

An Introduction to Matlab

Phil Spector

Statistical Computing Facility
Department of Statistics
University of California, Berkeley

1

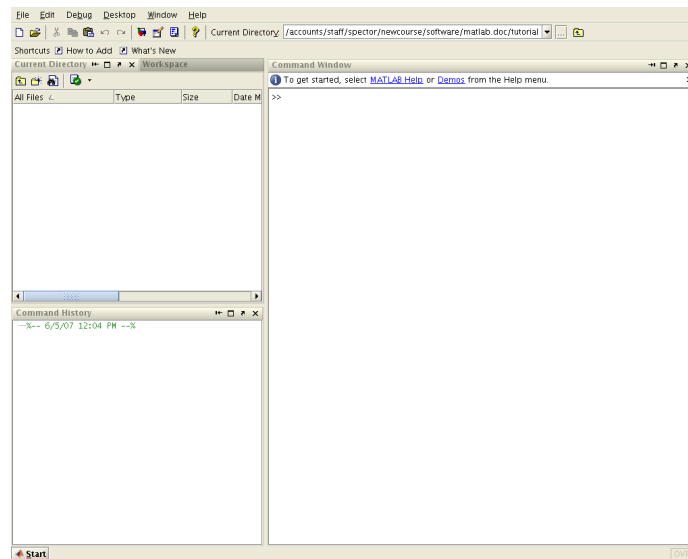
History of Matlab

- Developed by Cleve Moler in the late 1970s
- Designed to give easy access to EISPACK and LINPACK
- Rewritten in C in 1983; The Mathworks formed in 1984
- Always recognized as one of the leading programs for linear algebra
- Very popular among engineers and image analysts, among others
- Version 5, released in the late 1990s, added many new features

2

Accessing Matlab

By default, matlab will open a console with a file browser and command history window along with a window to input commands.



To start matlab without the console, use the `-nojvm` flag.

3

Accessing Matlab (cont'd)

For long running jobs, matlab can be run in the background, allowing you to start a job and log out with the job still running.

Under `csh`, the format is

```
matlab < matlab.in >& matlab.out &
```

Under `bash`, the format is

```
matlab < matlab.in > matlab.out 2>&1 &
```

Redirecting the error messages is critical, since they can't be retrieved otherwise.

4

Some Basics

- The percent sign (%) is the comment character.
- Names in matlab can be as long as 63 characters.
- Use single quotes, not double quotes.
- Use three periods (...) at the end of lines to allow continuation.
- Emacs-style editing is always available at the command line
- Output is displayed unless the input line is terminated by a semi-colon (;)
- If you forget to store an object in a variable, it is available as the variable `ans` until the next statement is executed.
- Help is available through the `help` (text), `helpwin` (separate window), `doc` (browser) or `lookfor` (apropos) commands.

Function vs. Command format

Matlab commands can be called as functions (with parentheses and comma-separated argument lists) or commands (separate tokens separated by spaces). The primary difference is that the function form uses the values of its arguments, while the command form treats its arguments as literal strings. Additionally, only the function form allows objects to be returned.

For example, to use the `load` command to load a saved matlab data file called `data.mat`, you could use command syntax:

```
load data.mat
```

or function syntax:

```
load('data.mat')
```

Basic Information Functions

- `disp` - displays the value of a variable
- `who` - displays currently active variables
- `whos` - displays types and sizes
- `size` - displays the size of a variable
- `class` - displays storage class of a variable
- `type` - displays contents of files or commands
- `clear` - removes variables from the workspace (**Edit**→**Clear Workspace**)
- `diary` - saves a transcript of your session

Data Entry: Overview

1. Entering a small data set – Values can be entered, separated by spaces, and surrounded by square brackets (`[]`). To separate rows, use a semi-colon (`;`), or enter each row on a separate line.
2. The import wizard (**File**→**Import Data** in the console) guides you through the data entry process.
3. The `load` command can be used to read saved matlab files, as well as numeric data stored in ASCII files.
4. The `dlmread` and `csvread` commands can read data delimited by a character other than a space.
5. The `fopen` function, in conjunction with the `textscan` function, can flexibly read character and/or numeric data.

Reading Numeric Matrices

The `load` command, in conjunction with the `-ascii` flag can be used to read files containing entirely numeric data. Suppose the file `num.txt` contains 20 lines, each with 10 space-delimited values.

Then the command

```
load -ascii num.txt
```

will create a 20×10 matrix called `num`.

The function form allows to name the result:

```
mymatrix = load('num.txt', '-ascii')
```

Saving Objects

The `save` command allows you to store some or all of your variables in a MAT-file. With no arguments, `save` stores all current variables in a file called `matlab.mat` in the current directory.

An alternative filename can be used as a single argument.

To save only certain variables, list them after the rest of the `save` command.

The `-ascii` flag can be used to save data in ASCII format.

The `load` command can be used to restore a MAT-file.

The `whos -file` command can be used to see the contents of a MAT-file.

```
>> save x.mat f mat vec  
>> save('x.mat', 'f', 'mat', 'vec')
```

Some Basic Matrix Functions

- `zeros` - create a matrix of all zeroes
- `ones` - create a matrix of all ones
- `diag` - create diagonal matrix or extract diagonal elements
- `reshape` - change dimensions of matrix or vector
- `rand` - create matrix of uniform random numbers
- `cat` - concatenate matrices
- `vertcat` - concatenate matrices by rows
- `blkdiag` - construct block diagonal matrix
- `eye` - create identity matrix

11

Matrix and Vector Indexing

- Parentheses are used for subscripts
- Indexes can be scalars, vectors, or matrices.
- To refer to an entire row or column of a matrix, replace the other index with a colon (:).
- Indexing a matrix with a single value indexes its columnwise representation – using `:` as a subscript converts it to a vector.
- The reserved keyword `end` can be used to refer to the last element in an object

```
>> vec = [1 3 5 7];
>> vec(end + 1) = 4
vec =
     1     3     5     7     4
>> mat = [7 2 3; 4 5 9; 8 12 1];
>> disp(mat(2,:))
     4     5     9
```

12

Matrix and Vector Indexing (cont'd)

Single rows and columns can be assigned without loops.

```
>> mat(2,:) = [10 11 12];
>> mat(:,2) = [10 11 12];
>> disp(mat)
     7     10     3
    10     11    12
     8     12     1
```

To replace more than one row or column, the replacement must conform to the target:

```
>> mat(1:2,:) = [100 101 102; 200 201 202];
>> mat(:,1:2) = [100 101 102; 200 201 202];
??? Subscripted assignment dimension mismatch.
>> mat(:,1:2) = [100 101 102; 200 201 202]';
>> disp(mat)
mat =
    100    200    102
    101    201    202
    102    202     1
```

Matrix and Vector Indexing (cont'd)

To specify contiguous indices, the colon operator (:) can be used. The two forms of the operator are `start:finish` and `start:increment:finish`.

```
>> mat = [1 7;2 3;4 5;9 12];
>> mat(2:3,:)
ans =
     2     3
     4     5
```

The colon operator can also be used to create vectors containing sequences – the `linspace` function provides more control.

For non-contiguous indices, ordinary vectors can be used:

```
>> mat([1 3 4],:)
ans =
     1     7
     4     5
     9    12
```

Matrix and Vector Indexing (cont'd)

Using a vector as an index extracts multiple elements:

```
>> x = [1 2 3 ;10 20 30;100 200 300];
>> x([1 3],[3 2])
ans =
     3     2
    300   200
```

Using a matrix as an index creates a similarly-dimensioned matrix with the corresponding elements of the indexed object:

```
>> x([1 3;3 2])
ans =
     1   100
    100    10
```

Note that indexing is performed as if the matrix was a vector.

Matrix and Vector Indexing (cont'd)

The `sub2ind` function can convert multiple indexes to single indexes, and is useful to extract scattered elements from a matrix. The first argument to `sub2ind` is a vector containing the size of the matrix – the remaining arguments are interpreted as indices:

```
>> x = [1 2 3 ;10 20 30;100 200 300];
>> indices = sub2ind(size(x),[2 3 1],[1 3 2])
ans =
     2     9     4
>> x(indices)
ans =
    10   300     2
```

The `ind2sub` function reverses the process:

```
>> [i,j] = ind2sub(size(x),[2 9 4])
i =
     2     3     1
j =
     1     3     2
```


Removing Elements of Matrices and Vectors

To remove a row or column of a matrix, set its value to an empty vector:

```
>> x = [7 2 4;9 8 12;4 7 8;12 1 4];
>> x(:,2) = []
x =
     7     4
     9    12
     4     8
    12     4
```

The same technique can be used to remove single vector elements:

```
>> y = [13 7 12 9 15 8];
>> y(3) = []
y =
    13     7     9    15     8
```

Logical Subscripts

Logical expressions on vectors or matrices result in logical variables of the same size and shape. These expressions can be used to extract elements which meet a particular condition.

```
>> x = rand(10,1) * 10;
>> x(x>5)
ans =
     7.0605     8.2346     6.9483     9.5022
>> length(x(x>5))
ans =
     4
```

Logical subscripts can be used for assignment:

```
>> x = 1:10;
>> x(mod(x,2) == 1) = 0;
>> x
x =
     0     2     0     4     0     6     0     8     0    10
```

Working with Logical Variables

Logical variables can be used for counting:

```
>> x = [7 12 9 14 8 3 2];  
>> sum(x > 10)  
ans =  
     2
```

The `find` command returns indices matching a logical expression:

```
>> find(x > 10)  
ans =  
     2     4
```

To retrieve row and column indices, use

```
[i,j] = find(mymat > 10);
```

The values of the found elements can be retrieved using the form:

```
[i,j,v] = find(mymat > 10);
```

Note that with logical expressions, `v` will simply be a vector of 1s.

Missing Values

In matlab, missing values are represented as `NaN` (not a number).

You can assign the value directly, but you must use the `isnan` function to test for their presence. (The `isfinite` and `isinf` functions may also be useful.)

Missing values will be propagated through computations.

Logical subscripts can be used to remove missing values:

```
>> vals = [7 12 NaN 19 42 3];  
>> vals(~isnan(vals))  
ans =  
     7     12     19     42     3
```

There are also specialized functions which ignore missing values: `nanmax`, `nanmin`, `nansum`, `nanmean`, etc.

Cell Arrays

Since matrices can only hold one type of data (numeric or character), cell arrays were introduced in Matlab 5 to allow a single object to contain different modes of data.

Cell arrays are indexed like matrices, but their elements can be scalars, matrices or other cell arrays.

Cell arrays are created by surrounding the input elements with curly braces ({}). Curly braces are also used for indexing.

```
>> xc = {[7 5 2] ['one' 'two'] [8 5 7;3 2 5]};
>> xc{3}
ans =
     8     5     7
     3     2     5
```

The `cell` function can be used to create an empty cell array.

Cell Arrays (cont'd)

Use curly braces for extracting single elements, and parentheses for extracting multiple elements:

```
>> mycell = {[7 9 12;3 7 2;4 9 8] ['one','two','three'] 14};
>> mycell{2}
ans =
onetwothree
>> mycell([1 3])
ans =
 [3x3 double] [14]
```

If the elements of a cell array are conformable, you can create a matrix using `cell2mat`

```
>> cx = {[7 9 12]; {'one' 'two' 'three'}; [8 14 21] ;[9 17 8]};
>> cell2mat(cx([1 3 4]))
ans =
     7     9    12
     8    14    21
     9    17     8
```

Strings

In ordinary matrices, matlab stores strings as an array of numeric codes. If all strings are the same length, this may be obscured:

```
>> strs = ['one';'two']
strs =
one
two
```

What if we look at the transpose?

```
>> strs'
ans =
ot
nw
eo
```

Strings are generally best stored in cell arrays:

```
>> strs = {'one' 'two'}
strs =
    'one'    'two'
```

Some String Functions

- `strcmp`, `strcmpi`, `strncmp` - compares two strings
- `strcat` - concatenate strings by row
- `strvcat` - concatenate strings by column
- `sprintf` - write formatted text to string
- `blanks` - create string of blanks
- `upper`, `lower` - convert case of strings
- `deblank`, `strtrim` - remove blanks from strings
- `regexp`, `regexpi`, `regexprep` - regular expression processing

Reading mixed data

The `textscan` function allows mixed data to be read from a file, returning the results in a cell array. First, the `fopen` function is used to open the file; the returned file handle is then passed to `textscan` along with a format.

```
mixed.dat:          >> f = fopen('mixed.dat','r');
3 7 John 15         >> x = textscan(f,'%n %n %s %n');
9 12 Harry 6       >> disp(x)
5 9 Willie 12      [3x1 double]    [3x1 double]
                   {3x1 cell}    [3x1 double]
```

The columns of the result are stored in `x{1}`, `x{2}`, `x{3}`, and `x{4}`. The numeric columns could be combined into a matrix using

```
>> xmat = [x{1} x{2} x{4}]
```

More about textscan

By specifying a numeric argument to `textscan`, you can control the number of elements read with a particular format, then continue using `textscan` with a different format. For example, to read a data with variable names stored as headers:

```
head.dat:          >> f = fopen('head.dat','r');
one,two,three     >> hdrs = textscan(f,'%s',3,'delimiter',',','');
12,18,24          >> names = hdrs{1}'
7,19.3,22.1       ans =
34,22.,19.4       'one'    'two'    'three'
12,81.2,15.9     >> x = cell2mat(...
                  textscan(f,'%n %n %n','delimiter',',',''))
x =
    12.0000    18.0000    24.0000
     7.0000    19.3000    22.1000
    34.0000    22.0000    19.4000
    12.0000    81.2000    15.9000
```

User Input

The `input` function will display a prompt for the user, and then return the result of evaluating their input as a matlab expression:

```
>> x = input('Give a vector of numbers> ');
Give a vector of numbers> [7 12 19]
>> x
x =
     7     12     19
```

To retrieve the user's input as an ordinary character string, pass an extra argument of `'s'` to `input`:

```
>> fname = input('Name of file? ', 's');
Name of file? myfile.out
>> f = fopen(fname, 'r');
```

27

Operators

Op	Function	Op	Function
+	Addition	<	Less than
-	Subtraction	>	Greater than
*	Multiplication	<=	Less than or equal
.*	Element-wise multiplication	>=	Greater than or equal
/	Division	==	Equality
./	Element-wise division	~=	Inequality
\	Left division	&	Element-wise and
.\	Element-wise left division		Element-wise or
^	Matrix power	~	Element-wise not
.^	Element-wise power	&&	Scalar and
'	Matrix transpose		Scalar or

The `any` and `all` functions are also useful for logical tests.

28

A Note about Multiplication and Division

The usual multiplication and division operators in matlab implement matrix operations: `*` is used for matrix (or scalar) multiplication and `/` is used for matrix “division”. (Essentially, the matlab expression `A / B` is roughly equivalent to BA^{-1} , while `A \ B` is roughly equivalent to $A^{-1}B$):

```
>> a = [1 3 2;7 -5 1;-9 3 2];
>> b = [5 -17 3];
>> a \ b'
ans =
    0.2000
    3.2000
   -2.4000
```

which is equivalent to $a^{-1}b'$.

29

Element-by-element operations

To do element-by-element operations, the operators must be preceded by a period.

```
>> x = [1 2 3 4 5];
>> y = [5 8 12 40 10];
>> x / y
ans =
    0.1381
>> x ./ y
ans =
    0.2000    0.2500    0.2500    0.1000    0.5000
```

The same is true for other operators such as exponentiation:

```
>> y ^ -1
??? Error using ==> mpower
Matrix must be square.
>> y .^ -1
ans =
    0.2000    0.1250    0.0833    0.0250    0.1000
```

30

Using Functions

Functions can return multiple values. For example, the `eig` function returns the eigenvectors and eigenvalues of a matrix. To retrieve both, use

```
[val, vec] = eig(mat);
```

If only a single variable is used as a target, consult the documentation.

Since function arguments don't have keywords, many functions accept either a series of values or key value pairs as arguments. For example, to use the `dlmread` function to read a tab-delimited file, use:

```
x = dlmread('myfile', 'delimiter', '\t');
```

Again, consult the documentation for available choices.

M-files

Once you have created a script of matlab commands, one way to access them is by storing a file, with suffix `.m`, in either the current directory, or a directory called `matlab` in your home directory. You can then access the script by typing its name (minus the `.m`) in your matlab session. For example, if the file `doplot.m` in the current directory contains

```
x = [1 2 3 4 5];  
y = [12 19 17 21 22];  
plot(x,y, '*')
```

then you can execute those commands by typing `doplot` in the matlab console.

The `which` command will tell what script matlab is using for a particular command.

Writing Functions

A function is simply an m-file that begins with the word `function`. For example, suppose the file `~/matlab/scale.m` contains

```
function [new,themax] = scale(x)
% SCALE scale a vector by dividing by the maximum value
%   new = scale(x) returns a vector new, like x, with each
%       element divided by the maximum value of
%       the original vector
%   [new,themax] = scale(x) also returns the maximum value
%       of the original vector in themax
themax = max(x);
new = x / themax;
```

then you can use the `scale` function as you would any other matlab function. The comments after the function line will be displayed by the `help` command.

You can use the `edit` command to edit M-files; alternatively, use any editor of your choice, or `File`→`New` or `Open` in the console.

Vectorization

Most functions can operate on vectors or matrices returning an object of the same size by applying the function to each element. For example, suppose you have a vector of values and you want their square roots:

```
>> x = [9 17 24 15 12 82];
>> sx = sqrt(x)
sx =
    3.0000    4.1231    4.8990    3.8730    3.4641    9.0554
```

When some functions (`mean`, `max`, `sum`, `median`, `prod`, `sort`, `std`, etc.) are used with matrices, they automatically operate on each column:

```
>> xmat = [1 7 9 ; 2 4 3 ; 8 7 12 ; 4 9 9];
>> mean(xmat)
ans =
    3.7500    6.7500    8.2500
```

Vectorization (cont'd)

Most functions that operate on matrices in this way accept an optional argument (called `dim`) that allow you to choose on which dimension the operation is performed:

```
>> x = [3 7 4;2 1 8;5 10 12;7 9 14];
>> mean(x)
ans =
    4.2500    6.7500    9.5000
>> mean(x,2)
ans =
    4.6667    3.6667    9.0000   10.0000
```

Alternatively, you can operate on transposes.

To force matlab to treat a matrix as a vector use a colon as the single subscript:

```
>> mean(x(:))
ans =
    6.8333
```

Vectorization (cont'd)

The functions `arrayfun` and `cellfun` can be used to apply a function to each element of an array or cell array.

Suppose you wish to find the length of each element of a cell array:

```
>> cx = {[ 1 2 3 4] {'dog' 'cat'} [7 12 19]};
>> length(cx)
ans =
    3
>> cellfun(@length,cx)
ans =
    4    2    3
```

The leading “at”-sign (@) creates a function handle.

If the results need to be put into a cell array, the `UniformOutput` parameter needs to be set to 0:

```
>> classes = cellfun(@class,cx,'UniformOutput',0)
classes =
    'double'    'cell'    'double'
```

Anonymous Functions

If an existing function or m-file is not available for use with `cellfun` or `arrayfun`, you can create an anonymous function using the `@`-operator. Suppose you want to find those elements of the cell array that have a class of “double”:

```
myfunc = @(x)strcmp(x,'double');
```

```
>> cellfun(myfunc,classes)
```

```
ans =
```

```
    1     0     1
```

or more succinctly:

```
>> cellfun(@(x)strcmp(x,'double'),classes)
```

```
ans =
```

```
    1     0     1
```

Non-conformable operands

Note that when performing matrix operations, the dimensions of the operands must conform (unless one is a scalar). Sometimes it makes sense to repeat the values in the smaller operand. While matlab will not do this automatically, the `bsxfun` function can be used. Suppose we wish to subtract the mean from each column of a matrix:

```
>> mat = [7 9 2;3 5 1];
```

```
>> mx = mean(mat);
```

```
>> mat - mx
```

```
??? Error using ==> minus
```

```
Matrix dimensions must agree.
```

```
>> bsxfun(@minus,mat,mx)
```

```
ans =
```

```
    2.0000    2.0000    0.5000
```

```
   -2.0000   -2.0000   -0.5000
```

Creating Matrices

One way to create a matrix in matlab is to simply refer to an element of that matrix:

```
>> clear mymat
>> mymat(5,3) = 7;
```

This creates a 5×3 matrix with zeroes in the unassigned positions.

More commonly the `zeros` or `ones` function is used. To create a matrix with some other value, multiply the call to `ones` by a constant:

```
>> 3 * ones(4,2)
ans =
     3     3
     3     3
     3     3
     3     3
```

Creating Matrices (cont'd)

To create a matrix from repeated rows, either use multiplication or use `ones` for indexing, and include a second subscript.

```
>> x = [7 9 5];
>> ones(4,1) * x
ans =
     7     9     5
     7     9     5
     7     9     5
     7     9     5

>> x = [7 9 5];
>> x(ones(4,1), :)
ans =
     7     9     5
     7     9     5
     7     9     5
     7     9     5
```

Operations like this can also be done with the `repmat` function:

```
>> repmat(x,4,1)
ans =
     7     9     5
     7     9     5
     7     9     5
     7     9     5
```

Catenation

In matlab, there's no need for a special function for catenation, since square brackets ([]) will do this automatically.

```
>> x = [5 9 3 7 2];
>> y = [2 3 1 8];
>> [x y]
ans =
     5     9     3     7     2     2     3     1     8
```

Rows or columns can be added to matrices in the same way:

```
>> mat = [7 9;8 12;13 19];
>> [mat ; 15 18]
ans =
     7     9
     8    12
    13    19
    15    18

>> mat = [7 9;8 12;13 19];
>> [mat';5 8 14]'
ans =
     7     9     5
     8    12     8
    13    19    14
```

Catenation (cont'd)

The same method can be used to combine conformable matrices:

```
>> x1 = [3 2;4 7];
>> x2 = [7 12 15;8 4 19];
>> x3 = [9 18 24 15 12;6 19 8 14 20];
>> [x1 x2; x3]
ans =
     3     2     7    12    15
     4     7     8     4    19
     9    18    24    15    12
     6    19     8    14    20
```

Some other functions that are useful for similar tasks include `kron`, `toeplitz`, `blkdiag`, `reshape`, `flipud`, `fliplr`, `cat`, `vertcat`, and `horzcat`.

Structures

Matlab structures allow mixed data to be stored in a single object, but also allows referring to elements by name instead of number. Suppose you wish to work with the name and annual profits of some companies:

You can simply refer to elements directly:

```
>> firms(1).name = 'Acme Rockets';  
>> firms(1).profits = [7500 10000 12000];  
>> firms(2).name = 'Ajax Widgets';  
>> firms(2).profits = [9000 8000 10000];
```

You can use the `struct` function to add a new entry:

```
>> firms(end + 1) = struct('name','Precision Drills','profits',...  
[9000 9500 9750]);
```

Structures (cont'd)

To extract an element of a structure into an array or cell array, use square brackets (`[]`) or curly brackets (`{}`).

```
>> thenames = {firms.name};  
>> theprofits = [firms.profits];  
>> size(theprofits)  
ans =  
     1     9
```

Notice that creating an ordinary array lost the structure. Cell arrays will preserve this structure:

```
>> cellfun(@mean,{firms.profits})  
ans =  
 1.0e+03 *  
     9.8333     9.0000     9.4167
```

Structures (cont'd)

To access fields using variables, surround the variable with parentheses:

```
>> fld = 'profits'
>> firms(2).(fld)
ans =
    9000    8000   10000
```

The `fieldnames` function will return the names of the fields in the structure.

```
>> fnames = fieldnames(firms);
>> for f=fnames'
    firms.(f{:})
end
ans =
Acme Rockets
    . . .
ans =
    9000    8000   10000
```

Sorting

In its simplest form the `sort` command sorts a vector:

```
>> x = [9 7 14 12 15 8];
>> sort(x)
ans =
    7    8    9   12   14   15
```

Note that the sort is **not** done in place.

An optional character variable can be used for a descending sort:

```
>> sort(x,'descend')
ans =
   15   14   12    9    8    7
```

When applied to matrices, each row or column (depending on an optional numeric argument) of the matrix is sorted.

Sorting (cont'd)

To sort each row of a matrix by the values of a vector, first use `sort` to return a permutation vector; then use that to index the desired dimension.

```
>> x = [12 19 15;7 8 12;14 5 9;10 16 2];
>> [xx,perm] = sort(x(:,1));
>> x(perm,:)
ans =
     7     8    12
    10    16     2
    12    19    15
    14     5     9
```

47

Tabulation

The `tabulate` function takes a vector as input, and returns a 3-column matrix with the unique values from the vector, their frequencies, and their percentages.

```
>> values = [1,2,5,2,2,5,2,1];
>> tabulate(values)
Value    Count    Percent
     1         2    25.00%
     2         4    50.00%
     3         0     0.00%
     4         0     0.00%
     5         2    25.00%
```

Notice that some values have counts of zero; they could be eliminated using indexing:

```
>> res = tabulate(values);
>> res(res(:,2) > 0,:)
ans =
     1     2    25
     2     4    50
     5     2    25
```

For just the unique values, the `unique` function can also identify unique rows and columns of matrices.

48

Random Numbers

The `random` function from the Statistics Toolbox allows generation of random numbers. The first argument to `random` is a string identifying the distribution. This is followed by the appropriate number of parameters (1, 2, or 3) for the distribution. Finally, an optional vector with the desired output dimensions can be specified.

Some of the available distributions include `beta`, `bin`, `chi2`, `exp`, `f`, `gam`, `geo`, `nb`, `norm`, `poiss`, `t`, `unif`, and `wbl`.

```
>> random('normal',[0 0 0 5 5 5],1,[1 6])
```

```
ans =
```

```
    0.7079    1.9574    0.5045    6.8645    4.6602    3.8602
```

The functions `cdf` and `pdf` provide cumulative and probability distributions for the same distributions as `random` supports.

(Base matlab provides `rand` (uniform) and `randn` (normal) generators.)

Programming

- `if expression1`
 ... *statements* ...
 `elseif expression2` % elseif and else are optional
 ... *statements* ... % multiple elseifs are allowed
 `else`
 ... *statements* ...
 `end`
- `for var = start:end` % or `start:inc:end`
 ... *statements* ...
 `end`
- `for var = x` % elements of vectors or columns of matrices
 ... *statements* ...
 `end`
- `while expression`
 ... *statements* ...
 `end`

Example

```
function seq = rep(x,n)
% REP - create vector with repeated values
%   seq = rep(x,n) returns a vector with the values
%       in x repeated n times
%       if x and n are vectors of the same length,
%       each element of x is repeated by the corresponding
%       element in n
if nargin ~= 2
    error('requires 2 arguments');
end
if length(n) ~= 1 && length(x) ~= length(n)
    error('x and n must be same length');
end
if length(n) == 1
    if length(x) == 1
        seq = x * ones(1,n);
        return
    else
        seq = reshape((x(ones(n,1),:))',1,n * length(x));
        return
    end
else
    seq = [];
    for i=1:length(x)
        seq = [seq x(i) * ones(1,n(i))];
    end
    seq = reshape(seq',1,sum(n));
    return
end
```

51

Another Example

```
function [vals,lens] = rle(x)
%RLE - run length encoding
%   [vals,lens] = rle(x)
%       vals consists of the values of runs
%       found in x; corresponding elements of
%       lens contain the length of the runs
%
%   are adjacent elements equal?
y = x(2:end) ~= x(1:end-1);
%   find where adjacent elements change
i = [find(y | isnan(y)) length(x)];
vals = x(i);
lens = diff([0 i]);
return
```

52

Sparse Matrices

A sparse matrix is one where only a small fraction of the values are non-zero. The `sparse` command can create a sparse matrix, either from an ordinary (full) matrix, or from a set of indices and values. Once a sparse matrix is created, all operations on it will be carried out using special routines.

To convert a sparse matrix to a full matrix, use the `full` function.

Remember that some operations (like adding a constant to each element of a matrix) will force matlab to use full storage.

The `issparse` function tests if a matrix is being stored as a sparse matrix.

Output Format

The `format` command allows easy control over the way output is displayed. (Alternatively, the `File`→`Preferences` dialog can be used with the console.)

The `short` format displays 4 digits after the decimal, while the `long` format displays 14 or 15. The `e` modifier specifies exponential, the `g` modifier uses the “best” format, and the `eng` modifier uses engineering notation.

Alternatively, the `hex` and `bank` formats can be used.

To suppress blank lines in the output, use `compact`; to display the blank lines, use `loose`

```
>> format bank
>> format long g
>> format compact
>> format('long','eng')
```

Graphics

Some of the more useful graphics functions include:

- `plot`, `plot3` - plots 2-d and 3-d curves from data
- `fplot` - plot 2-d curve from function definitions
- `loglog`, `semilogx`, `semilogy` - plot 2-d curve with logarithmic axes
- `hist`, `bar`, `barh`, `rose` - plot 2-d histograms/bar charts
- `mesh`, `meshz`, `meshc`, `waterfall` - plot 3-d mesh surface from regular data
- `surf`, `surfc`, `surf1` - plot shaded surface from regular data
- `griddata` - prepare data for use with `mesh`, etc.

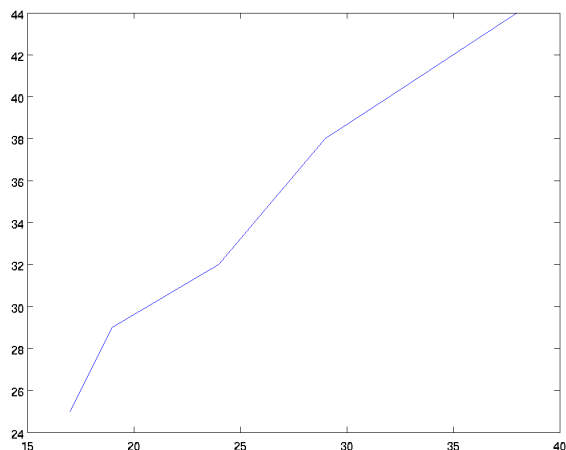
Plots created by matlab can be modified when created, or interactively.

55

Simple Graphics

The basic use of the plotting commands is very straight forward. For example, to create a plot of two vectors, the input on the left produces a window with the output on the right.

```
>> x = [17 19 24 29 32 38];  
>> y = [25 29 32 38 40 44];  
>> plot(x,y)
```



56

Colors, Markers and Linestyles

Graphics properties can be modified by combining symbols from the following columns into a third argument to `plot`:

Symbol	Color	Symbol	Marker	Symbol	Linestyle
b	blue	.	point	-	solid line
g	green	o	circle	:	dotted line
r	red	x	cross	-.	dash-dot line
c	cyan	+	plus sign	--	dashed line
m	magenta	*	asterisk		
y	yellow	s	square		
k	black	v	triangle (down)		
w	white	^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

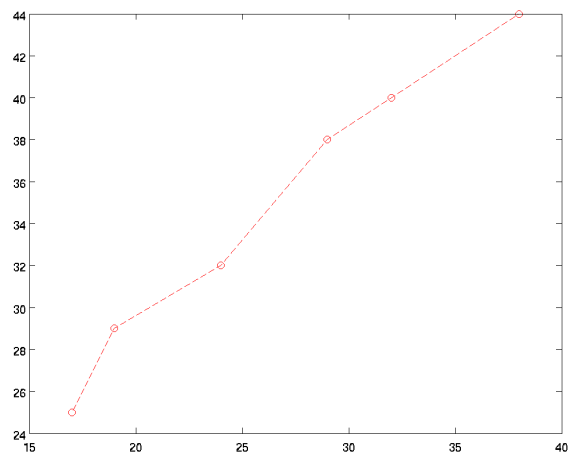
adapted from “Mastering Matlab 5,” by Hanselman & Littlefield

57

Colors, Markers and Linestyles (cont'd)

To replot the previous example, using a red dashed line with circles for the data points:

```
>> x = [17 19 24 29 32 38];  
>> y = [25 29 32 38 40 44];  
>> plot(x,y,'ro--')
```

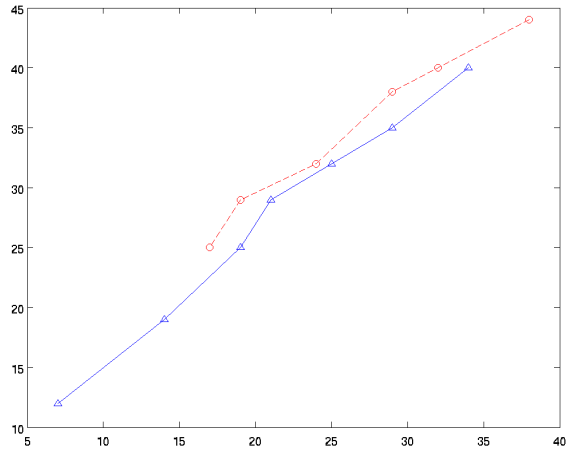


58

Multiple Lines on a Plot

By providing additional vectors to the `plot` command, it's easy to put multiple plots in a single figure.

```
>> x = [17 19 24 29 32 38];  
>> y = [25 29 32 38 40 44];  
>> a = [7 14 19 21 25 29 34];  
>> b = [12 19 25 29 32 35 40]  
>> plot(x,y,'ro--',a,b,'b^-')
```



59

Adding Information to Plots

The following functions and commands are available:

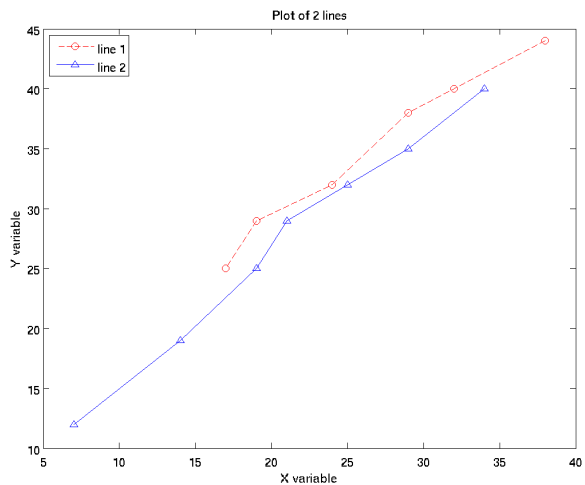
- `xlabel`, `ylabel` - add x- or y- axis labels
- `axis` - control axes properties
- `grid` - use `grid on` or `grid off`
- `box` - use `box on` or `box off`
- `title` - add a title above the plot
- `text` - add text to a plot
- `gtext` - add text to a plot using pointing device
- `textlabel` - create a label using \LaTeX
- `annotation` - annotate a plot
- `legend` - add a legend to a plot

Note: The `plotedit` on command allows interactive editing.

60

Annotated Plot

```
>> x = [17 19 24 29 32 38];  
>> y = [25 29 32 38 40 44];  
>> a = [7 14 19 21 25 29 34];  
>> b = [12 19 25 29 32 35 40]  
>> plot(x,y,'ro--',a,b,'b^-')  
>> title('Plot of 2 lines')  
>> xlabel('X variable')  
>> ylabel('Y variable')  
>> legend('Location',...  
'NorthWest',...  
{'line 1' 'line 2'})
```



61

Multiple Figures on a Page

The `subplot` function allows you to divide the plotting figure into subplots, and to select an active subplot for graphics commands.

The basic form of the command is

```
subplot(nrows,ncols,nwhich)
```

This sets the current plotting region to the `nwhich`-th plot within an `nrows` by `ncols` grid, counting by rows. After calling `subplot`, plotting commands will be executed in the specified subplot. The optional argument `'replace'` will replace an existing subplot.

Finally, the command

```
subplot('Position',[left bottom width height])
```

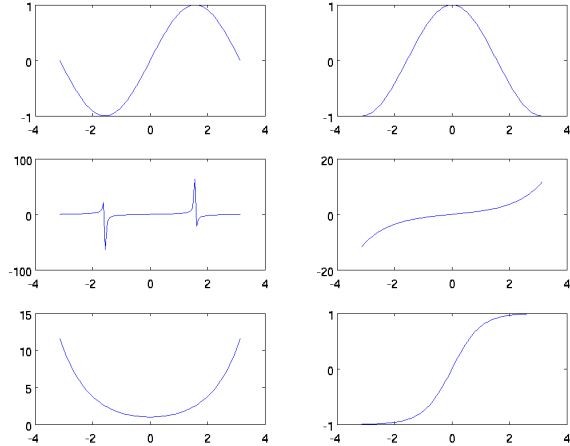
creates a `width`×`height` subplot at the specified position.

62

Multiple Figures on a Page (cont'd)

To create a 3×2 grid of trigonometric functions on a single figure:

```
>> xvals = linspace(-pi,pi,100)
>> subplot(3,2,1)
>> plot(xvals,sin(xvals))
>> subplot(3,2,2)
>> plot(xvals,cos(xvals))
>> subplot(3,2,3)
>> plot(xvals,tan(xvals))
>> subplot(3,2,4)
>> plot(xvals,sinh(xvals))
>> subplot(3,2,5)
>> plot(xvals,cosh(xvals))
>> subplot(3,2,6)
>> plot(xvals,tanh(xvals))
```



63

Printing / Saving Plots

The `print` command is used both to print plots, and to save them to files in a variety of formats. The default printing behaviour is defined in the system `printopt.m` file, and can be overridden with a local copy.

Examples:

```
print % sends current figure to default printer
print -dpng plot.png % stores a png file in plot.png
print -dpng plot % same as previous example
print('-dpng','plot') % alternative to last 2 examples
```

Some other format options include `-dps`, `-dtiff`, `-dpsc` (color PostScript), and `-depsc` (encapsulated PostScript).

64

Modifying Plots

While plots can be customized through a GUI (via `plottedit` on, or `Tools`→`Edit Plot` in the Figure window), it is sometimes useful to make modifications in scripts. To facilitate this, matlab uses a system known as handle graphics. If the return value from a plotting function is stored, it can be passed to the `get` or `set` routine to examine or modify its properties.

Many times the desired property is in the handle of the parent of the plot; the parent's handle can be retrieved as follows:

```
>> hand = plot(x,y)
>> phand = get(hand, 'Parent')
```

Passing this handle to the `get` function with no arguments will display all the available parameters and their current values, which can be modified using the `set` function.

Modifying Plots (cont'd)

Suppose we wish to change the location of the tick marks on the x-axis of a plot. First, create the plot, saving the handle in a variable, and use it to access the parent handle:

```
>> x = [9 12 14 17 22 28];
>> y = [10 19 18 23 25 31];
>> h = plot(x,y);
>> ph = get(h, 'Parent');
```

The output of the command `get(ph)` includes the following line:

```
      . . .
XTick = [ (1 by 8) double array]
      . . .
```

To use 3 ticks at 8, 16, 24 we could use

```
>> set(ph, 'XTick', [8 16 24])
```

mlint M-file checker

Matlab provides a program checker that goes beyond simple syntax checking, and can sometimes point out ways of improving your program. It's accessed by the `mlint` command, followed by the name of the M-file to be checked.

```
function val = findlarge(vec,lim)
    val = [];
    for v = vec
        if v > 0 & v > lim
            val = [val v];
        end
    end
end
```

```
>> mlint findlarge
```

```
L 3 (C 5-7): FOR may not be aligned with its matching END (line 7).
L 4 (C 9-10): IF may not be aligned with its matching END (line 6).
L 4 (C 18): Use && instead of & as the AND operator in
           (scalar) conditional statements.
L 5 (C 6-8): 'val' might be growing inside a loop.
           Consider preallocating for speed.
```

Timing

The `tic` and `toc` functions can be used to measure the elapsed time of code execution. Call `tic` before the statements to be called, and `toc` after they are done:

```
>> tic;x1 = 3 * ones(10000,100);t1=toc;
>> tic;xx=3;x2 = xx(ones(10000,100));t2=toc;
>> tic;x3= repmat(3,[10000 100]);t3=toc;
>> [t1 t2 t3]
ans =
    0.0182    0.0588    0.0193
```

The `cputime` function can also be used, but its resolution may be much lower than `tic/toc`:

```
>> t0=cputime();x1 = 3 * ones(10000,100);t1=cputime() - t0;
>> t0=cputime();xx=3;x2 = xx(ones(10000,100));t2=cputime() - t0;
>> t0=cputime();x3= repmat(3,[10000 100]);t3=cputime() - t0;
>> [t1 t2 t3]
ans =
    0.0100    0.0700    0.0300
```

Profiling

Profiling a program means determining which parts of the program are using the most CPU time. This helps to pinpoint bottlenecks, and suggests what parts of a program need to be improved in order to speed it up. To start profiling a program, type

```
profile on
```

in the matlab console. Once your program is done, type

```
profile off
```

to stop profiling, and

```
profile viewer
```

to see the results. To store the results for further analysis, use

```
stats = profile('info')
```

Debugging M-Files

To start debugging an M-File, type

```
debug
```

in the matlab console.

To see where to set appropriate breakpoints, either examine the M-file, or use

```
dbtype filename
```

where *filename* is the name of the M-file you're debugging. When you identify a breakpoint, type

```
dbstop filename num
```

where *num* is the line number at which you want to stop. Now, run the command in the normal way.

Debugging M-Files (cont'd)

When the editor opens on your code it will show the currently executing line, and the console prompt will change from

>>

to

K>>

At this point, you can examine and change the values of variables, as well as use the following commands:

Command	Function
dbstep	Advance one line in the program
dbcont	Continue executing
dbstop	Add breakpoint
dbclear	Remove breakpoint(s)
dbquit	Exit the debugger

Optimization Toolbox

The Optimization Toolbox offers a wide variety of techniques, each with a common interface. The main functions are described in the table below:

Function	Purpose
fminunc	Unconstrained Minimization
fmincon	Constrained Minimization
fminbns	Bounded Minimization
linprog	Linear Programming
quadprog	Quadratic Programming
lsqnonneg	Least Squares
lsqlin	Constained Linear Least Squares

The two required arguments are the function to be optimized, and the starting values. Additional arguments are used to specify constraints, which can be both linear and non-linear, and can be equalities or inequalities.

Linear Constraints

The optimization routines allow up to six arguments to specify linear constraints:

- Linear inequality (provide **A** matrix and **b** vector)

$$A \cdot x \leq b$$

- Linear equality (provide **Aeq** matrix and **beq** vector)

$$Aeq \cdot x = beq$$

- Lower and upper bounds (provide **lb** and **ub** vectors)

$$lb \leq x \leq ub$$

Use **-Inf** and **Inf** for unbounded variables.

If you don't need all six arguments, pass an empty vector (`[]`) as a placeholder.

Linear constraints (cont'd)

Consider the following problem

$$\begin{aligned} \text{Minimize} & : 2x_1^2 + x_1x_2 + 2x_2^2 - 6x_1 - 6x_2 + 15 \\ \text{Subject to} & : x_1 + 2x_2 \leq 5 \\ & 4x_1 \leq 7 \\ & x_2 \leq 2 \\ & -2x_1 + 2x_2 = -1 \end{aligned}$$

The following arguments can be used:

```
A = [1 2;4 0;0 1];
```

```
b = [5 7 2];
```

```
Aeq = [-2 2];
```

```
beq = -1;
```

```
[x,fval] = fmincon(@(x)2 * x(1)^2 + x(1) * x(2) + 2 * x(2)^2 ...  
- 6 * x(1) - 6 * x(2) + 15,[1 1],A,b,Aeq,beq);
```

Non-linear constraints

Non-linear constraints are implemented by providing a function which returns two values – the result of evaluating any inequality constraints, and the result of evaluating any equality constraints, in both cases evaluated with respect to 0.

For example, suppose we have the following problem:

$$\begin{aligned} \text{Maximize} & : y = 2x_1 + x_2 \\ \text{Subject to} & : x_1^2 + x_2^2 \leq 25 \\ & x_1^2 - x_2^2 \leq 7 \end{aligned}$$

We could create a suitable M-file function (say, `mycon.m`) as follows:

```
function [c,ceq] = mycon(par)
    c = [par(1)^2 + par(2)^2 - 25 par(1)^2 - par(2)^2 - 7];
    ceq = [];
    return
```

Non-linear constraints (cont'd)

To avoid using global variables to evaluate the constraint function, an anonymous function can be used:

```
[x,fval] = fmincon(@(par)-2*par(1)-par(2),...
    [1 1], [], [], [], [], [], @(par)mycon(par));
```

Note that the the function evaluates the negative of the stated problem, since we wish to maximize the function.

Alternatively, an anonymous function could be used directly. The `deal` function allows the return of multiple arguments:

```
[x,fval] = fmincon(@(par)-2*par(1)-par(2),...
    [1 1], [], [], [], [], [], ...
    @(par)deal([par(1)^2 + par(2)^2 - 25 par(1)^2 - par(2)^2 - 7], []));
```

Options for Optimization Routines

Options for the routines are controlled by an argument created by the `optimset` function. Typing `optimset` with the name of a function displays the default options:

```
>> optimset fminunc
ans =

    Display: 'final'
  MaxFunEvals: '100*numberofvariables'
    MaxIter: 400
    TolFun: 1.0000e-06
    TolX: 1.0000e-06
  FunValCheck: 'off'
    OutputFcn: []
    PlotFcns: []
  ActiveConstrTol: []
  BranchStrategy: []
  DerivativeCheck: 'off'
  . . .
```

For example to change the maximum number of function evaluations from its default, the following call to `optimset` could be passed to `fminunc`

```
optimset('MaxFunEvals',500)
```

Optimization with Data: Background

In a small village, inhabitants suffer from an eye disease. Fifty subjects at each of several ages were examined, with the following number of blind found:

Age (x)	20	35	45	55	70
Number blind (out of 50)	6	17	26	37	44

We wish to maximize a logistic likelihood for this data, using the following likelihood function:

$$L = k - \sum_{i=1}^5 y_i \log p(\alpha, \beta, x_i) - \sum_{i=1}^5 (50 - y_i) \log(1 - p(\alpha, \beta, x_i))$$

where y_i represents the number blind out of 50, and x_i represents the age.

Evaluating the Likelihood

The following lines, placed in the file `likli.m`, define a function to evaluate the likelihood:

```
function x = likli(par,blind,age)
    pp = @(par,x)1 ./ (1 + exp(-(par(1) + par(2) * x)));
    x = -sum(blind .* log(pp(par,age))...
        + (50 - blind) .* log(1. - pp(par,age)));
```

Notice that the function accepts three arguments, but the optimization routines expect just one, the vector of parameters. We can solve the problem with an anonymous function:

```
>> age = [20 35 45 55 70];
>> blind = [6 17 26 37 44];
>> mylikli = @(par)likli(par,blind,age)
>> [x,fval] = fminunc(mylikli,[-3.5,.1])
```

79

Other features

- Demos (Demos tab in the Help window)
- Animated Plots (`getframe` and `movies`)
- Graphical User Interfaces (`guide`)
- Other Toolboxes (Statistics, Image, Wavelet, Simulink)
- Mathwork's File exchange at <http://www.mathworks.com/matlabcentral/fileexchange>
- Loren Shure's Mathworks Blog on the Art of Matlab at <http://blogs.mathworks.com/loren/>

80