MATLAB Tutorial Chap 3

by: David Groppe

Topic 1 – Yet more kinds of variables

Last week you were introduced to two basic types of MATLAB variables: strings (or "character arrays") and numeric arrays (e.g., scalars, vectors, and matrices). In this topic, you'll learn how to use a couple of other very handy variable classes: cell arrays and structure arrays.

1.A Cell Arrays:

A sometimes aggravating trait of MATLAB is that it has been built in way that is great for processing numbers but isn't so great for processing text. For example, say we would like to store a set of filenames that we're going to use in an analysis: *odbl1.edf*, *odbl2.edf*, and *odbl3.edf*. We could do that with a matrix of characters:

```
>> fnames=['odbl1.edf'; 'odbl2.edf'; 'odbl3.edf']

fnames =

odbl1.edf
odbl2.edf
odbl3.edf
```

We could access the matrix just like a numeric matrix. For example, to extract the second filename we would enter:

```
>> fnames(2,:)

ans =

odbl2.edf
```

This strategy fails though if some of the filenames differ in length. For example, if we replace *odble3.edf* with *odbl33.edf*:

```
>> fnames=['odbl1.edf'; 'odbl2.edf'; 'odbl33.edf']
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

To store a set of strings of different length, we use a *cell array*. Cell arrays are accessed with curly brackets (i.e., { and }) and can contain things of different length. For example:

```
>> clear fnames
>> fnames{1}='odbl1.edf';
>> fnames{2}='odbl2.edf';
>> fnames{3}='odbl33.edf';
>> fnames
```

```
fnames =

    'odbl1.edf'    'odbl2.edf'    'odbl33.edf'
```

The elements of a cell array are accessed pretty much just like those of numeric or cell arrays. For example, to get the second filename we would enter:

```
>> fnames{2}

ans =

odbl2.edf
```

EXERCISE

1.1) Write a *for* loop to create a cell array called *fnames* that contains the following file names:

- odbl11.edf
- odbl12.edf
- odbl13.edf
- odbl15.edf
- odbl16.edf
- odbl18.edf

1.B Struct Arrays:

Often one would like to store information of different types together (e.g., numeric and string). For example, you might want to store someone's EEG data along with a string indicating the code for the participant who generated that data and text indicating how that data's been processed (e.g., have any trials been rejected due to artifacts?). You can make such a complicated variable by using a *struct array*. To illustrate, copy the file *prsent34demo.set* to your current working directory from the wiki.

It is an EEGLAB "set file", which contains the EEG data (and associated information) from one participant of an experiment. Although the extension of the file is *.set*, it is really a MATLAB *.mat* file and can be loaded like so:

```
>> clear
>> load prsent34demo.set -MAT
```

The "-MAT" extension tells MATLAB that the file you're loading is really a *.mat* file, even though it doesn't have a *.mat* extension. Now, if you look at the variables in your workspace you'll see a struct array called *EEG*.

```
>> whos
  Name       Size                 Bytes  Class     Attributes

  EEG        1x1                8119896  struct
```

Now look at the contents of *EEG*:

```
>> EEG

EEG =

              setname: 'prsent34demo'
             filename: 'prsent34demo.set'
             filepath:
'/Users/dgroppe/Documents/MATLAB/MK2MAT/'
              subject: 'prsent34 06/23/09 '
                group: ''
            condition: ''
              session: []
             comments: 'prsent34 CALS'
               nbchan: 31
               trials: 209
                 pnts: 256
                srate: 250
                 xmin: -0.1000
                 xmax: 0.9200
                times: [1x256 double]
                 data: [31x256x209 single]
               icaact: []
              icawinv: [31x31 double]
            icasphere: [31x31 double]
           icaweights: [31x31 double]
         icachansind: [1x31 double]
            chanlocs: [1x31 struct]
          urchanlocs: []
            chaninfo: [1x1 struct]
                 ref: 'bimastoid(mostly)'
               event: [1x418 struct]
             urevent: [1x209 struct]
    eventdescription: {1x12 cell}
               epoch: [1x209 struct]
    epochdescription: {}
              reject: [1x1 struct]
               stats: [1x1 struct]
             specdata: []
          specicaact: []
          splinefile: ''
       icasplinefile: ''
              dipfit: []
             history: [1x1845 char]
               saved: 'yes'
                 etc: [1x1 struct]
             bindesc: {1x5 cell}
            cal_info: [1x1 struct]
```

```
              condesc: {'SameVSdifferent Phonemic
Restoration Task'}
       crw2set_params: [1x1 struct]
              datfile: []
           icfeatures: [1x31 struct]
              icfreqs: [257x1 double]
             iclabels: {1x31 cell}
            icspectra: [31x257 double]
    rawtrials_per_bin: [74 61 56 6 12]
```

Each line of the contents of *EEG* is called a "field." For example, the original location of the set file (on my laptop) is stored in the field "filepath." To access the contents of a field of a struct array, simply type in the name of the struct array, a period, and the name of the field:

```
>> EEG.filepath

ans =

/Users/dgroppe/Documents/MATLAB/MK2MAT/
```

Let's update the file path to the file's new location:

```
>> EEG.filepath=pwd;
```

Check to make sure that command worked.

Struct arrays can contain other struct arrays and thus are a very flexible (and potentially complex) way of storing lots of information together. For example, the information about each electrode is stored in the field *chanlocs*:

```
>> EEG.chanlocs

ans =

1x31 struct array with fields:
    theta
    radius
    labels
    sph_theta
    sph_phi
    sph_theta_besa
    sph_phi_besa
    X
    Y
    Z
    ref
```

chanlocs is also a struct array composed of 31 elements. To access the first element (corresponding to the first electrode) enter:

```
>> EEG.chanlocs(1)

ans =

              theta: -22
             radius: 0.7200
             labels: 'lle'
         sph_theta: 22
           sph_phi: -39.6000
    sph_theta_besa: -129.6000
      sph_phi_besa: -68
                 X: 0.7144
                 Y: 0.2886
                 Z: -0.6374
               ref: 'bimastoid'
```

We access the fields of this stuct array by adding on another period and field name. For example, we could change the name of the first electrode by doing this:

```
>> EEG.chanlocs(1).labels='Left lower eye';
```

Check to make sure that worked.

EXERCISE

1.2) Write a *for* loop that will display the names of all 31 electrodes stored in the *EEG* variable used in the previous section.

Topic 2 – Functions and global variables

Last week you were introduced to the concept of a MATLAB "m-file" and you learned how to make m-file "scripts" that executed simple programs. In this topic, you'll learn about a different type of m-file, "functions." In addition, you'll learn about how the variables in a function are generally "local" to that function (i.e., not observable in your command-line workspace or by other functions) and how to make "global" variables that are potentially shared by your command-line workspace and all MATLAB functions.

2.A Functions:

2.A.1 A simple function

Use the MATLAB editor to create a new m-file called *simple_script.m*. Then add the following lines:

```
in=4;
increment=2;
out=in+increment;
```

Now run the script. You should have the following variables in your workspace:

```
>> whos
  Name            Size              Bytes  Class
Attributes

  in              1x1                   8  double
  increment       1x1                   8  double
  out             1x1                   8  double
```

And the variable *out* should equal 6:

```
>> out

out =

     6
```

This is exactly what would have resulted if you had typed each line of the script into the command line one at a time. In other words, running a script is no different than entering text directly into the command line.

Let's try something different. Create an m-file called *simple_function.m* and type the following text:

```
function out=simple_function(in)
% function out=simple_function(in)
% Adds 2 to the input

increment=2;
out=in+increment;
```

You've just created a "function." A function differs from a script in that a function requires input variables and it can output new variables. For example, we can input the value *4* into your function and it outputs *6*:

```
>> clear
>> simple_function(4)

ans =

     6
```

However, if you tried to execute the function without any input variables (i.e., like a script), you would get an error:

```
>> simple_function
??? Input argument "in" is undefined.

Error in ==> simple_function at 6
out=in+increment;
```

Also notice that the header for a function m-file provides the help documentation just like it did for a script m-file.

```
>> help simple_function
  function out=simple_function(in)
  Adds 2 to the input
```

Now look at the contents of your command workspace:

```
>> whos
  Name        Size              Bytes  Class     Attributes

  ans         1x1                   8  double
```

Note that none of the variables in *simple_function.m* are in your workspace (e.g., there's not a variable called *increment* like there was when we ran *simple_script.m*). This illustrates another key difference between functions and scripts. Functions have their own "workspace" that is local to the function. When a function is called, a workspace is created that is private to that function. Other functions can't access the variables in it and you can't access them from the command line. When the function finishes, its private workspace is erased. Let's use the MATLAB debugger to illustrate. Use your mouse or the command *dbstop* to put a break point at the last line of *simple_function.m* (i.e., "out=in+increment;")[1]. Now call the function again:

```
>> simple_function(4)
```

The debugger should pause the function right before the last line and the command line should look like this:

```
K>>
```

This indicates that the debugger is active and that you are in a script or function. Since we're in a function, we can now access its local workspace.

```
K>> whos
  Name            Size              Bytes  Class
Attributes

  in              1x1                   8  double
  increment       1x1                   8  double
```

[1] This was introduced in Week 1's tutorial materials (*matlab_chap1.pdf*, pg 11 to be exact). The easiest way to add a breakpoint is with a mouse. In the MATLAB editor window, click on the dash to the right of the line number where you want to pause the function. The dash should become a red dot, indicating that a breakpoint is now there. If the dot is grey, that means the function has been changed and needs to be resaved before you can use the debugger.

Un-pause the function with the *dbcont* command:

```
K>> dbcont
```

Now you're back in the command line workspace and the function's workspace no longer exists:

```
>> whos
  Name        Size              Bytes  Class     Attributes

  ans         1x1                   8  double
```

2.A.2 Global variables

It's possible to make variables that can be shared across multiple workspaces by declaring them "global." For example,

```
>> global msg
>> whos
  Name        Size              Bytes  Class     Attributes

  ans         1x1                   8  double
  msg         0x0                   0  double    global
```

Now we can store something in *msg*:

```
>> msg='Here I am!';
```

And we can modify *simple_function.m* to use what's stored in *msg*:

```
function out=simple_function(in)
% function out=simple_function(in)
% Adds 2 to the input

global msg %this line makes "msg" accessible to this
           %function's local workspace
disp(msg);

increment=2;
out=in+increment;
```

Try running the function:

```
>> simple_function(3)
Here I am!

ans =

     5
```

Using global variables is particularly helpful when dealing with huge variables because it

can save memory (i.e., without a global variable, multiple slightly different versions of the variable may be created in different function workspaces). Indeed some EEGLAB and MATLABmk functions require that EEG datasets be stored in global variables. *ic_prop.m* is one such function. It plots the properties of an *EEG* variable's independent components. The commands below will plot the properties of the first independent component of *prsent34demo.set*, which represents blink activity.

```
>> global EEG
>> load prsent34demo.set -MAT
>> ic_prop(1);
```

2.A.3 Functions with multiple inputs and outputs

Let's make our simple function a little bit more complicated by adding additional inputs and outputs. Replace the previous version of *simple_function.m* with the following:

```
function [added, mltpld]=simple_function(in,fctr)
% function [added, mltpld]=simple_function(in,fctr)

added=in+fctr;
mltpldd=in*fctr;
```

We could call this function in a couple of ways. The first way is similar to what we used previously for *simple_function.m*, except now we have two inputs separated by commas.

```
>> simple_function(3,2)

ans =

     5
```

When you enter that command, MATLAB finds the function *simple_function.m*, passes the inputs 3 and 2 to it, and stores the first output of the function in the variable *ans* (i.e., the output that corresponds to the variable *added* in *simple_function.m*). This is unsatisfactory though since we don't know what the value of the second output is (i.e., the output that corresponds to the variable *mltlpd* in *simple_function.m*).

To be able to store both output values we need to call the function in a different way by doing something like the following:

```
>> [out1, out2]=simple_function(3,2)
```

The outputs of the function will now be stored in the variables *out1* and *out2*. Note that to specify multiple output variables for a function, we use square brackets (instead of parenthesis) and separate the variables with commas (actually the commas are optional, but they make the code slightly more readable).

EXERCISE

2.1) Write a function that takes a participant's number and outputs a edf and log file name. For example:

```
>> [edffile, logfile]=fnames(8);
>> edffile
edffile =

odbl8.edf
>> logfile
logfile =

odbl8.log
```

Topic 3 – Loading data from text files & variations on optional function arguments

One way to import data into MATLAB from other file formats (e.g., an Excel spreadsheet) is to store the data as a tab-delimited text file and to load it into MATLAB using the *textread* function. For example, the text file *odbl_summ.txt* on the lab wiki contains the median reaction times and proportion correct responses from 16 participants in two target detection tasks (coded as "LC" and "SC").

The target classes were designed to be much easier to discriminate in one task than the other. The first and third columns contain the reaction times in the easier and harder target detection tasks respectively. The second and fourth columns contain the accuracy (proportion of correct responses) in the easier and harder target detection tasks respectively. Each row corresponds to a different participant.

To load the data into MATLAB copy *odbl_summ.txt* to your current working directory and type the following into the MATLAB command line:

```
>> [easy_rt easy_acc hard_rt hard_acc] ...
=textread('odbl_summ.txt','%f %f %f %f','headerlines',1);
```

A few things to note about this command:

- We used three periods at the end of the first line to indicate that the command continues on the next line. You can use three periods line-after-line to make commands continue for as many lines as you wish.
- The string *headerlines* followed by *1* tells MATLAB that the first line of the file odbl_summ.txt is part of the file's header and it will be ignored. We'll talk more about such pairs of optional input arguments later in this topic.
- Each column of odbl_summ.txt is being stored in a separate output variable. *easy_rt* contains the first column. *easy_acc* contains the second column, etc…
- The first argument of the function call needs to be a string. If you had entered *odbl_summ.txt* instead of *'odbl_summ.txt'*, MATLAB would have interpreted *odbl_summ.txt* as a variable and given you an error since no such variable exists in your workspace.
- The second argument of the function call tells MATLAB what type of information is stored in each column. *%f* indicates that a column contains floating point numbers. *%s* would indicate that a column contains strings. In our case we use *'%f %f %f %f'* because we have four columns of floating point numbers. Enter *help textread* to see the full lists of data types *textread* recognizes.

Let's look at the reaction times in the two tasks:

```
>> [easy_rt hard_rt]

ans =

   392   488
   430   560
   370   468
   478   536
   432   476
   484   548
   436   528
   436   506
   492   588
   444   576
   468   540
   404   474
   442   512
   544   668
   454   498
   408   476
```

It looks like everyone was faster on average in the easier task. Here's an easier way to see if that's true:

```
>> [easy_rt hard_rt easy_rt-hard_rt]

ans =

   392   488   -96
   430   560  -130
   370   468   -98
   478   536   -58
   432   476   -44
   484   548   -64
   436   528   -92
   436   506   -70
   492   588   -96
   444   576  -132
   468   540   -72
   404   474   -70
   442   512   -70
   544   668  -124
   454   498   -44
   408   476   -68
```

Yup, people were always faster on the easier task by around 50 to 125 milliseconds.

The next thing we'd like to do is to evaluate the statistical significance of that difference. Since it's a within-subjects comparison, a paired sample t-test would be an

appropriate way to do this. Let's find out if MATLAB has a built-in function for paired sample *t*-tests:

```
>> lookfor ttest
mattest - Two-sample t-test for identifying differentially
expressed genes
barttest - Bartlett's test for dimensionality of the data
in X.
ttest - One-sample and paired-sample T-test.
ttest2 - Two-sample T-test with pooled or unpooled variance
estimate.
```

Bingo! *ttest.m* is exactly what we need. Now let's find out how to use it:

```
>> help ttest
```

ttest.m is a somewhat flexible function which can be passed up to four inputs, but only one input is required. For example, to analyze our data we could enter the following:

```
>> [h p ci stats]=ttest(easy_rt,hard_rt)
```

However, by default, this function executes a two-tailed test. Since we have a priori reason to believe that people should be faster on the easier task, we actually want to do a one-tailed test (i.e., our alternative hypothesis is that RT should be less for *easy_rt*). We can do this by passing additional inputs to *ttest.m*:

```
>> [h p ci stats]=ttest(easy_rt,hard_rt,.05,'left')
```

The third argument specifies the alpha level and the fourth argument specifies the tail of the test. This illustrates one way MATLAB deals with optional function inputs. Each input is given a particular meaning (e.g., the third input is the alpha level) and you can ignore the lattermost inputs if you choose. However, you can't ignore inputs between the first and last one you specify. For example, when using *ttest.m*, to specify the tail of the test we have to also specify the alpha level. If we tried to enter the following:

```
>> [h p ci stats]=ttest(easy_rt,hard_rt,'left')
```

We would get an error because MATLAB would interpret *'left'* as our alpha level.

An alternative way of dealing with optional inputs is illustrated by the function *corr.m*, which computes the correlation between two variables.

```
>> [r p]=corr(easy_rt,hard_rt)

r =

    0.8502


p =

   3.0231e-05
```

By default, the *p*-value returned by *corr.m* is from a two-tailed test of the null hypothesis of no correlation. However, we had a priori reason to believe that RT should be positively correlated across the tasks. Thus we should do an upper tailed test.

```
>> [r p]=corr(easy_rt,hard_rt,'tail','right')
```

To specify the tail of the test, we needed to pass two input arguments. The first, the string *'tail'*, tells MATLAB which input variable should be set equal to the next argument, *'right'*. This is very different than the way *ttest.m* works. *ttest.m* knows which input corresponds to the desired tail of the test by THE ORDER OF THE INPUTS. Again, *corr.m* knows which input corresponds to the desired tail of the test BECAUSE IT IS PRECEDED BY THE STRING *'tail'*.

Let's add a second optional argument. One danger of using linear correlation is that it is highly influenced by outliers. A simple way to determine if our significant correlation between reaction times is due to outliers is to compute a "rank" correlation coefficient, which is very insensitive to outliers. Kendall's *tau* is a popular correlation coefficient and can be computed thusly:

```
>> [r p]=corr(easy_rt,hard_rt,'tail','right', ...
'type','kendall')

r =

    0.6555


p =

   2.5662e-04
```

The correlation is still highly significant and is clearly not an artifact of outliers.

Note that the order of these two sets of optional inputs doesn't matter. *corr.m* does the same thing as above if we enter:

```
>> [r p]=corr(easy_rt,hard_rt,'type','kendall', ...
'tail','right')
```

EXERCISE

3.1) Perform a *t*-test on the accuracy data across the two target detection tasks. Is there any evidence of a speed-accuracy trade off?

3.2) The built-in MATLAB function *corrcoef.m* provides pretty much the same information as *corr.m* but it also returns confidence intervals of the estimated correlation coefficient. Compute the correlation between the accuracy data across the two target detection tasks and provide upper and lower bounds of the 95% confidence interval on that estimate.

Chap 3 Summary:

You should have learned the following from this week's topics:

- What cell and struct array variables are and how to use them
- The difference between a local and global variable and how to make a variable global
- The difference between a "function" m-file and a "script" m-file and how to make function m-files
- How to load data into MATLAB from a text file using the *textread* function
- The two different ways MATLAB deals with optional input arguments