More MATLAB

Last lecture focussed on MATLAB Matrices (Arrays) and vectors which are fundamental to how MATLAB operates in its key application areas — including **Multimedia data processing**

We continue our brief overview of MATLAB by looking at some other areas:

- Basic programming and essential MATLAB
- MATLAB data and system management

MATLAB Statements and expressions

We have already met some simple expressions with MATLAB matrices but let's formalise things:

- MATLAB is an *expression* language;

  the expressions you type are interpreted and evaluated.

- MATLAB statements are usually of the form:

  variable = expression, or simply: *expression*

- Expressions are usually composed from operators, functions, and variable names.

- Evaluation of the expression produces a matrix, which is assigned to the variable for future use and/or is then displayed on the screen .

- If the variable name and = sign are omitted, a variable `ans` (for answer) is automatically created to which the result is assigned.

◀◀
▶▶
◀
▶
Back
Close

Important Note: MATLAB is case-sensitive

- MATLAB is **case-sensitive** in the names of commands, functions, and variables.
- For example,

  `IM` is not the same as `im`.

Statement Termination

- A statement is *normally terminated* with the **carriage return**.
- A statement can be continued to the next line with three or more periods followed by a carriage return.

```
>> A = 3 + ...
4
A =
     7
```

- On the other hand, several statements can be placed on a single line if separated by commas or semicolons.

```
>> A= 3 + 4; B = 2*A; C = B + A
C =
    21
```

Back
Close

Statement Termination (cont.)

- If the last character of a statement is a **semicolon**, the printing is suppressed, but the assignment is carried out.

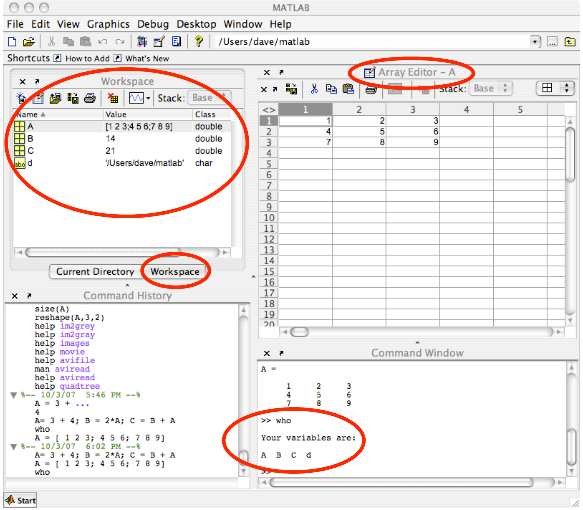
  Recall: This is essential in suppressing unwanted printing of intermediate results.

- Unwanted printing to the command window significantly slows down MATLAB processing:
  - Useful for debugging
  - Avoid in intensive loops/recursion *etc.* when not debugging.

MATLAB Variable Spaces

You can find out what variables exist in you program in two ways:

- The command `who` (or `whos`) will list the variables currently in the workspace.
- The MATLAB IDE **Workspace** window lists them and their type and value. Clicking on a Matrix/Array structure brings up an **Array Editor** which can be useful.



Name	Value	Class
A	[1 2 3;4 5 6;7 8 9]	double
B	14	double
C	21	double
d	'/Users/dave/matlab'	char

```
size(A)
reshape(A,3,2)
help im2grey
help im2gray
help images
help movie
help avifile
man aviread
help aviread
help quadtree
%-- 10/3/07  5:46 PM --%
A = 3 + ...
4
A= 3 + 4; B = 2*A; C = B + A
who
A = [ 1 2 3; 4 5 6; 7 8 9]
%-- 10/3/07  6:02 PM --%
A= 3 + 4; B = 2*A; C = B + A
A = [ 1 2 3; 4 5 6; 7 8 9]
who
```

<>	1	2	3	4	5
1	1	2	3		
2	4	5	6		
3	7	8	9		

```
A =

     1     2     3
     4     5     6
     7     8     9

>> who

Your variables are:

A  B  C  d

>>
```

```
>> whos
  Name      Size         Bytes  Class
  A         3x3           72  double array
  B         1x1          8  double array
  C         1x1          8  double array
  d          1x18        36  char array
Grand total is 29 elements using
124 bytes
```

Back
Close

Clearing Variables, etc.

A variable can be cleared/deleted from the workspace with the command:

`clear` *variablename*.

The command `clear` alone will clear all nonpermanent variables.

Other forms of `clear` include:

`clear global`: removes all global variables.
`clear functions`: removes all compiled M- and MEX-functions.
`clear all`: removes all variables, globals, functions and MEX links.

MATLAB Sessions

A MATLAB begins when the application starts up and ends when quits MATALAB:

- Generally **on exit** MATLAB **all variables are lost**.
- **Unless**, however, you save your MATLAB workspace or a selection of variables:

  Invoking the command `save` before exiting causes **all variables** to be written to a (binary format) file named `matlab.mat`.
- When one later reenters MATLAB, the command `load` will restore the workspace to its former state.
- `save FILENAME` will save all variables to the named file
- `save FILENAME X Y Z ....` will save the listed variables (`X Y Z` in this case) to the named file
- See `help save` and `help load` for more details

◀◀
▶▶
◀
▶
Back
Close

Managing MATLAB

The following few slides summarise a few useful commands for managing MATLAB from the command window:

help	help facility
which	locate functions and files
demo	run demonstrations
path	control MATLAB's search path
why	*Try it and see!*

Useful: A runaway display or computation can be stopped on most machines without leaving MATLAB with CTRL-C (CTRL-BREAK on a PC).

Managing Variables and the Workspace

who	list current variables
whos	list current variables, long form
save	save workspace variables to disk
load	retrieve variables from disk
clear	clear variables and functions from memory
pack	consolidate workspace memory
size	size of matrix
length	length of vector
disp	display matrix or text

Back
Close

Files and the Operating System

cd	change current working directory
pwd	show current working directory
dir, ls	directory listing
delete	delete file
getenv	get environment variable
!	execute operating system command
unix	execute operating system command; return result
diary	save text of MATLAB session

Controlling the Command Window

clc	clear command window
home	send cursor home—to top of screen
format	set output format

Example:

```
>> A=rand(2, 2);
>> format long; A
A =
   0.07596691690842   0.12318934835166
   0.23991615355658   0.18390778828241 7
>> format short; A
A =
    0.0760    0.1233
    0.2399    0.1839
```

Back
Close

Starting and Quitting from Matlab

quit	terminate MATLAB
startup.m	Special (M-file) executed when MATLAB is started
matlabrc.m	MATLAB master startup M-file

If exists, `startup.m` is actually called from `matlabrc.m`. It is recommended to create/modify your own `startup.m` for operations that are supposed to be done every time MATLAB starts.

◀◀
▶▶
◀
▶
Back
Close

For, While, If statements

In their basic forms, these MATLAB flow control statements operate like those in most computer languages. Note: these keywords should NOT be capitalised.

For:

For example, for a given n, the statement:

```
x = []; for i = 1:n, x=[x,iˆ2], end
```

or

```
x = [];
for i = 1:n
     x = [x,iˆ 2]
end
```

will produce a certain n-vector and the statement

Note: `x = []; for i = n:-1:1, x=[x,iˆ2], end`

will produce the same vector in reverse order.

Note: a matrix may be empty (such as `x = []`).

Matrix Elements: Vectorise NOT loops

Avoid using `for` loops *etc.* to index and manipulate matrix elements where ever possible, **Vectorise**: **loops significantly slow down Matlab**.

For example:

```
x(1:n)
```

is **MUCH MORE ELEGANT** than

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

For more details and examples see:

- Mathworks Code Vectorisation guide
- Cambridge University Engineering Dept. : Matlab vectorisation tricks (Web Page) More resources available following the link.

Back
Close

While:

The general form of a `while` loop is:

```
while relational expression
  statements
end
```

The statements will be repeatedly executed as long as the `relational expression` **remains true**.

For example:

```
n = 0;
while  n < 10
  n = n + 1
end
```

◀◀
▶▶
◀
▶
Back
Close

If:

The general form of a simple `if` statement is

```
if relational expression
     statements
end
```

The statements will be executed only if the `relational expression` **is true**.

Simple example:

```
if  grade_average >= 70
   pass = 1;
end;
```

Back
Close

If .... elseif .... else:

Multiple branching is also possible, as is illustrated by

```
for m = 1:k
    for n = 1:k
        if m == n
            a(m,n) = 2;
        elseif abs(m-n) == 2
            a(m,n) = 1;
        else
            a(m,n) = 0;
        end
    end
 end
```

In two-way branching the elseif portion **would**, of course, be **omitted**.

Relational Operators

The relational operators in MATLAB are

$<$	less than
$>$	greater than
$<=$	less than or equal
$>=$	greater than or equal
$==$	equal
$\sim=$	not equal.

Note that "=" is used in an assignment statement while "==" is used in a relation (the same as Java or C).

Back
Close

Logical Operators

Relations may be connected or quantified by the logical operators

&	and
|	or
∼	not

Truth table:

A	B	$A\&B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A\|B$
0	0	0
0	1	1
1	0	1
1	1	1

A	$\sim A$
0	1
1	0

Simple example:

```
if  ( (grade_average >= 60) & (grade_average < 70))
   pass = ’2.1’;
end;
```

◀◀
▶▶
◀
▶
Back
Close

Relational Operators, Scalars and Matrices

When applied to **scalars**, the result is actually the scalar 1 or 0 depending on whether the relation is true or false.

For example:
`a< 5, b > 5, c == 5,` and `a == b`.

When applied to **matrices** of the **same size**, the result is a matrix of 0′s and 1′s giving the value of the relation between corresponding entries.

For example: `a = rand(5); b = triu(a); a == b` gives:

```
ans =   1       1       1       1       1
        0       1       1       1       1
        0       0       1       1       1
        0       0       0       1       1
        0       0       0       0       1
```

Logical Data Type and Submatrices

Scalars/matrices obtained through relational operators are recognised as logical data type. For example

```
>> a=rand(1,6)
a =
    0.7431    0.3922    0.6555    0.1712    0.7060    0.0318
>> b=a>0.5
b =
     1     0     1     0     1     0
>> whos a b
  Name         Size               Bytes  Class      Attributes
  a            1x6                   48  double
  b            1x6                    6  logical
```

Submatrices can be addressed using a logical vector of the same size as a mask. A convenient way to obtain submatrices with elements satisfying certain condition.

```
>> a(b)
ans =
    0.7431    0.6555    0.7060
```

produces a subvector with elements > 0.5. This is identical to `a([1 3 5])` if we use an index vector instead.

◀◀
▶▶
◀
▶
Back
Close

Logical Data Type and Submatrices (cont.)

Note: Logical matrices are different from double matrices. Double matrices can't be used as masks:

```
>> a([1 0 1 0 1 0])
??? Subscript indices must either be real positive integers or logicals.
```

Logical data type can be obtained also by casting double matrices using `logical` function.

```
>> a(logical([1 0 1 0 1 0]))
ans =
    0.7431    0.6555    0.7060
```

`find` function returns a matrix comprising indexes of non-zero elements (for both double or logical matrices).

```
>> b
b =
     1     0     1     0     1     0
>> find(b)
ans =
     1     3     5
>> a(find(b))
ans =
    0.7431    0.6555    0.7060
```

Back
Close

Matrix relations in While and If

A relation between matrices is interpreted by `while` and `if` to be true if each entry of the relation matrix is **nonzero**.

So, if you wish to execute *statement* when matrices A and B are **equal** you could type:

```
if  A == B
    statement
end
```

However if you wish to execute *statement* when A and B are **not equal**, you have to be more careful.

Since that the seemingly obvious `if A ~= B,` *statement*`,` `end` will not give what is intended since

- *statement* would execute **only if** *each* of the corresponding entries of A and B differ.

◀◀
▶▶
◀
▶
Back
Close

The `Any` Function

To execute, *statement* when A and B are **not equal**, we can use the `any` operator:

```
if    any(any(A ~= B))
   statement
end
```

The functions `any` and `all` can be creatively used to reduce matrix relations to vectors or scalars.

- `any` returns **true** if **any** element of a **vector** is a nonzero number.
- Two `any`'s are required above since `any` is a vector operator.

Alternatively, more simply, we could ‘*invert*’ the logic:

```
if  A == B  else
   statement
end
```