to multiply them:

>> **a\*b**

??? Error using ==> \*

Inner matrix dimensions must agree.

What happened? The multiplication symbol by itself when applied to vectors or matrices is interpreted to mean matrix multiply. We could have done this by transposing the second vector:

>> **a\*b'**

ans =

36

This gave (1\*2 + 2\*5 + 3\*8) = 36, the usual scalar product of two vectors.  
If we want the element by element product, we must put a period before the multiplication symbol:

>> **a.\*b**

ans =

2 10 24

The same procedure must be followed in doing division and exponentiation. For example, if you want the square of each element in a vector you must use:

>> **a.^2**

ans =

1 4 9

Forgetting the period will lead to:

>> **a^2**

??? Error using ==> ^

Matrix must be square.

The fact that MATLAB gives vector and array results with most of its built-in functions is one of its main features. The fact that sin(t), with t, a vector gives the value of *sin* of each element makes it trivial to look at a plot of the function.  
  
  
  
**2.7 Plotting Results**  
  
  
The following set of commands produces a plot of *sin(t)* and *cos(t)* for *t* from 0 to 4pi.

>> **t=0:pi/16:4\*pi;**

>> **plot(t,[sin(t);cos(t)])**

>> **title('sin(t) and cos(t)')**

>> **xlabel('t')**

>> **ylabel('sin and cos')**

**Figure 2.1 Sin(t) and Cos(t)**

|  |
| --- |
|  |

The five lines in this sequence do the following:

1) Sets up a vector t with the elements: 0, pi/16, pi/8, .... 4pi.  
2) Creates a matrix with its first row: sin(0),sin(/16),sin(/8)...sin(4). and its second row: cos(0),cos(/16),cos(/8)...cos(4). and plots the two curves.  
3) Puts a title at the top of the plot.  
4) Labels the x axis.  
5) Labels the y axis.

If you want to save your figure to use in a later Matlab session or for someone else to see in such a session use the **Save As** command in the file menu on the plot figure. This will save the figure with the name you specify and an appendage of .fig. You may then retreive the file with the open command in another Matlab session.

Two three dimensional plotting routines are available. Both are demonstrated in the EXPO package. The *mesh*, command constructs a three dimensional plot of the values in a matrix vs the indices that specify positions in the matrix. The *contour*, command gives a contour plot of a matrix interpreted in the same way.  
  
You can create up to four subplots in the same window. The first two digits in the argument of subplot, specify how many plots and their orientation as:

|  |  |  |
| --- | --- | --- |
| First Two Digits | No. of Plots | Orientation |
| 12 | Two | side by side |
| 21 | Two | above and below |
| 22 | Four | in the Quadrants |

The last digit then specifies the particular plot.  
  
  
  
**2.8 Find Roots of Equations: Using Polynomial Approximations**  
  
  
We will demonstrate several MATLAB functions in this section. You may find them useful in solving a variety of problems, but our demonstration will show simply how to find a root of one equation. The typical problem is: find x such that f(x)=0. Our demonstration will be concerned with finding a root of sin(x)=0. We know there are roots at 0 and all multiples of . In more complicated problems we would not know that however and might have to spend some time determining just what intervals to search for a root. Let's start by looking at a plot of our function:

>> **t=0:.1:6;**

>> **plot(t,sin(t))**

>> **grid**

Try that and you can see that there is a root between 3 and 4:  
  
  
**Figure 2.2 Looking for a root**

|  |
| --- |
|  |

We could redefine t to span the interval (3,4) and plot that result to home in on the root, but will investigate using a polynomial fit of the values for our function:

>> **t=3:.25:4;**

>> **c=polyfit(t,sin(t),3)**

c =

0.1537 -1.4420 3.5107 -1.5619

>> **roots(c)**

ans =

5.6727

3.1415

0.5704

The *polyfit*, function finds a polynomial that fits the data given it in the least squares sense. In our case, it found that:

p(t) = 0.1537t^3 - 1.4420t^2 + 3.5107t - 1.5619

closely approximates sin(t) over the interval (3,4). The program *roots* finds all roots of a polynomial. We can recognize that the procedure found an answer very close to the one that we know is correct: .

[Continue on to Chapter 3](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 3**

**Example Problems Solved with MATLAB**

[3.1 Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor815415)  
[3.2 Controlling Your Screen Output with disp and format](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor815512)  
[3.3 Solution of Simultaneous Equations](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor816798)  
[3.4 Data Fitting](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor817409)  
[3.5 Using Polynomials](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor817928)  
[3.6 Trial and Error Solutions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor818840)  
[3.7 Contour Plots and a Simple Function](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor820044)  
[3.8 Easy Editing During a MATLAB Session](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor820692)  
[3.9 Communicating with Other Systems and Languages]](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat3.html#anchor821141)

**3.1 Discussion**  
  
The following problems illustrate many techniques helpful in solving scientific problems. In order to make our listing of variables less cumbersome with the *disp* , (for display) function, its description here is a preliminary digression.  
  
  
**3.2 Controlling Your Screen Output with** *disp* **and** *format*  
  
It is impossible to make the printed output from a MATLAB session look good with just the default way of printing results. This is particularly difficult in looking at results stored in matrices. If you give the name of an array, you always get that name followed by spaces and an equal sign. Even if you just give something like a message in quotes you get *ans* , as the assumed name with the spaces and equal sign. One way to avoid this is with the *disp* , function. You can avoid messy extraneous printing in your error messages with it as in the *ssec2* , function. You can also print a table of data that looks like a table as seen in the function *dsply*.



Note that images of Matlab functions were created for these notes using a Macintosh. Color is used to designate important features of the functions and the lines are number so we can reference individual lines if needed. The colors used are:

* Blue is used to designate key Matlab words such as **function**, **if**, **return**, and **end**.
* Red is used to show all comments
* Green is used to show strings
* Black is used for all else

If we follow the example to set up the values in *distm:* ,

>> **distm=[0 0 2.3 .66**

**1 2 2.7 .77**

**3 6.5 2.2 .60];**

we could look at it by simply:

>> **distm**

distm =

0 0 2.3000 0.6600

1.0000 2.0000 2.7000 0.7700

3.0000 6.5000 2.2000 0.6000

but that is not very informative about its contents. The function *disp* , allows us to avoid repeating the name of the variable:

>> **disp(distm)**

0 0 2.3000 0.6600

1.0000 2.0000 2.7000 0.7700

3.0000 6.5000 2.2000 0.6000

The *disp* , function is thus used in *dsply* , to allow the user to add a heading to the matrix listing:

>> **dsply(distm,' time distance accel force')**

time distance accel force

0 0 2.3000 0.6600

1.0000 2.0000 2.7000 0.7700

3.0000 6.5000 2.2000 0.6000

We are stuck with having to adopt MATLAB's standard output format for data, but we have made some progress. We also get an additional blank space before the next MATLAB prompt: '>>'. Most of these blank space can be eliminatedby setting:

>> **format compact**

Then that use of the function dsply , would give:

>> **dsply(distm,' time distance accel force')**

time distance accel force

0 0 2.3000 0.6600

1.0000 2.0000 2.7000 0.7700

3.0000 6.5000 2.2000 0.6000

We will set our environment with *format compact* in the rest of these notes. The *format* command has several uses. Here is what *help* shows:

>> **help format**

FORMAT Set output format.

All computations in MATLAB are done in double precision.

FORMAT may be used to switch between different output

display formats as follows:

FORMAT Default. Same as SHORT.

FORMAT SHORT Scaled fixed point format with 5 digits.

FORMAT LONG Scaled fixed point format with 15 digits.

FORMAT SHORT E Floating point format with 5 digits.

FORMAT LONG E Floating point format with 15 digits.

FORMAT SHORT G Best of fixed or floating point format with 5 digits.

FORMAT LONG G Best of fixed or floating point format with 15 digits.

FORMAT HEX Hexadecimal format.

FORMAT + The symbols +, - and blank are printed

for positive, negative and zero elements.

Imaginary parts are ignored.

FORMAT BANK Fixed format for dollars and cents.

FORMAT RAT Approximation by ratio of small integers.

Spacing:

FORMAT COMPACT Suppress extra line-feeds.

FORMAT LOOSE Puts the extra line-feeds back in.

Suppose you want to see the elements in the vector formed by:

>> **v=exp(-10\*(1:5))**

v =

1.0e-04 \*

0.4540 0.0000 0.0000 0.0000 0.0000

Obviously, that does not give very much information in the default SHORT format mode. Now let's try two other options:

>> **format long**

>> **disp(v)**

1.0e-04 \*

Columns 1 through 3

0.45399929762485 0.00002061153622 0.00000000093576

Columns 4 through 5

0.00000000000004 0.00000000000000

>> **format short e**

>> **disp(v)**

4.5400e-05 2.0612e-09 9.3576e-14 4.2484e-18 1.9287e-22

The LONG mode gives more decimal places but again does not cover the entire range in our vector. The SHORT E , or LONG E , option is required to see what has really been stored in *v.*  
  
We will impose format compact in the rest of the notes and occasionally use other formats to show what has been produced.  
  
  
  
**3.3 Solution of Simultaneous Equations**   
  
Scientific problems always seem to involve the solution of simultaneous linear equations. For a few equations (two or three at most) this may be done by hand. For more than three equations it is advisable to use a computer. We will look at the MATLAB solution of such equations by way of the example: Solve for x1, x2, and x3 satisfying:

3.5x1 + 2x2 = 5

-1.5x1 + 2.8x2 + 1.9x3 = -1

- 2.5x2 , 3x3 = 2

We have already seen how to do this, but it is worthwhile repeating to make sure we have not forgotten the details. In MATLAB the solution would be found as seen next:

>> **a=[3.5 2 0**

**-1.5 2.8 1.9**

**0 -2.5 3];**

>> **b=[5 -1 2];**

>> **a\b'**

ans =

1.4421

-0.0236

0.6470

The answer may be interpreted as: x1 = 1.4421 , x2 =-0.0236 , x3 = 0.6470 . Now see that this looks even better with *disp:*

>> **disp(a\b')**

1.4421

-0.0236

0.6470

**3.4 Data Fitting**  
  
A second common operation in scientific work is the fitting of experimental data by means of an approximating polynomial. The following set of data gives the values for yi measured at values of xi.

i xi yi

1 0 1.9

2 2 2.9

3 4 4.7

4 5 5.1

5 9 8.0

We wish to find various polynomials that might be used for smoothing this data or interpolating between points in the table. First we will try a linear approximation. This means we want to find c0 and c1 such that

y ~= c0 + c1x

at each of the data points. Since there are five points and only two unknowns this is very unlikely. MATLAB has a function called *polyfit* , that finds the coefficients that ``best" fit the data with a polynomial in the least squares sense. The function requires three arguments: x, y and n : x and y are vectors of data and n is the order of the polynomial. We see it used in the session:

>> **x=[0 2 4 5 9];**

>> **y=[1.1 2.9 4.7 5.1 8];**

>> **cl=polyfit(x,y,1);**

>> **disp(cl)**

0.7587 1.3252

Coefficients returned by the function are always listed in the order of decreasing powers of x . Thus the ``best" linear approximation is:

y ~= 0.7587x + 1.3252

The procedure for finding the ``best" quadratic fit is shown next:

>> **cq=polyfit(x,y,2);**

>> **disp(cq)**

-0.0176 0.9199 1.1232

The best quadratic fit was thus found to be:

y ~= - 0.0176x2 + 0.9199x + 1.1232

**3.5 Using Polynomials**  
  
We will now look at methods that allow polynomial coefficients like cl and cq to be used in MATLAB for interpolation, smoothing and comparison with experimental data.  
  
The function *polyval* , is the main tool for finding the value of:  
  
for either a single value of x or at each x in a vector or matrix. If we continue the session from Example 2, we see that the linear approximation can be compared to the original data by:

>> **yla=polyval(cl,x);**

>> **disp(yla)**

1.3252 2.8426 4.3600 5.1187 8.1535

and the quadratic approximation by:

>> **yqa=polyval(cq,x);**

>> **disp(yqa)**

1.1232 2.8927 4.5217 5.2834 7.9790

A table to compare the linear approximation is found by:

>> **ela=y-yla;**

>> **mla=[x;y;yla;ela]';**

>> **dsply(mla,' x y yla ela')**

x y yla ela

0 1.1000 1.3252 -0.2252

2.0000 2.9000 2.8426 0.0574

4.0000 4.7000 4.3600 0.3400

5.0000 5.1000 5.1187 -0.0187

9.0000 8.0000 8.1535 -0.1535

Similarly the quadratic approximation looks like:

>> **eqa=y-yqa;**

>> **mqa=[x;y;yqa;eqa]';**

>> **dsply(mqa,' x y yqa eqa')**

x y yqa eqa

0 1.1000 1.1232 -0.0232

2.0000 2.9000 2.8927 0.0073

4.0000 4.7000 4.5217 0.1783

5.0000 5.1000 5.2834 -0.1834

9.0000 8.0000 7.9790 0.0210

The average absolute error in the linear fit may be found by:

>> **disp(sum(abs(ela))/5)**

0.1590

This may be compared to the quadratic fit by:

>> **disp(sum(abs(eqa))/5)**

0.0827

There is also a MATLAB function called *polyder* , that allows the derivative of a polynomial to be found. Here is what it gives when we apply it to the two sets of polynomial coefficients we found:

>> **polyder(cl)**

ans =

0.7587

>> **polyder(cq)**

ans =

-0.0351 0.9199

These show that the derivative of 0.7587x + 1.3252 is 0.7587 and the derivative of:

- 0.0176x^2 + 0.9199x + 1.1232 .

is -0.0351x + .9199 . In these cases we could see by inspection what the derivatives of our polynomials should be, but for more complicated cases, this function can be very useful.  
  
MATLAB's graphics allow us to see a comparison of the polynomial approximations with the original data in our example:

>> **xs=0:.1:10;**

>> **yl=polyval(cl,xs);**

>> **yq=polyval(cq,xs);**

>> **plot(xs,yl,xs,yq,'r')**

>> **hold**

Current plot held

>> **plot(x,y,'g\*')**

>> **title('Linear and Quadratic Approximations')**

>> **xlabel('x')**

>> **ylabel('y')**

>> **text(4,3,'Experimental Data Green \*')**

>> **gtext('<- Linear Approximation')**

>> **gtext('Quadratic Approximation ->')**

>> **print**

Here is the plot:  
**Figure 3.1 Approximations to Experimental Data**

|  |
| --- |
| http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/images/lqa.gif |

**3.6 Trial and Error Solutions**  
  
The solution of nonlinear problems nearly always requires the use of a trial and error procedure to find a close approximation to a root of an equation. Roots of polynomials may be found with the MATLAB function *roots* , demonstrated in the chapter on programs. Other functions could be approximated by polynomials and then roots found as shown in that same chapter.  
  
A general purpose program that will find a root of a single equation by one version of the secant method is called *ssec2*. We will demonstrate it by finding the roots of the  
cubic equation:

x3 - 2x2 - 4.25x + 7.5 = p(x) = 0

Two functions are actually used: *ssec1* , and *ssec2.* , These functions are described more fully in the chapter on MATLAB functions. The second of these programs uses the first one. The *ssec1* , program takes two ``guessed" values for x and finds the value of f(x) at each. Then it predicts a new value that would give 0 for a result if the function were linear. The *ssec2* , program starts with two values for x and uses *ssec1* , up to nmax , times to try to make the difference between two successive guesses less in magnitude than erc. Each time after *ssec1* , is used, the value it predicts is used to replace the ``oldest" guess.  
  
We will concentrate here on the use of *ssec2*, , since that is the one that will actually find a root of an equation. The comments in the function may be seen by:

>> **help ssec2**

function x=ssec2(xs,fx,c,erc,nmax)

xs gives an interval to seek a solution to f(x) = 0 in.

fx is a character vector that tells how to find f(x).

c is a vector of parameters in f(x)

erc gives the difference between successive values of x

allowed when convergence is deemed to be achieved.

nmax is the maximum number of trials, before quitting.

If erc and nmax are not given, then the program uses:

erc=.0001 and nmax=10

If only two arguments are given, c is set to null.

The program uses the function ssec1 for each iteration.

Example 1:>> cs=[1 -2 -4.25 7.5];

>> disp(ssec2([2 3],'polyval(c,x)',cs,.0001,10))

Example 2:>> ssec2([3 4],'sin(x)')

The first example shown in the comments for the function finds one root of the cubic in this discussion. We can see that it finds the root at x = 2.5 by trying it:

>> **cs=[1 -2 -4.25 7.5];**

>> **disp(ssec2([2 3],'polyval(c,x)',cs))**

2.5000

Other roots may be found as seen next:

>> **disp(ssec2([0 2],'polyval(c,x)',cs))**

1.5000

>> **disp(ssec2([-3 -1],'polyval(c,x)',cs))**

-2.0000

Of course all these could also have been found with roots, by simply:

>> **disp(roots(cs))**

2.5000

1.5000

-2.0000

The program *ssec2* , does not always give a root in the specified interval and sometimes fails to converge if the initial interval is too large as shown in the next session.

>> **disp(ssec2([-5 0],'polyval(c,x)',cs))**

1.5000

>> **disp(ssec2([2 5],'polyval(c,x)',cs))**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ssec2 did not converge, xl: 2.5155 xr: 2.4987 \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2.4987

**3.7 Contour Plots and a Simple Function**   
  
  
Contour plots frequently give the user a picture of a function's behavior particularly in the vicinity of stationary points. The function:

f(x,y) = 2 - [(x-1)2 + 4(y-1)2 + 2xy]

has a stationary maximum near (0,1). Here is a MATLAB function that will give a matrix of values of the function for vector of x and y :



Note particularly that as vectors are added to the matrix, the last value of y is used first and then we work toward the first value of y .  
  
The following session gave a contour plot of this function for both x and y in the interval (-2,2):

>> **xs=-2:.1:2;**

>> **ys=xs;**

>> **m=prof(xs,ys);**

>> **cs=contour(xs,ys,flipud(m),[-15 -10 -5 -2 0 .5]);**

>> **clabel(cs)**

>> **xlabel('x')**

>> **ylabel('y')**

>> **title('Contour Plot of Dyson Function')**

The function *flipud*, which flips a matrix upside down, is used with the *contour* command in order to make the graph appear right-side-up. (This is because in MATLAB 4, the appearance of the graph has been flipped in the up/down direction.)  
  
  
**Figure 3.2 A Contour Plot**

|  |
| --- |
| http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/images/dyson.gif |

**3.8 Easy Editing During a MATLAB Session**   
  
If you have been trying some of the example problems in these notes, you have probably discovered that it is difficult to type an entire line without error. When you start generating your own commands you will find it even more difficult to type in exactly what you want the first time you try something.Fortunately, MATLAB provides a means for trying out corrections on a line-by-line basis. On the right side of the SUN keyboard, in the lower part of the R-functions key pad, there are four keys with black arrows --- called cursor keys. The up-pointing arrow moves the `line to be selected' upward toward earlier commands in the MATLAB session; the down arrow moves it downward. Each time you press either of these arrows, a new line appears at the bottom of your session ready to be executed with a RETURN . To edit this line, press the left arrow to move the cursor letter-by-letter toward the start of the line until the point is reached where you want to erase, change or insert one or more characters. Alternating left and right arrows allows you to edit any part of the line. When you are done press RETURN.  
  
  
  
  
**3.9 Communicating with Other Systems and Languages**   
  
In this section we will examine ways to use MATLAB with other languages such as FORTRAN and C. MATLAB has several shortcomings such as:

a) its lack of speed in looping operations,  
b) its somewhat limited character handling ability,  
c) its limited ability to read data from files,  
d) its limited control over windows and menus.

Both FORTRAN and C can now be used to create functions to be called in MATLAB that can give tremendous improvements in speed as well as the other limitations listed. Section I.7.4 in the MATLAB User's Guide shows how one can develop a FORTRAN version of a function called *yprime* that can speed up the solution of a model of an orbiting object.  
  
An easier way to use FORTRAN programs in MATLAB is by simply using a FORTRAN program to create a data file that can be read into MATLAB. The program *tprop2* is stored in *~chbe301/prog4*, so chemical engineering students can execute it by simply typing its name. It can be used to created a file of thermodynamic data that can then be loaded into a MATLAB session and plotted. For an illustration of its use, see the [Chbe 301 Notes:](http://www.owlnet.rice.edu:80/%7Echbe301/2c.html#2.7.2)

Data stored in the file created with *tprop2* may be displayed in MATLAB with the program *tplot1*:

>> **load water.dat**

>> **help tplot1**

function tplot1(m,com,n0,n1,n2,fx,fy)

Turns the plot hold OFF.

Plots column n2 vs column n1 in m with lines of constant

column n0 connected by solid lines and marked with

different symbols.

Adds title and axis labels.

Argument List

Arg Specifies

m matrix of data from thermo properties FORTRAN program.

com name of the compound in single quotes.

n0 column with data to sort on: 1 is p, 2 is t

n1 column of data for x axis

n2 column of data for y axis

fx fraction to shift text in x direction.

fy fraction to shift text in y direction.

If these are not given, 0.1 is used for each.

Values of nk

k Specifies

1 Pressure kPa

11 log(Pressure)

2 Temperature C

3 Volume m3/kg

4 Entropy kJ/kg-K

5 Enthalpy kJ/kg

6 Int. Eng. kJ/kg

7 Gibbs Eng. kJ/kg

We can plot and then print the data by:

>> **tplot1(water,'Water',1,2,5, 0.3, .07)**

>> **print**

Here is what we get:  
  
**Figure 3.3 Steam Enthalpy Plot**

|  |
| --- |
| http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/images/steamh.gif |

It can be seen from this figure that water boils at about 99°C at a pressure of 100 kPa and at about 212°C when the pressure is 2000kPa. These values agree with those in steam tables.

[Continue on to Chapter 4](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 4**

**Arithmetic Operations on Elements in Arrays**

[4.1 Operations on Scalars](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html#anchor852722)  
[4.2 Operations on Vectors of the Same Length](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html#anchor853518)  
[4.3 Operations of a Vector with a Scalar](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html#anchor854123)  
[4.4 Matrices Used with the Operators](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat4.html#anchor854896)

Many of the basic operators on arrays were demonstrated or at least used in the introductory material on MATLAB. We will try to include a more complete list of those operators that are primarily arithmetic here and to emphasize the way each treats different types of arrays. First we will look at the operators associated with one or two symbols as they perform on scalars. This Includes:

|  |  |
| --- | --- |
| **Symbol(s)** | Performs the Operation on Scalars |
| **+** | Addition |
| **-** | Subtraction |
| **\*** | Multiplication |
| **/** | First Divide: Left over right |
| **\** | Second Divide: Right over left |
| **^** | Raise to a Power |
| **,** | Catenates to form vectors when used with [ and ] |
| **;** | Catenates to form matrices when used with [ and ] |
| **:** | Used in setting the range for a vector |

**4.1 Operations on Scalars**   
  
  
First let's see how these perform with scalars. We will use the *disp* function to display our results. Here are some simple arithmetic operations:

>> **disp(3+5)**

8  *<-- Three* ***plus*** *five is eight*

>> **disp(3-5)**

-2  *<-- Three* ***minus*** *five is negative 2*

>> **disp(3\*5)**

15  *<-- Three* ***times*** *five is fifteen*

>> **disp(3/5)**

0.6000 *<-- Three* ***divided by*** *five is six tenths*

>> **disp(3\5)**

1.6667 *<-- Five* ***divided by*** *three is one and two thirds*

>> **disp(5^3)**

125  *<-- Five* ***cubed*** *is one hundred twenty five*

Note that blank lines are frequently erased from these notes. The second divide operator gives the second number over the first one. Raising a number to a power is done with the "^" symbol.  
  
Now let's look at catenation: combining scalars. We use square brackets in MATLAB to catenate things together:

>> **disp([3,5])**

3 5

>> **disp([3 5])**

3 5

The comma is not even necessary. We can also do the same thing with character data but the way it prints is quite different:

>> **disp(['a' 'b'])**

ab

What does the semicolon do?

>> **disp([3;5])**

3

5

>> **disp(['a';'b'])**

a

b

It puts the elements in separate rows. There is alternate way to do that too:

>> **[3**

**5]**

ans =

3

5

The colon produces a vector from two or three scalars:

>> **disp(1:3)**

1 2 3

>> **disp(0.1:3)**

0.1000 1.1000 2.1000

>> **disp(3:2)**

ans =

[]

Two scalars (a and b) separated by a colon give a vector with elements:  
  
a, a+1, a+2, ... a+n  
where n is the largest integer such that a+n < b  
  
Three scalars separated by colons give the same sort of vector, but with  
a spacing between elements equal to the middle scalar as seen in the examples:

>> **disp(1:.5:2)**

1.0000 1.5000 2.0000

>> **disp(3:-1:1)**

3 2 1

**4.2 Operations on Vectors of the Same Length**  
  
  
Now let's see what care must be exercised in using the operators on vectors and matrices. First the operators used with vectors of the same length will be shown where they make sense.

>> **disp([1 2]+[-2 2])**

-1 4

>> **disp([1 2]-[-2 2])**

3 0

>> **disp([1 2]\*[-2 2])**

??? Error using ==> \*

Inner matrix dimensions must agree.

Oops! That last one does not work on vectors. If we want to multiply elements together, we need to use a period to indicate that is our intention:

>> **disp([1 2].\*[-2 2])**

-2 4

>> **disp([1 2]./[-2 2])**

-0.5000 1.0000

>> **disp([1 2].\[-2 2])**

-2 1

>> **disp([1 2].^[-2 2])**

1 4

The same holds for the two divides and the power operator.  
  
Catenation of vectors is the same way as with scalars:

>> **disp([[1 2],[-2 2]])**

1 2 -2 2

>> **disp([[1 2];[-2 2]])**

1 2

-2 2

There does not seem to be a use for the colon with vectors to produce sequences of numbers as with scalars, but it does help in selecting elements from a vector.

>> **v=-3:3;**

>> **disp(v)**

-3 -2 -1 0 1 2 3

>> **disp(v(3:5))**

-1 0 1

**4.3 Operations of a Vector with a Scalar**  
  
Which of our operators work with a scalar and a vector? Let's try them:

>> **disp(3+[1 2])**

4 5

>> **disp(3-[1 2])**

2 1

>> **disp(3\*[1 2])**

3 6

>> **disp(3/[1 2])**

??? Error using ==> /

Matrix dimensions must agree.

We got further than before. At least multiply worked but the first divide does not.

>> **disp(3\[1 2])**

0.3333 0.6667

>> **disp(3./[1 2])**

3.0000 1.5000

The first divide does not require a period to indicate element by element operation, but our second divide does require a period if element by element division is required.

>> **disp([1 2]/3)**

0.3333 0.6667

>> **disp([1 2]\3)**

0

1.5000

>> **disp([1 2].\3)**

3.0000 1.5000

The first and third of these are simply scalar division repeated. In the first case we got each element of the vector divided by three. In the last case we got 3 divided by each element in the vector. In the middle case, MATLAB thought we wanted to solve for x and y such that:  
  
1x + 2y = 3  
  
It gave us one such solution arranged as a column vector.  
  
  
The power operator works the same way as division, but it does not matter whether we raise the elements in the vector to a scalar power or vice versa:

>> **disp(3.^[1 2])**

3 9

>> **disp([1 2]^3)**

??? Error using ==> ^

Matrix must be square.

>> **disp([1 2].^3)**

1 8

**4.4 Matrices Used with the Operators**   
  
We will not demonstrate all the possible operations between matrices and other arrays. Operations of scalars with matrices follow the same behavior that we have seen with vectors.

>> **A=[1 2 3**

**2 3 4];**

>> **disp(A+2)**

3 4 5

4 5 6

>> **disp(A\*2)**

2 4 6

4 6 8

>> **disp(A/2)**

0.5000 1.0000 1.5000

1.0000 1.5000 2.0000

>> **disp(2/A)**

??? Error using ==> /

Matrix dimensions must agree.

>> **disp(2./A)**

2.0000 1.0000 0.6667

1.0000 0.6667 0.5000

Each of the above cases was identical to what we saw with scalars and vectors. We would expect all the other operators to follow as before so we will leave them for you to demonstrate.  
  
How about a matrix with another matrix of the same size? Here again things are pretty much as with operations on a pair of vectors. A few examples should convince us of how element by element operations work:

>> **A1=[-1 0 2**

**0 2 4];**

>> **disp(A+A1)**

0 2 5

2 5 8

>> **disp(A.\*A1)**

-1 0 6

0 6 16

>> **disp(A./A1)**

Warning: Divide by zero

-1.0000 Inf 1.5000

Inf 1.5000 1.0000

>> **disp(A.\A1)**

-1.0000 0 0.6667

0 0.6667 1.0000

[Continue on to Chapter 5](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat5.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 5**

**Logical Operators and Functions**

[5.1 The Logical Operators: *not*, *and*, *or*, *exclusive or*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat5.html#anchor971946)  
[5.2 Associated Logical Functions: *any*, *all*, *find*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat5.html#anchor972526)  
[5.3 Looking for Special Numbers: *NaN* and *Inf*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat5.html#anchor972894)

MATLAB supplies the usual logical operations that find use particularly in writing MATLAB programs. These consist of logical operators used for comparisons as shown next:

**Logical Comparison Operations**

**Example Symbol Meaning**

>> **a = [1 2 3] ;** Equality of numerics --

>> **disp(2 == a)** Thus, only the second

0 1 0 == element of vector a is

equal to 2.

>> **disp('b'=='abB')**

0 1 0 == Equality of character

data

>> **disp(2 <= a)**

0 1 1 <= Less than or equal for

numerics

>> **disp('b'<='abcB')**

0 1 1 0 <= Less than equal for

character data

The examples shown in the table are typical, but the other comparisons shown next:

|  |  |
| --- | --- |
| **Symbol** | Meaning |
| **<** | Less than |
| **>** | Greater than |
| **>=** | Greater than or equal |
| **~=** | Not equal |

may also be done.  
  
The six logical operators all perform element by element comparisons of elements in the same positions in arrays. They also give us a direct comparison of a scalar with all the elements in any array.  
  
The logical comparisons of scalars are what we would expect. They all give 0 for false or 1 for true:

>> **disp(3<5)**

1

>> **disp(5<3)**

0

>> **disp(3<3)**

0

>> **disp(3>=3)**

1

>> **disp(3==3.1)**

0

>> **disp(3~=3.1)**

1

All of those are easy to understand.  
  
The logical operators all perform directly as we would expect on vectors as long as they are the same length.

>> **disp([1 2]<[-2 2])**

0 0

>> **disp([1 2]<=[-2 2])**

0 1

>> **disp([1 2]>[-2 2])**

1 0

>> **disp([1 2]>=[-2 2])**

1 1

>> **disp([1 2]==[-2 2])**

0 1

>> **disp([1 2]~=[-2 2])**

1 0

The logical operators all do quite nicely in comparing a scalar to the elements of a vector, but you need to tell whether the comparison or the vector generator is to be done first:

>> **disp(2>0:4)**

??? disp(2>0:4)

Improper function reference. A "," or ")" is expected.

>> **disp(2>(0:4))**

1 1 0 0 0

Additional parentheses are needed to made this clear.  
  
Operations on each element of matrices is shown next:

>> **m=[1 2 3**

**2 3 4]**;

>> **disp(m<2)**

1 0 0

0 0 0

>> **m1=[-1 0 2**

**0 2 4]**;

>> **disp(m>m1)**

1 1 1

1 1 0

**5.1 The Logical Operators:** *not, and, or, exclusive or*  
We may also want to string the logical results with 'not,' 'or,' 'and,' and `exclusive or' operations. The ways that `not,' 'or,' 'and,' and `exclusive or' operations perform are shown in the next Matlab session:  
  
**More Logical Operations**

>> **disp([0 0 1 1] & [0 1 0 1])**  *<-- & = Logical `and'*

0 0 0 1

>> **disp([0 0 1 1] | [0 1 0 1])**  *<-- | = Logical `or'*

0 1 1 1

>> **disp(~[0 1])**  *<-- ~ = Logical `not'*

1 0

>> **disp(xor([0 0 1 1],[0 1 0 1]))**  *<-- xor = Logical `exclusive or'*

0 1 1 0

The `not' operator (~) helps transform answers that we get for logical comparisons to ones that we need. It can be applied to any array:

>> **disp(m)**

1 2 3

2 3 4

>> **disp(~0)**

1

>> **disp(~[0 1])**

1 0

>> **disp(~m>2)**

0 0 0

0 0 0

>> **disp(~(m>2))**

1 1 0

1 0 0

The examples shown with the matrix m, illustrate two characteristics of MATLAB. In MATLAB logical operations, any non-zero number is treated as being 'true'. Secondly, the *not* operation is done before the comparison if there are no parentheses as in the first case with m. Thus from ~m, we got all zeros or ones and then none of those were found to be greater than 2.  
  
The *and* operation on logical elements is done with the symbol: "&" . The *or* operation on logical elements is done with the symbol: "|" . These operations work with pairs of logical elements to produce results as seen in the vector operations used in our table:

>> **disp([0 0 1 1]&[0 1 0 1])**

0 0 0 1

>> **disp([0 0 1 1]|[0 1 0 1])**

0 1 1 1

These extend to pairs of matrices of the same shape and to a matrix combined with a scalar with no problems:

>> **m=[0 0**

**1 1];**

>> **m1=[0 1**

**0 1];**

>> **disp(m&m1)**

0 0

0 1

>> **disp(m|m1)**

0 1

1 1

>> **disp(m&1)**

0 0

1 1

The *exclusive or* operation on logical elements is done with the function: *xor*. This operation accepts a pair of logical elements to produce results as seen in the vector operations used in our table:

>> **disp(xor([0 0 1 1],[0 1 0 1]))**

0 1 1 0

Be careful to note *xor* is a function, and the two vectors of logical elements must be entered in standard function format.  
  
  
**5.2 Associated Logical Functions:** *any, all, find*  
These three functions find frequent use in making programming decisions about what needs to be done in solving problems. The function *any* tells us whether there are any non-zero elements in a vector as in:

>> **v=[-2 1 3 5];**

>> **disp(any(v<1))**

1

>> **disp(any(v>6))**

0

Since the logical comparison returns 0 for each false comparison and 1 for each true comparison, the function *any* allows us to find if **any** of the comparisons are true. The *all* function tells us if every comparison is true:

>> **disp(all(v<1))**

0

>> **disp(all(v<6))**

1

Both of these functions can be applied to matrices. They simply tell us the condition found for each column of the matrix.

>> **disp(m)**

1 2 3

2 3 4

>> **disp(any(m<2))**

1 0 0

The *find* function allows us to locate non-zero elements in an array. It gives the indices of all non-zero elements as seen next:

>> **disp(v)**

-2 1 3 5

>> **disp(find(v>3))**

4

>> **disp(find(v>0))**

2 3 4

You cannot use *find* with matrices:

>> **disp(m)**

1 2 3

2 3 4

>> **disp(find(m>2))**

??? Error using ==> find

Argument must be a vector.

**5.3 Looking for Special Numbers: NaN and Inf**  
  
  
We have already run into two special numbers that can be generated by arithmetic operations. Dividing a finite number by zero can lead to an infinite "number" indicated by MATLAB as *Inf* . We get a warning message when this occurs. It would be nice to be able to find which element in an array took on such a value. Dividing 0 by 0 leads to another special case: NaN (otherwise know as Not a Number.) Both these special numbers can be seen in:

>> **ex=[0 1 2]./[0 2 0]**

Warning: Divide by zero

ex =

NaN 0.5000 Inf

The functions *isnan* , *finite* and *isinf* allow us to look for such special cases:

>> **disp(isnan(ex))**

1 0 0

>> **disp(finite(ex))**

0 1 0

>> **disp(isinf(ex))**

0 0 1

Note also that the logical operations work in a reasonable fashion with the exceptional numbers.

>> **disp(ex==NaN)**

0 0 0

>> **disp(Inf>ex)**

0 1 0

>> **disp(Inf>=ex)**

0 1 1

>> **disp(NaN>ex)**

0 0 0

>> **disp(NaN>=ex)**

0 0 0

>> **disp(NaN==NaN)**

0

The NaN value essentially can not be compared with any other number, including itself . It is not equal to another copy of the same thing, and it is not greater than or equal to another NaN. The NaN value is unique in that it is the only value which is not equal to itself.

[Continue on to Chapter 6](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

## ****CHAPTER 6****

## ****Using MATLAB to Display Results****

[6.1 Two-Dimensional Plotting Functions in MATLAB](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#twodplot)

[6.1.1 Using fplot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#fplot)  
[6.1.2 Annotating fplot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#annotate.fplot)  
[6.1.3 Making a bar graph](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#bar)  
[6.1.4 Making a pie chart](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#pie)  
[6.1.5 Using plot and yxplot](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#plot)

[Line types and colors for plotting functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#line.color)

[6.2 Miscellaneous Graphics Commands](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6b.html)  
[6.3 Three-Dimensional Plotting Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6c.html)  
[6.4 Advanced Features of Plotting](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html)

[6.4.1 Using Plot Handles](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html#handles)  
[6.4.2 Printing and Saving Graphics Files](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html#print)  
[6.4.3 Images and Graphs](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html#images)

Every student in science or engineering will find many problems that have solutions that are best presented in graphical form. There are many ways to use computer systems to help construct various types of plots. One advantage of using a computer to do so is the fact that much experimental data is now analyzed on computers and therefore this data is directly available in stored form for feeding to a graphics package.  
  
**6.1 Two-Dimensional Plotting Functions in MATLAB**

#### 6.1.1 Using fplot

The new version of the function plotting program: *fplot* has a number of options that will allow you to construct a variety of two dimensional plots. Here are first notes about the function given in the help comments:

>> help fplot

FPLOT Plot a function.  
FPLOT(FNAME,LIMS) plots the function specified by the string variable FNAME between the x-axis limits specified by LIMS = [XMIN XMAX]. LIMS = [XMIN XMAX YMIN YMAX] gives optional y-axis plotting limits.

Further down in the help notes we find one way to specify the function that will be plotted that is most useful:

FNAME may be an eval-able string with variable x, such as 'sin(x)', 'diric(x,10)' or '[sin(x),cos(x)]'

This means that we form a string that specifies any legal Matlab command. It might be as simple as *'sim(x)'* or involve more complicted expressions such as: *'exp(-2\*x).\*sin(x) -5cos(x)'.* Let's see what one example given near the end of the help notes gives:

>> fplot('[tan(x),sin(x),cos(x)]',[-2\*pi 2\*pi -2\*pi 2\*pi])

#### Figure 6.1 Use of fplot



Our example gave the names of the functions to be plotted: tan(x), sin(x) and cos(x) in that order. Note that:

* the independent variable must be given as x (it may be omitted for very simple functions)
* the names of more than one function must be enclosed in square brackets so they form a vector,
* the vector of function names must be enclosed by quotes so that the argument is a string.

The second argument given the plot function said to plot these function for a range of x from negative two Pi to positive two Pi. It also said to limit y to the same range. Without this specification about the range for y, we would have had a plot that just showed a very large jump in tan(x) at each of its discontinuities and the other functions would have looked like straight lines.

#### 6.1.2 Annotating fplot

The plot constructed so far is **very** barren of information. It may be quite useful for a few minutes, but it would not be acceptable as the solution to an assignment or in a report. In order to be used as a solution, we need to include at least a title and put labels on the axes. It may also be very helpful to include information that tells which curve on the plot goes with each function and to add a grid that makes it easier to locate points. The commands: *xlabel* and *ylabel* allwo you to add albels to the axes. A title for the plot may be placed using the command: *title. The command* legendwill allow you to automatically differentiate multiple curves. All of these commands are simple enough that an example of their use should show you the basic dieas behind them. If you need more information about the way they are used, try *help* with them. Here are examples of each applied after the plot of the trigonometric functions was created. The first command *set* is used to change the font used so that text on the axes and in the title are readable.

>> set(gca,'FontName','Helvetica','Fontsize',12)

>> xlabel('x')

>> ylabel('f(x)')

>> title('Trigonometric Functions')

>> legend('tan(x)','sin(x)','cos(x)')

>> grid

#### Figure 6.2 Annotated Plot



Note that the *legend* command uses color to designate the lines in the order they were drawn. This works fine as long as they are viiewed on a color screen, but this can all be lost if you print to a black and white printer. In such a case, it would be advisable to use lines that have different symbols on them. That is one of the options of *fplot* that will be useful.

Two types of graphs are used extensively in presentations: bar graphs and pie charts. Both may be easily used in Matlab. The *help* command applied to each will tell you most of what you need to know, but here is an example of the use of each.

#### Products from Separation Plant mol/s

|  |  |  |  |
| --- | --- | --- | --- |
| **Compound/Stream** | **Stream 1** | **Stream 2** | **Stream 3** |
| n-Pentane | 1.985 | 0.015 | 0 |
| n-Hexane | 0.020 | 2.930 | 0.050 |
| n-Heptane | 0 | 0.015 | 4.985 |

#### 6.1.3 Creating a bar Graph

First here is a session used to create a bar graph:

>> ns=[1.985 .015 0;.02 2.93 .05; 0 .015 4.985]'

ns =

1.9850 0.0200 0

0.0150 2.9300 0.0150

0 0.0500 4.9850

>> bar(1:3,ns)

>> legend('n-Pentane','n-Hexane','n-Heptane')

>> xlabel('Stream Number')

>> title('Products from Separation Plant')

>> ylabel('Flow in mol/s')

Here is the figure produced by this sequence of commands:

#### ****Figure 6.3 Use of bar****

******

#### 6.1.4 Creating a pie Chart

Here is a session used to produce a pie chart giving the mol fractions of each compound in the feed to the plant.

>> sum(ns)*<-- This sums over columns of ns to give the total in each stream*

ans =

2.0050 2.9600 5.0350

>> sum(ns') *<-- This sums over rows to give the total flow of each compound*

ans =

2 3 5

>> pie(sum(ns'))*<-- to produce a pie chart of the feed flows*

>> title('Feed to the Separations Plant')

>> gtext('Total Flow is 10 mol/s')

Note that the legend from the provious bar figure is still while we do this. Here is the figure:

#### ****Figure 6.4 Use of pie****

******

#### 6.1.5 Using plot and yxplot

The *plot* command may be used to plot sets of points that might represent experimantal data or could be from functions. Here is the beginning of the notes form help applied to the command:

>> help plot

PLOT Plot vectors or 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 line up.

An important note about line types and colors is given in the help for plotting functions:

Various line types, plot symbols and colors may be obtained with  
PLOT(X,Y,S) where S is a character string made from any or all the following 3 colunms:

y yellow . point - solid

m magenta o circle : dotted

c cyan x x-mark -. dashdot

r red + plus -- dashed

g green \* star

b blue s square

w white d diamond

k black v triangle (down)

^ triangle (up)

< triangle (left)

> triangle (right)

p pentagram

h hexagram

For example, PLOT(X,Y,'c+:') plots a cyan dotted line with a plus

at each data point; PLOT(X,Y,'bd') plots blue diamond at each data

point but does not draw any line.

Note that this same option to change line type or color applies in the use of *fplot.*

The *plot* command may be used in a variety of ways, but just like *fplot*, it produces plots that are poorly annotated. This is partly alleviated in the program *yxplot*. The program is stored in *~chbe301/matlab/chbe303*. It allows one to plot any one set of y data vs a set of the same length of x data points and adds a title and axis names. A listing of *yxplot* is given below:



An example of using the function is shown in the following session:

>> **t=0:.5:7; <-- A vector of times: 0, .5, 1.0, ..7.0**

>> **d=[0 1 1.6 2 2.1 1.9 1.6 1.3 1.1 1 .9 .95 1 1.05 1.03]; <-- Distances**

>> **yxplot('time','distance','Lab Experiment 1',t,d,'c')**

>> **grid**

The plot is shown below:

#### ****Figure 6.5 Use of yxplot****

******

The [color](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6.html#line.color) you choose for the curve can be particularly important when you want to print your graph or when you want to insert it in notes such as this. The default color is yellow. Yellow shows up well against a black background but it is not suitable with light backgrounds such as white. The *print* command places the graph in a rectangle centered in the middle of the page. The *orient* command can be used to change the way a page is oriented when printed.

>> **help orient**

ORIENT Hardcopy paper orientation.

ORIENT LANDSCAPE causes subsequent PRINT operations

from the current figure window to generate output in

full-page landscape orientation on the paper.

ORIENT PORTRAIT returns to the default PORTRAIT

orientation with the figure window occupying a rectangle

with aspect ratio 4/3 in the middle of the page.

ORIENT TALL causes the figure window to map to the

whole page in portrait orientation.

ORIENT, by itself, returns a string containing

the current paper orientation, either PORTRAIT or

LANDSCAPE (but not TALL). ORIENT is an M-file that sets

the Paper Orientation and Paper Position properties of

the current figure window.

See also PRINT.

[Continue on to Section 6.2: Miscellaneous Graphics Commands](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6b.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

### 6.2 Miscellaneous Graphics Commands

There are many plotting functions that perform different functions in Matlab. Here is a list of some that may be particularly helpful in your work:

### 

|  |  |
| --- | --- |
| **Function** | Performs |
| **clf** | Clears the graphics window |
| **contour** | Allows you to generate a contour plot of a three dimensional object |
| **hold** | Holds the current plot so you can add to it |
| **subplot** | Allows you to put many in the same window |
| **figure** | Allows you to create multiple graphics windows |
| **gco** | GCO Get handle to current object |
| **close** | Allows you to close graphics windows |
| **plotyy** | Creates 2d plot with different y axes on 2 sides |
| **loglog** | Creates a log-log plot |
| **semilogy/semilogx** | Create semi-log plots with one coordinate on a log scale. |
| **gtext** | Plce text in a plot with the mouse |
| **ginput** | Get data using the mouse from a graph |

There are also graphics functions that allow you to input information using the mouse. Two of these may be very helpful. The first one is *gtext*.. It was illustrated in chapter 3. Here is what the *help* function tells us about the function:

>> **help gtext**

GTEXT Place text on a graph using a mouse.

GTEXT('string') displays the graph window, puts up

a cross-hair, and waits for a mouse button or

keyboard key to be pressed. The cross-hair can be

positioned with the mouse (or with the arrow keys on

some computers). Pressing a mouse button

or any key writes the text string onto the graph at

the selected location. See also: GINPUT

This function can simplify the process of placing text in the graphics window to specify notes about the plot. The other function is called *ginput*. Help used on this second function shows:

>> **help ginput**

GINPUT Graphical input from a mouse or cross-hair.

[X,Y] = GINPUT(N) gets N points from the graph

window and returns the X- and Y-coordinates in

length N vectors X and Y. The cross-hair can be

positioned using a mouse (or by using the Arrow

Keys on a terminal). Data points are entered by

pressing a mouse button or any key on the keyboard.

[X,Y] = GINPUT gathers an unlimited number of

points until the return key is pressed.

[X,Y,BUTTON] = GINPUT(N) returns a third argument

BUTTON that contains a vector of integers specifying

which mouse button was used (1,2,3... from left) or

ASCII numbers if a key on the keyboard was used.

[X,Y] = GINPUT(N,'sc') returns X and Y 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

corner. Otherwise, data-coordinates are returned.

There are a number of uses for this function. Here is a simple modification, using *ginput* of the *yxplot* function:



Now if you execute *yxploti* you can add whatever message you want at the point that you designate on the screen. Here is a sample run and its plot:

>> **xs=0:.1:5;**

>> **ys=xs.\*sin(xs);**

>> **yxploti('x','xsin(x)','Pointing out the Maximum','^ Maximum',xs,ys,'c')**

#### ****Figure 6.6 Use of yxploti and ginput****

****

[Continue on to Section 6.3: Three-Dimensional Plotting Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6c.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

### ****6.3 Three-Dimensional Plotting Functions****

The *mesh* function allows you to view a three dimensional object from various locations relative to the coordinate axes. The function *mesh* is described with *help* as:

>>  **help mesh**

MESH 3-D mesh surface.

MESH(X,Y,Z,C) plots the colored parametric mesh defined by four matrix

arguments. The view point is specified by VIEW. The axis labels are

determined by the range of X, Y and Z, or by the current setting of AXIS.

The color scaling is determined by the range of C, or by the current

setting of CAXIS. The scaled color values are used as indices into the

current COLORMAP.

MESH(X,Y,Z) uses C = Z, so color is proportional to mesh height.

MESH(x,y,Z) and MESH(x,y,Z,C), with two vector arguments replacing the

first two matrix arguments, must have length(x) = n and length(y) = m

where [m,n] = size(Z). In this case, the vertices of the mesh lines are

the triples (x(j), y(i), Z(i,j)).

Note that x corresponds to the columns of Z and y corresponds to the rows.

MESH(Z) and MESH(Z,C) use x = 1:n and y = 1:m. In this case, the height,

Z, is a single-valued function, defined over a geometrically rectangular

grid.

MESH returns a handle to a SURFACE object.

AXIS, CAXIS, COLORMAP, HOLD, SHADING and VIEW set figure, axes, and surface

properties which affect the display of the mesh.

See also SURF, MESHC, MESHZ, WATERFALL.

The function *meshgrid* used to produce data for three dimensional plots of functions is described by:

>> **help meshgrid**

MESHGRID Generation of X and Y arrays for 3-D plots.

[X,Y] = MESHGRID(x,y) transforms the domain specified by vectors x and y

into arrays X and Y that can be used for the evaluation of functions of

two variables and 3-D surface plots.

The rows of the output array X are copies of the vector x and the columns

of the output array Y are copies of the vector y.

For example, to evaluate the function x\*exp(-x^2-y^2) over the

rectangle:

-2 < x < 2, -2 < y < 2,

[X,Y] = meshgrid(-2:.2:2, -2:.2:2);

Z = X .\* exp(-X.^2 - Y.^2);

mesh(Z)

[X,Y] = MESHGRID(x) is an abbreviation for [X,Y] = MESHGRID(x,x).

[X,Y,Z] = MESHGRID(x,y,z) produces packed 3-D arrays that can be used to

evaluate functions of three variables and 3-D volumetric plots.

See also SURF, SLICE.

Here is a rather nice picture of a three dimensional object.

>>  **[X,Y]=meshgrid(-1.5:.1:1.5,-1.5:.1:1.5);**

>>  **Z=sin(3\*X.^2+2\*Y.^2)./(X.^2+Y.^2);**

Warning: Divide by zero.

>>  **Z=sin(3\*X.^2+2\*Y.^2)./(X.^2+Y.^2+1e-10);**

>>  **mesh(Z)**

>>  **ht=title('Cowboy Hat')**

ht =

6.0001

>>  **set(ht,'FontSize',14)**

#### ****Figure 6.7 Use of**** meshgrid ****and**** mesh

|  |
| --- |
| http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/images/fig67.gif |

The function *meshex* allows the user to rotate an object about axes. It lists as:



Here is a session in which the example suggested in *meshgrid help* is seen from various points:

>> **[x,y]=meshgrid(-2:.2:2,-2:.2:3);**

>> **z=x.\*exp(-x.^2-y.^2);**

>> **mesh(z)**

This showed the default position of the viewpoint. Now here is an example of 7 different views:

>> **meshex(z,[0 0],[10 -5],7)**

Here is the last view shown:

#### ****Figure 6.8 Another View of a Three Dimension Object****



The *surf* command produces a three dimensional picture similar to the one from *mesh*. It is more colorful since each segment of the surface is colored rahter than just shown as a line. Here is the result of using *surf* on the same data used to make Figure 6.7.**Figure 6.9 Use of *surf***

******

[Continue on to Section 6.4: Advanced Features of Plotting](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat6d.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

### ****6.4 Advanced Features of Plotting****

**6.4.1 Using Plot Handles**

The function *fplot* returns the x and y coordinates used on the figure. On the other hand *plot* give handles to each of the curves shown on the plot. Thus we won't try to deal with handles associated with plots from *fplot*, but use an our example a case where we start with:

>> t=-2\*pi:pi/8:2\*pi;

>> hplot=plot(t,sin(t),t,cos(t).\*exp(-.5\*abs(t)))

hplot =

2.0001

5.0001

Each of the handles: hplot(1) and hplot(2) is associated with one of the curves that is shown. Note that if you have forgotten to save the handle to an object, you can find its value with the *gco* command. This is very handy if you have a curve (or any other object) you wish to remove from a figure, just click on it and issue the command:

>> delete(gco)

Each of the annotation commands: *xlabel*, *ylabel*, *title*, *text*, *gtext* and *legend* also returns one or more handles.

>> hxl=xlabel('time im ms')

hxl =

6.0001

>> hyl=ylabel('sin(t) and cos(t)e^{-\alpha\*|t|}')

hyl =

7.0001

>> ht=title('Experimenting with Handles')

ht =

8.0001

Note the very limited use of TeX characters in the label on the Y axis. This feature was used to make the Greek character for alpha and to make the exponential a superscript. It has more extensive possibilities in annotating graphs. You may also see that the exponetial part of the Y label is so small that it is unreadable in its default size and font. This is where the handles become very useful. The numeric values for the handles may be used to "fine tune" your plots. Here are a couple of examples. We may see that the font used for labeling both axes are too small. We can correct that by:

>> set(hxl,'FontSize',12)

>> set(hyl,'FontSize',12)

We can make the title even larger:

>> set(ht,'FontSize',14)

We may not like the color used to plot the first curve. To find which color was used the *get* command is appropriate:

>> get(hplot(1),'Color')

ans =

0 0 1

Those three numbers give the amount of Red, Green and Blue in the color. To change to red, we will use: [1 0 0], but note that the numbers are not limited to being 0 or 1. They may also be fractions. Here is red:

>> set(hplot(1),'Color',[1 0 0])

We may even see after we use the larger font that we had a typo in the x label and need to correct it:

>> set(hxl,'String','Time in ms')

We can add text to label on of the curves in the color associated with the curve:

>> a=get(hplot(2));

>> text(-3,.8,'Color of cos\*exp(-|t|/2)','Color',a.Color)

Note that *a* is a structure and *a.color* is the color associated with the second curve. Our final plot looked like:

#### ****Figure 6.10 Use of Handles****



#### 6.4.2 Printing and Saving

Graphics Files

There are a variety of ways to save the graphics results that you produce in Matlab. You can of course print the window directly to a connected printer. This is easily done with the Print... Command under the File Menu at the top of the Figure Window. This File menu is shown in Figure 6.2. The selection of Print...brings up a window with lots of options.



The default settings should produce a copy of the figure on your default printer. You can chose a variety of other options such as:

* print on other paper type if your printer supports them
* print in Landscape mode rather than portrait
* print with a large number of possible Device Options
* produce a file rather than printing immediately
* open a window to do a Page Setup

You will find that after you print your figure or even if you send an image of it to a file, you will not have something that you can work with in Matlab in a different session. Matlab now allows you to recreate your figure by using the Save As... option under the File Menu. If you save your figure this way Matlab creates an *m* file together with a *mat* file. The *m* file can then be executed to recreate the figure. I saved Figure 6.10 calling it figure1. Here is a listing of the first part of the m file it produced:

whale% cat figure1.m

function figure1()

% This is the machine-generated representation of a MATLAB object

% and its children. Note that handle values may change when these

% objects are re-created. This may cause problems with some callbacks.

% The command syntax may be supported in the future, but is currently

% incomplete and subject to change.

% % To re-open this system, just type the name of the m-file

% at the MATLAB prompt.

%The M-file and its associtated MAT-file must be on your path.

load figure1

a = figure('Color',[0.8 0.8 0.8], ...

'Colormap',mat0, ... 'Position',[517 351 560 420]);

b = axes('Parent',a, ...

'Box','on', ...

'CameraUpVector',[0 1 0], ...

'Color',[1 1 1], ...

...............

Looking at my home directory, I find there is a *mat* file produced as well as the *m* file.

whale% ls figure1\*

figure1.m figure1.mat

If I quit my Matlab session and come back another time, I can restore the same graph by simply typing.:

>> figure1

However, when you use the command *who*, you find all of the handle variables have not been reproduced. This may limit what you can do if you want to change details about the figure. If you need to make such changes, you will have to become acquainted with the commands: *findobj, gca, gcf,* and even more complicated *set* and *get*. I found for example that the all the handles could be determined with:

>> Hs=findobj

Hs =

0

1.0000

3.0001

9.0001

5.0001

4.0001

You can see that these are not the same numeric values set when the objects were first formed.

A second topic in this section is concerned with the production of *gif* files for use in web pages. This section might be applied to copying a number of windows besides Matlab figures, but currently it is quite important in Matlab since there is no direct way to produce *gifs* or *jpeg* files from Figures. Here is the procedure we have used to produce most of the figures in the chbe301 and 303 web pages. If you have a whole or part of a window you want to make into a gif file, then:

* type xv in an xterm: wsname% xv This will create a colorful window that you should click on with the RIGHT button to bring up an xv controls window.
* Move all windows that you are using so they do not cover any part of the window you want to copy.
* Click on Grab in the xv controls window. This brings up an xv grab window.
* Change the delay to 5 seconds
* Click the Grab button and move the cursor to the window to be copied.
* Wait 5 seconds until you hear a beep.
* Click on the window and a copy of that window should appear.
* Bring the copy to the front and select the part you want to make into a gif file. Do this with the left mouse button by holding it down starting at one corner and wiping across to the other corner of the rectangle you want.
* Hit the Crop button on the xv controls.
* Save the file as GIF in Full Color.
* Give the file a name with the appendage gif.
* Quit xv unless you have another window to copy.

#### 6.4.3 Images and Graphs

Matlab can produce and manipulate two different types of figures: graphs from programs like plot and images that might have originated as scanned bit maps. The figure at the top of the main pages in these notes is an image. A copy of it is stored in /home/chbe301/public\_html/images/089-sm.jpg. We can load that file in a Matlab session with the imread command:

>> Symb=imread('/home/chbe301/public\_html/images/089-sm.jpg');

It loads as a three dimensional object with a large number of

elements.

>> size(Symb) ans = 115 118 3

If you look at a few of the elements, you see that they are

numeric:

>> Symb(1:3,1:3,1)

ans =

132 139 129

122 128 118

121 125 119

To see the images, we use:

>> image(Symb)

This produces a larger version of the figure that you have been

seeing. It looks better if it is made square:

>> axis image



You can manipulate such images in a number of ways such as by changing the colors used to map the number in the array and changing the location of the virtual light source for the image. If you produce an image that you want to save and store in a web page, you can do so with the *imwrite* command. It requires three arguments: the name of the image, the finame where you are going to store the data and the format you want to use. One of the allowable formats is 'jpg' (Joint Photographic Experts Group) and can be used in web pages.

[Continue on to Chapter 7: More Useful MATLAB Arithmetic Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 7**

**More Useful MATLAB Arithmetic Functions**

[7.1 Find and Evaluate Character Strings: *grep* and *eval*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1028533)  
[7.2 The Exponential and Logarithm Functions: *exp* and *log*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1029255)  
[7.3 General Powers and Roots: *x.^a,* and, *sqrt(x)*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1032101)  
[7.4 Hyperbolic Functions & Their Inverses: *cosh*, *sinh*, *tanh*, etc.](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1033113)  
[7.5 Trigonometric Functions & Their Inverses: *cos*, *acos*, *sin*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1034129)  
[7.6 Setting Nothing, Zeros and Random Values in Arrays](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat7.html#anchor1035004)

This chapter will include examples of the use of many functions that have been used in the examples or will be required in the Laboratory Assignments. The current list of functions discussed here is:

|  |  |
| --- | --- |
| **Function(s)** | **Performs** |
| grep and eval | Finds and evaluates character strings |
| exp and log | Exponential and logarithm functions |
| Power functions | Powers and roots |
| cosh, sinh, tanh, and acosh, asinh, atanh | Hyperbolics and their inverses |
| cos, sin, tan, and acos, asin, atan | Trig. functions and their inverses |
| [ ], zeros, rand | Setting nothing, zeros, and random numbers |
| polyfit, polyval and roots | Using Polynomials |

Many of these functions have been used in the examples shown previously, but you may need more information about them and the way they treat special arrays.  
  
**7.1 Find and Evaluate Character Strings: grep and eval**  
  
  
The *eval*, function greatly expands the way in which many programs may be used in MATLAB. It is essential in such programs as *ssec2* to allow the user to pass various functions to the program when different equations are to be treated. It finds frequent use as a means to evaluate a function that is to be plotted, find a root for or used as an argument in an integral or differential equation solution. The fact that I give the program this information in a character string has the added benefit of making it easy to then put it in a title or label.  
  
Suppose we want to tell our program to use the program *f(x)* in a certain calculation; where *f(x)* one time might be *sin(x)* and another time *exp(x)-2x*. It might even be a real complicated function that we have written and stored as *rcfunc.m*, in our matlab directory. I have found it to be most convenient to settle on exactly what the independent variable will be called. In most of the functions in these notes it is x or t. Then the function knows what to change when the function is to be evaluated for new data. Here are examples of the *eval* as it is used in the command window:

>> **c1='exp(x)';**

>> **x=1:.5:2;**

>> **disp(eval(c1))**

2.7183 4.4817 7.3891

>> **x=[-1 0 1];**

>> **disp(eval(c1))**

0.3679 1.0000 2.7183

>> **c2='x-sin(x).^2';**

>> **disp(eval(c2))**

-1.7081 0 0.2919

An alternate way to use *eval*, in functions was used in several programs. In it the user simply supplied the function name. It is shown next:

>> **c3='sin';**

>> **x=[-1 0 1];**

>> **disp(eval([c3,'(x)']))**

-0.8415 0 0.8415

This works fine for simple functions like *sin*, or *exp*, but it could not be used directly for a function as simple as *x-sin(x)*. You would have to create a function in an m-file, to return that function. That is not so difficult, but it can clutter up your MATLAB directory unless you remove the ones you no longer need.  
  
  
**7.2 The Exponential and Logarithm Functions:** *exp* **and** *log*  
  
  
The exponential and logarithm functions are inverses to one another at least for some ranges of their arguments. We should always expect that:

exp(log(x)) = x for x 0

but there are cases where:

log(exp(x)) x

Here are some cases that demonstrate the first equality and show that sometimes the second is true as well:

>> **z=[0 0.1 1 2 pi]+i\*[-1 0 0 2 0];**

>> **disp(z-exp(log(z)))**

1.0e-15 \* *<-- Note that this is very small*

-0.0612 -0.0139 0 -0.4441 0

>> **disp(z-log(exp(z)))**

1.0e-16 \* *<-- So are these.*

0 -0.6939 0 0 0

Here are ones in which the first is still true, but not the second:

>> **z=[1 1 1]+i\*pi\*(2:4);**

>> **disp(z-exp(log(z)))**

1.0e-14 \* *<-- These are small.*

-0.0222 - 0.0888i 0.0999 + 0.1776i -0.0444 - 0.3553i

>> **disp(z-log(exp(z)))** *<-- But these are not small.*

0 + 6.2832i 0 + 6.2832i 0 +12.5664i

The logarithm function always gives the principal value for its result so that its imaginary part is in the range (-,). The last example shows that when you take the exponential of a number with its imaginary part outside this range, you get back from the log function a number that differs by a multiple of 2i.  
  
  
  
**7.3 General Powers and Roots: x.^a, and, *sqrt(x)***  
  
  
Integer powers of any non-zero number may be found by:

>> **x=[-5.5 2 2+i];**

>> **disp(x.^2)**

30.2500 - 0.0000i 4.0000 3.0000 + 4.0000i

>> **disp(x.^-3)**

??? disp(x.^-

Missing variable or function.

>> **disp(x.^(-3))**

-0.0060 - 0.0000i 0.1250 0.0160 - 0.0880i

You just have to careful about supplying required parentheses. Roots may also be found with the same function:

>> **x=[-5.5 2 2+i];**

>> **disp(x.^(1/3))**

0.8826 + 1.5287i 1.2599 1.2921 + 0.2013i

Even complex powers may be found:

>> **x=[-5.5 2 2+i];**

>> **disp(x.^(1-i))**

1.0e+02 \*

0.1700 + 1.2613i 0.0154 - 0.0128i 0.0335 - 0.0119i

The *sqrt*, function is a special case and has the same properties that any other fractional power has. We always get the positive root of real positive numbers and for other numbers we get the principal value. In complex variables courses it is shown that all the fractional powers and powers of complex numbers are really multiple valued so MATLAB returns the principal values for these. It does so using the definition:

x^a = exp(a\*log(x)) for x 0

The principal value of the *log* is used in finding the powers with this definition for a fraction or complex number.  
  
  
**7.4 Hyperbolic Functions & Their Inverses:** *cosh, sinh, tanh*, etc.  
  
  
The hyperbolic functions *cosh*, and *sinh*, are defined in terms of the exponential function:

cosh(x) = (exp(x) + exp(-x))/2  
sinh(x) = (exp(x) - exp(-x))/2

Then *tanh(x)* is given as:

tanh(x)=sinh(x)/cosh(x)

All could be eliminated from MATLAB's vocabulary, but they appear in many solutions of problems and make our programs shorter when we use them. Here are a few examples:

>> **x=[-5.5 2 2+i];**

>> **disp(cosh(x))**

1.0e+02 \*

1.2235 0.0376 0.0203 + 0.0305i

>> **disp(sinh(x))**

1.0e+02 \*

-1.2234 0.0363 0.0196 + 0.0317i

>> **disp(tanh(x))**

-1.0000 0.9640 1.0148 + 0.0338i

The inverse hyperbolic functions are defined so that:

cosh(acosh(x)) = x  
sinh(asinh(x)) = x  
tanh(atanh(x)) = x

We could solve for any of these and express it in terms of the *log* function. They all have the same properties that *log* does: they are multivalued and MATLAB gives you the principal value. Thus you do not always get back x from: acosh(cosh(x)), asinh(sinh(x)) or atanh(tanh(x)).  
  
Here are examples involving *cosh*, and *acosh*:

>> **x=[-5.5 2 2+i];**

>> **w=cosh(x);**

>> **disp(acosh(w))**

5.5000 2.0000 2.0000 + 1.0000i

>> **u=acosh(x);**

>> **disp(u)**

-2.3895 - 3.1416i 1.3170 1.4694 + 0.5074i

>> **disp(cosh(u))**

-5.5000 + 0.0000i 2.0000 2.0000 + 1.0000i

**7.5 Trigonometric Functions & Their Inverses: *cos, acos, sin***   
  
The trigonometric functions can also be related to the exponential function through Euler's formula:

exp(ix) = cos(x) + isin(x)

The same equation with -x substituted for x gives:

exp(-ix) = cos(x) - isin(x)

where we have made use of the even and odd properties of *cos* and *sin*. If we add the two equations and divide by 2, we get:

cos(x) = (exp(ix) + exp(-ix))/2

and then:

sin(x) = (exp(ix) - exp(-ix))/(2i)

The next session illustrates some of these equations:

>> **disp(cos(x))**

0.7087 -0.4161 -0.6421 - 1.0686i

>> **e1=exp(i\*x);**

>> **e2=exp(-i\*x);**

>> **disp((e1+e2)/2)**

0.7087 -0.4161 -0.6421 - 1.0686i

The inverse functions satisfy the equations:

cos(acos(x)) = x  
sin(asin(x)) = x  
tan(atan(x)) = x

The inverses may again be related to the *log* function. They are multivalued with the principal values reported by MATLAB. Here are some examples involving the *cos* function:

>> **x=[-5.5 2 2+i];**

>> **disp(cos(acos(x)))**

-5.5000 + 0.0000i 2.0000 2.0000 + 1.0000i

>> **disp(acos(cos(x)))**

0.7832 2.0000 2.0000 + 1.0000i

This included an example to illustrate that:

acos(cos(x)) may not be = x

**7.6 Setting Nothing, Zeros and Random Values in Arrays**   
  
  
As you work with programs for this course, you will find that a number of programs define arrays with nothing in them. This is a legacy from APL where it was necessary to start with ``empty" arrays before you could build up answers. In MATLAB this is rarely required , but it can be done as shown in defining the array empty:

>> **clear**

>> **empty=[];**

>> **who**

Your variables are:

empty

>> **empty**

empty =

[]

The variable *empty*, is there, but we have not stored anything in it yet. Empty arrays could be used as starting points in looping operations. We can add numbers in *empty*, to form a vector by:

>> **empty=[empty 1]**

empty =

1

You can accomplish the same thing by:

>> **who**

Your variables are:

empty

>> **notemp=[notemp 1]**

notemp =

1

The important point to see in this example is that *notemp*, did not exist, but I was able to add to it. Thus I do not need an empty array to start with. Here is another example that followed the previous one so that the only arrays that had been defined were *empty* and *notemp*:

>> **newvec(4)=1**

newvec =

0 0 0 1

>> **newmat(2,3)=3.5**

newmat =

0 0 0

0 0 3.5000

>> **mat2(2,:)=[-1 3 8]**

mat2 =

0 0 0

-1 3 8

Thus if a vector or matrix does not exist, but we assign one particular element a value, MATLAB produces an array big enough to have that element in it. We could also have created *mat2*, shown in our last session by replacing rows in an array that was set initially with zeros in it:

>> **mat3=zeros(2,3)**

mat3 =

0 0 0

0 0 0

>> **mat3(2,:)=[-1 3 8]**

mat3 =

0 0 0

-1 3 8

There is no need to do this in MATLAB, but it may speed computations up some for large arrays.  
  
In addition to arrays of all zeros as shown in the example, you can also make arrays of ones with the *ones*, function, or square identity matrices with the *eye*, function. Use *help*, to find out more about these useful functions.  
  
Most engineering problems are posed in such a way that it appears that we can ``solve" them to obtain very specific answers. We think of such problems as being ``deterministic". Given the dimensions of an object, we think we can determine its volume or surface area. In most cases, we have to rely on measurements or specifications that are approximate. Suddenly, we are faced with real problems that are ``non-deterministic". Our measurements have a certain `randomness' about them and this carries over to our results. In non-deterministic problems, we have to be content with finding average or mean properties. A lot of computer time has been spent in modeling and computing these properties for a variety of problems by performing computer ``experiments". This requires some way to generate `random' numbers. There are several ways to do this. The *rand* function gives a random number between 0 and 1. It may be used in several ways:

1) generate a single random number in (0,1) with ``equal" likelihood for each point in the interval with *rand*,  
2) generate a whole matrix of random numbers in (0,1) with ``equal" likelihood for each point with *rand(n,m),*  
3) generate random numbers in (0,1) with a normal distribution by executing: *rand('normal')* before asking for *rand* or *rand(n,m).*

Here are a few examples:

>> **rand**

ans =

0.2190

>> **v=rand(1,100);**

>> **sum(v)/100**

ans =

0.5194

You may be surprised to know that if you try these same instructions you may get the same results. On the other hand, if I repeat them a second time, I will not:

>> **rand**

ans =

0.8024

>> **v=rand(1,100);**

>> **sum(v)/100**

ans =

0.5040

The program that generates these numbers always starts with the same number (called a `seed') but this gets changed each time you ask for a new random number.

[Continue on to Chapter 8](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 8**

**Functions that Operate on Arrays**

[8.1 The *sum* and *cumsum* Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1080909)  
[8.2 The *prod* and *cumprod* Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1082635)  
[8.3 The *max* and *min* Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1084173)  
[8.4 The *sort* Function](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1084917)  
[8.5 The *mean* and *median* Functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1086464)  
[8.6 Moments and the Standard Deviation Function: *std*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1087347)  
[8.7 The *diff* Function and an Approximate Derivative](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat8.html#anchor1088143)

There are a whole arsenal of function in MATLAB that will be useful in your work. We will discuss and demonstrate the following functions in this chapter.

|  |  |
| --- | --- |
| **Function** | **Performs on vectors** |
| sum | Sums the elements |
| cumsum | Gives a running sum of the elements |
| prod | Multiplies the elements together |
| cumprod | Gives a running product of the elements |
| max | Finds the maximum value |
| min | Finds the minimum value |
| sort | Sorts the elements |
| mean | Finds the mean value of the elements |
| std | Finds the standard deviation of elements |
| diff | Gives the differences between adjacent elements |

All of these also perform the same operation on each column of a matrix. Let's look at each function in a MATLAB session. We start it by setting up some test vectors and matrices. We will include some complex arrays as well as real ones.

>> **vr=[3 -3 -5.2 12.4];**

>> **vc=vr+i\*[-2 3 -1 2.2];**

>> **mr=[3 14 -5**

**-3 15 -2**

**-5.2 1 10**

**12.4 16 11];**

Now we will look at the way each function treats these arrays.  
  
  
**8.1 The** *sum* **and** *cumsum* **Functions**  
  
Here is a test of the *sum* and *cumsum* functions.

>> **disp(sum(vr))**

7.2000

>> **disp(sum(vc))**

7.2000 + 2.2000i

>> **disp(sum(mr))**

7.2000 46.0000 14.0000

The *sum* function just sums the elements in both real and complex vectors. It sums each column in a matrix and gives a vector of these sums. If we want to sum over rows, we need to transpose the matrix first and transpose the result if we want a single column result.

>> **disp(sum(mr))**

12.0000

10.0000

5.8000

39.4000

Now that we have seen what the *sum* function does, it is not too hard to figure out what *cumsum* gives:

>> **disp(cumsum(vr))**

3.0000 0 -5.2000 7.2000

>> **disp(cumsum(vc))**

3.0000 - 2.0000i 0 + 1.0000i -5.2000 7.2000 + 2.2000i

>> **disp(cumsum(mr))**

3.0000 14.0000 -5.0000

0 29.0000 -7.0000

-5.2000 30.0000 3.0000

7.2000 46.0000 14.0000

The result is always an array of the same size as the array operated on. The last element is always what *sum* gave. The rest are partial sums over 1, then 2 then 3, etc. elements in the vector or columns of a matrix. For example, operating on the real vector: *vr*, we get:

3 as the sum of the first element

3+(-3)=0 as the sum of the first two elements

3+(-3)+(-5.2)=-5.2 as the sum of the first three elements

3+(-3)+(-5.2)+12.4=7.2 as the sum all four elements

These four results are then placed in a vector result.  
  
  
**8.2 The** *prod* **and** *cumprod* **Functions**  
  
  
Next the *prod* and *cumprod* functions will be tested. The *prod* function acts just like *sum* except the elements are multiplied together:

>> **disp(prod(vr))**

580.3200

>> **disp(prod(vc))**

5.4444e+02 - 8.6268e+02i

>> **disp(prod(mr))**

1.0e+03 \*

0.5803 3.3600 1.1000

The complex product may not be familiar, but the other results should not be surprising. You can check the two real cases with a calculator. You will find that the product over the columns of the matrix were displayed in short format mode and truncation shows only

0.5803 \* 10^3 or 580.3

rather than 580.32.  
  
The cumulative product function behaves in an analogous fashion to the cumulativesum function:

>> **disp(cumprod(vr))**

3.0000 -9.0000 46.8000 580.3200

>> **disp(cumprod(vc))**

1.0e+02 \*

0.0300 - 0.0200i -0.0300 + 0.1500i 0.3060 - 0.7500i 5.4444 - 8.6268i

>> **disp(cumprod(mr))**

1.0e+03 \*

0.0030 0.0140 -0.0050

-0.0090 0.2100 0.0100

0.0468 0.2100 0.1000

0.5803 3.3600 1.1000

We will leave it to the exercises to figure out these results.  
  
  
**8.3 The** *max* **and** *min* **Functions**  
  
  
The *max* and *min* functions find the largest and smallest elements in a vector or each column of a matrix. This is done in the algebraic sense and if the elements are complex, the magnitude of complex numbers is used in the comparison. Here is what our example arrays have for maxima:

>> **disp(max(vr))**

12.4000

>> **disp(max(vc))**

12.4000 + 2.2000i

>> **disp(max(mr))**

12.4000 16.0000 11.0000

The real arrays again offer no surprises, but the example chosen for the complex vector fails to tell us exactly what the criteria was since the last element had both the largest real part and largest magnitude. This can be easily remedied:

>> **vc(3)=-5.2-12.4\*i;**

>> **disp(max(vc))**

-5.2000 -12.4000i

>> **disp(vc)**

3.0000 - 2.0000i -3.0000 + 3.0000i -5.2000 -12.4000i 12.4000 + 2.2000i

>> **disp(abs(vc))**

3.6056 4.2426 13.4462 12.5936

Thus we see that the criterion is based on the magnitude of the numbers.  
  
Now the *min* function is seen to give results based on the same procedures used in the *max* function except we now look for the smallest elements or those with the smallest magnitudes.

>> **disp(min(vr))**

-5.2000

>> **disp(min(vc))**

3.0000 - 2.0000i

>> **disp(min(mr))**

-5.2000 1.0000 -5.0000

There were no questions about what *min* would find, were there?  
  
  
**8.4 The** *sort* **Function**  
  
  
Here is a function that goes along with the last pair. It gives the complete order for all the elements in a vector or in each column of a matrix. Again the magnitudes of complex numbers are used in the comparison. The ordering is done from smallest to largest.

>> **disp(sort(vr))**

-5.2000 -3.0000 3.0000 12.4000

>> **disp(sort(vc))**

3.0000 - 2.0000i -3.0000 + 3.0000i 12.4000 + 2.2000i -5.2000 -12.4000i

>> **disp(sort(mr))**

-5.2000 1.0000 -5.0000

-3.0000 14.0000 -2.0000

3.0000 15.0000 10.0000

12.4000 16.0000 11.0000

If we want to sort the matrix over rows, we need to do a little transposing:

>> **disp(sort(mr')')**

-5.0000 3.0000 14.0000

-3.0000 -2.0000 15.0000

-5.2000 1.0000 10.0000

11.0000 12.4000 16.0000

If we want to know where the elements in the sorted form stood in the original array we can use *sort* in the form:

>> **m=[2 5 8**

**-1 7 5**

**0 2 10];**

>> **[ms,mi]=sort(m)**

ms =

-1 2 5

0 5 8

2 7 10

mi =

2 3 2

3 1 1

1 2 3

**8.5 The** *mean* **and** *median* **Functions**

Now let's look at the difference between the mean and median of our arrays. Here are the mean values:

>> **disp(mean(vr))**

1.8000

>> **disp(mean(vc))**

1.8000 - 2.3000i

>> **disp(mean(mr))**

1.8000 11.5000 3.5000

In each case MATLAB simply adds the elements and divides by the number of elements. We could have found the same result for the vectors by:

>> **disp(sum(vr)/length(vr))**   
 1.8000

>> **disp(sum(vc)/length(vc))**

1.8000 - 2.3000i

The matrix mean could also have been found as long as we use the length to get its number of rows:

>> **disp(sum(mr)/length(mr))**

1.8000 11.5000 3.5000

Now look at the median values applied to vectors with three elements:

>> **disp(median([-1 5 6]))**

5

>> **disp(median([-1 -1 6]))**

-1

It simply orders the elements and reports the middle one. In our example vectors, we have an even number of elements. Traditionally, we would expect to order them and find the mean of the two in the middle.

>> **disp(median(vr))**

0

>> **disp(median(vc))**

4.7000 + 2.6000i

Looking back at the sorted vectors, we see that the mean of the middle elements is what is returned by the median function.  
  
  
**8.6 Moments and the Standard Deviation Function:** *std*  
  
  
Let us now look at a single vector: x of n elements:

x1, x2, .... xn

The mean will be designated as: xmean. The kth moment about the mean is defined as:

i=n

µk =[sum(xi -xmean )k ]/n

i=1

The standard deviation is then defined as:

s = µ2

The *std* function gives this value. Let's check to see that the definitions given above are consistent with what we expect:

>> **x=[.1 1 1.2 .5 .8 .9];**

>> **xbar=mean(x)**

xbar =

0.7500

>> **xmxb=x-xbar**

xmxb =

-0.6500 0.2500 0.4500 -0.2500 0.0500 0.1500

>> **n=length(x)**

n =

6

>> **mu2=sum(xmxb.^2)/n**

mu2 =

0.1292

>> **sqrt(mu2)**

ans =

0.3594

>> **std(x)**

ans =

0.3594

It checks!  
  
  
**8.7 The** *diff* **Function and an Approximate Derivative**  
  
We will demonstrate the *diff* function only for vectors. It can be used on matrices and gives the same thing for each column of the matrix that is does for a vector. For a vector with elements:

x1, x2, .... xn

diff gives a vector with elements:

x2 - x1, x3 - x2, .... xn - xn-1

If we apply diff to the vector:

>> **x**

x =

0.1000 1.0000 1.2000 0.5000 0.8000 0.9000

we get:

>> **disp(diff(x))**

0.9000 0.2000 -0.7000 0.3000 0.1000

As an application of diff look at calculating the derivative of the function sin(t) using the numerical approximation:

df/dt at tn ~ (f(tn+1) - f(tn))/(tn+1 - tn)

Here is a session used to do just that:

>> **t=0:pi/32:pi;**

>> **sint=sin(t);**

>> **dsint=diff(sint);**

>> **dt=diff(t);**

>> **dfdt=dsint./dt;**

The length of each of the vectors: sint, dsint, dfdt is:

>> **disp(length(dfdt))**

32

Thus we will make our comparison over the same number of values of t:

>> **t=t(1:32);**

>> **plot(t,[dfdt;cos(t)])**

>> **title('Numerical Approximation to dsin(t)/dt')**

>> **xlabel('t')**

>> **text(1.5,.5,'\_\_\_ Appx. with dt = pi/32')**

>> **text(1.5,.4,'--- cos(t)')**

Here is what we got:  
  
  
**Figure 8.1 Numerical Approximation to a Derivative**

|  |
| --- |
| http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/images/deriv.gif |

[Continue on to Chapter 9](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat9.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 9**

**The Main Group of Matrix Operations**

[9.1 Matrix Multiplication](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat9.html#anchor1057455)  
[9.2 Determinants and Matrix Division](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat9.html#anchor1058758)

We will expand on the use of matrix operations on arrays in this chapter. These operations give MATLAB (for Matrix Laboratory) its name. As we saw in the section on element by element operations, there are many cases where MATLAB thinks we want to do matrix operations unless we specify (with the use of a period) otherwise. First a trivial, but important operation: forming the transpose. This is done in MATLAB with the single quote: '. It essentially reverses rows and columns in a matrix. If the matrix is complex in MATLAB it also takes the conjugate of each element. If we want to keep from getting a conjugate, we need to insert a period before the quote. Here are some examples:

>> **disp([1 2]')**

1

2

>> **a=[1 2 3**

**4 5 6];**

>> **disp(a')**

1 4

2 5

3 6

>> **z=a+i\*[-1 0 1;-2 1 2];**

>> **disp(z)**

1.0000 - 1.0000i 2.0000 3.0000 + 1.0000i

4.0000 - 2.0000i 5.0000 + 1.0000i 6.0000 + 2.0000i

>> **disp(z')**

1.0000 + 1.0000i 4.0000 + 2.0000i

2.0000 5.0000 - 1.0000i

3.0000 - 1.0000i 6.0000 - 2.0000i

>> **disp(z.')**

1.0000 - 1.0000i 4.0000 - 2.0000i

2.0000 5.0000 + 1.0000i

3.0000 + 1.0000i 6.0000 + 2.0000i

**9.1 Matrix Multiplication**  
  
  
Matrix multiplication finds extensive applications in a number of areas of mathematics and solution of equations. Students are usually introduced to it in the solution of $N$ simultaneous linear equations in $N$ unknowns. If the equations are written:

a11x1 + a12x2 + ... + a1NxN = b1

a21x1 + a22x2 + ... + a2NxN = b2

. . . .

aN1x1 + aN2x2 + ... + aNNxN = bN

we find that they express what we want to solve, but they are cumbersome to write out each time we look at another problem. Looking carefully at this and similar problems, we see that there are three "entities" that we will deal with.  
  
The set of unknowns: x1, x2, ..., xN. The set of coefficients multiplying the unknowns:

a11 a12 ... a1N

a21 a22 ... a2N

. . . .

aN1 aN2 ... aNN

The set of given elements: b1, b2, ..., bN  
  
It would be a lot easier to call these x, A and b and to write our set of equations as:

Ax = b.

We can do just that if:  
  
1) We write our "vectors" x and b as matrices with a single column,  
  
2) We define matrix multiplication of A (with elements: aij) by B (with elements: bjk) to give C (with elements cik) such that: if A is an M row by N column matrix and B is an N row by P column matrix, then AB = means that:  
  
giving C , an M row by P column matrix  
  
3) We understand that two matrices are equal if and only if all elements in like positions are equal.  
  
Two very important points about this operation:

a) Arrays A and B can be multiplied together in the matrix sense only if the number of columns in the first array is equal to the number of rows in the second array.  
b) The product: AB is nearly always **not** equal to BA even when both products are defined.

Many of the error messages that you have seen in these notes (and probably in your practice sessions) when you tried to multiply arrays together occur as a result of the violation of the first of these points. Here is a MATLAB session to illustrate the first point:

>> **a=[1 2 3**

**4 5 6];**

>> **a\*[1 2]**

??? Error using ==> \*

Inner matrix dimensions must agree.

>> **a\*[1 2 3]**

??? Error using ==> \*

Inner matrix dimensions must agree.

>> **disp(a\*[1 2]')**

??? Error using ==> \*

Inner matrix dimensions must agree.

>> **disp(a\*[1 2 3]')**

14

32

The matrix a has three columns, therefore it can multiply only matrices with three rows. Row vectors of any length have only one row. A column vector of exactly three rows is the only one that worked. Here is a typical case that shows how matrix multiply depends on the order in which the matrices are listed:

>> **b=[1 2**

**3 4**

**5 6];**

>> **disp(a\*b)**

22 28

49 64

>> **disp(b\*a)**

9 12 15

19 26 33

29 40 51

Matrix multiply gives a direct way to find the vector product of two vectors of the same length or of a vector with itself. For real vectors with N elements, this is defined as:   
  
It is given in MATLAB by:

>> **x=[1 2 3];**

>> **disp(x\*x')**  *<-- Note the transpose using a quote*

14

>> **y=[-1 0 1];**

>> **disp(x\*y')** *<-- Note the transpose using a quote*

2

**9.2 Determinants and Matrix Division**  
  
  
MATLAB gives us two matrix division operations. We will consider only the one that solves the set of equations: AX=B for X, given A and B. This is done with:

>> **X=A\B**

The simplest case is the one where the number of unknowns equals the number of equations. In that case and if the determinant of A is **not** 0 (or not so small that the computer can not find one divided by it), then there should be a unique solution to the set of equations. This is frequently represented by:

A-1B

where A-1 is defined as the inverse matrix to A .The matrix product of the inverse of A with B then gives our desired solution.  
  
Let's see what can be done in MATLAB to demonstrate this solution procedure. Suppose we want to solve:

3x1 + x2 - 2x3 = 1

x1 + 2x2 - x3 = 0

-2x1 - x3 = 5

We set up our matrix a:

>> **a=[3 1 -2**

**1 2 -1**

**-2 0 -1];**

The determinant can be found with *det*:

>> **disp(det(a))**

-11

It is a long way from zero, so we expect a unique solution.

>> **b=[1 0 5]';**

>> **disp(a\b)**

-1.1818

-0.7273

-2.6364

Now let's check that answer (conveniently stored in ans):

>> **disp(a\*ans)**

1.0000

0

5.0000

We could also have carried out the solution using the inverse matrix function: *inv*. Here is how that would have worked:

>> **ai=inv(a);**

>> **disp(ai)**

0.1818 -0.0909 -0.2727

-0.2727 0.6364 -0.0909

-0.3636 0.1818 -0.4545

>> **disp(ai\*b)**

-1.1818

-0.7273

-2.6364

We arrived at the same answer as before. Since matrix inversion takes many more operations than the solution of the same number of equations, we would never follow this route unless we had the same set of equations to solve (with different values for b) many times.  
  
  
Our main restriction on use of B\A is that the number of rows in each matrix must be the same. In most cases where the number of equations is larger than the number of unknowns; i.e. the number of rows of A is larger than the number of columns, there is no solution that will satisfy all the equations. In that case, we say that the system of equations is over specified. MATLAB then finds the best solution it can by making the sum of the squares of the errors in the equations as small as possible. This is said to give the ``least square error" solution. An example is:

4x1 + 2x2 = 3

x1 - x2 = -1

3x1 + 4x2 = 7

MATLAB gives the following for the two unknowns that come closest to satisfying all three equations:

>> **a=[4 2**

**1 -1**

**3 4];**

>> **b=[3 -1 7]';**

>> **disp(a\b)**

0.0378

1.6324

If we multiply the matrix by that result, we get:

>> **disp(a\*[.0378;1.6324])**

3.4160

-1.5946

6.6430

This procedure finds frequent use in generalized curve fitting.  
  
Finally, if there are more unknowns than there are equations (or if the matrix is square but singular) then there are an infinity of solutions. Here, MATLAB gives us one such solution. A singular example is shown next:

>> **a=[ 1 2 3**

**4 3 2**

**5 5 5];**

 >> **disp(det(a))**

0

 >> **disp(a\[1 2 3]')**

Warning: Matrix is close to singular or badly scaled.

Results may be inaccurate. RCOND = 1.110223e-17

0.0250

0.7500

-0.1750

The answer given works well when we multiply it by a:

>> **disp(a\*[.025 .75 -.175]')**

1

2

3

[Continue on to Chapter 10](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 10**

**MATLAB m-Files**

[10.1 Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor905182)  
[10.2 Loops with *for* and *while*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor905563)  
[10.3 Tests with *if* Statements](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor908058)  
[10.4 Global and Local Variables Used in Functions and m-files](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor910424)  
[10.5 The Heading of a MATLAB Function](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat10.html#anchor912374)

**10.1 Discussion**

You will find that you can solve many problems in MATLAB that would require complicated FORTRAN programs with just the programs that are furnished you. Writing programs in MATLAB is quite simple for most other engineering problems. Programs are edited and stored in regular text files, but must have: .m after their names. As soon as a program is defined and saved in: the directory that MATLAB was entered from (or the user's home directory if it was activated from the Applications' menu) or the user's MATLAB directory the program can be used. Please note four important points:

1) The program is edited and saved just like any other file. This may be done in an window or any other way that is convenient for you.  
2) The program must be saved in a file that has .m appended to it.  
3) The program is executed in a MATLAB Command Window.  
4) Only the name (without the .m suffix) is given when you want to execute the program.

One bit of advice about the programs (in MATLAB even more than in other languages): Add comments that describe what the function does and give an example of its use in another comment. This not only helps when you list the function in Unix or a text editor, it also can be a concise way to see what the function does with the *help* command.

You will find it most convenient to store all MATLAB programs in a single directory. If you call this directory *MATLAB* (and it is a subdirectory of your home directory), they will automatically be accessible to you in MATLAB no matter what directory you are in when you enter it.

**10.2 Loops with** *for* **and** while

MATLAB contains several flow control procedures that are similar (but not identical to) those in FORTRAN. These include FOR and WHILE (instead of DO) loops and IF statements. The form of a FOR loop is:

for k=1:n

.........  
(lines of program)  
.........

end

The lines of the program between the *for* and its *end* will be repeated n times. We can demonstrate this in a MATLAB session:



WHILE loops allow you to repeat a set of instructions until some logical condition is no longer satisfied. It's form is:

while logic

.........  
(lines of program)  
.........

end

Here is a simple example:



**10.3 Tests with *if* Statements**

The usual form of an IF statement is:

if(logical)

.........  
(lines of program)  
.........

end

Help applied to *if* shows more about the way it works:

>> **help if**

IF   Conditionally execute statements. The simple form is:

                IF variable, statements END

     The statements are executed if the variable has all

     non-zero elements. The variable is usually the result

     of expr rop expr where rop is ==, <, >, <=,

     >=, or ~=

     For example:

               IF I == J,

                  A(I,J) = 2;

               ELSEIF ABS(I-J) == 1,

                  A(I,J) = -1;

               ELSE

                  A(I,J) = 0;

               END

This is convenient, since it allows one to test directly to see if character strings match. Here is a simple program to make sure we understand the *help* description.



>> **tstif2('abc','abc')**

Same

>> **tstif2('abc','abb')**

Not the same

Unfortunately, the program can give some problems if you try to compare strings of different lengths:

>> **tstif2('aaa','a')**

Same

>> **tstif2('abc','aa')**

??? Error using ==> ==

Matrix dimensions must agree.

Error in ==> /marsh/chbe301/matlab/chbe303/tstif2.m

On line 2  ==> if a==b

To avoid this problem, you can use the function *strcmp* to compare character strings as in the program *tstif3*.



Here are the same examples shown to give problems if you simply use "==":

>> **tstif3('aaa','a')**

Not the same

>> **tstif3('abc','aa')**

Not the same

The logical test may be applied to a value that has been set in a variable or it may consist of a complex determination involving relational operations combined with logical operations. Here are examples of each:



Another set of examples begins with the program shown below. It was edited and saved as a file called root1.m with a standard editor and was then stored in my MATLAB directory so that I could always access it from any active directory. I moved it to: ~chbe301/matlab/chbe303 so others can execute the program by simply typing its name: *root1*. It lists as:



In MATLAB, I can use *root1* with the following results:

>> **clear**

>> **root1**

    3.1415

>> **who**

Your variables are:

 c         i         r         t

The "program" is nothing more than a sequence of MATLAB instructions that you can test in a MATLAB session. The program has the advantage that you can use it over again with different data to solve similar problems. You should note that all the variables defined in the program become variables in your active workspace. This has an advantage in that they could now be used for further calculations. Its disadvantages are:

1) you can alter values of variables of the same name in your workspace,  
2) you add to the number of variables and thus may make it harder to find one that you want with *who*.

A brief explanation of each line in the program *root1*is given next:

|  |  |
| --- | --- |
| **Line(s)** | **Performs** |
| 1-3 | Comments that give the main purpose of the program and how to use it. |
| 4 | Makes a vector called t with elements: 3 3.25 3.5 3.75 4. |
| 5 | Finds the coefficients in a cubic that best fits sin(t) over the interval given in t . |
| 6 | Finds the three roots of the cubic. |
| 7 | Defines a vector with elements: 1 2 3 and starts a loop in which i takes on each of these values. |
| 8 | Tests to see if a root is real (if not it will not be tested further). It really asks if the imaginary part of the root is 0. |
| 9 | Tests to see if a root is in the interval (3,4). It asks if the root (known to be real) is greater than 3 and less than 4. |
| 10 | If the root is in the interval (3,4) and is real it is listed. |
| 11 | Marks the end of the inner test. |
| 12 | Marks the end of the outer test. |
| 13 | Marks the end of the for loop. |

Note that the indentation is not required, but just as in FORTRAN it helps make the program readable.

Obviously the program shown as *root1* is not very useful since it is restricted to solving a single problem. We can make it much more general by changing to the format of a MATLAB function with arguments that give us the desired flexibility. Our first example of a function is shown as *root2*:



Now we can find the same answer as before by:

>> **clear**

>> **disp(root2(3,4,.25,3))**

    3.1415

>> **who**

Your variables are:

   degree    dt        tleft     tright

But we can also find other roots as the example in the comment suggests:

>> **disp(root2(4,7,.01,4))**

    6.2836

The major changes from *root1* to *root2* are listed below:

|  |  |
| --- | --- |
| **Line(s)** | **Performs** |
| 1 | The heading shows that the function returns the value last put in r, and has four arguments with each argument defined. |
| 2-6 | Comments that give the main purpose, the meanings of the arguments and an example use. |
| 7 | Now the interval and number of points may be changed. |
| 13 | Now the program returns the value found for the root rather than just listing it. |

Be sure to note that the variables t, c, rv and r do not appear in our variable list for the workspace after the program is executed.

We are still restricted to finding roots of a single equation: sin(t)=0. Wouldn't it be nice to be able to do this for any function? The command *eval* provides the means for doing exactly that as seen in the function *root3* :



Now as we change the function specified as a character vector in the last argument of *root3* we can look for the root of anything. The only changes that were required to do this were:

1) Insert line 9 to take the name we gave and assuming there is a single argument of the function, evaluate *f(x)* for each value of x.  
2) Change line 10 to replace what was given by sin(t) with the values stored in f.

We will show it used here for our same old problem.

>> **disp(root3(3,4,.25,3,'sin(x)'))**

    3.1415

Note that I changed the required name of the variable sought from t to x as a transition to the usage in the secant functions: *ssec1* and *ssec2*.

Even simple functions like *root2* and *root3* can have problems either in the programs or in the way they are used. A large part of using any programming language is finding out how to find errors and correct them. The next chapter will go into this in some detail.

Now let's look at the *ssec1* and *ssec2* functions. The first lists as:



The comments explain most of what we need to understand and use it. If we execute the example lines, we see:

>> **cs=[1 -2 -4.25 7.5];**

>> **disp(ssec1([2 3],'polyval(c,x)',cs))**

    2.2105

The function has the values as seen in the next lines at the two points: 2 and 3.

>> **disp(polyval(cs,[2 3]))**

   -1.0000    3.7500

A linear fit of the data found would be:

>> **disp(polyfit([2 3],[-1 3.75],1))**

    4.7500  -10.5000

or: 4.75x-10.5

If we set this equal to 0 and solve for x we get: 2.2105 just as *ssec1* found. If we try *ssec1* again, but change the interval to (3, 2.2105) and then repeat as *ssec2* does, we should get closer to our root.

>> **disp(ssec1([3 2.2105],'polyval(c,x)',cs))**

    2.3586

>> **disp(ssec1([2.2105 2.3586],'polyval(c,x)',cs))**

    2.5912

The process shown above is what *ssec2* uses to iterate for a solution. A listing of the function is shown next:



Note some of the very important features in this trial and error program. It will always terminate since it keeps track of how many times it executes *ssec1*. It anticipates that it may not succeed and tells the user if it does not find a root to the desired accuracy. It can accept 2, 3 or 5 arguments. If two are given, they must specify the interval to be searched and the function of x. The third argument can be a vector of constants and the last two are the allowed error and number of iterations. If the vector of constants is not given, it is set to a null vector. If the error and iteration parameters are not supplied, they are set to default values. The program was demonstrated in the examples chapter so we will not repeat that here.

**10.4 Global and Local Variables Used in Functions and m-files**

In MATLAB all variables are assumed to be local in a function unless they are explicitly declared to be global in that function. If those same variables are to be used in another function, or in the command window, they must also be declared there. Thus data (unless declared global in that function) must be passed in the argument list of the function as we did with the polynomial coefficients in the functions *ssec1* and *ssec2*. However all variables defined in the MATLAB command window are readily used in MATLAB m-files which are **not** functions. Here is a short m-file called *tglob1* to show what happens in using such a file:



Here is a session used with *tglob1*:

>> **a1=100.35;**

>> **v1=[2 5 1.2];**

>> **v2=[55 2 1];**

>> **tglob1**

Here are some variables that you have defined.

a1 =

  100.3500

v1 =

    2.0000    5.0000    1.2000

Now give a vector that you have defined. **v2**

v2 =

    55     2     1

If we make the variables global, it has no effect on the use of them in the m-file:

>> **global a1 v1 v3**

Warning: Future versions of MATLAB will require that you declare a variable to be  
         global before you use that variable.

Warning: Future versions of MATLAB will require that you declare a variable to be  
         global before you use that variableevising this example, first clear the variables a1, v1, v2, and v3.

>> **clear a1 v1 v2 v3**

Now declare a1, v1, and v3 as global variables and re-enter the values for a1, v1, and v2.

>> **global a1 v1 v3**

>> **a1=100.35;**

>> **v1=[2 5 1.2];**

>> **v2=[55 2 1];**

Now enter the value for v3 and run tglob1.

>> **v3=[4 4 4];**

>> **tglob1**

Here are some variables that you have defined.

a1 =

  100.3500

v1 =

    2.0000    5.0000    1.2000

Now give a vector that you have defined. **v3**

v2 =

    4     4     4

>> **v2**

v2 =

     4     4     4

Note that the variable v2 has been reset as a result of this execution in the command window workspace.

In a function, the situation is quite different. The m-file *tglob2* is similar to *tglob1*, but its heading shows that we have defined it to be a function:



We will use it in a workspace that has been cleared of all variables:

>> **a1=55.5;**

>> **v1=[1 4 2];**

>> **v2=[8 3 1.1];**

>> **tglob2**

Here are some variables that you have defined.

???  Undefined function or variable 'a1'.

Error in ==> /marsh/chbe301/matlab/chbe303/tglob2.m

On line 4  ==> a1

Obviously, the function knows nothing about the variable a1 and we can anticipate that it will not know about v1 . Making them global will allow the program to associate values with the names, but only if we globalize the variables in the command window **and** the function.  To do this, clear the variables again, declare them as global, and re-enter their values.

>> **clear a1 v1 v2**

>> **global a1 v1**

>> **a1=55.5;**

>> **v1=[1 4 2];**

>> **v2=[8 3 1.1];**

Now run tglob2.

>> **tglob2**

Here are some variables that you have defined.

??? Undefined function or variable 'a1'.

Error in ==> /marsh/chbe301/matlab/chbe303/tglob2.m

On line 4  ==> a1

In this case, the variables were globalized in the command window, but not in the function, so the function still didn't know what a1 and v1 were. If we add a global statement in the function as well, the function will work properly.



>> **tglob2a**

Here are some variables that you have defined.

a1 =

   55.5000

v1 =

     1     4     2

Now give a vector that you have defined.

>> **v2**

v2 =

     [ ]

The whole program runs, but it does not get the values we stored for v2 when it tried to put them in. The vector v2 is still defined in the command window:

>> **v2**

v2 =

    8.0000    3.0000    1.1000

but the function again does not know about those values. All the function knows about are variables defined internally or globally. It does not acknowledge those set in the command window.

If we try to run *tglob2* and give v3 (a new vector) as the variable to be read in, we get:

>> **v3=[4 4 4];**

>> **tglob2a**

Here are some variables that you have defined.

a1 =

   55.5000

v1 =

     1     4     2

Now give a vector that you have defined. **v3**

??? Undefined function or variable 'v3'.

After *tglob2a* prints the statement: ``Now give a vector that you have defined.'' you have to give (numeric) values for the vector elements in order to to continue:

Now give a vector that you have defined. **[1 1 1]**

v2 =

     1     1     1

If we make v3 global, we can get it into our function.  Again, clear v3, since we already referred to it.  Next, declare it as global and reassign its value.   Then run tglob2a

>> **clear v3**

>> **global v3**

>> **v3=[4 4 4];**

>> **tglob2a**

Here are some variables that you have defined.

a1 =

   55.5000

v1 =

     1     4     2

Now give a vector that you have defined. **v3**

??? Undefined function or variable 'v3'.

Of course, that is because we also need to globalize v3 as in the function, which we will now call *tglob2b*.



>> **tglob2b**

Here are some variables that you have defined.

a1 =

   55.5000

v1 =

     1     4     2

Now give a vector that you have defined. **v3**

v2 =

     4     4     4

The alternate to using global variables is the use of extensive lists of arguments in functions.

**10.5 The Heading of a MATLAB Function**

The information given after ``function" in the heading of a function tells:

1) what array(s) (if any) is/are returned,  
2) the name of the function,  
3) the arguments (if any) passed to the function.

All except the first of these is probably familiar to anyone who has written programs in other languages. The way that information is returned in MATLAB is quite different from most languages. A single array may be returned as shown in the *root2* function among others. In the heading on line 4 (following three lines of comments) we see: r=root2 and later (on line 11) a value is assigned to r. This is typical of the way any one array can be returned. Several values may be assigned by the function and returned as seen in the function *tstret*:



We can return any set of arrays we want. If we used *tstret* without specifying that multiple returns are expected, we get only the first one:

>> **disp(tstret)**

     1     2     3

If we ask for more, we get up to the three shown in the heading of the function:

>> **[x,y,z]=tstret;**

>> **disp(x)**

     1     2     3

>> **disp(y)**

     xyz

>> **disp(z)**

     1     2     3

     4     5     6

Note that the variables returned do not have to be of the same shape or even type as seen in the session with *tstret*.

Another feature in MATLAB programs is the ability to use one program with a variable number of arguments. The variable: *nargin* may be tested inside any function program to find out how many arguments were used in the session that called it. This was used for example to make the *ssec2* program use default values for the two arguments erc and nmax if the user did not supply them and to assume that no parameters were needed if none were passed.

[Continue on to Chapter 11](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

**CHAPTER 11**

**Finding Errors and Testing Programs**

[11.1 Discussion](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html#anchor934791)  
[11.2 Diagnostic Messages](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html#anchor934940)  
[11.3 Using echo](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html#anchor935686)  
[11.4 The keyboard Command](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Mat11.html#anchor936537)

**11.1 Discussion**   
  
  
An effective way to debug long programs is to:

1) break the program into small parts and test each part separately,  
2) list intermediate results in all calculations so they can be checked,  
3) make use of the standard routines that do array operations so you reduce the number of looping operations,  
4) test individual lines in a program in the Matlab Command window to make sure you understand the consequences of each,  
5) learn what the error messages mean and where to look for their causes,  
6) make use of any debugging tools that the system may provide.

So far we have spent most of our time on ways to follow the first four of these suggestions. This chapter will focus on the last two points.  
  
Matlab is quite easy to debug since it is an interpretive language and you can test individual lines in it. In spite of this, complicated problems that involve a lot of looping operations may be hard to follow. Finding errors in such programs may take a lot longer than the original writing of the program.  
  
**11.2 Diagnostic Messages**   
  
  
The *MATLAB User's Guide* does not provide a comprehensive list of error messages. Diagnostics from individual operations (such as from matrix divide) are mentioned, but we could use something more complete. We will mainly be concerned with what you would find as a message when you try to execute one of your programs. Messages from direct executions in the command window are usually easy to interpret. First let's look at some of the errors that can arise in the use of a program that has been tested many times. These messages will all come from incorrect arguments given when the program was called. The programs *ssec1* and *ssec2* will be our test programs. Suppose we try:

>> **ssec2(3,'sin(x)')**

Error in ==> /marsh/masc223/matlab/ssec1.m

On line 13 ==> x=xs(2);

Error in ==> /marsh/masc223/matlab/ssec2.m

On line 24 ==> x=ssec1(xs,fx,c);

The error message is not a great help. Even if we list the program, it is not clear why we should have an error. If we think some, it could be that a) xs did not exist, or b) it did exist, but it did not have a second element. What about the possibility that it was a matrix? Would that have caused an error message? Try it:

>> **xs=[1 2**

**3 4];**

>> **disp(xs(2))**

3

Now, we can be pretty sure that it was one of the two listed reasons. How do we find which? You can in this case list the programs and find that xs was in the argument list for both *ssec1* and *ssec2*. In fact it was the first argument and we just gave a scalar for it! Let's try a second stab at using *ssec2*:

>> **ssec2([2 3],sin(x))**

??? Undefined function or variable.

Symbol in question ==> x

Oh well that is easy to understand, we need to define x. We do so and try again:

>> **x=2;**

>> **ssec2([2 3],sin(x))**

??? Index exceeds matrix dimensions.

Error in ==> /marsh/masc223/matlab/ssec2.m

On line 26 ==> if (abs(xs(2)-xs(1))<=erc)

That looks very confusing. We know we made xs, have two elements! What else could be wrong?  
  
At this point we may be tempted to use some sort of trace if we are used to debugging in several other languages. One of our problems is that we do not know how we got to the point where the error occurred. The *echo* command in Matlab can help. If we use *echo* by itself, all commands in script files will be listed before they are executed. If we use *echo* *funcnm*, then the commands in the function called *funcnm* will be listed. Let's try that here.

>> **echo ssec2**

>> **ssec2([2 3],sin(x))**

??? Index exceeds matrix dimensions.

Error in ==> /marsh/masc223/matlab/ssec2.m

On line 26 ==> if (abs(xs(2)-xs(1))<=erc)

Well, that did not help. The error was found before any instructions were executed. Now we will have to rely on the help lines in the function to guide us:

>> **help ssec2**

function x=ssec2(xs,fx,c,erc,nmax)

xs gives an interval to seek a solution to f(x) = 0 in.

fx is a character vector that tells how to find f(x).

c is a vector of parameters in f(x)

erc gives the difference between successive values of x

allowed when convergence is deemed to be achieved.

nmax is the maximum number of trials, before quitting.

If erc and nmax are not given, then the program uses:

erc=.0001 and nmax=10

If only two arguments are given, c is set to null.

The program uses the function ssec1 for each iteration.

Example 1:>> cs=[1 -2 -4.25 7.5];

>> disp(ssec2([2 3],'polyval(c,x)',cs,.0001,10))

Example 2:>> disp(ssec2([3 4],'sin(x)'))

If we compare the examples shown by the help command with what we tried, we can see that we left out the quotes around the function. Note that **nearly all bugs are simple after you find them!**  
  
Here is one more case that is similar to the last one.

>> **ssec2([2 3],'sin(x)',.001,10)**

??? Input argument is undefined.

Symbol in question ==> nmax

Error in ==> /marsh/masc223/matlab/ssec2.m

On line 23 ==> for k=1:nmax

This one is easy to understand, although it seems to be saying that we forgot to give it nmax, we can see that really we omitted the third argument. Matlab simply counts arguments, and when five are required it assumes the last one is the one you left out.  
  
  
**11.3 Using echo**   
  
  
The *echo* command looks like it would be very helpful, but our first try did not produce any results since the program did not execute at all. Here are a pair of very simple functions that will allow us to see what sort of information we could get from *echo* . The function *tstecho1* calls *tstecho2* ;n times:



It also passes two parameters to the function *tstecho2*. Here is *tstecho2*:



If we just execute the first function without *echo* on, we see:

>> **tstecho1(2,3,-4)**

ans =

-0.9167

Now let's turn *echo* on *tstecho1* and try again:

>> **echo tstecho1 on**

>> **tstecho1(2,3,-4)**

function z=tstecho1(a,n,c)

% function tstecho1(a,n,c)

% Testing the echo function

z=0;

for k=1:n

z=z+tstecho2(a,c\*k);

end

ans =

-0.9167

We get a listing of each line of the program that is to be executed. Normally that is not too helpful. We could have gotten the same thing in an edit or shell window. Let's turn that *echo* off and *echo* the function that is called:

>> **echo tstecho1 off**

>> **echo tstecho2 on**

>> **tstecho1(2,3,-4)**

function z=tstecho2(a,c)

% function tstecho2(a,c)

% Testing the echo function

% Called by tstecho1

z=a/c;

function z=tstecho2(a,c)

% function tstecho2(a,c)

% Testing the echo function

% Called by tstecho1

z=a/c;

function z=tstecho2(a,c)

% function tstecho2(a,c)

% Testing the echo function

% Called by tstecho1

z=a/c;

ans =

-0.9167

Now we see the listing for the function that is called, each time it is executed. We still do not get information about the arguments of the function when it is called however. That is usually the crucial information that can help us find ``bugs".  
  
  
**11.4 The *keyboard* Command**   
  
  
Now here is a command that sounds like it could be very helpful in debugging programs. Remember our problem in trying to decide which of two possibilities was causing *ssec2* to bomb. We thought it was either because xs did not exist or it was not a vector. Here was the error message we were trying to interpret:

>> **ssec2(3,'sin(x)')**

Error in ==> /marsh/masc223/matlab/ssec1.m

On line 13 ==> x=xs(2);

Error in ==> /marsh/masc223/matlab/ssec2.m

On line 24 ==> x=ssec1(xs,fx,c);

Suppose we put in a *keyboard* command right before line 13 in *ssec1*:

fs1=eval(fx);

keyboard

x=xs(2);

Then when we execute *ssec2* , we get:

>> **clear**

>> **ssec2(3,'sin(x)',[],.001,10)**

K>> **xs**

xs =

3

K>>Error in ==> /marsh/masc223/matlab/ssec1.m

On line 14 ==> x=xs(2);

Error in ==> /marsh/masc223/matlab/ssec2.m

On line 24 ==> x=ssec1(xs,fx,c);

Note the ``K" before the double prompt when the keyboard command is in effect. When we stop in the program, we can then list any variables such as xs to see what values they have assumed at that point in the calculation. The first time the program stopped, we typed xs to see what had happened to that variable. The second time it stopped, we typed, in lower case, the letters R-E-T-U-R-N and then pressed the RETURN key to signal the end of the list that we wanted to examine \. This may be crucial when you are trying to figure out what went wrong in a subroutine that is four deep in a complicated set of functions.

[Continue on to Chapter 12](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

## ****CHAPTER 12: New Features of Matlab 5.0****

This chapter shows information gleaned from use of *help* on new features of Matlab 5.0 We hope to investigate these topics and commands and add notes about and/or incorporate into our 301 and 303 functions. The commands dealing with structures have been put into [Chapter 13](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html) and are illustrated in a few functions that may be useful in creating a new starting program for our Chemical engineering packages.

### [multidimensional arrays](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#mularr)

* [cat](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#cat)
* [repmat](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#repmat)
* [ndgrid](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#ndgrid)
* [interpn](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#interpn)
* [sum with multidimensional arrays](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#sum)
* [slice a multidimensional array](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#slice)

### [objects, methods, class directories](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#obmecl), etc.

* objects (see helpdesk)
* methods (see helpdesk)
* [class](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#class)
* [switch](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#switch)
* [case](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#case)

### [other functions](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#othfun)

* [isequal](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#isequal)
* [logical](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#logical)
* [relational operators](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#relop)
* [bar3](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#bar3)
* [barh](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#barh)
* [pie3](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/New_Features.html#pie3)

### 1.0 multidimensional arrays

>>help cat

CAT Concatenate arrays. CAT(DIM,A,B) concatenates the arrays A and B along the dimension DIM. CAT(2,A,B) is the same as [A,B]. CAT(1,A,B) is the same as [A;B]. B = CAT(DIM,A1,A2,A3,A4,...) concatenates the input arrays A1, A2, etc. along the dimension DIM.

When used with comma separated list syntax, CAT(DIM,C{:}) or CAT(DIM,C.FIELD) is a convenient way to concatenate a cell or structure array containing numeric matrices into a single matrix.Examples:

a = magic(3); b = pascal(3); c = cat(4,a,b)

produces a 3-by-3-by-1-by-2 result and

s = {a b};

for i=1:length(s),

siz{i} = size(s{i});

end

sizes = cat(1,siz{:})

produces a 2-by-2 array of size vectors.

>>help repmat

REPMAT Replicate and tile an array. B = REPMAT(A,M,N) replicates and tiles the matrix A to produce the M-by-N block matrix B. B = REPMAT(A,[M N]) produces the same thing. B = REPMAT(A,[M N P ...]) tiles the array A to produce a M-by-N-by-P-by-... block array. A can be N-D.

REPMAT(A,M,N) when A is a scalar is commonly used to produce an M-by-N matrix filled with A's value. This is much faster than A\*ONES(M,N).

Example:

repmat(magic(2),2,3)

repmat(NaN,2,3)

See also MESHGRID.

>>help ndgrid

NDGRID Generation of arrays for N-D functions and interpolation. [X1,X2,X3,...] = NDGRID(x1,x2,x3,...) transforms the domain specified by vectors x1,x2,x3, etc. into arrays X1,X2,X3, etc. that can be used for the evaluation of functions of N variables and N-D interpolation. The i-th dimension of the output array Xi are copies of elements of the vector xi.

[X1,X2,...] = NDGRID(x) is the same as [X1,X2,...] = NDGRID(x,x,...).

For example, to evaluate the function x2\*exp(-x1^2-x2^2-x^3) over the range -2 < x1 < 2, -2 < x2 < 2, -2 < x3 < 2,

[x1,x2,x3] = ndgrid(-2:.2:2, -2:.25:2, -2:.16:2);

z = x2 .\* exp(-x1.^2 - x2.^2 - x3.^2);

slice(x2,x1,x3,z,[-1.2 .8 2],2,[-2 -.2])

NDGRID is like MESHGRID except that the order of the first two input arguments are switched (i.e., [X1,X2,X3] = NDGRID(x1,x2,x3) produces the same result as [X2,X1,X3] = MESHGRID(x2,x1,x3)). Because of this, NDGRID is better suited to N-D problems that aren't spatially based, while MESHGRID is better suited to problems in cartesian space (2-D or 3-D).

See also MESHGRID, INTERPN.

>>help interpn

INTERPN N-D interpolation (table lookup).

VI = INTERPN(X1,X2,X3,...,V,Y1,Y2,Y3,...) interpolates to find VI, the values of the underlying N-D function V at the points in arrays Y1,Y2,Y3,etc. For an N-D V, INTERPN should be called with 2\*N+1 arguments. Arrays X1,X2,X3,etc. specify the points at which the data V is given. Out of range values are returned as NaN's. Y1,Y2,Y3,etc. must be arrays of the same size or vectors. Vector arguments that are not the same size are passed through NDGRID to create the Y1,Y2,Y3,etc. arrays. INTERPN works for all N-D arrays with 2 or more dimensions.

VI = INTERPN(V,Y1,Y2,Y3,...) assumes X1=1:SIZE(V,1),X2=1:SIZE(V,2),etc.

VI = INTERPN(V,NTIMES) expands V by interleaving interpolates between every element, working recursively for NTIMES. VI = INTERPN(V) is the same as INTERPN(V,1).

VI = INTERPN(...,'method') specifies alternate methods. The default is linear interpolation. Available methods are:

'linear' - linear interpolation

'cubic' - cubic interpolation

'nearest' - nearest neighbor interpolation

INTERPN requires that X1,X2,X3,etc. be monotonic and plaid (as if they were created using NDGRID). Variable spacing is handled by mapping the given values in X1,X2,X3,etc. and Y1,Y2,Y3,etc. to an equally-spaced domain before interpolating. For faster interpolation when X1,X2,Y3,etc. are known to be equally spaced and monotonic, use the methods '\*linear', '\*cubic', or '\*nearest'.

See also INTERP1, INTERP2, INTERP3, NDGRID.

>>help sum

SUM Sum of elements.

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. For N-D arrays, SUM(X) operates along the first non-singleton dimension.

SUM(X,DIM) sums along the dimension DIM.

Example: If

X = [0 1 2

3 4 5]

then sum(X,1) is [3 5 7] and sum(X,2) is

[ 3

12];

See also PROD, CUMSUM, DIFF.

>>help slice

SLICE Volumetric slice plot.

SLICE(X,Y,Z,V,Sx,Sy,Sz) draws slices along the x,y,z directions at the points in the vectors Sx,Sy,Sz. The arrays X,Y,Z define the coordinates for V and must be monotonic and 3-D plaid (as if produced by MESHGRID). The color at each point will be determined by 3-D interpolation into the volume V. V must be an M-by-N-by-P volume array.

SLICE(X,Y,Z,V,XI,YI,ZI)

draws slices through the volume V along the surface defined by the arrays XI,YI,ZI.

SLICE(V,Sx,Sy,Sz)

or

SLICE(V,XI,YI,ZI)

assumes X=1:N, Y=1:M, Z=1:P.

SLICE(...,'method')

specifies the interpolation method to use. 'method' can be 'linear', 'cubic', or 'nearest'. 'linear' is the default (see INTERP3).

H = SLICE(...)

returns a vector of handles to SURFACE objects.

Example: To visualize the function x\*exp(-x^2-y^2-z^2) over the range -2 < x < 2, -2 < y < 2, -2 < z < 2,

[x,y,z] = meshgrid(-2:.2:2, -2:.25:2, -2:.16:2);

v = x .\* exp(-x.^2 - y.^2 - z.^2);

slice(x,y,z,v,[-1.2 .8 2],2,[-2 -.2])

See also MESHGRID, INTERP3.

### 2.0 objects, methods and class directories, etc.

Class has help on-line. Objects and methods are described in the helpdesk.

>>help class

CLASS Create object or return object class. CLASS(OBJ) returns the class of the object OBJ. Possibilities are:

double -- double precision floating point number array

(this is the traditional MATLAB matrix or array)

sparse -- 2-D real (or complex) sparse matrix

struct -- Structure array

cell -- cell array

char -- Character array

<class\_name> -- Custom object class

Within a constructor method, CLASS(S,'class\_name') creates an object of class 'class\_name' from the structure S. This syntax is only valid in a function named <class\_name>.m in a directory named @<class\_name> (where <class\_name> is the same as the string passed into CLASS). On VMS, the method directory has the name #<class\_name>.

CLASS(S,'class\_name',PARENT1,PARENT2,...) also inherits the methods and fields of the parent objects PARENT1, PARENT2, ...

See also ISA, SUPERIORTO, INFERIORTO, STRUCT.

>>help switch

SWITCH Switch among several cases based on expression. The general form of the SWITCH statement is:

SWITCH switch\_expr

CASE case\_expr,

statement, ..., statement

CASE {case\_expr1, case\_expr2, case\_expr3,...}

statement, ..., statement

...

OTHERWISE,

statement, ..., statement

END

The first CASE where the switch\_expr matches the case\_expr is executed. When the case expression is a cell array (as in the second case above), the case\_expr matches if any of the elements of the cell array match the switch expression. If none of the case expressions match the switch expression then the OTHERWISE case is executed (if it exists). Only one CASE is executed and execution resumes with the statement after the END.

The switch\_expr can be a scalar or a string. A scalar switch\_expr matches a case\_expr if switch\_expr==case\_expr. A string switch\_expr matches a case\_expr if strcmp(switch\_expr,case\_expr) returns 1 (true).

Example (assuming METHOD exists as a string variable):

switch lower(METHOD)

case {'linear','bilinear'}, disp('Method is linear')

case 'cubic', disp('Method is cubic')

case 'nearest', disp('Method is nearest')

otherwise, disp('Unknown method.')

end

See also CASE, OTHERWISE, IF, WHILE, FOR, END.

>>help case

CASE SWITCH statement case.

CASE is part of the SWITCH statement syntax, whose general form is:

SWITCH switch\_expr

CASE case\_expr,

statement, ..., statement

CASE {case\_expr1, case\_expr2, case\_expr3,...}

statement, ..., statement

...

OTHERWISE,

statement, ..., statement

END

See SWITCH for more details.

### 3.0 Other functions

>>help isequal

ISEQUAL True if arrays are identical. ISEQUAL(A,B) is 1 if the two arrays are the same size with the same contents and 0 otherwise.

ISEQUAL(A,B,C,...) is 1 if all the input arguments are equal.

See also EQ.

>>help logical

LOGICAL Convert numeric values to logical.

LOGICAL(X) returns an array that can be used for logical indexing or logical tests. Logical arrays are also created by the relational operators (==,<,>,~, etc.) and functions like ANY, ALL, ISNAN, ISINF, and ISFINITE.

A(B), where B is a logical array, returns the values of A at the indices where the real part of B is nonzero (B must be the same size as A).

Most arithmetic operations remove the logicalness from an array. Seldom necessary, the easiest method is add zero (i.e. A+0).

See also ISLOGICAL, RELOP, OPS.

>>help islogical

ISLOGICAL True for logical array.

ISLOGICAL(X) returns 1 if X is a logical array and 0 otherwise.

Logical arrays must be used to perform logical 0-1 indexing.

See also LOGICAL.

>>help relop

Relational operators.

< > Relational operators.

The six relational operators are <, <=, >, >=, ==, and ~=. A < B does element by element comparisons between A and B and returns 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. A and B must have the same dimensions unless one is a scalar. A scalar can be compared with anything.

& Logical AND.

A & B is a matrix whose elements are 1's where both A and B have non-zero elements, and 0's where either has a zero element. A and B must have the same dimensions unless one is a scalar. A scalar can be operate with anything.

| Logical OR.

A | B is a matrix whose elements are 1's where either A or B has a non-zero element, and 0's where both have zero elements. A and B must have the same dimensions unless one is a scalar. A scalar can be operate with anything.

~ Logical complement (NOT).

~A is a matrix whose elements are 1's where A has zero elements, and 0's where A has non-zero elements.

xor Exclusive OR.

xor(A,B) is 1 where either A or B, but not both, is non-zero.

>>help bar3

BAR3 3-D bar graph.

BAR3(Y,Z) draws the columns of the M-by-N matrix Z as vertical 3-D bars. The vector Y must be monotonically increasing or decreasing.

BAR3(Z)

uses the default value of Y=1:M. For vector inputs,

BAR3(Y,Z)

or

BAR3(Z)

draws LENGTH(Z) bars. The colors are set by the colormap.

BAR3(Y,Z,WIDTH)

or

BAR3(Z,WIDTH)

specifies the width of the bars. Values of WIDTH > 1, produce overlapped bars. The default value is WIDTH=0.8

BAR3(...,'detached')

produces the default detached bar chart.

BAR3(...,'grouped')

produces a grouped bar chart.

BAR3(...,'stacked')

produces a stacked bar chart.

BAR3(...,LINESPEC)

uses the line color specified (one of 'rgbymckw').

H = BAR3(...) returns a vector of surface handles.

Example:

subplot(1,2,1), bar3(peaks(5))

subplot(1,2,2), bar3(rand(5),'stacked')

See also BAR, BARH, and BAR3H.

>>help barh

BARH Horizontal bar graph.

BARH(X,Y) draws the columns of the M-by-N matrix Y as M groups of N horizontal bars. The vector X must be monotonically increasing or decreasing.

BARH(Y)

uses the default value of X=1:M. For vector inputs,

BARH(X,Y)

or

BARH(Y)

draws LENGTH(Y) bars. The colors are set by the colormap.

BARH(X,Y,WIDTH)

or

BARH(Y,WIDTH)

specifies the width of the bars. Values of WIDTH > 1, produce overlapped bars. The default value is WIDTH=0.8.

BARH(...,'grouped')

produces the default vertical grouped bar chart.

BARH(...,'stacked')

produces a vertical stacked bar chart.

BARH(...,LINESPEC)

uses the line color specified (one of 'rgbymckw').

H = BARH(...)

returns a vector of patch handles.

Examples:

subplot(3,1,1), barh(rand(10,5),'stacked'), colormap(cool)

subplot(3,1,2), barh(0:.25:1,rand(5),1)

subplot(3,1,3), barh(rand(2,3),.75,'grouped')

See also PLOT, BAR, BAR3H.

>>help pie3

PIE3 3-D pie chart.

PIE3(X) draws a 3-D pie plot of the data in the vector X. The values in X are normalized via X/SUM(X) to determine the area of each slice of pie. If SUM(X) <= 1.0, the values in X directly specify the area of the pie slices. Only a partial pie will be drawn if SUM(X) < 1.

H = PIE3(X,EXPLODE)

is used to specify slices that should be pulled out from the pie. The vector EXPLODE must be the same size as X. The slices where EXPLODE is non-zero will be pulled out.

H = PIE3(...)

returns a vector containing patch, surface, and text handles.

[Continue on to Chapter 13](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html)

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)

## ****CHAPTER 13: Structures, Cells and Other Arays****

Data used in the Chemical Engineering modules has been stored in matrices. Since matrices must be rectangular and can hold **either** numeric **or** character data, this placed limits on the way the data was stored during use in the programs. It also made it rather difficult to extract data from the property data files and put the data in the arrays where they would be used. In Matlab versions through 4.2 this was the only choice for storage of data. Now Matlab 5.1 offers several choices that have definite advantages since they can store mixed (both numeric and character) data, can store data in higher dimensional arrays than matrices and can be used to avoid having to spend as much effort in storing and extracting the data from files. This chapter will show the basic ideas that were used in utilizing structures in the *start301* program developed by Mike Cohen during the Summer of 1998.

First let's a brief review of the way chemical property data is stored in a chemical engineering session. The main types of arrays are:

### Table 13.1 Types of Arrays Used in ceng301 Modular Simulation

|  |  |  |  |
| --- | --- | --- | --- |
| *Type of Array* | *Example* | *Stores* | *Might hold* |
| numeric scalar | nc | Number of Compounds | 4 |
| string matrix | cnms | Compound Names | ammonia  nitrogen  hydrogen  argon |
| numeric vector | mw | molecular weights | 17.0300  28.0130  2.0160  39.9480 |
| numeric matrix | Aabc | Antoine Constants | 15.494 13.45 12.78  13.915 2363.20 658.22  232.320 832.78 -22.62  -2.854 8.08 2.36 |

For a complete list of compound properties stored by use of the start program see the section on this program in the [Ceng 301 Notes](http://www.owlnet.rice.edu/%7Eceng301/31.html).

Cells could be used to store all four types of data in the same array. We will give an example where data for the same four compounds in Table 13.1 were put in a file with *start301a* and loaded with *start301b* (these are the programs that *start301* replaced. We see the familiar listing of compounds and reactions at the end of this process:

Here are your compounds' formulae and names:

No. Formula Name

1 NH3 ammonia

2 N2 nitrogen

3 H2 hydrogen

4 Ar argon

Here are your reactions:

1 N2 + 3 H2 --> 2 NH3

We can consturct an empty cell array as indicated in the help notes about the command by that name:

»help cell

CELL Create cell array.

CELL(N) is an N-by-N cell arrray of empty matrices.

CELL(M,N) or CELL([M,N]) is an M-by-N cell array of empty

matrices.

CELL(M,N,P,...) or CELL([M N P ...]) is an M-by-N-by-P-by-...

cell array of empty matrices.

CELL(SIZE(A)) is a cell array the same size as A containing

all empty matrices.

Thus we can create a cell to hold the data shown in Table 13.1 by:

»celldat=cell(4,1) <-- Making a empty, cell, column vector with place

celldat = for four "things".

[]

[]

[]

[]

»celldat{1,1}=nc <-- Putting nc in the first place.

**Note use of "{" and "}"**

celldat =

[4]

[]

[]

[]

»celldat{2,1}=cnms; <-- Adding cnms in the second place.

»celldat{3,1}=mw; <-- mw is put in the third place

»celldat{4,1}=Aabc; <-- The matrix Aabc is put in the fourth place.

»celldat <-- When we list the cell, we see the data we placed

celldat = only in the scalar: nc

[ 4] <-- nc is 4

[4x9 char ] <-- cnms is a 4 by 9 character matrix.

[4x1 double] <-- mw is a numeric column vector with 4 elements

[4x3 double] <-- Aabc is a 4 by 3 numeric matrix

The use of braces ( **"{" and "}"**) to access the contents of a cell must be remembered. If you use parentheses instead you will find that it can lead to confusion:

»celldat(1,1) <-- This produces a cell with the number 4 in it

ans =

[4]

»celldat{1,1} <-- This gives us the number 4

ans =

4

»celldat(4,1) <-- A cell having Aabc in it

ans =

[4x3 double]

»celldat{4,1} <-- The numeric values of the coefficients

ans =

1.0e+03 \*

0.0155 2.3632 -0.0226

0.0134 0.6582 -0.0029

0.0128 0.2323 0.0081

0.0139 0.8328 0.0024

The same data may also be placed in a structure. Here is information to get started:

»help struct

STRUCT Create or convert to structure array.

S = STRUCT('field1',VALUES1,'field2',VALUES2,...)

creates a structure array with the specified fields and values. The value arrays VALUES1, VALUES2, etc. must be cell arrays of the same size, scalar cells or single values. Corresponding elements of the value arrays are placed into corresponding structure array elements.

The size of the resulting structure is the same size as the value cell arrays or 1-by-1 if none of the values is a cell.

STRUCT(OBJ)

converts the object OBJ into its equivalent structure. The class information is lost.

Example:

s = struct('type',{'big','little'},'color','red','x',{3 4})

See also CLASS, CELL, GETFIELD, SETFIELD, RMFIELD, FIELDNAMES.

From this help, we can make a structure with the same data we put in *celldat*:

»strucdat=struct('nc',nc,'cnms',cnms,'mw',mw,'Aabc',Aabc)

strucdat =

nc: 4

cnms: [4x9 char ]

mw: [4x1 double]

Aabc: [4x3 double]

Now if we want to access this data that we stored, we can use:

»strucdat.nc

ans =

4

»strucdat.mw

ans =

17.0300

28.0130

2.0160

39.9480

Matlab ***structures*** seem to offer the most efficient way to store and then retreive data. The main advantages of structures over cells are:

* The sturcture fields identify the data,
* Data stored in the structure can be readily retrived without use of braces
* Listing the contents of a structure is much simpler than for a cell.

Two functions and a starting data base with part of the data in our data files were developed to explore the possibility of replacing our starting programs *start301a* and *start301b* with a single Matlab program. Help applied to the commands: [*fieldnames*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html#fieldnames)*,* [*getfield*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html#getfield)and[*setfield*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html#setfield)was used to understand ways that this might be done. The example programs: [*mkstruct*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html#mkstruct) and [*mkvars*](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html#mkvars) illustrate some of the advantages of using structures.

»help fieldnames

FIELDNAMES Get structure field names.

NAMES = FIELDNAMES(S)

returns the structure field names associated with the structure S as a cell array of strings.

See also GETFIELD, SETFIELD.

You must be careful in using *fieldnames* since it returns a [cell array](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/Structures.html#cell).

>> fnames=fieldnames(strucdat)

fnames =

'nc'

'cnms'

'mw'

'Aabc'

The *class* command tells us that:

>> class(fnames)

ans =

cell

One element of *fnames* may look like a string, but it is not.

>> class(fnames)

ans =

cell

>> fn1=fnames(1)

fn1 =

'nc'

>> 'nc'==fn1

??? Function '==' not defined for variables of class 'cell'.

>> class(fn1)

ans =

cell

If we want the contents of a cell, we need to use braces:

>> fn1=fnames{1}

fn1 =

nc

>> 'nc'==fn1

ans =

1 1

>> class(fn1)

ans =

char

»help getfield

GETFIELD Get structure field contents.

F = GETFIELD(S,'field')

returns the contents of the specified field. This is equivalent to the syntax F = S.field.

S must be a 1-by-1 structure.

F = GETFIELD(S,{i,j},'field',{k})

is equivalent to the syntax

F = S(i,j).field(k).

In other words,

F = GETFIELD(S,sub1,sub2,...)

returns the contents of the structure S using the subscripts or field references specified in sub1,sub2,etc. Each set of subscripts in parentheses must be enclosed in a cell array and passed to GETFIELD as a separate input. Field references are passed as strings.

See also SETFIELD, FIELDNAMES.

»help setfield

SETFIELD Set structure field contents.

S = SETFIELD(S,'field',V)

sets the contents of the specified field to the value V. This is equivalent to the syntax S.field = V. S must be a 1-by-1 structure. The changed structure is returned.

S = SETFIELD(S,{i,j},'field',{k},V)

is equivalent to the syntax

S(i,j).field(k) = V;

In other words,

S = SETFIELD(S,sub1,sub2,...,V)

sets the contents of the structure S to V using the subscripts or field references specified in sub1,sub2,etc. Each set of subscripts in parentheses must be enclosed in a cell array and passed to SETFIELD as a separate input. Field references are passed as strings.

See also GETFIELD, FIELDNAMES.

The function *mkstruct* allows us to add data for one compound to an existing structure. The fields in the existing structure are identical to the names of data variables set by *start301a* and *start301b*. Here is a listing of the program.



The function *mkvars* extracts all the data in a structure like the one created with *mkstruct* and puts the data in variables of the same names as the fields in the structure. All the compounds are added to these variables. Here is a listing of *mkvars.*

**

Note that the *mkvars* function returns a string with the command that makes all the variables global. Thus if this returned string is *eval*ed, we will have a session ready to be used in the 301 modules.

The function *mkstruct* was used to produce a structure holding twelve types of data for twenty compounds. This procedure produced a structure called *stdat*. It was saved as the file: */home/ceng301/datbas/struct1.mat*. Here is an example of using the *mkstruct* function to add the compound ethanol to the structure. First we use *start301a* and *start301b:*

»start301b

Copyright 1996 Rice University

All rights reserved

If you have not run the FORTRAN program start301a to

produce a data file, do so now or you can just set

the names without any data. A blank reply for

the file name will let you just set the names.

Give the name of your data file:waterethanol

How many streams will there be?1

Here are your compounds' formulae and names:

No. Formula Name

1 H2O water

2 C2H5OH ethanol

Next we load the file that had data for our twenty compounds:

»load struct1

»who

Your variables are:

Aabc StStdh cpl form icpv stdat

AcF StdhkJ cps hcpl mw

Gibb TbpK cpv hcps nc

LJones Tdeg critP hcpv ne

LhlvkJ TmpK critT icpl ns

LhslkJ cnms critZ icps nst

The structurestdat has the form shown by:

»stdat

stdat =

20x1 struct array with fields:

cnms

form

mw

Aabc

LhlvkJ

StStdh

StdhkJ

TbpK

cpl

cpv

hcpl

hcpv

Here is the data for the first compound in the structure:

»stdat(1)

ans =

cnms: ' benzene '

form: ' C6H6 '

mw: 78.1130

Aabc: [14.1600 2.9488e+03 -44.5630]

LhlvkJ: 30.7600

StStdh: 2

StdhkJ: 82.9300

TbpK: 353.2600

cpl: [-7.2732 0.7705 -0.0016 1.8979e-06]

cpv: [18.5800 -0.0117 0.0013 -2.0790e-06 1.0500e-09]

hcpl: [4.7448e-10 -5.4937e-07 3.8527e-04 -0.0073 28.4482]

hcpv: [2.1000e-13 -5.1975e-10 4.2500e-07 -5.8700e-06 0.0186 70.2605]

Let's add the ethanol data to this data structure:

»stdat2=mkstruct(stdat',2) *<-- Note transpose of stdat*

stdat2 =

1x21 struct array with fields:

cnms

form

mw

Aabc

LhlvkJ

StStdh

StdhkJ

TbpK

cpl

cpv

hcpl

hcpv

»stdat2(21)<-- Seeing what was added for ethanol

ans =

cnms: ' ethanol'

form: ' C2H5OH'

mw: 46.0690

Aabc: [16.1950 3.4235e+03 -55.7150]

LhlvkJ: 38.5800

StStdh: 2

StdhkJ: -234.8000

TbpK: 351.4800

cpl: [-325.1300 4.1379 -0.0140 1.7035e-05]

cpv: [17.6900 0.1495 8.9480e-05 -1.9730e-07 8.3170e-11]

hcpl: [4.2588e-09 -4.6767e-06 0.0021 -0.3251 -272.9021]

hcpv: [1.6634e-14 -4.9325e-11 2.9827e-08 7.4750e-05 0.0177 -247.1590]

Let's save this new structure and reload it into a clear workspace, then use *mkvars* create a new set of variables for use in the 301 modules.

»save structnew stdat2

»clear

»load structnew

»who

Your variables are:

stdat2

Now use *mkvars*:

»mkvars(stdat2)

ans =

global cnms form mw Aabc LhlvkJ StStdh StdhkJ TbpK cpl cpv hcpl hcpv

»eval(ans) <-- This makes the variables global (you could also do

»who do it by: eval(mkvars(stdat2)).

Your variables are:

Aabc StdhkJ cnms form mw

LhlvkJ TbpK cpl hcpl stdat2

StStdh ans cpv hcpv

»cnms <-- We can now look at all the compounds in struct1 plus

ethanol that we added in the session.

cnms =

benzene

n-butane

n-butanol

n-butyric acid

n-heptane

n-hexadecane

n-hexane

n-octane

n-pentane

n-propanol

neon

nicotinic acid

nitric acid

nitric oxide

nitrogen

nitrogen dioxide

nitromethane

nitrous oxide

toluene

water

ethanol

»Aabc <-- checking to see that one of the numeric arrays is also

set correctly.

Aabc =

1.0e+03 \*

0.0142 2.9488 -0.0446

0.0140 2.2924 -0.0279

0.0147 2.9030 -0.1029

0.0158 4.0961 -0.0718

0.0139 2.9327 -0.0556

0.0142 4.2053 -0.1192

0.0141 2.8254 -0.0427

0.0142 3.3042 -0.0552

0.0140 2.5546 -0.0363

0.0152 3.0083 -0.0865

NaN NaN NaN

NaN NaN NaN

NaN NaN NaN

0.0169 1.3191 -0.0141

0.0134 0.6582 -0.0029

NaN NaN NaN

NaN NaN NaN

NaN NaN NaN

0.0143 3.2424 -0.0472

0.0165 3.9854 -0.0390

0.0162 3.4235 -0.0557

»

[Return to Table of Contents](http://www.owlnet.rice.edu/%7Eceng303/manuals/matlab/MatCont.html)