Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

Matlab workshop: Applications of Matlab

This workshop serves to provide a small taste what one can do with Matlab. We will explore image processing by finding where in the brain the motor cortex is located, audio processing by removing interference from a sound file and numerical methods by coming up with a solution to use Euler's method for solving Ordinary differential equations.

Contents

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

Finding the motor cortex of the brain

The purpose of the following activity is to familiarize you with a data driven approach of using Matlab.

We will be using brain scans (courtesy of Professor Bharat Biswal of the New Jersey Institute of Technology) and identifying where in these scans the motor cortex lies. To do so, we will use two different techniques. We will find the average active images and subtract the average inactive images to find the areas with the most activation. We will also try to correlate a reference signal (what we expect the outcome to be) with the actual images.

Experiment

The data was collected as follows. A subject was asked to do some finger tapping in 10 second intervals. 10 seconds OFF followed by 10 seconds ON for a period of 90 seconds. During this experiment, 36 different slices of the brain was recorded for activity each record being a 64 by 64 pixel image. So the data we load is a set of 64x64 matrices for each of the 36 slices times 90. You may also think of it as an image with point (x,y,z) taken over 90 seconds.

Loading data

Open up the data by using the load command.

```
2 -    clear all;clc;
3 -    load('MOTOR.mat');
4 -    motorSize = size(MOTOR);
```

Figure 1 loading the data

```
motorSize =

    64    64    36    90
```

Figure 2 size of data

The size of the data is 64x64x36x90. Let's understand the data.

Visualisation

It would help us understand if we visualized the data a little.

```
clear all;clc;
load('MOTOR.mat');
motorSize = size(MOTOR);

for i=1:36
    imagesc(MOTOR(:,:,i,1));
    pause;
end
```

Figure 3 printing code

The code in figure 3 uses the function imagesc (which stands for image scaled) to print the values of a given matrix. We are using a loop because we want to see all 36 slices at the first second. The pause command causes the loop to pause and wait for a key press; this way we can visualize the scans at our own pace.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

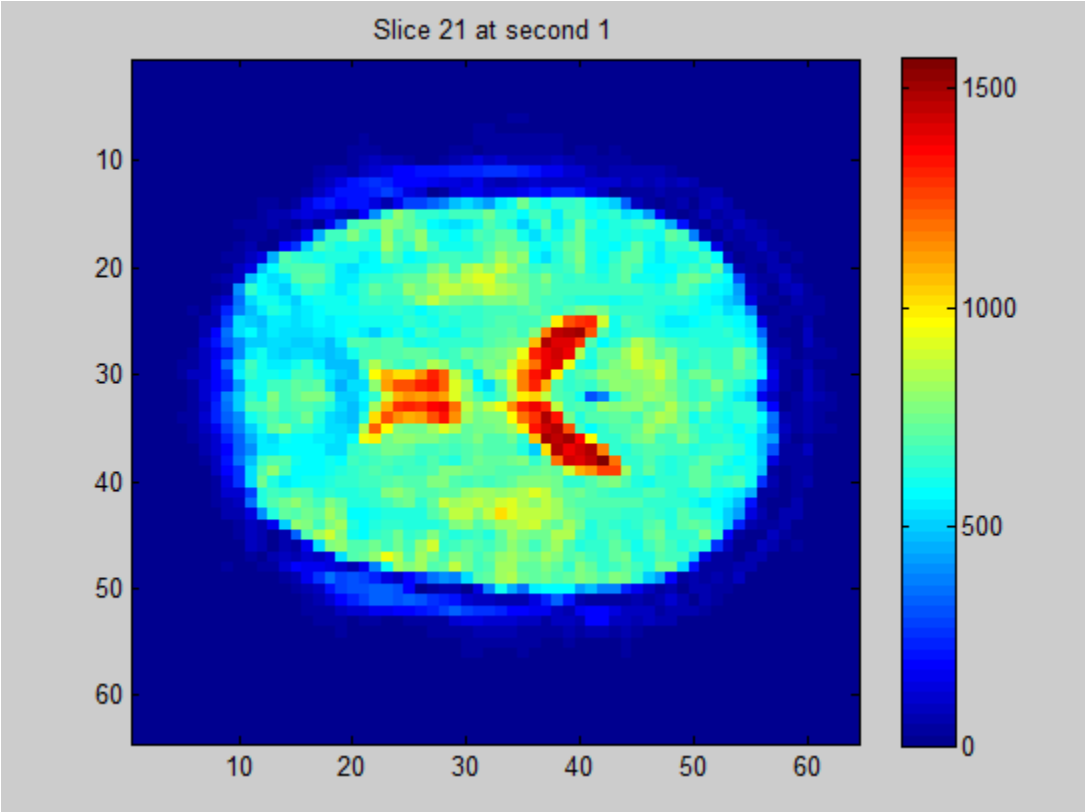

Figure 4 One brain scan

What you are looking at in figure 4 is one 64x64 matrix. Imagesc scales the values such that the red corresponds to high activity and the blue corresponds to low activity.

Difference method

The full script can be found in motorDiff.m

We can take the average pixel value during the ON times and subtract the average pixel value during the OFF times to find the most active pixels when finger tapping occurs.

Masking

In the experiment the data was interleaved as 10 points OFF, 10 points ON, etc. If we want to look at only the ON then only the OFF data, we need to create a mask to segregate the data.

```
%TASK mask
taskOFFMask = [1:10 21:30 41:50 61:70 81:90];
taskONMask = [11:20 31:40 51:60 71:80];
```

Figure 5 task mask

These masks will help us segregate the data. Next we need to take the average across time for each slice.

Getting the average

We can easily get the average of each pixel when ON and when OFF now that we have our masks.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

```
%get the average
MeanOFF = mean(MOTOR(:,:,:,taskOFFMask),4);
MeanON = mean(MOTOR(:,:,:,taskONMask),4);
```

Figure 6 Getting the average

In the above statement we are selecting all the ROWS, COLUMNS, SLICES and only the pixels in our mask to take the mean. The mean is only taken in the 4th dimension, this means that were are **averaging over time.**

It makes sense that we've taken the average over time when we look at the size of the MeanON.

```
>> size(MeanON)

ans =

    64    64    36
```

Figure 7 Size of the mean

Notice the column for time is gone, that is because those 90 numbers have now been replaced by one, the average over time.

Getting the difference image

Getting the difference image is really simple at this point

```
%take the differrence
DiffImg = MeanON - MeanOFF;
```

Figure 8 expression for the difference image

What we are doing is that we are subtracting each ON pixel by each OFF pixel.

Hiding values we don't care about

We only care about values that are positive since we are looking positive activity during the ON times.

```
%zeros all the values less than or equal to zero
diffmask = DiffImg<=0;
DiffImg(diffmask) = 0;
```

Figure 9 Zeroing values

We can set all the negative values to zero using the above masking technique.

Result

We can now take a look at what we get. We can use the following snippet for visualisation

```
for i=1:36
    imagesc(DiffImg(:,:,i));
    text = sprintf('Difference Image at slice %d',i);
    title(text);
    pause;
end;
```

Figure 10 Displaying the diffImg

The result shows us the highest positive difference between ON and OFF times. The regions with the most color correspond to the most activation when the finger tapping occured.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

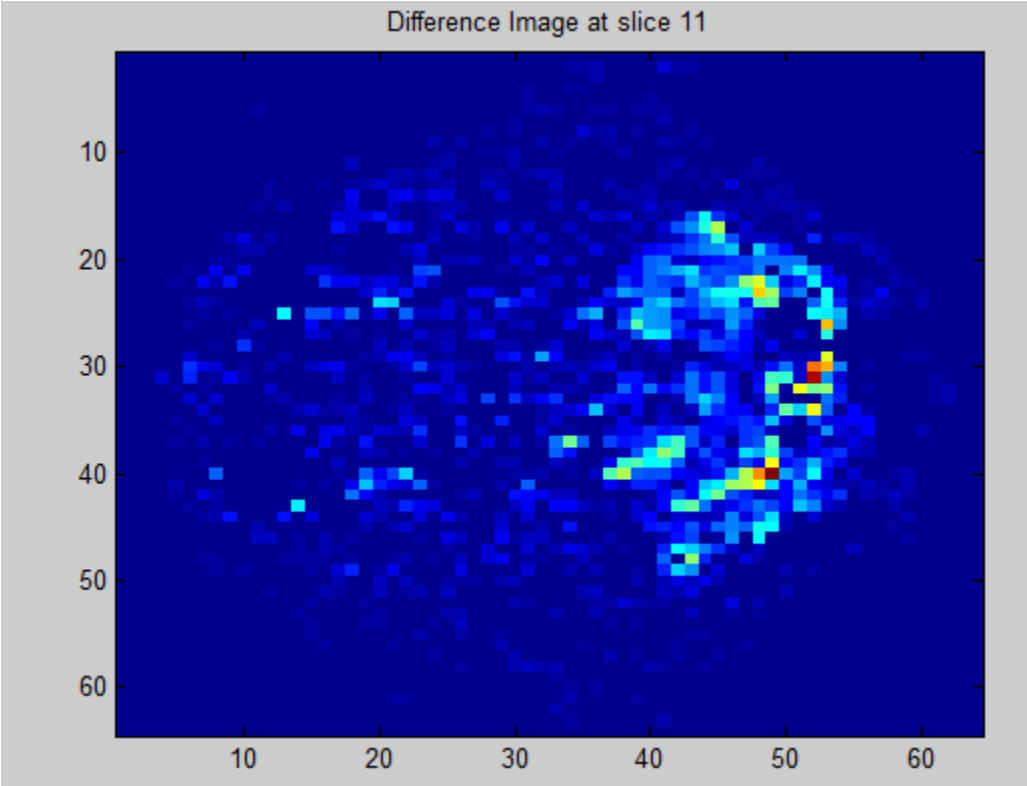

Figure 11 The result of the diff image

Correlation method

We can take a more statistical approach to find the regions with the highest activation. The code for this section can be found in motorCorr.m.

Creating a reference

We are trying to find all the points in the brain, or pixels, which have high activity when the finger tapping occurs. We would expect the ideal signal to look something like the signal in the figure below.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

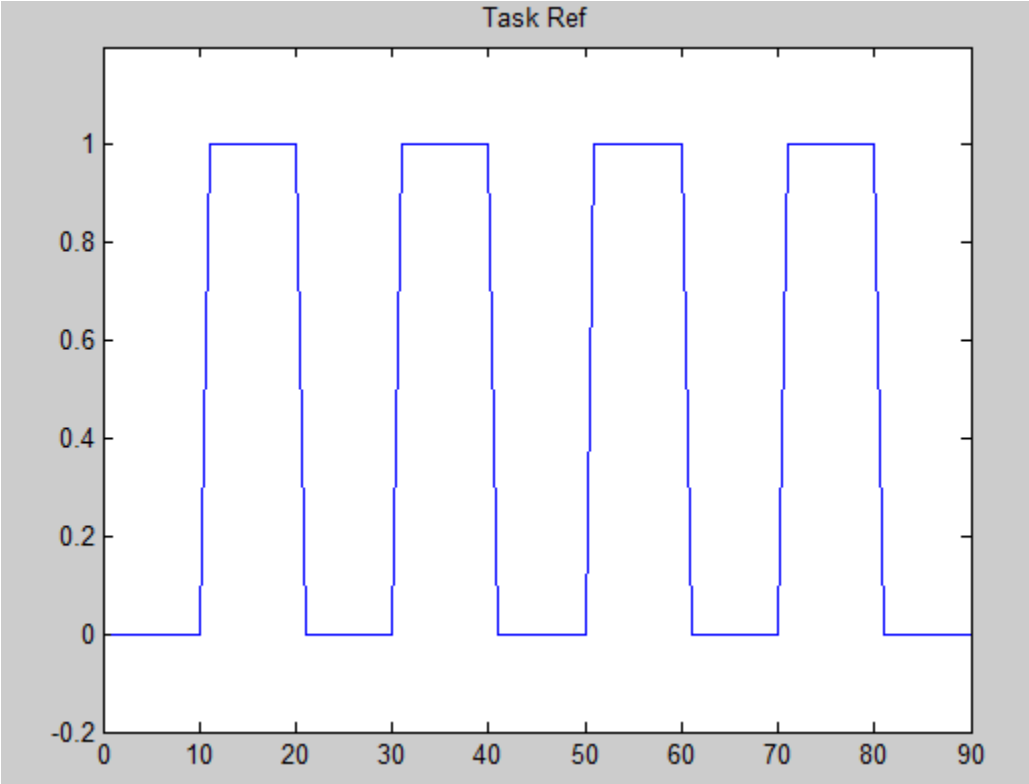

Figure 12 The expected activity signal

Let's create a variable that holds the above waveform.

```
%TaskRef
ON = ones(1,10);
OFF = zeros(1,10);
taskRef = [OFF ON OFF ON OFF ON OFF ON OFF];
```

Figure 13 TaskRef code

We can generate the plot in taskref graph above using the following code.

```
>> plot(taskRef);axis([0 90 -0.2 1.2]);
>> shg
>> title('Task Ref');
```

Figure 14 plotting the taskRef

Getting the correlation

We can now compare each pixel with the reference but first we need to reshape our matrix so that we have rows of pixels.

```
pixels = reshape(MOTOR, [],90);
```

Now we have a matrix where each row is a pixel's value over time.

```
>> size(pixels)

ans =

      147456          90
```

Figure 15 pixels matrix

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

Now we can find the correlation.

```
%find the correlation between each pixel and the reference
corrImage = zeros(64,64,36);
pixels = reshape(MOTOR, [],90);
corrImage = 1 - pdist2(pixels, taskRef, 'correlation');
corrImage = reshape(corrImage, 64,64,36);
```

Figure 16 finding correlation

We've made a new matrix now which shows us the pixels with the highest correlation. Let's look at only the positive correlation.

```
%zeros all the values less than or equal to zero
corrMask = corrImage<=0;
corrImage(corrMask) = 0;

for i=1:36
    imagesc(corrImage(:,:,i));
    text = sprintf('Correlation Image at slice %d',i);
    title(text);
    pause;
end;
```

Figure 17 visualizing our result

Using the above code we can see the pixels with the highest correlation to our reference signal.

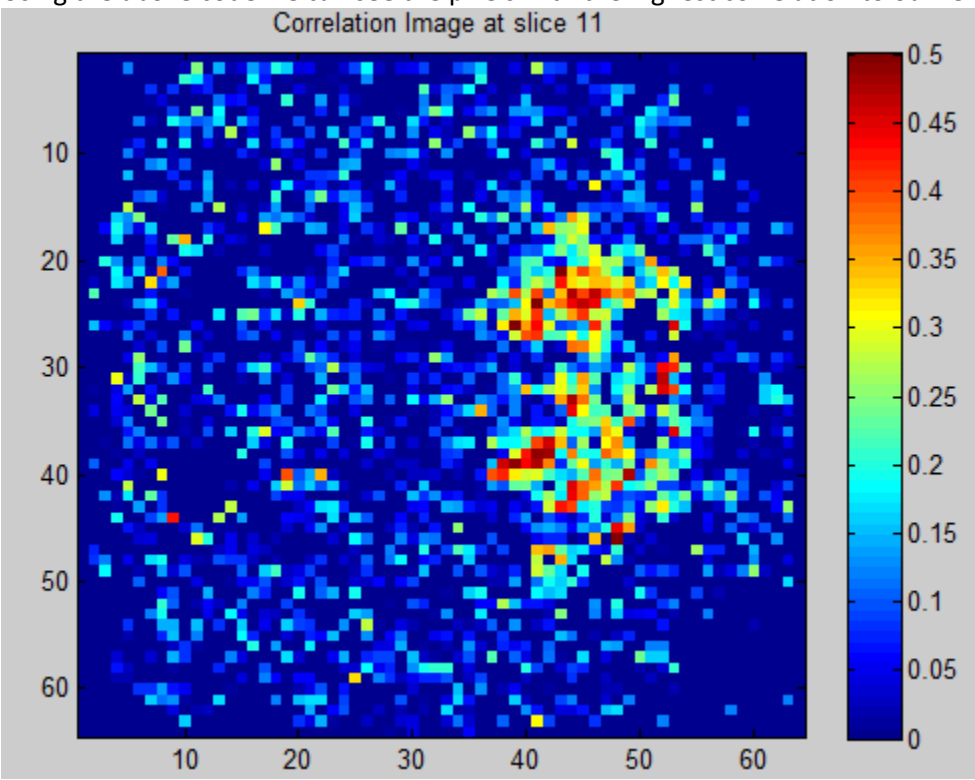

Figure 18 Correlation Image

We can see a lot of interference in this image but clearly there is a distinct bulge on the right side that has a high correlation to our reference. It is very similar to our results in the difference image part.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

Conclusion

Hopefully this example shows many ways to manipulate matrices in Matlab **without** using loops. It is important to remember to use a data driven approach when dealing with large data sets because a lot of computations can be done with more efficient algorithms; the built in Matlab functions take of just that.

Visualizing Linear Algebra

This section shows some ways to visualize concepts in linear algebra.

Seeing Eigen vectors

The code for this section can be found in Eigen.m. An Eigen vector is defined as

$$Ax = \lambda x$$

All it means is that if a vector X is an Eigen Vector then when it is Matrix Multiplied by A it will not change its direction only its magnitude. We can see these Eigen value in Matlab if we multiply a whole bunch of vectors and find the ones who's direction stays the same.

Let's start by creating a whole bunch of vectors of length 1.

```
vectors = randn(2,2000);
nVectors = normc(vectors);
X = nVectors(1,:);
Y = nVectors(2,:);

plot(X,Y,'ro');
```

We use the randn function to create a matrix where each column represents a vector. The first row will represent X values and the second row will represent Y values. We then normalize each column such that each vector has length 1. The plot function plots our points as Red Os.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

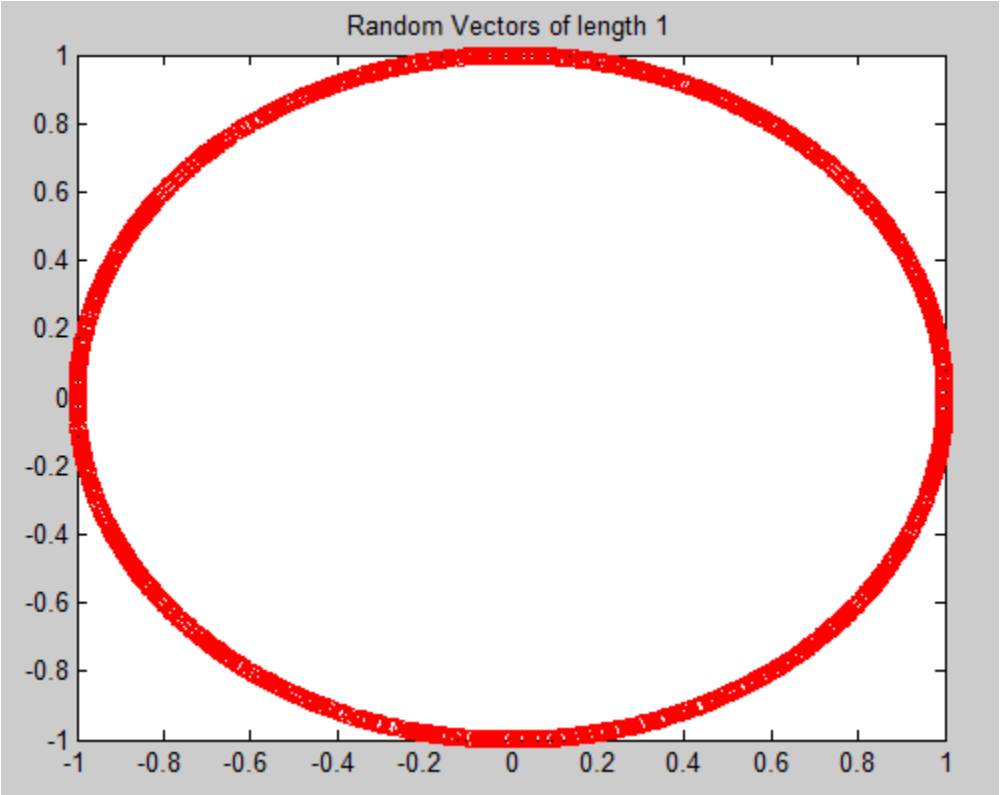

Figure 19 resulting picture is circle with radius 1

Now we can multiply our points by a matrix and plot the result. The statement "hold on" allows us to plot the new plot over the old plot.

```
matrix = [5 2; 3 3];
transform = matrix*nVectors;
Xt = transform(1,:);
Yt = transform(2,:);

hold on;
plot(Xt,Yt,'bo');
```

Figure 20 Transforming points

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

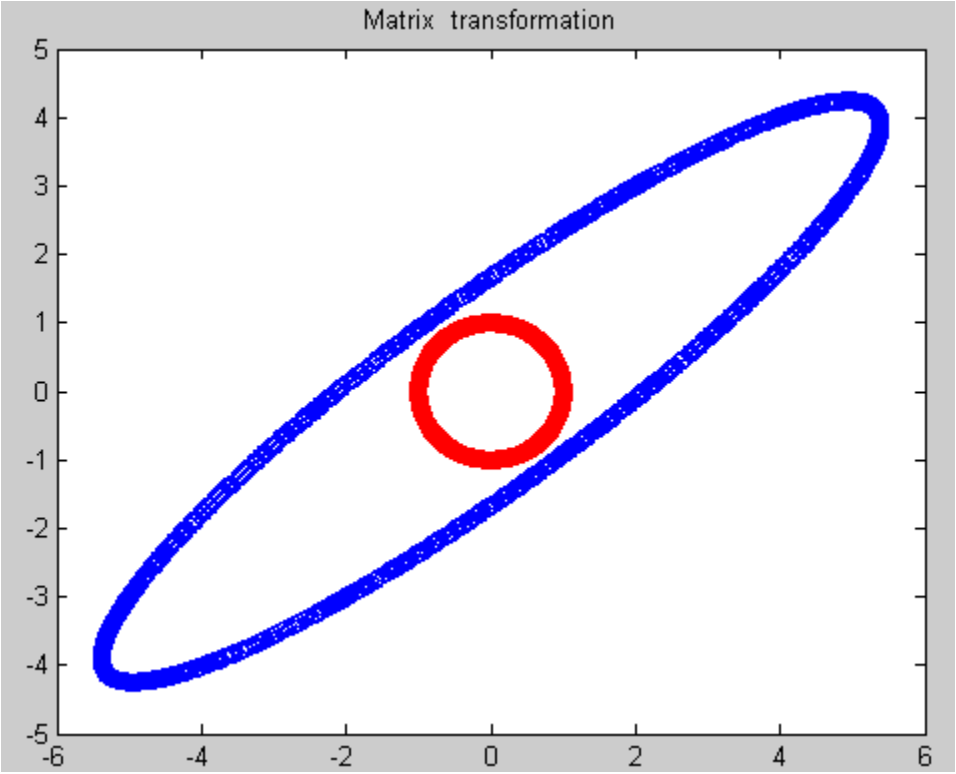

Figure 21 Result of matrix transformation

From the above figure you can see that the Eigen vectors are the vectors that correspond to the points furthest from the center of the ellipse and the closest points to the center of the ellipse. We can show this by actually finding the Eigen values.

```
>> [EigenVectors, EigenValues] = eig(matrix)

EigenVectors =

    0.7722   -0.4810
    0.6354    0.8767


EigenValues =

    6.6458         0
         0    1.3542
```

Now that we know the direction of the actual Eigen vectors we can plot a line in those directions to show where they are on our plot.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

```
[EigenVectors, EigenValues] = eig(matrix);
line = linspace(-5,5,1000);
Eig1Line = line*(EigenVectors(2)/EigenVectors(1));
Eig2Line = line*(EigenVectors(4)/EigenVectors(3));

plot(line,Eig1Line,'g.');
plot(line,Eig2Line,'g.');
```

Figure 22 script to plot Eignen Vector's direction

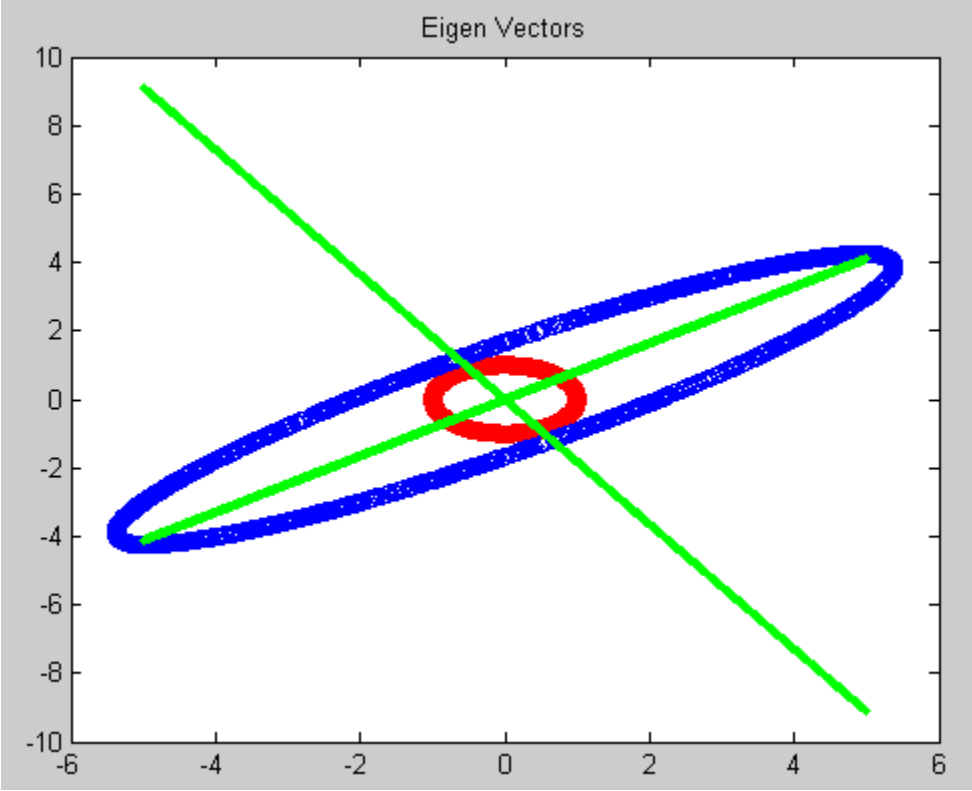

Figure 23 The visual description of Eigen Vectors

For extra credit, try changing the matrix and interpret the result.

Matrix Rotations

The code for this section can be found in rotation.m .The two dimensional rotational matrix is in the following form.

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

This matrix takes any vector and rotates it anti clockwise about the origin. Let's rotate a square. We need to first create a square.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

```
inc = 0.1;
side = -1:inc:1;
O = ones(1,length(side));
mO = -1*O;

right = [O;side];
left = [mO;side];
top = [side;O];
bottom = [side;mO];

square = [top, bottom, left, right];
```

Figure 24 Making a square

Now we can plot our result.

```
X = square(1,:);
Y = square(2,:);

plot(X,Y,'b.');axis([-2 2 -2 2]);
title('The Square');
```

Figure 25 Plotting the square as blue dots

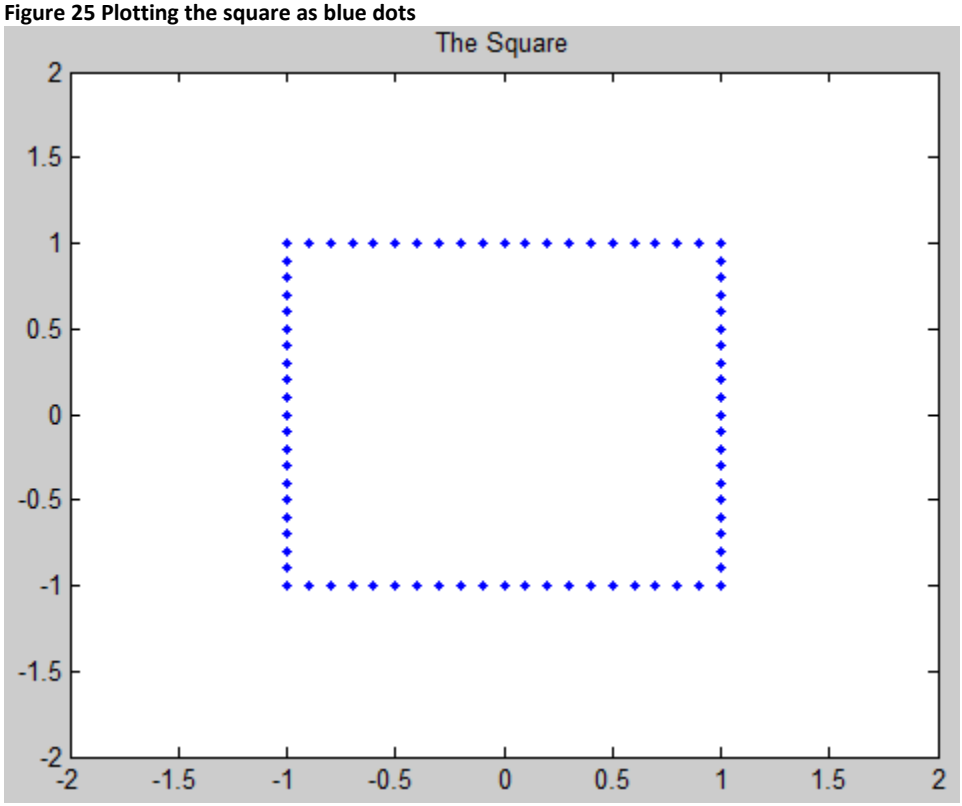

Figure 26 The original square

Now let's create the rotation matrix.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

```
%make one degree angle
angle = pi/180*30;
rotationMatrix = [cos(angle) -sin(angle); sin(angle) cos(angle)];
```

Finally, let's see what a rotation looks like.

```
rotatedSquare = rotationMatrix*square;
Xt = rotatedSquare(1,:);
Yt = rotatedSquare(2,:);

hold on;
plot(Xt,Yt,'r.');
```

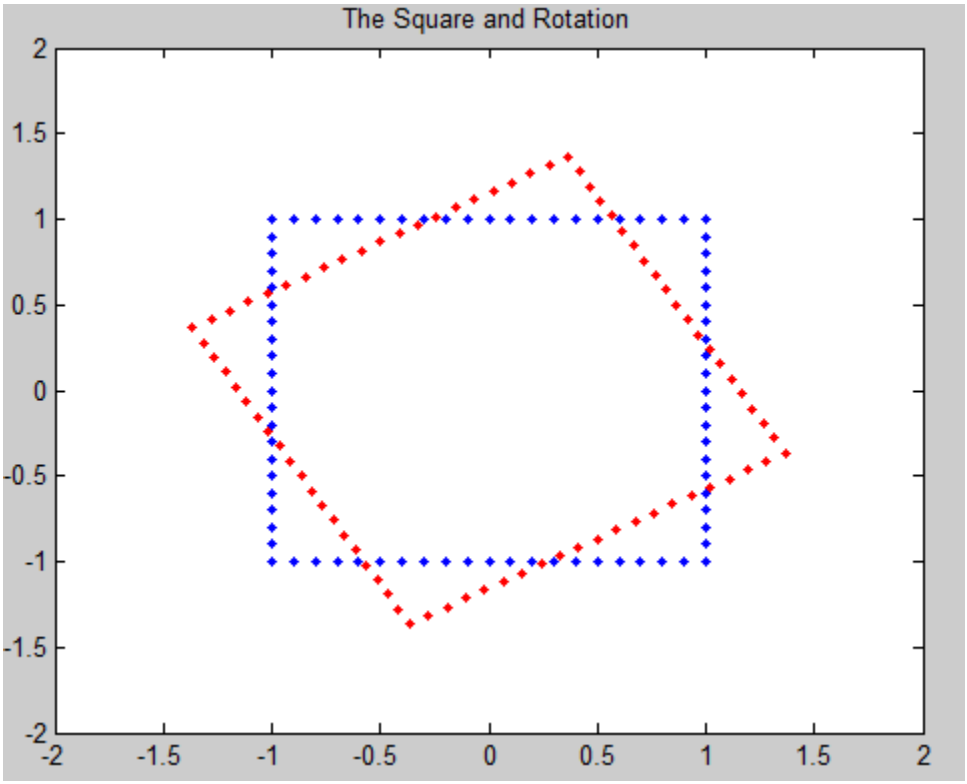

Figure 27 rotated square

We can take this one step further by looping over transformations. Let's put our code into a loot and see the visual spectacle of rotating a square several times.

```
for i=1:180
    rotatedSquare = rotationMatrix*rotatedSquare;
    Xt = rotatedSquare(1,:);
    Yt = rotatedSquare(2,:);
    plot(Xt,Yt,'r.');
    title('Rotating in a loop');

    hold on;
    pause;
end
```

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

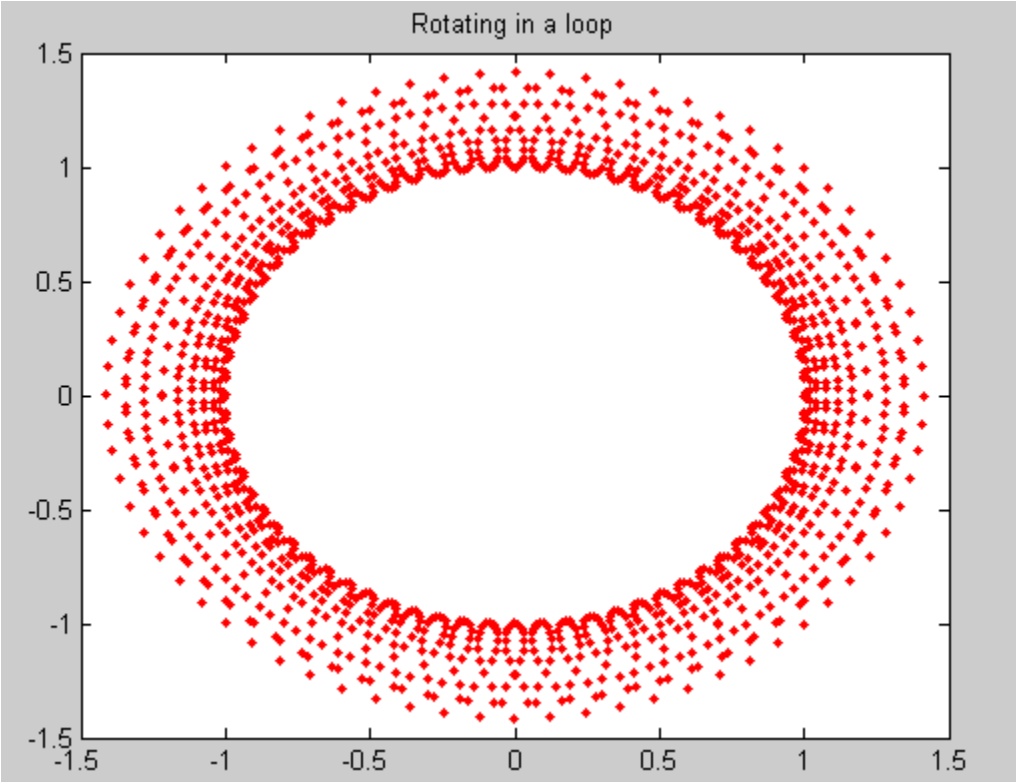

Figure 28 Square after 180 rotations at 35 degrees per rotation

Euler method for solving first order ODEs

The code for this section can be found in EulersMethod.m and TheODE.m. If you're taking Math 263 first order differential equations might seem easy. However, there are some differential equations that are unsolvable. How would you solve something like the following?

$$\frac{dy}{dx} = 2t + y + e^{y}$$

If we can't solve an equation we can always approximate the solution. And that is what Euler's method is for. Given an initial value we can approximate the solution at a point using the following formula.

$$y_{n+1} = y_n + h * f(t_n, y_n)$$

Where h is the step size and $f(t_n, y_n) = \frac{dy}{dx}$ is the function denoting the derivative of y at a point. Essentially, we take a point and add a little bit of its slope over and over until we reach the point we are looking for.

Solving using loops

Let's try an easy example first.

$$\frac{dy}{dx} = 2t + y \qquad y(0) = 1$$

Let's use Matlab to find y(1) using a step size of 0.2.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

```
1 -    stepSize = 0.2;
2 -    t = 0:stepSize:1;
3 -    yn=1;
4 -    for i=1:(1/stepSize)
5 -        yn = yn + stepSize*(2*t(i) +yn);
6 -    end
```

Figure 29 Euler's method

We start by setting up the values then applying the formula until we reach t=1.

```
yn =

    3.4650
```

Figure 30 Euler's method result

The result is seen in figure 10. Since we chose an easy example, we can compare our result with the actual result. The solution for the ODE is:

$$y(t) = 3e^t - 2(t+1)$$

Which means:

$$y(1) \approx 4.1548$$

Our answer is close but we can do better. Let's reduce the step size to 0.001.

```
1 -    stepSize = 0.001;
2 -    t = 0:stepSize:1;
3 -    yn=1;
4 -    for i=1:(1/stepSize)
5 -        yn = yn + stepSize*(2*t(i) +yn);
6 -    end
```

Figure 31 Eulers method with smaller step sizee

Now our answer is much better

```
yn =

    4.1508
```

Figure 32 Eulers method answer with step size 0.001

Generalizing a formula

Now that we've got an idea of how to use Euler's method, let's package this solution into a Matlab function.

Shabbir Hussain
EPTS COMP Tutor
Epts.comp@mcgilleus.ca

```
1      %this fucntion returns the result of euler's method
2      %given any function that represents the derivative of y
3      function result = eulersMethod(stepSize, initialValue, point, func)
4 -    t = 0:stepSize:point;
5 -    yn = initialValue;
6
7 -    for n=1:(1/stepSize)
8 -        yn = yn + stepSize*func(t(n),yn);
9 -    end
10
11 -   result = yn;
```

Figure 33 euler method function

We also need to define our ODE as a function.

```
1      function result = TheODE(t,y)
2 -    result = 2*t +y;
```

Figure 34 the ordinary differential equation as a function

Now we can call the eulersMethod function to solve our ode.

```
>> res = eulersMethod(0.001,1,1,@TheODE)

res =

    4.1508
```

Figure 35 calling the solver

Notice we used the @ symbol in order to pass a function as an argument.