Matlab bootcamp – Class 2

Written by Kyla Drushka, modified from notes by Darcy Ogden for course SIO113

Aside: scientific notation

The letter **`e`** represents shorthand notation for the exponential **`10^`** . For example

`>> 1e3`

is equivalent to 1000.

`>> 9.283e3`

is equivalent to 9283.

This is a handy way to enter small numbers:

`>> 4.229e-9`

is equivalent to 4.229 x 10^{-9}

Note that Matlab by default only prints out 5 digits (scaled using scientific notation if necessary). You can change the output using the command **`format`**. E.g.

`>> x=1234567.8901234`

produces

```
x =

   1.2346e+06
```

```
>> format long    % shows 15 digits using fixed-point format
>> x
```

produces

```
x =

     1.234567890123400e+06
```

This doesn't change the precision of the commands performed by Matlab, just the way the output looks.

Loading & saving variables I : Matlab-format (.mat) files

Save your entire workspace:

`>> save mystuff`

or

`>> save('mystuff')`

or

`>> save('mystuff.mat')`

all produce a file called "mystuff.mat", which contains all of the variables in your workspace.

```
>> save('mystuff','x1,'x2','x3','y*')
```
saves variables x1,x2,x3 and all variables beginning with the letter y into the file mystuff.mat

You can also include the directory that the file is saved in.
```
>> save('/mydirectory/mystuff')
```

Load .mat files that you have saved:

```
>> load('mystuff')
```
loads all of the variables in mystuff (note, this will overwrite any variables in your current workspace that have the same name)

```
>> load('mystuff','x*')
```
load just variables starting with the letter x

We will talk about some other formats for saving on Thursday.

Other useful commands:

```
>> d=dir('/mydirectory/')
```
Lists all files and subdirectories in the specified directory, in a structure containing the name, date, size, etc.

Selecting subsets of arrays: indexing

This is a very powerful and important aspect of Matlab!

Definitions:

element: a single value in an array. E.g.
```
>> g=[9 8 5 2];   % the second element of g is 8.
```

index: the "address" of an element within an array. It is specified using parentheses.
```
>> g(2)      % the value of the element at index 2 is 8
```

Examples:

```
>> r=[3 2 5 6 1 11] % r is a 1x6 row
>> r(3)    % the index is 3. This selects the third element of
r (which is equal to 5)

>> v=r(3)   % make a new variable, v, and set it equal to the
third element of r
```

>> x=r([2 4 5]) % select the 2^{nd}, 4^{th}, and 5^{th} elements of r and store them in a new vector called x. Note: specify multiple indices of r by containing them in square brackets.

The colon operator can also be used to specify a range of values:
>> r(2:4) % select the 2^{nd} to 4^{th} elements of r.

>> r(1:2:5) % select the 1^{st}, 3^{rd}, and 5^{th} elements of r.

If you don't know how many elements are in a certain dimension, you can indicate the last element as "`end`":

>> r(2:end) % selects from index 2 to the end of r.

>> x=r(:) % the colon alone selects ALL elements of r

Indices for matrices:

For matrices, indices take the following form:

```
matrix(rows,cols,dim3,dim4...)
```

i.e. the first entry corresponds to the row index, the second to the column index, etc. For example:

>> g=[9 8 7; 6 5 4] % create g, a 2x3 matrix

```
g =

     9     8     7
     6     5     4
```

>> g(2,:) % selects the elements in the second row and all columns of g

>> g(1,[1 3]) % selects the elements in the first row and only the first and third columns of g

Find

One powerful way to define indices in an array is to use the command **`find`**, which returns the indices of the *non-zero elements of its argument*: E.g.

>> m=[1 3 5 -2 -1 0 0 3]
>> find(m)

```
ans =
```

```
     1     2     3     4     5     8
```

Note,

```
>> find(~m)
```

returns the zero indices (~ is like "not", so you are asking to look for all **not**-zero elements)

```
ans =

     6     7
```

Logical expressions

The output of *logical expressions* is equal to 1 for true and 0 for false, so they can be used as the argument of the `find` command. For example `isnan` is a logical operator that returns 1 if its argument is NaN (not a number), and 0 otherwise:

```
>> v=[1 2 3 nan]
>> isnan(v)

ans =

     0     0     0     1
```

And since `find` returns the indices of all non-zero elements of its argument, combining `find` with the logical operator `isnan` returns the indices of the elements in `v` that are NaN:

```
>> find(isnan(v))

ans =

     4
```

That is, the 4th element of `v` is NaN

To generate logicals, you can use *relational operators* such as:

<
>
<=
>=
== (note, **two** equals signs are needed for the *relational* equals)
~= (not equal to)

E.g.

```
>> m=[1 3 5 -2 -1 0 0 3]
>> i=find(m==3)    % returns the index of elements of m that
are equal to 3, and stores the index in the variable i:

i =

     2     8
```

Indexing for matrices:

So far, we have just considered indexing for vectors. For a matrix, you can find the index in (row,column) format by specifying two outputs for find. E.g.

```
>> n=[2 3 4 9 11; 5 -1 0 -2 -2; 6 6 5 5 3]

n =

     2     3     4     9    11
     5    -1     0    -2    -2
     6     6     5     5     3

>> [rowind colind]=find(n==3)
rowind =

     1
     3

colind =

     2
     5
```

That is, n(1,2)=3 and n(3,5)=3

If you only specify one output when you apply `find` to a matrix, Matlab returns the "linear index", which means that it first reshapes your matrix into one long vector (column-wise) and then finds the index in that vector. E.g.:

```
>> i=find(n==3)

i =

     4
    15
```

Because when you stretch `n` into a vector, column by column, you get:

```
     2
     5
     6
     3              ← the 4th element is 3.
    -1
     6
     4
     0
     5
     9
    -2
     5
    11
    -2
     3              ← and the 15th element is 3.
```

No elements found:

If the argument of `find` is all zeros, it returns an empty matrix. E.g.

```
>> g=ones(3); % generate a 3x3 matrix of ones
>> find(g<0)
ans =

   Empty matrix: 0-by-1
```

Example of using indexing:

The file tao.mat contains monthly sea surface temperature data from 1992 to 2012 at a mooring in the equatorial Pacific Ocean. I downloaded the data from the TAO website.

```
>> load tao.mat
```

tao.mat contains two variables: **time** and **temperature** (one temperature value per time). Note **that missing data have been given the value -9.99** (something like this is often done to indicate bad or missing data, especially for datasets that are publically available).

Now plot:

```
>> plot(time,temperature)
>> datetick('x')  % this sets the x axis to be dates; more on
this later
>> ylabel('temperature, degrees C')
>> xlabel('date')
>> title('temperature from TAO')
```

There are obviously some bad values of temperature. We can identify these using `find`:

```
>> badi=find(temperature==-9.99) % badi returns the index of
data with the value -9.99 (i.e. missing data):

badi =

    49
    50
    51
   139
   140
   167
   191
   192
   193
   194
```

Let's set all the missing/bad data to `NaN` (not a number):

```
>> temperature(badi)=NaN; % now all the elements of
temperature corresponding to the index badi are Nan.
```

Matlab does not plot `NaN` values, so now our figure looks more reasonable:

```
>> plot(time,temperature)
>> datetick('x'); ylabel('temperature, degrees C')
```

A note on times/dates with Matlab:

Matlab uses a format for dates called a “serial date number” that is based on the number of days from a reference of Jan 1, year zero. E.g., September 24, 2013 at noon has a serial date number of 735501.5, i.e. we are 735501 days from Jan 1 0000, and noon represents half a day. You can convert using between yyyy-mm-dd and serial dates using `datenum` and `datestr`, e.g.:

```
>> datenum(2001,12,1)    % gives the serial date number for
December 1, 2001
>> now          % gives the serial date number for this moment
>> datestr(now)  % the date right now, as a string
>> datestr(datenum(2001,12,1))  % first convert Dec 1,2001 to
a number, then convert it to a string.
```

Now, we can use `badi` and `datestr` to get the dates of all the missing data in our temperature time series:

```
>> bad_dates=datestr(time(badi))

bad_dates =

16-Apr-1997
16-May-1997
16-Jun-1997
16-Oct-2004
16-Nov-2004
16-Aug-2007
16-Aug-2009
16-Sep-2009
16-Oct-2009
16-Nov-2009
```

Other logicals used for indexing:

You can also use some of these operators, which return logicals, for indexing; these return 1 for true and 0 for false:

`isnan` is a NaN
`isinf` is infinite (Inf or –Inf)
`isempty` is an empty matrix
`isprime` is a prime number

```
e.g.
>> a=[1 3 nan 9 nan 11]
>> ind=find(isnan(a))  % returns the index of all nan values
in a
```

Math with Matlab

Element-by-element math

It is straightforward to multiply, add, or divide each element of an array by a scalar. (We did this yesterday.)

```
>> x=rand(2,3,4);   % a random 3-dimensional array
>> y=2*x-1 ;        % this multiplies each element of x by 2
and subtracts 1
```

Things get more complicated when you are performing mathematical operations between two arrays. **Multiplication, division, and exponential operators have matrix algebra meanings** – and Matlab will do these by default when it sees a `*` `/` or `^` between two arrays.

To tell Matlab to perform element-by-element operations, add a period (.) before the operator symbol:

```
>> a=rand(2,3);          % a 2x3 matrix
>> b=[9 9 9;-8 0 1];     % another 2x3 matrix

>> c=a.*b  % multiply each element of a by the corresponding
element of b

>> c=a.^b  % raise each element of a to the power of the
corresponding element of b


>> c=a./b  % divide each element of a by the corresponding
element of b
```

Element-by-element operations require both arrays to have the same dimensions.
What if you forget? Usually, you just get an error, because the dimensions are incorrect for performing matrix math. E.g.:

```
>> a*b
```

produces

```
Error using  *
Inner matrix dimensions must agree.
```

But if the dimensions of the matrices happen to permit matrix math, you can get an unexpected result. E.g.

```
>> [1 1]*[1; 1]
ans =

     2
```

Matlab doesn't require a "." when you're dividing or multiplying an array with a scalar:

```
>> a/100
>> a*30
```

But if you put in one to be safe, it doesn't cause problems:

```
>> a./100
>> a.*30
```

Matrix math, briefly

By default, the `*` `/` and `^` commands are used for matrix algebra. If matrix `A` has dimensions m x n and matrix `B` has dimensions n x p, their product `A * B` is m × p. (Recall that the element of `A * B` in the ith row and jth column is the sum of the products of the elements from the ith row of `A` times the elements from the jth column of `B`.)

Symbolic algebra

Matlab can solve symbolic expressions. First, define the symbolic variables using **`syms:`**

```
>> syms x y   % defines symbolic variables x and y
```

Then you can evaluate expressions symbolically, e.g.

```
>> (x + y)*(x + y)

ans =

(x + y)^2
```

Simplify expressions using `simplify`:

```
>> simplify((x-y)^2+2*x*y)

ans =

x^2 + y^2
```

More useful built-in functions:
(taken from Matlab basics notes by Padmanabhan Seshaiyer)

Elementary trigonometric functions and their inverses
`sin, cos, tan, sec, csc, cot, asin, acos, atan, asec, acsc, acot`

Elementary hyperbolic functions and their inverses
`sinh, cosh, tanh, sech, csch, coth, asinh, acosh, atanh, asech, acsch, acoth`

Basic logarithmic and exponentiation functions
`log, log2, log10, exp, sqrt, pow`

Basic Statistical functions
`max, mean, min, median, std, var, sum`

Basic complex number functions
`imag, real, i, j, abs, angle, cart2pol`

Basic data analysis functions
`fft, ifft, interpn, spline, diff, del2, gradient`

Basic logical functions
`and, or, xor, not, any, all, isempty, is*`

Basic polynomial operations
`poly, roots, residue, polyfit, polyval,conv`

Function functions that allows users to manipulate mathematical expressions
`feval, fminbnd, fzero, quad, ode23, ode45, vectorize, inline, fplot, explot`

Basic matrix functions
`zeros, ones, det, trace, norm, eig`