Matlab bootcamp – Class 3

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

Aside: characters and strings

So far, we have just considered variables that are numeric. Variables can also be *characters* and arrays of characters, or *strings.* Characters and strings are defined using single quotation marks:

```
>> g='h'
```

creates a variable named `g`, whose value is the letter h.

```
>> k='my string'
```

creates a variable named `k` whose value is the string of characters 'my string' ; `k` has length 9 (spaces are also characters)

You can combine strings just like you would combine matrices, using square brackets and separating values with commas (though commas are optional):

```
>> newstring=['this', 'is' , 'my','string']
     newstring =

     thisismystring
```

Note that we haven't included any spaces , and so the output doesn't contain spaces. You have to include them explicitly:

```
>> stringwithspaces=['this', ' is' , ' my',' string']

     stringwithspaces =

     this is my string
```

You can combine string variables, also:

```
>> str1='my'
>> str2='string'
>> str3=[str1, ' ' str2]
```

(note that the `' '` inserts a space between `str1` and `str2`)

```
str3 =

my string
```

Numbers can also be strings – but be careful here. You still have to surround them with single quotation marks to indicate that they are numbers:

```
     >> b='6'    % defines the variable b, equal to the character '6'
     >> b=6      % defines the variable b, equal to the number 6
```

You can do math with numbers, but not with characters!

* if you are combining characters with numbers to form a string, you must convert the numbers to strings using the function `num2str`:

```
>> newname=['kyla' num2str(33)]
newname =

kyla33
```

If you don't convert the number to a string, Matlab will substitute the character corresponding to the ASCII character codes, which are defined for integers 0 to 127:

```
>> newname=['kyla' 33]  % forgot the num2str command

newname =

kyla!
```

...33 corresponds to an exclamation point in ASCII.

If statements

Syntax:

```
if expression
     statements
end
```

This evaluates an expression, and executes a group of statements if the expression is true. An evaluated expression is true when the result is not empty and contains all nonzero elements (logical or numeric).

The expression (`expression`) can be anything that generates a logical or numerical value. Particularly useful are relational operators (e.g. > , < , >= , <= , == , ~=) which produce a 1 if they are true. E.g.

```
>> if a>0        % the expression a>0 returns a 1 if a>0
>>   disp(a);    % display the value of a
>> end
```

Note that the indentation and spacing is optional, and just used to make things more clear. It works equally well to write this:

```
>> if a>4; disp(a); end
```

...This can be useful if you are typing directly into the command line.

You can include multiple parts to an expression by using the logical operators **&&** (and), **||** (or), **~** (not). E.g.

```
>> if a>b && b==2     % if a greater than b AND b equals 2,
>>     c=a+b;         % set c equal to a+b
>> end
```

`expression` can also have a numerical value. E.g.

```
>> if 1
>>   a=0;
>> end
```

1 is *always* nonzero, so the statement `a=0` will always be executed.

If *any* of the values in the evaluated expression are zero, the expression is false:

```
>> B=[1 2 3 4 0];
>> if B
>>     disp(B);
>> end
```

The `statement` will not be executed because `B` contains a zero, so the `expression` if false.

else and elseif

By adding the word `else`, the first set of statements (`statements1`) will be executed if `expression` is true. The second set of statements (`statements2`, below `else`) will be executed if `expression` is false.
Here is the syntax:

```
if expression
     statements1
else
     statements2
end
```

For example:

```
>> b=1;
>> if ~isempty(b) && b>2
>>     c=b;
>> else
>>     c=0;
>> end
```

The expression is only true if `b` is not empty *and* `b` is greater than 2; since `b` is only 1, Matlab does not evaluate the first statement (`c=b`), and instead moves onto the next statement (`c=0`).

Adding `elseif` tells Matlab to evaluate a series of expressions and only perform the statements that follow the first true expression it encounters. Here is the syntax:

```
if expression1
     statements1
elseif expression2
     statements2
elseif expression3
     statements3
end
```

`elseif` is like `else`, but it also has a condition attached to it. You can have as many `elseifs` as you need. Matlab will go sequentially through the list of expressions until it finds one that is true. When it does, it will only execute those statements after that `elseif`. It will then skip the remainder of the commands and go to `end`. For example:

```
>> if x==3
>>     u=10
>> elseif x==5
>>    u=30
>> else
>>     u=20
>> end
```

Loops: for and while

Using a *loop*, you can tell Matlab to execute a set of commands over and over again while a certain condition is met. There are two main ways of doing this: `for` loops and `while` loops.

for loops

syntax:

```
for index = values
     statements
end
```

`values` is a vector (list of numbers), and is often defined with a colon, e.g.

```
>> for k=1:10
>>     disp(k);
>> end
```

This runs through the values 1 to 10, and displays `k` each time.

Again, the indentation and line breaks are just for readability (although you have to include a semicolon after each line in `statements` so that Matlab knows you are starting a new line); you can equivalently type

```
     >> for k=1:10, disp(k); end
```

Loops are can be used for many different purposes. Any time you seem to be typing the same thing over and over, consider using a loop instead. For example, say you want to plot a bunch of data, each on its own figure.

```
     >> x=1:1000;              % define a vector x
     >> for i=1:5              % run through i=1,2,3,4,5
          y=randn(size(x));    % this creates a vector the same size
                               % as x and fills it with
                               % normally-distrbuted random numbers
          figure(i);           % open figure i
          clf;                 % clear the figure (in case
                               % something is already plotted there)
          plot(x,y,'.')        % plot y against x (using dots)
          disp(mean(y))        % print the mean value of y to the
                               % screen
     >> end                    % end of the for loop
```

Another example: loading a bunch of files that have a similar name: data1.mat, data2.mat, data3.mat, etc..., (don't forget to use num2str to convert the number to a string):

```
     >> for k=1:10
          filename=['data' num2str(k) '.mat'];
          load(filename);
     >> end
```

A long example: load Argo profile data. For each profile, calculate some basic statistics, and plot each temperature and salinity profile in a new figure that you then save.

```
clear
whos('-file','argo_data.mat'); % lists the variables in the file
load('argo_data.mat');         % load the file
figure(1)
clf
plot(lon,lat,'.-')              % plot lon vs lat
title('argo float trajectory')
xlabel('lon')
ylabel('lat');

% T and S are a 20x5 matrix containing temperature and salinity data at 20
% depths (given by the vector z) for 5 profiles.
%
% so, each column of T and S contains the data from a single profile.
% run through each profile and plot it on a new  figure, and save the data
% to a new file:
for j=1:length(lon)     % one profile per lon/lat pair
    figure(j)            % open figure j
    clf                  % clear the figure, in case something is already plotted there

    % first, plot temperature as a function of depth
    subplot(1,2,1)  % create a new subplot
    plot(T(:,j),z,'.-');  % plot temperature for the ith profile against z
    axis ij         % this inverts the y axis (so depth can go from 0 to 200)
    title(['Temperature for profile ' num2str(j)])
    ylabel('depth, m')
    xlabel('temperature, degC')
    xlim([15 30]);  % this will set the range for the x-axis...
                    % note: it can be helpful to use the same x range for each
                    % plot so that we can more easily compare them by eye.
    ylim([0 200]);  % range for the y-axis

    % next, plot salinity
    subplot(1,2,2)  % a second subplot beside the first one
    plot(S(:,j),z,'r.-');  % plot salinity for the ith profile in red
    axis ij         % this inverts the y axis (so depth can go from 0 to 200)
    title(['Salinity for profile ' num2str(j)])  % you need to use num2str when
                                                 % including numbers in titles,
                                                 % because they are strings

    ylabel('depth, m')
    xlabel('salinity, psu')
    xlim([36.0 37.5]);  % a different range for salinity than for temperature
    ylim([0 200]);  % range for the y-axis

    % calculate some stuff for each profile:
    profile_mean_T(j)=mean(T(:,j));
    profile_max_T(j)=max(T(:,j));

    % save the figure as a PDF:
    print('-dpdf',['argo_profile_' num2str(j)]);
end
```

while loops

While loops execute a set of statements over and over while a given condition is true.

```
while expression
     statements
end
```

Whereas `for` loops execute an explicit number of times (e.g. a loop beginning `for i=1:10` will execute 10 times), `while` loops execute statements indefinitely as long as expression remains true. To specify `expression`, use the same rules as you did with `if` statements: anything that generates a logical or numerical value. Particularly useful are relational operators (e.g. > , < , >= , <= , == , ~=) which produce a 1 if they are true. E.g.

```
>> g=21;
>> while g>10
     g=g-1;
     disp(g);
>> end
```

Once it starts executing statements within the `while` loop, Matlab *will not stop* until `expression` is no longer true!!! `while` loops should be written so that the variables in expression change when `statements` are executed – otherwise you can get stuck in an infinite loop! E.g.

```
>> count=0;
>> g=10;
>> while g>1    % g doesn't change inside the loop, so the condition
g>1 will always be true and the loop will execute indefinitely
     count=count+1;
     disp(count);
>> end
```

This is where the **CTRL+c** command comes in handy!

Other useful commands in loops

A `break` statement in a `for` or `while` loop will terminate the loop and move immediately to the following statement. Building on the previous example:

```
>> count=0;
>> g=10;
>> while g>1    % g doesn't change inside the loop, so the condition
g>1 will always be true and the loop will execute indefinitely
     count=count+1;
     disp(count);

     if count==10   % if count reaches 10, break out of the while
loop
          break
     end

>> end
```

A `continue` statement in `for` or `while` loop will skip the rest of the instructions within the loop and begin the next iteration.

Basic statistics

Basic statistics can be easily computed with Matlab. For example

```
>> x=rand(1000,1);

>> mean(x)
>> std(x)
>> var(x)
>> median(x)
>> mode(x)
>> max(x)
>> min(x)
>> range(x)
```

If x contains `NaN` values, **`mean`**, **`std`**, **`var`**, and **`median`** will all return `NaN`. You can deal with this in a few ways:

- using the functions **`nanmean`**, **`nanstd`**, **`nanvar`**, and **`nanmedian`**. However, these functions are in the "statistics toolbox" and so are not included in the basic version of Matlab, so you may have to devise your own methods:
- remove the `NaN`s first using indexing:

```
>> nani=find(isnan(x));
>> x(nani)=[];
```

** setting matrix elements to be [] removes them from the matrix!

- use indexing to only compute non-nan values – e.g:

```
>> not_nan_index=find(~isnan(x)); % index of non-nan elements of x
>> xmean=mean(x(not_nan_index));  % only compute the mean for the elements in not_nan_index, i.e. the non-nan elements
```

Basic fitting

You may want to fit a line or a curve to your data. Matlab has several functions to do basic curve fitting, all of which include some sort of uncertainty/error estimation. These include:

```
regress - multiple linear least-squares regression
robustfit - "robust" linear least-squares regression
polyfit - fits a polynomial function using least-squares regression
```

There is a few ins and outs to learning how and when to use these functions; I suggest reading the documentation so that you get the syntax correct and properly understand the output. Also, Dan Rudnick goes into fitting techniques in quite some detail in his statistics class (SIO221B), which is very useful. Here are a couple of basic examples to get you started:

Create a line with a known slope and intercept, plus some noise, and use a linear regression to fit a line. E.g. $y = 2x - 10 + \varepsilon$, where ε is random noise.

```
>> x=(1:10)';  % x is a 10x1 column vector (column vectors are needed
for the "regress" function)
>> y=[-5 -1 1 7 8 1 15 -3 50 33]'; % y has the same size as x
>> plot(x,y,'.');
% we want to fit a line to the data in "y"
% to determine the intercept, it is necessary to include an extra
column of ones in the fit:
>> xfit=[x(:) , ones(size(x(:)))];
% xfit is a 10x2 matrix consisting of the column vector x (writing
x(:) ensures that x will be a column vector) and a column of ones
that is the same length as x.
>> [b bint] = regress(y,xfit)   % note that y is specified first
b =

    4.2182
  -12.6000



bint =

    0.8928    7.5436
  -33.2334    8.0334
```
`b` gives the regression coefficients (slope, intercept – i.e. b(1)=slope, b(2)=intercept) and `bint` gives the 95% confidence intervals for each coefficient.

The `robustfit` function is a little easier, because you don't have to add the column of ones, as the intercept is included in the fit by default:

```
>> [b2,stats] = robustfit(x,y)  % note that x is specified first
b2 =

  -10.2021
    3.9468
```
`b` gives the regression coefficients (intercept, slope – note that this is the reverse order as what `regress` returned); `stats` gives some statistics for the fit.
Note that `robustfit` returns the intercept and then the slope – the opposite order of `regress`.

Let's see how well these fits did by reconstructing the line: (i.e. using the equation of a line to compute y for each value of x)

```
>> y_reconstruct1=b(1)*x + b(2);  % recall, b(1)=slope
>> y_reconstruct2=b2(2)*x + b2(1); % recall, b2(1)=intercept
>> hold on
>> plot(x,y_reconstruct1,'r--')
>> plot(x,y_reconstruct2,'k:')
```

An example of `polyfit`: create a line with a known slope and intercept, plus some noise, and use a linear regression to fit a line. E.g. $y = 4x^2 - 3x + 4 + \varepsilon$, where ε is random noise

```
>> x=(1:100)';  % x is a 100x1 column vector
>> y=4*x.^2 - 3*x + 4;  % y is a function of x
>> y=y+randn(size(y)); % add some random noise to y
>> figure(1),clf
>> plot(x,y,'.');
>> P = polyfit(x,y,2); % the third argument gives the degree of the
polynomial: in this case, 2 (i.e. x^2 is the highest power)


P =

    4.0000   -2.9987    3.8829
```

This is interpreted the coefficients from the highest order (order 2) down to order zero, ie. $y = 4.0000x^2 - 2.9987x + 3.8829$, which is a good estimate of our function.

Matlab has a handy function to reconstruct the polynomial: `polyval` , which requires a vector `P` as its argument, where `P` specifies the coefficients for a polynomial whose order is equal to the length of `P` plus 1 (that is, the same format that is returned by the `polyval` function).

```
>> y_reconstruct_poly=polyval(P,x)
>> hold on
>> plot(x,y_reconstruct_poly,'r--','linewidth',2) % make the line
thick so we can see it...
```

Another `polyval` example: compute $y = x^4 + 3x^2 + 2x$ for a vector x ranging from -100 to 100:

```
>> x=-100:1:100;
>> P=[1 0 3 2 0];
```

The values in `P` indicate the coefficients for x^4, x^3, x^2, x^1, and x^0, respectively. Zeros are used when a given power doesn't exist in the function (e.g. x^3 doesn't appear in our equation, so the second value in `P`, corresponding to the coefficient in front of x^3, is zero).

```
>> y=polyval(P,x); % evaluates the polynomial at each value of x
>> figure(3),clf
>> plot(x,y)
```