This tutorial was written by Jonas Larsson, Royal Holloway University of London
jonas.larsson@rhul.ac.uk

Matlab Tutorial 2

(The topics covered here assume basic knowledge of Matlab as covered in Tutorial 1)

1. Statistical analysis in Matlab

The Statistics and Machine Learning toolbox in Matlab has a very large number of statistical tests. Here we will cover only a very small number to get you started. When you have completed the tests in this tutorial, you are encouraged to consult the documentation and examples for the statistics toolbox, available by typing **`doc`** at the Matlab command window.

It is assumed that you have basic familiarity with statistics; we will not cover the theory or practice of statistical hypothesis testing or details of the tests below.

Importing and exporting data

Unless you generate your data from within Matlab, you will likely need to import it from another application. The easiest way to do so is if the data is saved as a text file with special characters as "delimiters" separating each field. In Excel or LibreOffice, for example, you can export a spreadsheet in "comma-separated values" or CSV format, which can be read into Matlab easily. Only numeric data can be imported (so if your data has text labels, they will not be imported, and any data you load must be in numeric format). You cannot import a file with both numeric and text data this way (to do so you need to use the **`readtable`** function, see below).

The course folder contains a comma separated data set in spreadsheet format, called Add.csv. (This is a public domain data set provided for David Howell's Fundamental Statistics for the Behavioral Sciences, www.uvm.edu/~dhowell/fundamentals7/DataFiles/DataSets.html).The first row of this file contains the data labels, and the following rows are the numeric data column by column. You can look at the contents of the file in the editor or by typing

```
type Add.csv
```

in the command line window. To load this data set use **`dlmread`**:

```
mydata=dlmread('Add.csv',',',1,0);
```

The first input is the file name, the second the delimiter used (here a comma as it's a CSV file), the third input the index of the first row to start reading from *starting from 0* (so 1 means start from the second row – because the first row is data labels), and the fourth argument the first *column* to start from (again starting at 0 – here it means we start from the first column). If you have a file with a different delimiter than commas, you just specify the correct delimiter when calling dlmread:

```
mydata=dlmread('Add.dat',' ',1,0); % load data with spaces as delimiters
```

If the first row contains labels, loading these into Matlab (as a cell array of strings) is rather more involved and requires you to manually open the file and load in the labels using low-level file operations (you may skip this section unless you're keen):

```
f=fopen('Add.csv'); % Open the file for reading
```

```
s=fgetl(f);         % Read the first line of the file containing the labels into
a comma-separated string
fclose(f);          % Close the file
mylabels=strsplit(s,',') % Split the string at every comma into a cell array
```

If you want to export data into a comma-separated file, you can use the function **`dlmwrite`**:

```
mymatrix=rand(100,10); % create some data
dlmwrite('mymatrix.csv',mymatrix,','); % save as comma-separated values
```

Saving labels in the top row of the file (so it can be easily imported to a spreadsheet) is also rather more involved (again, feel free to skip this section unless you're very keen). Assuming your column labels are in a cell array called **`mylabels`**, you would do the following:

```
f=fopen('mymatrix.csv','wt'); % Open the file for writing text
mylabelstring=char(strjoin(mylabels,',')); % Convert labels to comma-separated
text
fprintf(f,'%s\n',mylabelstring); % Write the labels to the file
fclose(f);                       % Close the file
dlmwrite('mymatrix.csv',mymatrix,'delimiter',',','-append');% save as comma-
separated values, appending values to the existing first line
```

Unless you want to export your data to other programs, using Matlabs built-in save function is much easier. To save all data in your workspace to a file called mydata.mat, type

```
save mydata
```

To only save some variables, list them after the name of the file:

```
save mydata mymatrix mylabels
```

Tables and dataframes
More recent versions of Matlab have a data type called a *table* (which is similar to dataframes in R if you are familiar with that software). In Octave, the corresponding type is called *dataframe* and requires the dataframe package to be installed and loaded (**`pkg load dataframe`**). A table is much like a spreadsheet in that it includes labels for each column and can store non-numeric information. To load a table from a spreadsheet, use the readtable function:

```
mytable=readtable('Add.csv'); % In Matlab only
```

For this to work, the first row of the data must contain column labels, and if you import from a comma-separated file as above, the labels must not contain commas. In Octave, you would write

```
mytable=dataframe('Add.csv'); % In Octave only
```

You can save a table (in Matlab) using **`writetable`**, see the documentation for details.

Some Matlab operations work directly with tables, but others require you to extract the data to a matrix first. To convert numeric data in a table in Matlab, all you need to do is to index it using curly brackets:

```
mydata=mytable{:,:}; % (Matlab) means copy all the table data into a matrix
```

In Octave, the corresponding function is a little different:

```
mydata=mytable.array; % (Octave) means copy all the table data into a matrix
```

Note that Octave has a completely unrelated function called **`table`** – don't confuse this with the Matlab one.

Random variables
Matlab has support for generating random variables from various distributions. To generate random variables uniformly distributed between 0 and 1 use **`rand`**:

```
r=rand(25,50); % create a matrix 25 rows by 50 columns with random variables
between 0 and 1
```

`randn` generates normally distributed variables with zero mean and standard deviation of 1:

```
r=rand(5,10); % create a matrix 5 rows by 10 columns with normally distributed
random variables with mean=0 and SD=1
```

To generate random variables with different means or standard deviations, just add the mean and multiply by the desired S.D.:

```
r=10+5*randn(1,10000); % create 10000 random variables with mean 10 and SD=5
```

You can check that this worked using the **`mean`** and **`std`** commands:

```
mean(r)
std(r)
```

Why aren't the means and the standard deviation exactly 10 and 5?

Another very useful command is **`randperm`**, which generates a sequence of integers between 1 and a specified number that have been permuted into random order – handy for generating randomly ordered data sets for bootstrapping, for instance. To generate a random permutation of all the numbers between 1 and 50, you would type

```
mypermutednumbers=randperm(50);
```

`randperm` can also generate numbers sampled without replacement from the given set of integers, e.g. to generate 10 integers randomly sampled *without* replacement (so no number is selected twice) from 1-50 you would type

```
withoutrep=randperm(50,10);
```

To generate 10 integers sampled *with* replacement (the same number may appear twice) use the command **`randi`** instead, which creates random integers (see documentation for details):

```
randi(50,1,10)
```

Also useful is the command perms which creates the full set of permutations of a vector of numbers – useful if you want to create a pseudorandom ordering:

```
p=perms([1 2 3])
```

Beware that the number of permutations increases as the factorial of the length of the input vector – don't try to use vectors longer than 10 elements (which has over 3 million permutations).

Statistical distributions

Matlab can compute many common (and some not-so-common) statistical distributions, such as the normal distribution and Student's t distribution:

`normpdf` – Normal probability density function
`normcdf` – Cumulative normal probability density function
`norminv` – Inverse normal probability density function
`tpdf` – Student's t probability density function
`tcdf` – Cumulative Student's t probability density function
`tinv` – Inverse Student's t probability density function

If you need to use these directly, please consult the documentation for details.

Correlation and regression

One of the most common statistical measures is the correlation between two variables. To compute this in Matlab use the command **`corrcoef`**. Load the data set in Add.csv and compute the correlation between the second and fifth columns (representing ADD scores and IQ scores from a public domain data set):

```
mydata=dlmread('Add.csv',',',1,0); % load the data
scatter(mydata(:,2),mydata(:,5)); % Make a scatter plot of the second and fifth
columns
[r,pvals]=corrcoef(mydata(:,2),mydata(:,5)) % compute the correlation
coefficient r and P-value for each coefficient for the second and fifth columns
```

`corrcoef` returns a matrix of correlations (and P-values) for the full set of combinations of the input data, so if we call the inputs X and Y the elements of the **`r`** and **`P`** matrices correspond to the following combinations respectively:

$$\begin{matrix} r_{XX} & r_{XY} \\ r_{YX} & r_{YY} \end{matrix}$$

$$\begin{matrix} P_{XX} & P_{XY} \\ P_{YX} & P_{YY} \end{matrix}$$

The correlations between X and X and between Y and Y are, of course, 1. Similarly, the correlation between X and Y is the same as the correlation between Y and X. To get the correlation and P-value between X and Y as a scalar, just select the corresponding element:

```
r(1,2)
P(1,2)
```

You can also use corrcoef to compute the correlation between all columns in a matrix (the bivariate correlation matrix):

```
mymatrix=rand(10,5);
[r,pvals]=corrcoef(mymatrix)
```

It can sometimes be more instructive to visualise a bivariate correlation matrix as an image, using the **imagesc** command (see Tutorial 1 for more information on visualisation with **imagesc**):

```
imagesc(r)
```

Linear regression in Matlab can be done in multiple ways; one of the simplest is using the command **regress** which is used both for simple and multiple regression.
For example to test if ADD scores (column 2) can be predicted by IQ (column 5), you could use linear regression as follows:

```
mydata=dlmread('Add.csv',',',1,0); % load the data
Y=mydata(:,2); % Use ADD scores as the dependent variables
nrows=size(mydata,1); % Get the number of rows in the data set
X=[ones(nrows,1) mydata(:,5)]; % Create a matrix of regressors nrows long. The
first column needs to be ones (1) for regress to work – this is important!
[B,BINT,R,RINT,STATS] = regress(Y,X); % Run the regression
disp(STATS) % print the regression statistics: the R-square statistic, the F
statistic and p value for the full model, and an estimate of the error variance
disp(B) % print the esti5mated beta coefficients (mean and slope)
```

To run a multiple regression, just add more regressors to your design matrix X, e.g.:

```
X=[ones(nrows,1) mydata(:,[5 8])]; % use columns 5 (IQ) and 8 (GPA) as
regressors
[B,BINT,R,RINT,STATS] = regress(Y,X); % Re-run the regression
disp(STATS) % print the regression statistics: the R-square statistic, the F
statistic and p value for the full model, and an estimate of the error variance
disp(B) % print the esti5mated beta coefficients (mean and slope)
```

`BINT` stands for the 95% confidence interval for the estimated beta values, `R` the residuals, an `RINT` the 95% confidence limits of the residuals (useful for identifying outliers). Consult the documentation for **regress** for more detailed information.

Comparing two group means
A standard test for comparing two group means is Student's t-test, which can be used on unpaired or paired data sets. The function ttest is used for paired, or one-sample t-tests, and ttest2 for two-sample (unpaired) t-tests.

First, load the data:

```
mydata=dlmread('Add.csv',',',1,0); % load the data
```

Now let's use **ttest** to investigate whether the mean IQ in the data set is different from the expected mean of 100:

```
iq=mydata(:,5); % IQ is the data in column 5
mean(iq) % show the mean. Is it significantly different from 100?
[H,P]=ttest(iq,100) % Perform a one-sample t-test of the hypothesis that IQ is
not equal to 100
```

H is 1 if the null hypothesis cannot be disproved, and P the P-value of the test. Was the mean IQ significantly different from 100? Check the documentation and try to re-run the test as a one-tailed test instead (make sure you use the appropriate tailed test, left or right).

If you have data in pairs, you can run the test on the difference by subtracting one set from the other, or calling ttest with the two data sets as inputs:

```
X1=rand(30,1);
X2=rand(30,1);
[H,P]=ttest(X1-X2) % test the difference
[H,P]=ttest(X1,X2) % test as separate inputs
```

The data set in the file Add.csv contains IQ values for both genders, with gender information in column 3 (males=1, females=2). Let's use a two-sample t-test – **ttest2** – to check whether there is a significant difference in IQ between males and females.

```
gender=mydata(:,3);                  % Extract gender information into a row vector
malerowindices=find(gender==1);      % Find the row indices for males
femalerowindices=find(gender==2);    % Find the row indices for females
maleiq=iq(malerowindices);           % Extract the IQ values for males
femaleiq=iq(femalerowindices);       % Extract the IQ values for females
disp([mean(maleiq) mean(femaleiq)]) % Display the mean IQ for males and females
[Htwotail,Ptwotail]=ttest2(maleiq,femaleiq) % Perform a two-sample two-tailed t-
test of the hypothesis that male and female IQ are different
[Honetail,Ponetail]=ttest2(maleiq,femaleiq,'tail','left') % Perform a two-sample
one-tailed t-test of the hypothesis that females have higher IQ than males
```

Comparing multiple group means with ANOVA

Matlab supports one-way, two-way, and multi-way ANOVA with the functions **anova1**, **anova2**, and **anovan** respectively. To run a one-way analysis of variance with equal numbers of measurements (balanced ANOVA), you can supply the data as a matrix with one column per group:

```
adata=randn(50,3); % create some randomly distributed data with zero mean
adata(:,1)=adata(:,1)+1+randn(50,1); % create a systematic difference between
groups
adata(:,3)=adata(:,3)-1+randn(50,1); % create a systematic difference between
groups
[P,anovatab]=anova1(adata) % run the analysis, plot results and return ANOVA
table
```

If you have unequal numbers of measurements across groups, you can instead use a grouping variable, which is a vector with one value per measurement which represents the group that measurement belonged to. The grouping variable can be numeric (coding group by a number), a string array or cell array:

```
mydata=dlmread('Add.csv',',',1,0); % load the data
addscore=mydata(:,2); % ADD scores as dependent variable in a single vector
groupvar=mydata(:,6); % EngL score coded as 1, 2, 3 as grouping variable
[P,anovatab]=anova1(addscore,groupvar)
```

It works the same if the grouping variable is a cell array:

```
for n=1:length(groupvar)
      groupvartext{n}=['Group ' num2str(groupvar(n))];
end
[P,anovatab]=anova1(addscore,groupvartext)
```

If you provide a third output stats this can be used to do post-hoc multiple comparisons:

```
[P,anovatab,stats]=anova1(addscore,groupvartext)
multcompare(stats)
```

See the documentation for **`multcompare`** for details.

To run a two-way or multi-way ANOVA (balanced or unbalanced), use anovan:

```
iq=mydata(:,5);
gender=mydata(:,3);
twowaygroupvar={iq gender};
variablenames={'IQ' 'Gender'};
[P,anovatab]=anovan(addscore,twowaygroupvar,'varnames',variablenames)
```

For a two-way balanced ANOVA, **`anova2`** can also be used. This requires your data to be in a matrix format where the change in one factor goes along the rows, the other along the columns, and multiple measurements for the same factor are in separate but adjacent rows. For the most part, it is a ***lot*** easier to use **`anovan`** than to force your data into this rather specific format. But if you are keen, check the documentation for more information.

General linear models

The above tests are all versions of the General Linear Model (GLM). Matlab has support for solving general linear models using the command **`glmfit`**, which is a bit like **`regress`** on steroids. If you are interested in using GLM to analyse your data then you need to be familiar with the statistical theory of general linear models, in which case the documentation for **`glmfit`** will make perfect sense – however covering this is beyond the scope of this tutorial.

Missing data

Many statistical functions in Matlab can handle missing data values, which should be coded as *NaN,* which is not your grandma but a computer's way to represent "not a number". You can indicate that a value is NaN by simply typing

```
myval=NaN;
```

or for an element in a vector, for example

```
myval(4)=NaN;
```

Mostt statistical functions will ignore NaN values and adjust the calculations accordingly; those that don't will usually complain.

NaNs also result when you do undefined operations, such as dividing zero by zero:

```
0/0

ans =
      NaN
```

2. Optimising data visualisation using handle graphics

Matlab has a wealth of functions to produce informative plots and graphs, but to make full use of the power of these functions you need to learn how to manipulate figure properties such as fonts, legends, lines, labels, and axis markers to clearly display your data. Many of these properties can be edited using conventional "point-and-click" on the figures, but not only is this often a bit clunky, it does not lend itself to automation, so can become very time consuming especially if you need to replot data. Handle graphics is a way of interacting with each element of a figure using Matlab commands, which allows rapid fine tuning of figures and data plots programmatically – allowing data to be plotted rapidly and with minimal user intervention for speed and consistency.

Overview of handle graphics

Whenever Matlab draws something in a figure window (and even the window itself) it internally creates a *graphics object* (basically that which is drawn) and assigns a *handle* to that object – this is a unique number that can be used to check and modify the properties of the object. You can think of it as an identifier that allows probing and manipulating the appearance of every graphics object. So, for example, if you plot a line, Matlab creates a line object, and assigns a unique handle to that line object which can be used to change line properties such as the style, colour, and thickness.

Getting and setting handle graphics properties

Every drawing command creates a handle to the drawn object. To use this handle, simply provide an output variable when calling the command, e.g.:

```
x=rand(1,10);
y=rand(1,10);
h=plot(x,y)
```

h is a handle to the plotted line and can be used to query and change the line properties. It is a number whose value is unique but otherwise arbitrary.

Some handles refer to "top-level" objects, such as a figure window, or the axes of a plot. To get the handle to the current figure window (the one you last opened or clicked on), use the command **gcf** (**g**et **c**urrent **f**igure):

```
figure
h=gcf
```

You can get the handle directly when creating a figure:

```
h=figure
```

To get the handle to the axes of the current plot, use **gca** (**g**et **c**urrent **a**xis):

```
h=gca
```

A handle is only useful as a way of querying or setting an object's properties. To query the properties, use get:

```
get(h)
```

which will print all the property names and their values – much like a structure. You can change almost any of these properties using the command set:

```
h=plot(x,y)
set(h,'linewidth',5)
```

To see the possible values that you can assign to each property, type set with only the handle input argument:

```
set(h)
```

This will print every property of the object referred to by the handle and the possible values each can take. Empty brackets like {} mean any value is possible (restricted to within the limits of that property, e.g. a line width can only be a positive numeric value).

Using handle graphics for data visualisation
To illustrate how handle graphics can be used, we will make a figure and use handle graphics to modify the look of each subplot programmatically.

```
% First we get some data to plot and make a basic line plot
mydata=dlmread('Add.csv',',',1,0); % load the data
f=figure; % Create a figure
mydata=sortrows(mydata,5); % Sort the data by IQ in ascending order
femalerows=find(mydata(:,3)==2); % plot data separately by gender
iqf=mydata(femalerows,5); % IQ
gpaf=mydata(femalerows,8); % GPA
hpf=plot(iqf,gpaf,'r'); % Plot female GPA vs IQ in red
hold on
malerows=find(mydata(:,3)==1); % plot data separately by gender
iqm=mydata(malerows,5); % IQ
gpam=mydata(malerows,8); % GPA
hpm=plot(iqm,gpam,'b'); % Plot male GPA vs IQ in blue
```

Inspect the plot, which looks pretty awful as is. First, let's create a legend. Second, let's change the plot styles: make the markers bigger, use different markers for males and females, and change line style and width for each gender.

```
hl=legend;
% for females make the markers large circles and filled in red with black
contours
set(hpf,'marker','o','markersize',8,'markerfacecolor','r','markeredgecolor','k')
;
% for males make the markers large squares and filled in blue with black
contours
set(hpm,'marker','s','markersize',8,'markerfacecolor','b','markeredgecolor','k')
;
% Make the lines thicker and use a dotted line for females and dashed for males
set(hpf,'linewidth',2,'linestyle',':')
set(hpm,'linewidth',2,'linestyle','--')

% Use meaningful labels for the legend and change the relative location
set(hl,'string',{'Females' 'Males'},'location','northwest')
% Make the legend fontsize a bit larger
set(hl,'fontsize',11)

% Remove the box for a more open-style plot
box off

% Add axis labels
xlabel('IQ')
ylabel('GPA')
```

```
% Add a title
title('GPA vs IQ by gender')

% make the axis limits a bit larger to add some whitespace around data
set(gca,'xlim',[60 150],'ylim',[0 4.5]) % Note that xlim and ylim are the min
and max limits of the x and y axes and are a property of the current axis (gca)

% Make the axes lines a bit thicker
set(gca,'linewidth',1.5)
% Change the figure font to italic - note how it affects all text in the figure
set(gca,'fontangle','italic')
% Make the title text 50% larger than the labels
set(gca,'titlefontsizemultiplier',1.5)

% resize the figure to 700 x 500 without moving it
figpos=get(gcf,'position'); % the first two numbers are position, the last two
size
newpos=[figpos(1:2) 700 500];
set(gcf,'position',newpos)
```

This basic demo illustrates how one can use handles to set and get properties. There are many more properties that can be changed, consult the documentation for details. Using handles can be very handy if you want to trial different plot styles etc without having to manually redo the plot every time; simply create a script to draw your figure and change the plot styles in the script, and then redraw the figure by simply running the script. If you code the key properties as variables (e.g. `mylinewidth=2; set(hpf,'linewidth',mylinewidth)`, then it becomes easy to apply changes consistently across multiple plots without having to change every number separately.

3. Basic numerical analysis techniques

Matlab is at its core a software package for numerical analysis and was developed originally for solving computational problems using linear algebra. As a result Matlab has very powerful support for numerical analysis based on matrix algebra and a large number of associated functions. The following is just a small sample of the functionality provided, if you are interested in venturing further please consult the documentation.

Matrix algebra and solving least squares problems

Matrix multiplication

Matrices are multiplied using the * operator (note that the .* operator does element-by-element multiplication), e.g.

```
C=A*B;
```

Note that the matrices need to be compatible in size (columns of A == rows of B)

Matrix division

Matrix division, using the backslash symbol \, is a shorthand for solving linear systems. E.g. to solve the matrix equation

```
A*X=B
```

you can type

```
X=A\B
```

The forward slash / has a different meaning – type **`help slash`** for information (note that the ./ operator does element-by-element division).

Other potentially useful commands are

`rank` – to compute the rank (number of independent columns) of a matrix
`inv` – the inverse of a matrix
`diag` – extracts the diagonal of a matrix
`eye` – creates an identity matrix of any size
`lscov` – least squares solution to matrix equations, similar to matrix division but more versatile, allows using weights for instance
`pinv` – pseudoinverse for solving underdetermined least squares problems

Non-linear function fitting

There are numerous ways to fit non-linear functions to data, for example for curve fitting. A standard and powerful technique is to use non-linear minimisation techniques with the command **`fminsearch`**, which uses a method called Nelder-Mead simplex minimisation. Consult the documentation for more information.

Convolution and filtering

Filtering data – for example, applying a moving average – can be done using the convolution functions. To filter a data vector – such as a time series of measurements – use the function **`conv`**:

```
dataSeries=rand(1,100); % create a random data series
movingAverage=ones(1,3); % make a filter vector representing the averaging of
three neighbouring data points - e.g. ones(1,5) would average over five nearby
points
movingAverage=movingAverage/sum(movingAverage); % Normalise so sum of filter
equals 1 (so filtering doesn't scale data)
filteredDataSeries=conv(dataSeries,movingAverage,'same'); % convolve (filter)
dataSeries with filter, keeping only the part of the convolution that
corresponds to the input data series
figure
plot(dataSeries,'b-'); % plot the original data
hold on % hold so new plot commands don't erase figure
plot(filteredDataSeries,'r-'); % plot filtered data
legend({'Raw data' 'Filtered data'});
```

Filtering in two dimensions (e.g. to filter an image) uses **`conv2`** but is otherwise equivalent:

```
dataImage=rand(100,200); % create an image with random data
movingAverage=ones(3,3); % make a filter image representing the averaging of 3x3
neighbouring data points (pixels) - e.g. ones(5,5) would average over the
nearest 5x5 pixels
movingAverage=movingAverage/sum(movingAverage(:)); % Normalise so sum of filter
equals 1 (so filtering doesn't scale data). Note use of colon operator (:) to
compute sum over entire image rather than just columns
filteredDataImage=conv2(dataImage,movingAverage,'same'); % convolve (filter)
dataImage with filter, keeping only the part of the convolution that corresponds
to the input image
figure
subplot(1,2,1)
imagesc(dataImage); % show the original image
axis image % Display image at correct aspect ratio (don't stretch image)
title('Raw data');
subplot(1,2,2)
imagesc(filteredDataImage); % show filtered image
axis image % Display image at correct aspect ratio (don't stretch image)
title('Filtered data');
```

Basic image processing

Images are represented as simple matrices where each element is one pixel (for grayscale images), or as arrays (height x width x 3) where each element is a red, green or blue pixel value (for RGB images), or as arrays (height x width x 4) where the fourth element is the transparency (alpha value) of the pixel. Images can be read in with the imread function. You can download an image of your choice (GIF, JPG or PNG all work) and load it, or use one of Matlab's built-in images for demos:

```
imdata = imread('ngc6543a.jpg');
size(imdata)
ans =
   650   600     3
```

Convert your image to grayscale with the **`rgb2gray`** function:

```
imgray = rgb2gray(imdata);
size(imgray)
ans =
   650   600
```

The image now has only 2 dimensions (red, green and blue information has been compressed into a single grayscale intensity for each pixel).

You can save the image with imwrite – the file name determines the format:

```
imwrite(imgray,'gray.png'); % save in PNG format
```

Example: creating visual stimuli with `meshgrid`

Here is a very brief example of how to use Matlab to create a visual stimulus often used in vision science: a grating in a circular aperture and a Gabor patch [the latter is a sinusoidal grating pattern windowed by (multiplied with) a 2-dimensional Gaussian function (like the normal distribution)].

First we create a coordinate system for the image, which we will use for computing the stimuli:

```
xsize=800;
ysize=600; % use an image size of 800x600 pixels
xcoord=linspace(-xsize/2, xsize/2, xsize); % create a vector xsize long that
goes from -xsize/2 to xsize/2 - this is our x coordinate system, with 0 at the
centre of the image
ycoord=linspace(-ysize/2, ysize/2, ysize); % create a vector ysize long that
goes from -ysize/2 to ysize/2 - this is our y coordinate system, with 0 at the
centre of the image

[x,y]=meshgrid(xcoord, ycoord); % Create two arrays of x and y coordinates
respectively, which give us for each pixel the corresponding x and y coordinates
- see help for meshgrid

% You can use imagesc to visualise x and y to see this more clearly

% Create a grating pattern tilted 45 degrees (pi/4) counterclockwise from the
horisontal
ang=pi/4; % angle in radians
sf=0.1; % spatial frequency - the number of sinusoidal cycles per pixel
ph=0; % phase of sinusoid
grating=sin(sf*(cos(ang)*x+sin(ang)*y)+ph); % compute grating as sinusoidal
function of x and y

figure
imagesc(grating); % show grating
colormap gray

% Now create a circular aperture by finding all the points within a 100 pixel
radius from centre
radius=100;
distfromcentre=sqrt(x.^2+y.^2); % .^ means to the power of for each element in
the matrix
circAperture=double(distfromcentre<radius); % set all pixels that meet condition
to 1, we have to convert the result to a numeric value (double) as the result is
a logical matrix which can't be used directly for calculations

% Mask the grating to create circular stimulus
circGrating=grating.*circAperture;
figure
imagesc(circGrating)
axis image
colormap gray

% Now let's make a Gabor patch: first, we create a Gaussian aperture
```

```
gaussAperture = (exp(-(0.02*distfromcentre))); % Gaussian function of radial
distance, scaled by 0.02

gaussAperture(gaussAperture<0.01)=0; % threshold window so small values are set
to zero

% Multiply the Gaussian aperture with grating to create Gabor
gabor=grating.*gaussAperture;

figure
imagesc(gabor)
axis image
colormap gray
```

4. Debugging techniques

When you start to learn programming you will frequently find yourself spending time debugging your code. It is often instructive to have your code print the values of key variables while it's running to test that it works properly. However, sometimes it is helpful to be able to halt execution of a programme allowing you to inspect the values of variables at a particular point. To instruct Matlab to stop execution temporarily, insert the command **`keyboard`** where you want to stop the programme. You can do this at multiple places in the code, if you like.

Once the programme halts, you can check current variables in the command window as usual – note that only those visible to the programme will be shown, not your normal workspace environment.

You can step through the subsequent code, one line at time, using the command **`dbstep`**. To resume execution, type **`dbcont`**. If you want to quit the code rather than letting it continue to run, type **`dbquit`**.

Check the documentation for more details.

5. Matlab coding style

Compared to more formal computer languages such as C, Matlab is very forgiving when it comes to coding style. This means it is easy to write code that is poorly structured and difficult to read. You are strongly encouraged to adopt good coding style, not just because it makes your code much easier to read and understand, but will also save a great deal of time when debugging code and by allowing you to re-use code rather than having to write similar code over and over again for different applications.

An excellent guide to coding style is the one by Richard Johnson (on Moodle). Also on Moodle is a one-page cheat sheet for quick reference. You are strongly encouraged to adhere to those guidelines.

Some personal tips:
- If you have to write the same code twice, make it into a function. Not only does this save time but also reduces the risk of errors.
- Avoid overly long or large functions – they quickly become unwieldy and hard to debug. Split them into a master function that calls smaller subfunctions.
- Use functions rather than scripts where possible. Functions don't have access to your workspace and therefore can only modify their output variables – hence they won't inadvertently change the value of a variable in your workspace, which can cause very difficult to find bugs. Scripts have no restrictions and can have unintended consequences.
- Watch out for variable or function names that are identical to Matlab built-in functions – they will "shadow" these functions, meaning that you can't access the original function anymore (until you clear the variables or functions). For example you could call a variable `mean`, and then the function `mean` would no longer be visible to your code. To check if a name is already taken by a Matlab function, type **`which`** followed by the variable name; if there is a function by that name Matlab will show where it is located, otherwise it will state "not found", meaning the name is free to use.
- Comment often but keep comments brief and concise. Make sure your comments keep up with changes in the code. Misleading comments are worse than no comments at all.

6. Student-led project

Below are some ideas for a project, if you don't already have one in mind.

1. Write a bootstrap statistics test function that takes two input data sets (vectors or matrices) and tests whether their means are significantly different using bootstrapping (sampling with replacement). The function should becalled like this

```
Pval=mybootstraptest(dataset1,dataset2,numiter,alpha)
```

where *numiter* is the number of resampling iterations to run and *alpha* is the alpha level for the test.

Additional options:
- make the fourth input optional so if the user doesn't specify it, an alpha of 0.05 is used (hint: use the **`nargin`** command to count how many input variables were provided)
- add a second output variable showing the P value of a standard t-test of the same data

2. Write a function that loads a comma-separated CSV data set from a file specified by the user and runs a 1-way ANOVA with the dependent and independent (grouping) variables specified by the user by their column index and returns the P value and the ANOVA table as output:

```
[Pval,anovatable]=mybootstraptest(filename,depvarcolumn,indepvarcolumn)
```

Additional options:
- allow multiple grouping columns to run 2-way or multi-way ANOVAs (hint: use **`anova2`** or **`anovan`**)

3. Write a function that loads an image and displays it in a figure with two subplots, with the other plot showing a histogram of the pixel values in the image. Hint: colour images are stored as 3D arrays, one slice per colour (height x width x 3). Use handle graphics to set the histogram properties (such as colour).

```
showimage(filename)
```

Additional options:
- add a second input argument which is a width of a local average filter to apply to the image; change the figure to have four subplots (2 rows and 2 columns) showing the image and histogram before and after filtering. Hint: filter each colour plane (R, G, B) separately