1. Implement k-means clustering on Brain MRI. Do not use the built-in Matlab function kmeans. Submit **one file** called `KMeans.m` (do not submit or modify Demo1.m)

```
segmentedImage = KMeans(InIm, numberofClusters, clusterCentersIn)
```

where:

segmentedImage is an image that codes each located cluster with a different intensity value

`InIm` is the input image, which can have multiple values at each pixel (more details below)

`numberofClusters` is the number of clusters to find in the image

`clusterCentersIn` is an optional parameter that specifies the starting centers of the clusters. If this is the empty list [], random initialization is used.

Hint: When random initialization is used, it is a good idea to run k-means several times (5 is good) with different random initializations and keep the best (in terms of distance fit).

Hint: For debugging, a good idea is to make an artificial image that is easily segmented, and to pass good cluster centers to the function to start from.

The script Demo1.m is provided, which does the following:

1) Loads t1 t2 and pd images

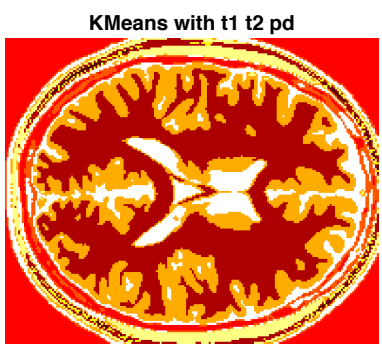
2) Applies k-means clustering with k=8 and random initialization

3) Creates two new images that encode the x and y coordinates of each pixel in the image

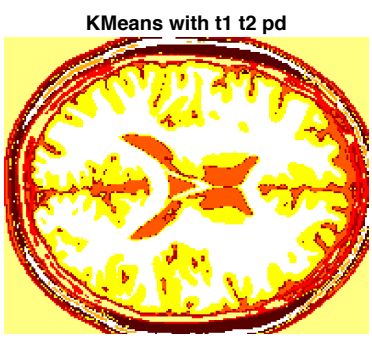
4) Adds these to the t1 t2 pd values so that each feature vector is both intensity and location

5) Repeats k-mean clustering to this image and displays the results

Figure below shows an example output of the program



Note that your result can be different. For example, the segment 1 (white color) can be another brain region (since cluster centers are initially randomly located). Therefore, the below result can be obtained by running the program another time:



2. Implement a line-finder using RANSAC. Submit **one file** called `RansacLine.m` (do not submit or modify Demo2.m). Call this function:

```
lines = RansacLine(edgeImageIn, noIter, fitDistance, noPts, minD)
```

`lines` is an n by 3 matrix parameterizing lines in the plane

`edgeImageIn` is a binary edge image

`noIter` is the number of iterations that you have to pick 2 points at random

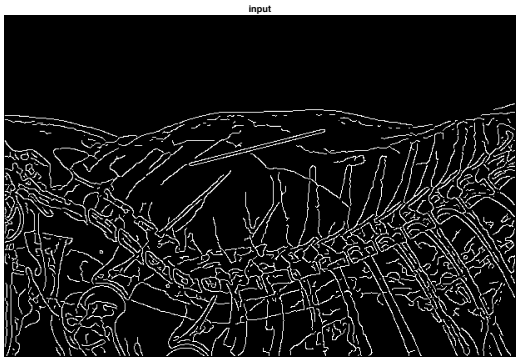
`fitDistance` is the maximum distance a pixel may lie from a line

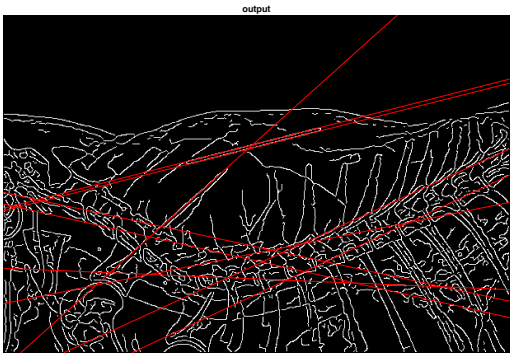
`noPts` is the minimum number of points that should vote for a line. Note that this is different from the implementation discussed in class where we pick the line with max votes. Here, we pick lines that have votes greater than `noPts`

`minD` is the minimum distance between the 2 randomly selected points. This improves RANSAC's performance because if the 2 original points are close, the line fitted can have inaccurate slope.

Hint hints for your `RansacLine` function**:** Use the matlab function "find" to get the coordinates of all of the pixel locations corresponding to an edge.

The input and output to your file are below.





And the following output is printed after running the code:
line #1: 0.0043569X + -0.00098433Y = 1
line #2: 0.0032443X + -0.00013411Y = 1
line #3: 0.0023285X + 0.0020893Y = 1
line #4: 0.0022508X + 0.0010309Y = 1
line #5: 0.0028469X + 0.0005698Y = 1
line #6: 0.0046447X + -0.001006Y = 1
line #7: 0.0020629X + 0.00097296Y = 1
line #8: 0.0041957X + 0.001061Y = 1
line #9: 0.0042487X + 0.001086Y = 1
Note that the number of lines can be different if you run the code again. For example, if we run the same code without changing anything, we might get 6 lines.
To make sure your code is correct, you can test your code with other images, such as:

```
inIM = imread('circuit.tif');
inIM = imread('gantrycrane.png');
```