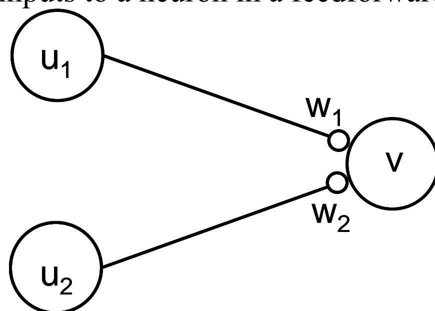
Eigenvectors, Eigenvalues, and Principal Components Analysis (PCA)

In this laboratory, we will analyze a set of data to find the "principal components" along which the data is scattered. Principal components analysis (PCA) is widely used in an enormous variety of settings to characterize the variability in a set of measurements. In sensory systems, it is the starting point for models that attempt to describe why sensory neurons respond to the features to which they are sensitive. In face-recognition technologies, it is used to classify the primary source of variation among different people's faces. A more complicated extension of PCA, known as independent components analysis (ICA), has been used to separate the voices of independent speakers who are talking simultaneously and has also been used to look for independent patterns in electroencephalographic (EEG) data. More generally, PCA provides an algorithm for compressing high-dimensional data down to a few (or several) dimensions that often can capture a large fraction of the variability in the full data set.

I. Principal components of a 2-dimensional data set

In this section of the tutorial, we will calculate the principal components of a randomly assigned set of two-dimensional data points (u_1,u_2). In general u_1 and u_2 could correspond to literally anything – pixel intensities of photographs, variables corresponding to dice rolls, number of wins by the Red Sox (=u_1) and Yankees (=u_2) in a season, etc.

To make the situation more directly neuroscience related, we will consider u_1 and u_2 to be the firing rates of the inputs to a neuron in a feedforward network, as shown below:



If time were sampled every, e.g. 1 second, we would gather many such data points **u** = (u_1, u_2) over the course of time and we could see how much variability there was in the data and whether there were any correlations between the rates u_1 and u_2. Although we will not prove this here, one of the reasons we choose this example is that Hebbian learning can be shown to align the synaptic weight vector **w** = [w_1 w_2] along the direction of the principal components of the variability in the (u_1,u_2) data points. Since the postsynaptic neuron's firing rate v = **w•u** is the projection of the input vector **u** along the weight vector **w**, this means that the postsynaptic neuron's firing rate v will be able to capture much of the variability in the inputs **u**. Under certain assumptions, it can be shown that this choice of weights assures that v carries the maximum information possible about its inputs.

In the following section, we will generate data points **u** from a 2-dimensional, rotated Gaussian distribution (i.e. an ellipse of data points). We will then show that principal components analysis finds the principal axes of this ellipse, with the first principal component corresponding to the long axis of the ellipse and the second principal component corresponding to the short axis of the ellipse.

II. Generating rotated Gaussian data

Let's jump right in and generate our random data points. First, as usual, let's open a .m file and name it something identifying like Lab8_2DPCA.m. Then let's comment our code and clear all variables:

```
%Generate a 2-D cloud of data points (u1,u2) and then
%find the principal components of this distribution

clear all;
close all;
```

To generate the Gaussian random data oriented in a tilted ellipse relative to the x-y axes, we will first generate an elliptical cloud of data oriented with elongated direction along the x axis. We will then rotate this data. First let's generate the non-rotated ellipse of random data by independently choosing random x-values and random y-values from a Gaussian distribution:

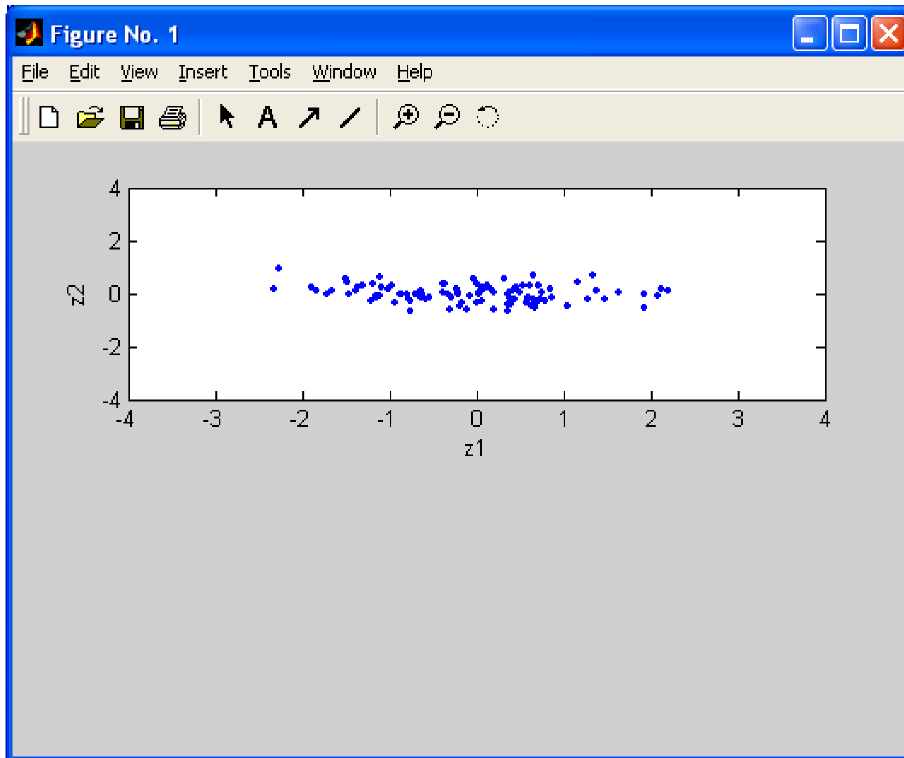
```
%GENERATE ELLIPSE OF RANDOM DATA TILTED AT AN ANGLE THETA
%RELATIVE TO THE X-AXIS
%first generate non-rotated ellipse by independently generating
%random x-values (placed in the variable z1) and y-values (in variable z2)
N = 100; %number of random data points
z1_vect = randn(1,N); %generates a 1x100 vector of values chosen from
                      %a Gaussian distribution of mean 0 and std dev 1
z2_vect = 0.33*randn(1,N); %Gaussian random of mean 0 and std dev 0.33
```

In this code, z1 gives the random x-coordinates of the points and z2 gives the random y-coordinates of the points. The command **randn(M,N)** generates an M x N array of random numbers chosen from a Gaussian distribution of mean zero and standard deviation 1.

Let's plot the data by adding the lines:

```
figure(1)
subplot(2,1,1)
plot(z1_vect,z2_vect,'.')
xlabel('z1')
ylabel('z2')
axis([-4 4 -4 4])
```

We will add a second subplot with the rotated data next. First, run the above and you should see something like the following:



Next let's rotate the data counterclockwise by an angle $\theta = 60^0 = \pi/3$ radians. This can be accomplished (see a math text for explanation of the geometry behind this) by multiplying each of the (z_1,z_2) data points by the rotation matrix

$$\mathbf{R} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

to get our final x-axis and y-axis values (u_1,u_2):

$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} \cos(\theta) * z_1 - \sin(\theta) * z_2 \\ \sin(\theta) * z_1 + \cos(\theta) * z_2 \end{pmatrix}$$

For a given point with coordinates (z_1,z_2) this gives the corresponding x and y values of the point after it has been rotated counterclockwise by an angle θ. If we would like to rotate every data point, then we can first organize our original (**z**) data into a matrix of points:

z_mat = [z1_vect; z2_vect]

Do this and run your code to see that it produces a matrix with 100 columns, each of which specifies the x- and y-coordinates of a single data point. Then add a semicolon at the end of this line to suppress the output.

To get a corresponding matrix of points that are rotated by the angle theta, we simply multiply this matrix of data points by the rotation matrix **R**, as follows:

```
theta = pi/3   %rotate counterclockwise by this angle [radians]
Rotation_mat = [cos(theta) -sin(theta);sin(theta) cos(theta)] %rotation matrix
```

```
u_mat = Rotation_mat*z_mat; %data points after rotation
```

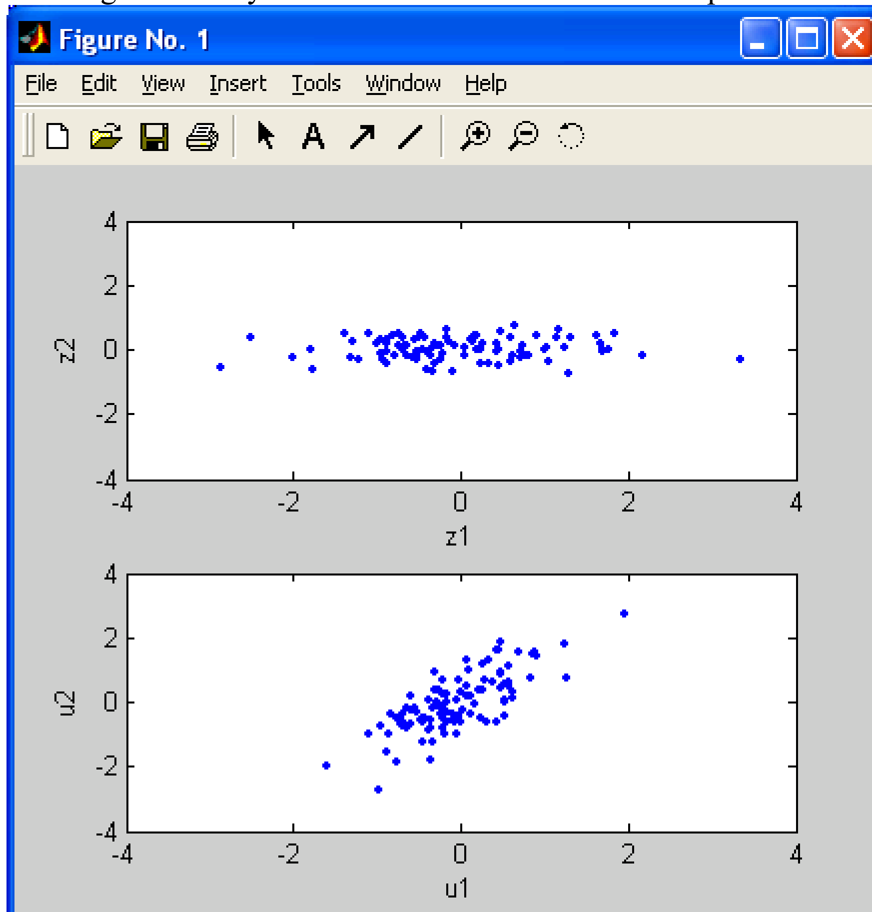
This produces a matrix u_mat whose 100 columns each contain the coordinates of a single (now rotated) data point, with x-values of all the data points in the first row and y-values of all the data points in the second row. (You can check this by typing size(u_mat) in the Command Window).

Let's then plot these points by adding to the end of our code the following:

```
subplot(2,1,2)
plot(u_mat(1,:),u_mat(2,:),'.')
xlabel('u1')
ylabel('u2')
axis([-4 4 -4 4])
```

Recall that u_mat(1,:) returns the vector containing all values of the first row of the matrix u_mat and u_mat(2,:) returns the vector containing all values of the 2nd row of the same matrix.

Running this code you should find the final rotated ellipse of data in the bottom subpanel:



Great! Now we have a data set of random points to work with. Your code at this point should read:

```
%Generate a 2-D cloud of data points (u1,u2) and then
```

```
%find the principal components of this distribution

clear all;
close all;

%GENERATE ELLIPSE OF RANDOM DATA TILTED AT AN ANGLE THETA
%RELATIVE TO THE X-AXIS
%first generate non-rotated ellipse by independently generating
%random x-values (placed in the variable z1) and y-values (in variable z2)
N = 100; %number of random data points
z1_vect = randn(1,N); %generates a 1x100 vector of values chosen from
                %a Gaussian distribution of mean 0 and std dev 1
z2_vect = 0.33*randn(1,N); %Gaussian random of mean 0 and std dev 0.33

figure(1)
subplot(2,1,1)
plot(z1_vect,z2_vect,'.')
xlabel('z1')
ylabel('z2')
axis([-4 4 -4 4])

z_mat = [z1_vect; z2_vect]; %data points before rotation
theta = pi/3   %rotate counterclockwise by this angle [radians]
Rotation_mat = [cos(theta) -sin(theta);sin(theta) cos(theta)] %rotation matrix
u_mat = Rotation_mat*z_mat; %data points after rotation

subplot(2,1,2)
plot(u_mat(1,:),u_mat(2,:),'.')
xlabel('u1')
ylabel('u2')
axis([-4 4 -4 4])
```

III. Finding the principal components of the data

Now that we have the data points (u_1,u_2) generated, we'll next generate the covariance matrix **Q** with elements $Q_{ij}=\langle u_i u_j \rangle$. If we are very clever, we can actually generate this in a single line of code! Note that the "outer product" of the vector u by itself is given by:

$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix}\begin{pmatrix} u_1 & u_2 \end{pmatrix} = \begin{pmatrix} u_1^2 & u_1 u_2 \\ u_2 u_1 & u_2^2 \end{pmatrix}$$

If we average this across all of the data points, we get the correlation matrix **Q**:

$$\left\langle \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}\begin{pmatrix} u_1 & u_2 \end{pmatrix} \right\rangle = \begin{pmatrix} \langle u_1^2 \rangle & \langle u_1 u_2 \rangle \\ \langle u_2 u_1 \rangle & \langle u_2^2 \rangle \end{pmatrix} = \mathbf{Q}$$

To perform this average, we need to sum over all data points and then divide by the number of data points N (and recall that each data point corresponds to a column of the matrix u_mat). This can be simply done with the command (type this below the u_mat assignment):

```
%DEFINE CORRELATION MATRIX Q
Q_mat = (u_mat*u_mat')/N
```

You should convince yourself that this code does indeed do the operations described in the equations above. To do this, I recommend trying this out by hand on a much simpler data set that you can write on a piece of paper, e.g. try an example with N=3 data points by choosing u_mat = [1 2 2; 4 2 5] and confirm that the result is a 2 x 2 matrix whose values correspond to the averages across the 3 data points of the quantities in the definition of **Q**. In doing this, recall that the *transpose* of this matrix, u_mat', is defined by interchanging the rows and columns, i.e. the first row of u_mat becomes the first column of u_mat' and the second row of u_mat becomes the second column of u_mat' so that u_mat' = [1 4; 2 2; 2 5]. Then, verify your by-hand solutions by entering the values for u_mat and the formula for Q_mat in the MATLAB Command Window.

Let's now return to our main code: If you leave the semicolon off the Q assignment and run your code you should get values for Q. They will differ from run to run due to the random number generator, but you should notice certain things:

▪ First, the variance along the x-direction, Q_{11}, should be smaller than the variance along the y-direction, Q_{22}, consistent with the data having been rotated by 60^0 (see subplot 2 of Figure 1—it will be helpful in visualizing the data to resize your Figure 1 window so that the plots appear to be square, thus more naturally reflecting that both axes go over the same range [-4,4]). This can be done with the command "axis square", "axis equal", or "axis image" – consult the help menus for the distinctions between these.

▪ Second, the "off-diagonal" elements Q_{12} and Q_{21} should be equal to each other and positive because the data are positively correlated (i.e. tend to both be positive or both be negative). *Terminology note: "off-diagonal" elements is common terminology for those elements not lying along the diagonal line running from top-left to bottom-right of the matrix and therefore not having indices equal to each other. The elements lying along the diagonal are, not surprisingly, called the "diagonal elements" or the "elements along the diagonal".*

Finally, let's calculate the principal components of the data, i.e. the directions along which the long and short axes of the ellipse lie. These directions correspond to the eigenvectors of Q. The eigenvalues of Q then correspond to the standard deviations of the corresponding lengths of the cloud of points along these directions.

To obtain these eigenvectors and eigenvalues we use the MATLAB command **eig** as follows:

```
%GENERATE EIGENVALUES AND EIGENVECTORS OF Q
%EIGENVECTORS ARE ASSIGNED AS THE COLUMNS OF THE 1ST MATRIX BELOW
%THE CORRESPONDING EIGENVALS ARE ASSIGNED AS THE DIAGONAL ELEMENTS OF THE
2ND MATRIX BELOW
[Eigenvect_mat,Eigenval_mat] = eig(Q_mat)
```

Leave the semicolon off this command so that you can inspect the result. In the final two outputs, you should see something like (the exact numbers will vary due to the random number generator):

```
Eigenvect_mat =

  -0.8752    0.4837
```

```
    0.4837    0.8752

Eigenval_mat =

    0.1143         0
         0    0.9166
```

The eigenvector matrix consists of two column vectors ξ_1 = [-0.8752; 0.4837] and ξ_2 = [0.4837; 0.8752] which are called the "principal component vectors" of this data set (*Note: ξ is the Greek letter "xi" and is often used to denote eigenvectors*). By MATLAB convention, these vectors are of length 1. They are also perpendicular, as expected: this can be checked by noting that the dot product $\xi_1 \cdot \xi_2 = \cos(\theta)$ where θ = the angle between the vectors (and the lengths are equal to 1 so that they do not appear in this expression). For perpendicular vectors, $\theta = 90^0$ so $\cos(\theta) = 0$. This is indeed the case, as can be verified by calculating the dot product between ξ_1 and ξ_2 (try this by typing at the Command Prompt Eigenvect_mat(:,1)'*Eigenvect_mat(:,2), which will take the dot or "inner" product of the two vectors).

The eigenvalue matrix above is a "diagonal matrix" (translation: all nonzero elements are on the diagonal) whose diagonal elements give the eigenvalues corresponding to the eigenvectors. The eigenvalues λ_1=.1143 (for the short axis of the ellipse) and λ_2 = .9166 (for the long axis of the ellipse) give the variance of the data along the directions corresponding to the two eigenvectors. To obtain the standard deviations in the spread of the data along the axes of the ellipse of data, we must take the square root of these eigenvalues.

Let's put these values into separate variables of their own:

```
xi1_vect = Eigenvect_mat(:,1)  %first eigenvector, length =1
xi2_vect = Eigenvect_mat(:,2)   %second eigenvector, length =1
sigma1 = sqrt(Eigenval_mat(1,1))  %length of ellipse along 1st eigenvect direction
sigma2 = sqrt(Eigenval_mat(2,2))  %length of ellipse along 2nd eigenvect direction
```

Run this code and check your output to make sure the eigenvectors and eigenvals were assigned correctly. You should find that the sigma's are quite close to the standard deviation of the ellipses we defined for the cloud of data in the previous section (1 along the long axis, 0.33 along the short axis).

Finally, let's plot the eigenvectors (with length given by the corresponding eigenvalue) on top of our data cloud to confirm our claim that these correspond to the standard deviations of the long and short axes of the ellipse of data points. First let's define vectors pointing along the eigenvectors but with lengths equal to the corresponding sigmas:

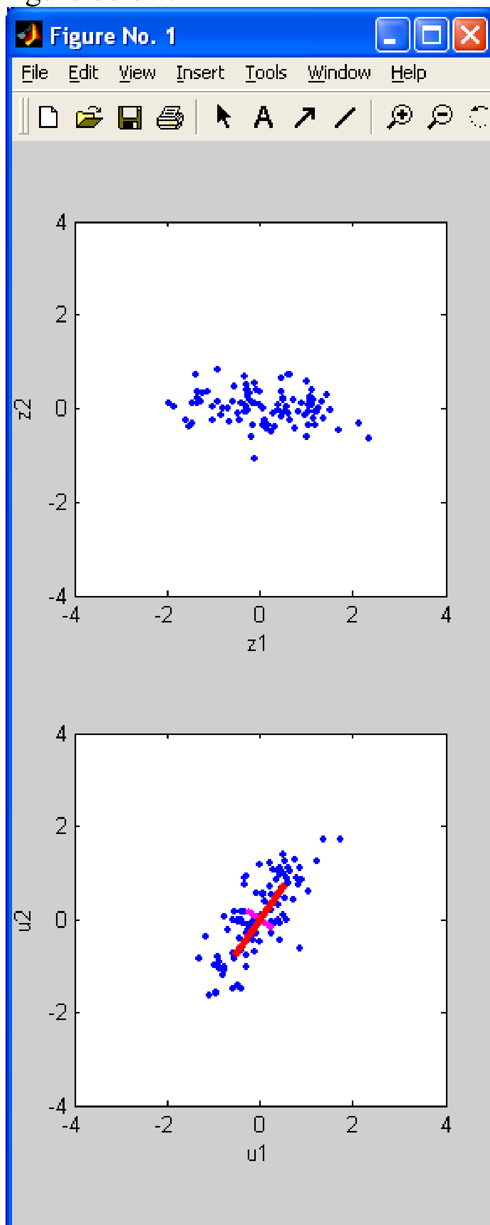
```
sigma1_vect = sigma1*xi1_vect;  %first eigenvector stretched to length sigma1
sigma2_vect = sigma2*xi2_vect;  %2nd eigenvector stretched to length sigma2
```

The above just give two individual data points that correspond to the endpoints of the vectors we'd like to draw. For each vector, let's draw a line from this endpoint to negative of its value (so that the line crosses through the whole ellipse on both sides of the origin). To draw these vectors, we'll plot a line from between the points (-sigma1_vect(1),-sigma1_vect(2)) and

(sigma1_vect(1),sigma1_vect(2)) for the first vector and correspondingly for the second vector. To do this, add to the plotting section:

```
hold on
plot([-sigma1_vect(1) sigma1_vect(1)],[-sigma1_vect(2) sigma1_vect(2)],'m','linewidth',3)
plot([-sigma2_vect(1) sigma2_vect(1)],[-sigma2_vect(2) sigma2_vect(2)],'r','linewidth',3)
hold off
```

The **'linewidth'** argument to the plot command tells MATLAB to plot these lines with a thickness of 3 points so that they are more visible. Your end result should look something like the figure below. If your standard deviation bars along the directions of the principal components do not look perpendicular to each other, this is probably an artifact of your plot panel not being square—you should resize your window so that the plot panel is square as in the figure below.



Note how nicely we can see in these plots that the eigenvectors of the correlation matrix **Q** extracted the principal components of the data: the first principal component corresponds to the eigenvector with the largest corresponding eigenvalue and represents the direction along which the data set has its maximum standard deviation – this direction is useful because it indicates the strongest trend in the data if we have to characterize the data set with a single vector. In this case, we note that the trend pulled out by the first principal component is that the x- and y-values are correlated maximally along an axis oriented 60^0 clockwise from the x-axis. The second principal component will always be perpendicular to the first and, in the case of higher-dimensional data sets will indicate the next largest direction of spread of the data (under the constraint that this direction must be chosen to be perpendicular to the first direction; likewise for higher dimensional data, all higher principal components are chosen to be perpendicular to the previously chosen principal components and, in order, represent the largest direction of remaining variance). Thus, one can think of PCA as fitting a data set to a multi-dimensional ellipse with the eigenvector/components corresponding to the direction of the axes of the ellipse and the corresponding eigenvalues to the standard deviation along these directions.

In practice, one can often account for most of the variability in a set of high-dimensional data with only a relatively small number of principal components. For example, in our case, we capture most of the variability

in the data with only 1 principal component as seen by the relative spreads along the two principal components in the neighboring graph.

Congratulations—you've written a program that "discovers" the structure in a set of data! Your final code for this section should read:

```
%Generate a 2-D cloud of data points (u1,u2) and then
%find the principal components of this distribution

clear all;
close all;

%GENERATE ELLIPSE OF RANDOM DATA TILTED AT AN ANGLE THETA
%RELATIVE TO THE X-AXIS
%first generate non-rotated ellipse by independently generating
%random x-values (placed in the variable z1) and y-values (in variable z2)
N = 100; %number of random data points
z1_vect = randn(1,N); %generates a 1x100 vector of values chosen from
                 %a Gaussian distribution of mean 0 and std dev 1
z2_vect = 0.33*randn(1,N); %Gaussian random of mean 0 and std dev 0.33

figure(1)
subplot(2,1,1)
plot(z1_vect,z2_vect,'.')
xlabel('z1')
ylabel('z2')
axis([-4 4 -4 4])

z_mat = [z1_vect; z2_vect]; %data points before rotation
theta = pi/3   %rotate counterclockwise by this angle [radians]
Rotation_mat = [cos(theta) -sin(theta);sin(theta) cos(theta)] %rotation matrix
u_mat = Rotation_mat*z_mat; %data points after rotation

subplot(2,1,2)
plot(u_mat(1,:),u_mat(2,:),'.')
xlabel('u1')
ylabel('u2')
axis([-4 4 -4 4])

%DEFINE CORRELATION MATRIX Q
Q_mat = (u_mat*u_mat')/N

%GENERATE EIGENVALUES AND EIGENVECTORS OF Q
%EIGENVECTORS ARE ASSIGNED AS THE COLUMNS OF THE 1ST MATRIX BELOW
%THE CORRESPONDING EIGENVALS ARE ASSIGNED AS THE DIAGONAL ELEMENTS OF THE
2ND MATRIX BELOW
[Eigenvect_mat,Eigenval_mat] = eig(Q_mat)

xi1_vect = Eigenvect_mat(:,1)  %first eigenvector, length =1
xi2_vect = Eigenvect_mat(:,2)   %second eigenvector, length =1
sigma1 = sqrt(Eigenval_mat(1,1))  %length of ellipse along 1st eigenvect direction
sigma2 = sqrt(Eigenval_mat(2,2))  %length of ellipse along 2nd eigenvect direction
```

```
sigma1_vect = sigma1*xi1_vect;  %first eigenvector stretched to length sigma1
sigma2_vect = sigma2*xi2_vect;  %2nd eigenvector stretched to length sigma2

hold on
plot([-sigma1_vect(1) sigma1_vect(1)],[-sigma1_vect(2) sigma1_vect(2)],'m','linewidth',3)
plot([-sigma2_vect(1) sigma2_vect(1)],[-sigma2_vect(2) sigma2_vect(2)],'r','linewidth',3)
hold off
```