Matlab Tutorial

A Jump Start into Matlab
by

Klaus Moeltner
Department of Resource Economics
University of Nevada, Reno (UNR)
Mail Stop 204 / Reno, NV 89557-0105
phone: (775) 784-4803
email: moeltner@cabnr.unr.edu
web: http://www.ag.unr.edu/moeltner

August 15, 2007

Specifically designed as a 1/2 day "bare-bones" introduction to Matlab for beginning Graduate Econometrics students

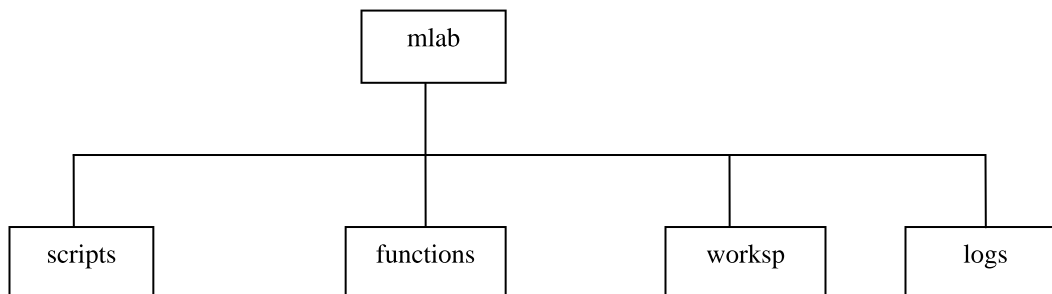
Comments & suggestions are always welcome!

Tutorial web site:
http://www.ag.unr.edu/moeltner/matlab_stuff.htm

Module I. Work Environment

Folder Structure

Create the following folder environment on your c:\ drive or any other path destination.



For example, for this tutorial, you could first create a folder **c:\mlab_tut\.** Then, within this folder, create the folder structure shown above.

Your main command scripts go into "scripts". Your sub-routines go into "functions". "Worksp" (short for "workspace") contains your data and other saved elements. Output goes into "logs".

Matlab Windows and Matlab Path

When you open Matlab you will first see the **command window**. This is where you can enter interactive commands.

File/new/m-file will open an **editor window**. This is where you write your programs (scripts and functions). We'll come back to this in a moment. To close the editor window click on the red "x" in the upper right hand corner. If you have multiple editor files open, close them individually by clicking on the specific "x" in the bottom bar of the editor window. By the way, scripts and functions are called "m-files" in Matlab jargon.

Desktop/workspace will open the **workspace window**, which shows you all elements currently in your workspace. For matrix elements, you can double-click on them & Matlab will open a separate spreadsheet showing the detailed contents of the matrix. To close the workspace window, simply click on the little "x" in the upper right hand corner.

File/set path will allow you to "set the path", i.e. to tell Matlab which folders to visit when looking for files. To add the entire folder environment created above to the path, simply click on "add with subfolders", then find your "mlab" folder created above in the browser window, click "OK" and "save". Your folder cluster will now be "on the path", at the very top.

Module II. A Basic Script with Programming Essentials and Random Data Generation

Open a new editor. Save the file as c:\mlab_tut\mlab\scripts\script1. A basic script will have the following structure:

- A comment (note) to yourself about the script
- set seeds for random draws (that way you will always be able to replicate your work precisely)
- set timer (that way you'll always know how long your program takes to run)
- open log file (tell Matlab where to send your output)
- Main script. Will likely contain elements such as:
  - load or generate data
  - call sub-routines (functions)
  - format output for your log file
  - save output elements to your workspace
- stop timer & capture run time for your log file
- close log file

Comments

Any line or cluster of elements in your editor starting with "%" will be interpreted as a comment and NOT executed as a command. Lets' start with the following comment:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is practice script 1 for the Matlab tutorial%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Set Random Number Seeds

You need to set seeds for the uniform and normal distribution. All other random numbers will be based on those. Example: Choose seed state "37" (any other number will do as well. FYI, I always use "37", so if you ever want to compare your results with mine and your program contains random draws, please do the same).

```
rand('state',37); % set arbitrary seed for uniform draws
randn('state',37); % set arbitrary seed for normal draws
```

Set timer

```
tic;  % start stop watch
```

Open log file:

```
[fid]=fopen('c:\mlab_tut\mlab\logs\script1.txt','w');
if fid==-1;
   warning('File could not be opened');
   break
else;
   disp('File opened successfully');
end;
```

The path after `fopen` shows where you would like your output to be sent. Note the apostrophes at the beginning and end of the file name. The 'w' tells Matlab to over-write any file by the same name in the same destination. In most cases this is what you'd want to happen. if not, change the "w" to 'a'. Matlab will then append (=add on to) the existing file. The `if - else - end` loop is optional, but useful. It

will immediately interrupt your program if the destination file can't be opened for any reason (most likely because you already have it open in, say, Word, or your path is wrong). The "`break`" command will stop the execution of your script.

Main script:
Let's generate some "data", run a basic OLS regression, and format the resulting output.

This is how you generate basic scalars: (note: Matlab IS case-sensitive)
```
n=1000; % set sample size
```
Note: In Matlab, each command generally ends with a semicolon (**;**). If you want to see elements you create in the command window while your script is running, simply omit the "**;**".

Generate a vector of "1"'s:
```
x1=ones(n,1);
```
Note: Elements in round parentheses usually refer to the *dimensions* of a matrix or vector. i.e *(number of rows, number of columns)*. Here we have n rows and 1 columns. "ones" is a built-in function generating "1"'s.

Generate a vector of random normals with mean -1.4 and standard deviation (std) 1:
```
x2=-1.4+randn(n,1);
```
Note: `randn` generates standard normal draws with mean 0 and std 1. By adding the "-1.4" we're moving the mean to the left. The `(n,1)` just indicates the dimensions of the resulting vector or matrix.

Generate another normal vector with mean 3 and std 2 (i.e. variance of '4").
```
x3= 3+2*randn(n,1);
```

Collect the three vectors into an "X" matrix:
```
X=[x1 x2 x3];
```
Note: Brackets [...] are used to combine numerical arrays (=scalars, vectors, matrices). The dimensions have to be compatible, of course. `[a b c]` combines elements horizontally (row dimensions must agree). `[a;b;c]` stacks elements vertically (column dimensions must agree).
Note: We could have skipped a few lines of script by defining X as:
```
X=[ones(n,1) -1.4+randn(n,1) 3+2*randn(n,1)];
```

Define the column dimension of X:
```
k=size(X,2);
```
The "`size`" command extracts a desired dimesnion for the element that appears first in parentheses (here: X). "2" calls for the column dimension. "1" calls for the row dimension.

Create a vector of coefficients ("betas").
```
b=[1.2 0.4 0.8]';
```
Note: The apostrophe at the end is the transpose operator. Equivalently, we could have used:
```
b=[1.2;0.4;0.8];
```
Note "beta" is a reserved name in Matlab - don't use it to label user-defined stuff.

Create a vector of zero-mean normal error terms and compose your dependent variable.
```
eps=1.2*randn(n,1);
y=X*b+eps;
```

Compute all relevant elements of an OLS regression:
```
bols=inv(X'*X)*X'*y;% get coefficient estimates; inv denotes "inverse"
```

```
e=y-X*bols;% Get residuals. Note the asterisk to perform matrix
multiplication.
% Matrix dimensions must be conformable.
s2=e'*e/(n-k); %get the regression error (estimated variance of "eps").
Vb=s2*inv(X'*X); % get the estimated variance-covariance matrix of bols
se=sqrt(diag(Vb));% get the standard erros for your coefficients;
% note the nested command structure: sqrt( ) takes the suare root of
% whatever is in ( ).  Diag ( ) extracts the diagonal from a square matrix
% in ( ).  We can easily combine the two commands.
t=bols./se; % get your t-values.  The dot-operator performs multiplication
% or division element-by-element.
```

Format all relevant output:

```
out=[bols se t]; %combine bols, se, and t vectors into a single matrix
fprintf(fid,'Output table for betas \n'); %label output; the "\n"
% combo is equivalent to "Enter" on your keypad, i.e it moves the next
% piece of output to a new line;
fprintf(fid,'coeff\t\tstd\t\tt-value\n'); % label each column of your output;
% \t inserts extra tabs. Play & experiment with \n and \t until your output
% looks the way you like it.
fprintf(fid,'%6.3f\t%6.3f\t%6.3f\n',out'); %define the numerical format for
% each column. Here: up to six digits total, with 3 decimal positions. The
% matrix to be plotted (here: "out") appears at the end in TRANSPOSED form.
fprintf(fid,'\n'); % this enters a blank line in your log file

fprintf(fid,'Squared regression error=\t%6.3f \n',s2); % for scalars you can
% define all output formats in a single line
fprintf(fid,'\n');
```

Save output elements:
Let's save things into a workspace element labeled "script1_stuff" (you can choose any name you like).
If you want to save EVERYTHING you created along the way:

```
save c:\mlab_tut\mlab\worksp\script1_stuff;
```

If you only want to save selected elements:

```
save c:\mlab_tut\mlab\worksp\script1_stuff y X bols;
```

Stop timer & capture run time:

```
finish = toc; %this will show run time in seconds
fprintf(fid,'Time elapsed in seconds \n\n');
fprintf(fid,'%6.3f\n',finish);
```

Note: To see the run time in minutes use `toc/60`; "finish" is just an arbitrary name.

Close log file

```
st=fclose(fid);
if st==0;
   disp('File closed successfully');
else;
   warning('Problem with closing file');
end;
```

To run the script, simply click on the "run" button in the upper toolbar (page symbol with downward arrow). This will automatically save it as well.

You can now open your log file in any text editor (e.g. Word) and inspect your output. Your raw output tables will copy nicely into Excel for further formatting.

If anything goes wrong, a red error message with the exact number of the offending line will appear in your command window. Also, as you write your script, any blatant errors (such as unbalanced matrices or brackets, dimension violations, etc) will be indicated on the right border of your editor window as dark red lines. You can ignore the orange lines (they are more cosmetic suggestions). We'll talk about error messages & de-bugging later.

Module III. Importing data / Descriptive statistics /Working with functions

In preparation we will save an existing data file in the correct format to be imported by Matlab. Download the files "**script2_data.xls**" and "**script2_data**.dta into your c:\mlab_tut\ folder. They are identical data sets. the first is in Excel format, the second in STATA format.

Preparing data in Excel:

- Open the original Excel file.
- Make sure your there are no empty cells in your data. (Replace blank cells with something like "999" if needed).
- Eliminate any rows with text or variable names.
- save your file as "Text (tab delimited)" (*.txt format), under the name "from_xl";
- close the file

Preparing data in STATA:

- Open the original STATA file.
- Make sure your there are no empty cells in your data. (Replace missing values with something like "999" if needed).
- in STATA's command window or do-file type:
  `outfile using c:\mlab_tut\from_stata,wide;` (semicolon at end only in do-file)

Main Matlab script:

Since your Editor is already open, to start a new script, choose File/New/Open from the *Editor* menu. Start your script following the structure outlined above:

```
rand('state',37); % set arbitrary seed for uniform draws
randn('state',37); % set arbitrary seed for normal draws

tic;  % start stop watch

[fid]=fopen('c:\mlab_tut\mlab\logs\script2.txt','w');
if fid==-1;
   warning('File could not be opened');
   break
else;
   disp('File opened successfully');
end;
```

Load data:

The data flow from a door-to-door fundraising campaign conducted in Pitt County, North Carolina, during the fall of 2005. The details of this field experiment are described in Landry et al. (QJE, 2006). Forty-three solicitors interacted with an average of 39 households each for a total sample size of 1682 observations. All observations are based on *actual interactions*, i.e. the "door didn't open" cases are not considered in this data set. The main focus of this research was on the effect of solicitor characteristics and lottery designs on donation outcomes. Each respondent (or "household") is visited only once, so there are no multiple observations per household in the data.

```
%load data from Excel:
load c:\mlab_tut\from_xl.txt;
data = from_xl; %rename dataset with a simpler name - optional
clear from_xl;% erase original (duplicate) data set - optional

% or from STATA:
```

```
load c:\mlab_tut\from_stata.raw;
data = from_stata;
clear from_stata;
```

Note: If you recycle an existing name (here "data"), the new element simply replaces the old one, regardless of dimensions. There will NO warning message, so be careful.

Describe your data (always a good idea since Matlab doesn't use or show variable labels)

```
% describe contents of data
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%1  runid           running observation id
%2  orig_solid      original solicitor ID
%3  amount          contributions (2005 dollars)
%4  gender          solicitor Gender (1=male)
%5  race            solicitor Race (1=white)
%6  height          solicitor height (inches)
%7  weight          solicitor weight (lbs)
%8  beauty          beauty index
%9  spunk           sum of personality scores
%10 gen_resp        observed gender of respondent (1=male)
%11 raceresp        observed race of respondent(1=white)
%12 avg_HHsize      average household size (Census block)
%13 inc000          Census tract median income (2005 dollars)
%14 lottery         fundraising package included lottery component
```

Define your dependent variable and your matrix of explanatory variables:

```
y = data(:,3); %":" as a dimension indicator means "all rows" or "all
columns"; here it means "all rows" since it describes the row dimension
X = data(:,4:end); % the second ":" means from - to; here from the 4th column
to the last
% Used in this context "end" indicates the last row, column, or element of a
numerical array
n=length(y); %same as n=size(y,1)
k=size(X,2);
```

Descriptive Statistics:

For a vector, summary stats are straightforward:

```
%Summary statistics:
my = mean(y);
stdy = std(y);
Vy = var(y);
Sy = sum(y);
```

You can now capture them for your log:

```
out =[my stdy Vy Sy];
fprintf(fid,'Summary stats for y ("amount") \n');
fprintf(fid,'mean\t\tstd\t\tvariance\t\tsum\n');
fprintf(fid,'%6.3f\t%6.3f\t%6.3f\t%6.3f\n',out');
fprintf(fid,'\n');
```

For a matrix, we have to indicate over which dimension(s) statistics are to be taken:

```
mX = mean(X,1);%mean for each column; result will be 1 by k
% note: add a transpose at the end if you prefer to see this as a k by 1
stdX = std(X,1);%std for each column; result will be 1 by k
VX = var(X,1);% variance for each column; result will be 1 by k
```

```
covX = cov(X,1);% covariance across columns; result will be k by k
corrX = corr(X);% sample correlation across columns; result will be k by k
sX = sum(X,1);% sum for each column; result will be 1 by k
```

So "1" in the second position of the stats command indicates that the statistic is to be taken over all rows (one result per column). Use "2" if you want the stats to be taken across columns (one result per row). In most cases, you will probably have all this info from your preparatory work in Excel or STATA, so we'll omit any log output here.
Note: "`corr(X)`" is the only basic stats command that does not allow for a dimension indicator. It will take the sample correlation across columns by default. If you need it across rows, use `corr(X');`

Write a simple OLS function with single output
We will now out-source or delegate the actual OLS computation to a separate function.

Open a new m-file in your Editor.
The first line should read:

```
function [bols] = ols1(y,X);
```

Function m-files ALWAYS start with the word "function", followed by a set of output elements to be returned to the main script in brackest [...] (here just "bols"). This is followed by an equality sign, the name of the function (here "ols1") and a set of inputs sent to the function by the main script (here y, X) in parentheses.

Next, it is a good idea to describe purpose, inputs and output in a few commented lines:

```
% PURPOSE: simple OLS regression
% INPUTS:
%           n by 1 vector y (dependent variable)
%           n by k matrix X (explanatory variables)
% OUTPUT:   k by 1 vector of regression coefficients
%
%
% written by: A bright Math week student
```

Note: All consecutive (uninterrupted) lines of such comments will be shown when you type "`help ols1`" in Matlab's main command window.

Next follows the main body of the function. This follows exactly the same rules as hold for scripts. Here we have just 1 line:

```
bols = inv(X'*X)*X'*y;
```

Save your function to your .../mlab/functions/ folder under then name "ols1";

Call the function from the main script:
Back in the main script, type:

```
bols = ols1(y,X);
```

Note that the order of inputs sent to the function must be EXACTLY as defined in the function itself. You could now proceed as in script 1, with computation of residuals, standard errors, t-values etc. in the main script:

```
e=y-X*bols;
s2=e'*e/(n-k);
Vb=s2*inv(X'*X);
se=sqrt(diag(Vb));
t=bols./se;
```

However, if this is a set of procedures you suspect will be used many times over in your research, you may want to make them part of the function. Here is how:

<u>OLS function with multiple outputs:</u>
Open a new m-file in your Editor.
The first line should read:

```
function [bols,se,t,s2] = ols2(y,X);
```

Add a few comments:

```
% PURPOSE: simple OLS regression
% INPUTS:
%           n by 1 vector y (dependent variable)
%           n by k matrix X (explanatory variables)
% OUTPUT:   bols: k by 1 vector of regression coefficients
%           se: k by 1 vector of standard erros
%           t: k by 1 vector of t-values
%           s2: squared regression error
%
% written by: A brilliant Math week student
```

Then add the function body:

```
n=length(y);% an alternative to "size(y,1)" for vectors
k=size(X,2);

bols=inv(X'*X)*X'*y;
e=y-X*b;
s2=e'*e/(n-k);
Vb=s2*inv(X'*X);
se=sqrt(diag(Vb));
t=bols./se;
```

Save your function to your .../mlab/functions/ folder under then name "**ols2**";

<u>Call the function from the main script:</u>
Back in the main script, type:

```
[bols,se,t,s2] = ols2(y,X);
```

Note that input and output arguments have to be defined and sequenced exactly as in the actual function.

Then compile key output into a single matrix and send it to your log as before:

```
out=[bols se t]; %combine bols, se, and t vectors into a single matrix
fprintf(fid,'Output table for betas \n');
fprintf(fid,'coeff\t\tstd\t\tt-value\n');
fprintf(fid,'%6.3f\t%6.3f\t%6.3f\n',out');
fprintf(fid,'\n');
fprintf(fid,'Squared regression error=\t%6.3f \n',s2);
```

```
fprintf(fid,'\n');
```

Save your key elements, stop the timer, and close your log as before:

```
save c:\mlab_tut\mlab\worksp\script2_stuff y X bols;

finish = toc; %this will show run time in seconds
fprintf(fid,'Time elapsed in seconds \n\n');
fprintf(fid,'%6.3f\n',finish);

st=fclose(fid);
if st==0;
   disp('File closed successfully');
else;
   warning('Problem with closing file');
end;
```

Practice:

Write a "script2b"using the same data as "script2" above. Write function "ols3" which has identical inputs and outputs to "ols2", but out-sources the computation of `s2, se`, and `t` to yet another function "ols3b".

The solution is given on the "Matlab Tutorial" web site.

Module IV. Reshaping and Indexing Arrays

(Goes with *script3*)

There are two useful commands to quickly replicate or reshape a vector or matrix. They are `repmat` and `reshape.`

Repmat:

Let's start by creating a simple row vector:

```
a=[1 2 3 4];
```

Now replicate this row 3 times along the row dimension (i.e. stacked vertically):

```
A1=repmat(a,3,1);
A1 =

     1     2     3     4
     1     2     3     4
     1     2     3     4
```

Now the same along the column dimension (i.e. aligned horizontally):

```
A2=repmat(a,1,3);
A2 =

   1   2   3   4   1   2   3   4   1   2   3   4
```

Now 3 times down and twice across:

```
A3=repmat(a,3,2);
A3 =

     1     2     3     4     1     2     3     4
     1     2     3     4     1     2     3     4
     1     2     3     4     1     2     3     4
```

Reshape:

reshape has the following generic structure:

```
B = reshape(a,b,c)
```

"a" is the vector or matrix to be reshaped. "b" indicates the number of rows in the reshaped output (here arbitrarily labeled "b") . "c" gives the number of columns in the reshaped output. Thus, the product *bxc* must always equal the total number of elements in "a".

Consider matrix A1 from above. Try the following 3 reshape versions:

```
%fun with reshape
B1=reshape(A1,12,1);
B2=reshape(A1,6,2);
B3=reshape(A1,4,3);
```

As you can see "reshape" picks elements from the original matrix column-by column and re-arranges them according to the dimensions specified by "b" and "c".

Practice:

Starting with matrix A1, create the following results using *repmat* and *reshape* (hint: you may also need to use some other simple matrix transformations)

```
P1 =

     1
     2
     3
     4
     1
     2
     3
     4
     1
     2
     3
     4

P2 =

     1     3     1     3     1     3
     2     4     2     4     2     4

P3 =

     1     3     1     3     1     3
     2     4     2     4     2     4
     1     3     1     3     1     3
     2     4     2     4     2     4
     1     3     1     3     1     3
     2     4     2     4     2     4

P4 =

     1     3     1     3
     1     3     1     3
     1     3     1     3
     2     4     2     4
     2     4     2     4
     2     4     2     4
```

Solutions can be found in "script3".

Indexing:

Indexing refers to identifying specific elements of a vector or matrix, possibly to then perform mathematical operations on them or to change the original construct. You can index elements directly or using the `find( )` function. Direct indexing is quicker than "find" if you want to immediately collect the indexed elements into a new construct, and / or transform them in some fashion.

Vector example:
Consider again vector
`a=[11 2 6.7 0]`. Assume you'd like to identify all positions in "a" that equal "0".

```
% Indexing
f1=find(a==0); %note the double-equal sign!

f1 = 4
```

So "0" happens to be located in the fourth position of a.

Now find all elements >2

```
f2=find(a>2);
f2 = 1     3

f3=find(isnan(a)); % find all elements that are "not numbers",
%i.e. unidentified, such as might result from a division by 0.
f3 = Empty matrix: 1-by-0

f4=find(isinf(a)); %find all elements that are "infinity",
% such as might result when you take the exponent of an already large number
f4 = Empty matrix: 1-by-0
```

You can now use the index to delete or alter the elements in "a":

```
b=a;% just to preserve the original a
b(f1)=[]; %delete all f1-indexed elements
b = 11     2     6.7

b=a;
b=a(f2); %keep all f2-indexed elements
b = 11     6.7

b=a;
b(f2)=15; %convert all f2-indexed elements to "15"
b = 15     2    15     0
```

We could have used **direct indexing** for all of these cases:

```
b=a;
b(b==0)=[];
b = 11     2     6.7
```

or faster:

```
b=a(a~=0)
b = 11     2     6.7

b=a(a>2)
b = 11     6.7
```

Matrix example:
For matrices, the "find" function implicitly stacks all columns on top of each other and works on the resulting vector. An example:

```
A=[1 0 3 0; 0 11 0 0; 2 5 11 0]';

A =
     1     0     2
     0    11     5
     3     0    11
     0     0     0

f1=find(A==0)
f1 =

     2
     4
     5
     7
     8
    12

[r1,c1]=find(A==0) %returns row and column index for indexed elements
r1 =
     2
     4
     1
     3
     4
     4

c1 =

     1
     1
     2
     2
     2
     3

[r1 c1] %makes explicit where exactly indexed elements are located
ans =

     2     1
     4     1
     1     2
     3     2
     4     2
     4     3
```

You can use indexing to delete entire rows of a matrix. For example, lets' assume we want to delete all rows in A for which the element in the first column equals zero:

```
f=find(A(:,1)==0);
A(f,:)=[];
A =

     1     0     2
     3     0    11
```

Practice:

Start with the following matrix:

```
M=3*[repmat((1:1:3)',2,1) [1 2 3 4 5 6]' ones(6,1)]
M =
     3     3     3
     6     6     3
     9     9     3
     3    12     3
     6    15     3
     9    18     3
```

Using indexing & subscripting, convert M to the following constructs (before each transformation, define `M1=M`, `M2=M`, etc to preserve the original M):

```
M1 =

     3     3     3
     6     6     3
     9     9     3
     3   100     3
     6   100     3
     9   100     3

M2 =
     3     3     3
     6     6     3
     9     9     3
     3    12   100
     6    15   100
     9    18   100

M3 =
     3     3     3
     6     6     3
     9     9     3

M4 =
     0     0     0
     6     6     0
     9     9     0
     0    12     0
     6    15     0
     9    18     0
```

Module V. Working with Cell Arrays

Matlab's **cell array feature** is best described as its ability to partition a vector or matrix into smaller "chunks" or "cells", and to reverse the process. Two key functions are `mat2cell` and its reciprocal `cell2mat`.

Let's first create a "data set" with a balanced grouping structure:

```
m1=reshape(repmat((1:1:8),4,1),32,1); %design first column
n=length(m1);
m2=2+1.4*randn(n,1); %design second column
m3=randsample([0 1],n,true)'; %this samples randomly n elements from the
% set [0 1] with replacement (drop "true" if you ever want to sample from a
% set of mubers w/o replacement)
M=[m1 m2 m3];
k=size(M,2);
```

```
M =
    1.0000    2.8485    1.0000
    1.0000    0.7423    1.0000
    1.0000    0.2894         0
    1.0000   -1.2272    1.0000
    2.0000    3.9684         0
    2.0000    1.7587         0
    2.0000    2.5686    1.0000
    2.0000    2.6728    1.0000
    3.0000    3.1910         0
    3.0000    1.4119    1.0000
    3.0000    1.7615         0
    3.0000    3.5988    1.0000
    4.0000    2.3928    1.0000
    4.0000    2.9976    1.0000
    4.0000    0.9373         0
    4.0000    0.9873    1.0000
    5.0000    0.2395    1.0000
    5.0000    3.5056    1.0000
    5.0000    0.5663         0
    5.0000    1.0289         0
    6.0000    0.1339    1.0000
    6.0000    0.7411    1.0000
    6.0000    2.0422         0
    6.0000    1.4392         0
    7.0000    2.1870    1.0000
    7.0000    0.8248    1.0000
    7.0000    1.6415    1.0000
    7.0000    3.9315         0
    8.0000    3.3584         0
    8.0000    2.4747         0
    8.0000    2.6756    1.0000
    8.0000    2.1512    1.0000
```

Assume the first column is an index for individuals. Thus, each of 8 individuals is associated with 4 observations (i.e. rows). Now collect all data for a given individual as a separate matrix (i.e. partition M into 8 chunks):

```
MC=mat2cell(M,4*ones(8,1),k)

MC =
    [4x3 double]
    [4x3 double]
    [4x3 double]
    [4x3 double]
    [4x3 double]
    [4x3 double]
    [4x3 double]
    [4x3 double]
```

Mat2cell has the following generic structure: out=mat2cell(a,b,c). a is the array to be partitioned. b is a vector that contains the row dimensions for each partitioned chunk. c is a vector that contains the column dimension for each partitioned chunk. The elements in b must SUM to the original row dimension (here: n). The same holds for c with respect to the original column dimension.

The result of our transformation is an 8 x 1 "**cell array"**

To access or inspect the contents of an individual cell:

```
mc1=MC{1}; %inspect contents of first cell
```

Note the curly brackets!

To access subsets of an individual cell:

```
mc1c2=MC{1}(:,2); %get second column of first cell
```

Note the curly brackets to access cell 1, followed by the usual round parentheses to access parts of an array, here the second column.

Now we can perform transformations or computations on each cell, and - if desired- re-assemble the transformed cells into a new matrix. Examples:

Ex1: Within each cell, we want to sort rows such that rows with zeros in the last column come first.

```
%Ex.1:
MC1=MC; %to preserve the original MC
for i=1:8
    MC1{i}=sortrows(MC{i},3); %means sort the rows in MC{i} in order
    %of the elements in the third column
end
MC1=cell2mat(MC1);
```

You can verify that now each panel is sorted according to the specified rule.

Ex2: Sum the elements of the 2nd column for each panel

```
Msum=zeros(8,1); %pre-allocate a zero vector to collect results
for i=1:8
    Msum(i)=sum(MC{i}(:,2));
end

Msum =

   10.8763
   11.4209
   11.9963
    8.1127
```

```
    4.5738
   10.0745
    6.2778
   10.9186
```

Now assume the panels are not balanced, i.e. the number of rows per panel is not equal across panels. We can use the fundraising data from Module 3. Each solicitor visited a different number of households.

```
% or from STATA:
load c:\mlab_tut\from_stata.raw;
data = from_stata;
clear from_stata;

% describe contents of data
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%1  runid              running observation id
%2  orig_solid         original solicitor ID
%3  amount             contributions (2005 dollars)
%4  gender             solicitor Gender (1=male)
%5  race               solicitor Race (1=white)
%6  height             solicitor height (inches)
%7  weight             solicitor weight (lbs)
%8  beauty             beauty index
%9  spunk              sum of personality scores
%10 gen_resp           observed gender of respondent (1=male)
%11 raceresp           observed race of respondent(1=white)
%12 avg_HHsize         average household size (Census block)
%13 inc000             Census tract median income (2005 dollars)
%14 lottery            fundraising package included lottery component
```

The second column in "data" contains the solicitor ID's. Let's first sort all observations by solicitor:
`data=sortrows(data,2);`
A quick inspection of `data(:,2)` will assure you that the panel sizes are not equal.

```
y = data(:,3);
X = data(:,4:end);
n=length(y);
k=size(X,2);
```

As before, we would like to partition "X" into solicitor-specific cell arrays.
To implement `mat2cell`, we first need a vector that contains the number of rows associated with each individual (= solicitor). This requires a bit of playing with `for` loops and `if` conditions.

```
j=0;
for i=2:n
    if sol_id(i)~=sol_id(i-1)% check if subsequent elements in sol_id are
        % equal or not
        j=i-1-sum(sol_vec); %j will give the panel size
        sol_vec=[sol_vec;j];
    end
end
sol_vec=[sol_vec;n-sum(sol_vec)]; % tag on size of last panel
```

You can verify that sol_vec contains 43 elements - that's the total number of panels (solicitors). The sum of elements in sol_vec equals the total sample size "n".

You can verify that each cell is indeed associated with a specific individual by using mat2cell on the sol_id vector (note: solicitor ids are not consecutive numbers in this case; however, each cell in the following "test" array should contain a vector of identical numbers)

```
test=mat2cell(sol_id,sol_vec,1);
test{1};
test{32}
test{end};
```

We can now implement the `mat2cell` command for our X matrix and perform cell-by-cell operations as before.

```
XC=mat2cell(X,sol_vec,k);
```

Practice: (for solutions see script 4)

<u>Ex 1:</u> Group the following vector into cells with identical elements:

```
a=[0 3 5 0 3 5 0 3 5 0 3 5]';
```

<u>Ex 2:</u> Same as Ex1, but now transpose the contents of each cell and re-assemble the parts in a new matrix. Show that the same result (starting with sorted vector a) could be accomplished using "`reshape`".

<u>Ex.3:</u> Create the following matrix, sort it by its first column and partition it into cells:

```
rand('state',37); % set arbitrary seed for uniform draws
randn('state',37); % set arbitrary seed for normal draws
g1=randsample([1:1:5],30,true)';
g2=ones(30,1);
g3=8*rand(30,1);
g4=randn(30,1);
G=[g1 g2 g3 g4];
```

Then sort each cell by its last column and re-assemble cells into matrix "GC".