Assignment 4

For the fourth assignment I utilized python 3.7 alongside opencv2 and numpy in order to conduct two different types of image segmentation on a given set of images and display before and after comparison pictures. The two image segmentation algorithms used are the Basic Global Thresholding algorithm and the K-Means Clustering algorithm, both of which consist of multiple steps which will be discussed further on. Once each image is run through a given algorithm, it is saved for later display and written to a ‘plots’ folder inside the ‘resources’ folder.

I began the project by importing all my necessary libraries inside both the *main.py* module, and the *threshold.py* module which contains the algorithms and their associated logic. I then set the matplotlib library to use the ‘TkAgg’ backend in order to force my ide to display any matplot images in a separate window. Next I read in all the given images using opencv2’s *imread()*  function as grayscale e.g.:

*im\_1001a = cv2.imread('resources/Fig1001(a)(constant\_gray\_region).tif', 0)*

The *main.py* module was then used for testing the implementation of my image segmentation algorithms and upon their completion it was updated with functions to run the algorithms on all the imported images and then display their results.

For the first image segmentation algorithm, Basic Global Thresholding, I first created an algorithm called *find\_grey\_avg()* which computes the average grey value of an entire grayscale image and then returns it. This is done by summing all grey values within the image and then dividing by the total number of pixels e.g.:

*dimensions = im.shape  
avg = np.sum(im) / (dimensions[0] \* dimensions[1])*

For example, on the given image ‘Fig1026(a)(headCT-Vandy).tif’, as seen below, the average grey value is calculated as 80.



Figure : image 1026a

I then designed an algorithm to compute the global threshold value of the given image. This function, *find\_global\_threshold()*, takes in the image array to be processed, the initial global threshold, and the minimum difference between the new and old global threshold. It then recursively iterates through the array, repeatedly breaking it in half and calculating the average values of the pixels. This continues until the difference between a successive set of global thresholds is less than the specified minimum difference, upon which the recursion completes and the final global threshold is returned. For example, for the previously used image ‘Fig1026(a)(headCT-Vandy).tif’ the histogram of its values looks like:

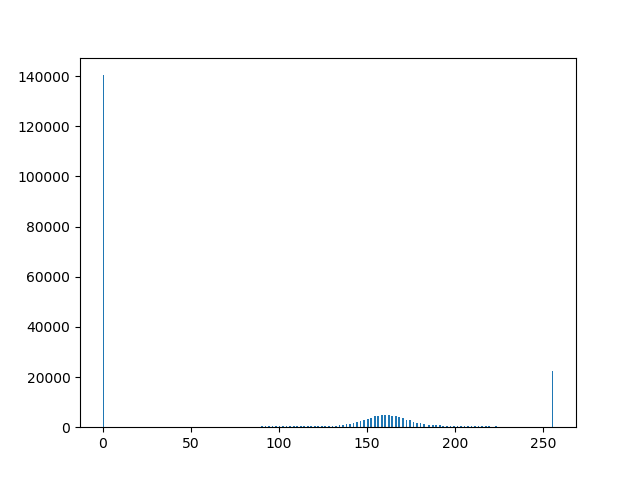


Figure : image 1026a histogram

Using a starting global threshold of the average grey value of the entire image and a minimum difference of 2, the algorithm then returns a final global threshold value of 91. This function can be seen below:

*def find\_global\_threshold(im: np.ndarray, diff: int, t: int):  
 t\_prev = t  
 count1 = 0  
 count2 = 0  
 sum1 = 0  
 sum2 = 0  
 with np.nditer(im, op\_flags=['readonly']) as it:  
 for x in it:  
 if x[...] > t:  
 count1 += 1  
 sum1 += x[...]  
 else:  
 count2 += 1  
 sum2 += x[...]  
  
 t = int(((sum1 / count1 + sum2 / count2) / 2))  
 if abs(t - t\_prev) < diff:  
 return t  
 else:  
 return find\_global\_threshold(im, diff, t)*

As seen above, I chose to use the *numpy.nditer()* function and style for the sake of efficiency and consistency as I am using *numpy.ndarray* types for my image arrays.

Finally, I set up the *global\_threshold()* function for using the found global threshold to set the new image values. Taking in the image array to be processed and the global threshold, it then iterates through the array and sets all values greater than the threshold value to 1 and the rest to 0. This segmented image is then returned for saving and display purposes, where it is also normalized to achieve a visible white/black output. The *global\_threshold()* function is shown below:

*def global\_threshold(im: np.ndarray, t: int):  
 im2 = im.copy()  
 with np.nditer(im2, op\_flags=['readwrite']) as it:  
 for x in it:  
 x[...] = 1 if x[...] > t else 0  
  
 return im2*

And the resulting image when performed on the previously specified input image using the same values:



Figure : Global Thresholding on image 1026a

Note that there is an additional function, *global\_threshold\_apply(),* which combines the previous steps for Basic Global Thresholding:

*def global\_threshold\_apply(im: np.ndarray, diff: int):  
 t = find\_global\_threshold(im, diff, find\_grey\_avg(im))  
  
 return global\_threshold(im, t)*

Next is the second image segmentation algorithm, K-Means Clustering. Currently, this algorithm is contained within a single function called *k\_means()*. To start, the clustering function takes in the image array to be processed, the number of clusters to use, and the type of cluster initialization to use. The options for cluster initialization are ‘rand’ for random unique values between the minimum and maximum values of the image pixels and ‘spaced’ which selects evenly spaced values within the minimum and maximum values of the image pixels according to the number of clusters specified. For the purposes of the project, the latter initialization type will always be used. An assortment of variables for holding values are then initialized, and the input image array is flattened into a one-dimensional array. Next, the starting cluster centers are chosen according to the specified initialization type e.g.:

*if initial == 'rand':  
 centroids = random.sample(range(min\_val, max\_val+1), k)  
elif initial == 'spaced':  
 idx = np.round(np.linspace(min\_val, max\_val, k)).astype(int)  
 centroids = idx*

The next major step in the algorithm is a loop which handles the logic for assigning each pixel in the image array to its closest cluster and updating the cluster centers accordingly. This loop runs until the cluster centers no longer move. Within the loop, it iterates through each pixel and calculates the distance between it and each cluster center, then sets the index for the nearest cluster in an array called *cluster\_array[]* which tracks the nearest cluster for every pixel in the image. This segment of code can be seen below:

*for i in range(im\_flat.size):  
 dist\_array = np.zeros([k])  
 for j in range(k):  
 dist\_array[j] = np.linalg.norm(im\_flat[i]-centroids[j])  
 cluster\_array[i] = int(np.argmin(dist\_array))*

Note the use of *numpy.linalg.norm()* for calculating the distance between points instead of simply subtracting and taking the absolute value. This function was chosen in order to better handle any future addition of handling non-single dimensional values.

After assigning each pixel to a cluster, the function then takes the average of the assigned values for each cluster to update the appropriate cluster center e.g.:

*for i in range(k):  
 count = 0  
 summ = 0  
 for j in range(cluster\_array.size):  
 if k == cluster\_array[j]:  
 count += 1  
 summ += im\_flat[j]  
 centroids[i] = int(summ/count) if count > 0 else centroids[i]*

The loop containing the above logic then continues to iterate using the new cluster centers until they no longer change.

Lastly, the algorithm takes the final cluster centers and associated pixels and assigns each pixel its associated cluster’s center value. This final array is then reshaped back into a two-dimensional image array and returned e.g.:

*for i in range(k):  
 for j in range(im\_flat.size):  
 if cluster\_array[j] == i:  
 im\_flat[j] = centroids[i]  
  
return im\_flat.reshape([dimensions[0], dimensions[1]])*

An example output, performed on given image ‘Fig1060(a)(car on left).tif’ is shown below with the original image followed by a processed version which has had the K-Means Clustering algorithm performed on it with a K(number of clusters) value of 3.



Figure : image 1060a

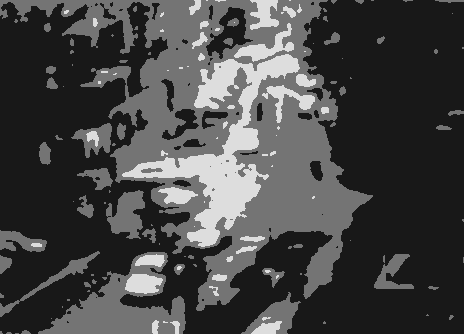


Figure : K-Means Clustering on image 1060a with K = 3

Additionally, a second processed image using the same original image and algorithm but with a K size of 5 is shown as well. This illustrates the difference a change in the number of clusters can have.

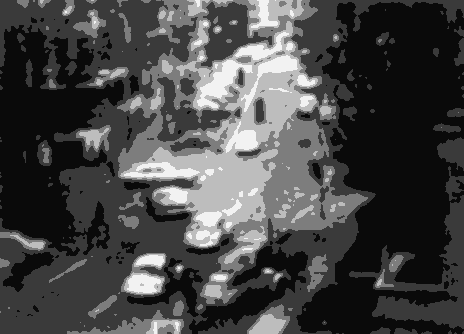


Figure : K-Means Clustering on image 1060a with K = 5

For an additional visual, displayed below are the histograms for the original image as well as for the outputs of the K-Mean Clustering algorithm performed with a K of 3 and 5.

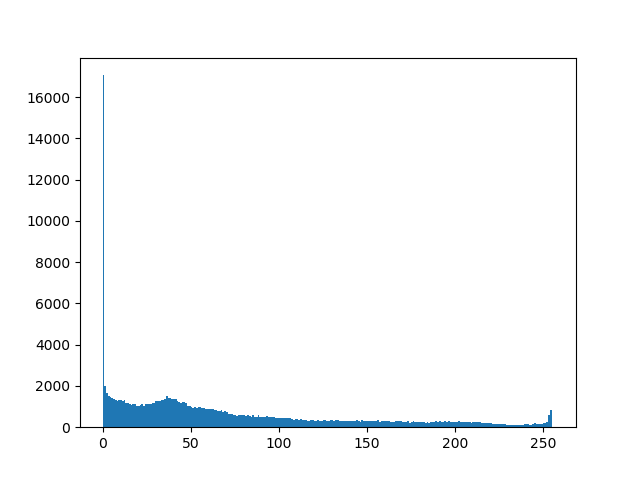


Figure : image 1060a histogram

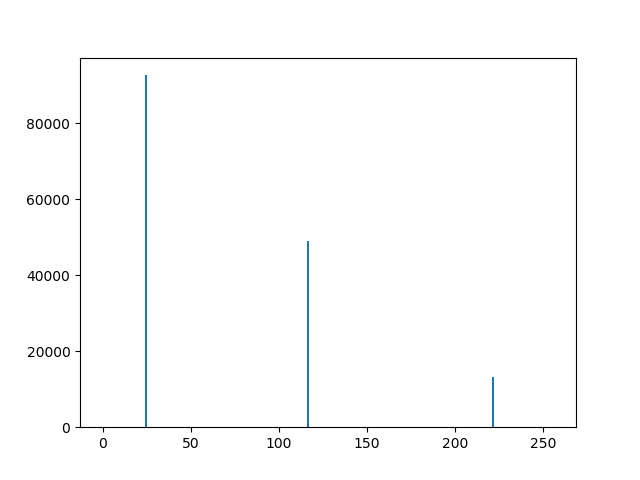


Figure : K-Means Clustering on image 1060a with K = 3 histogram

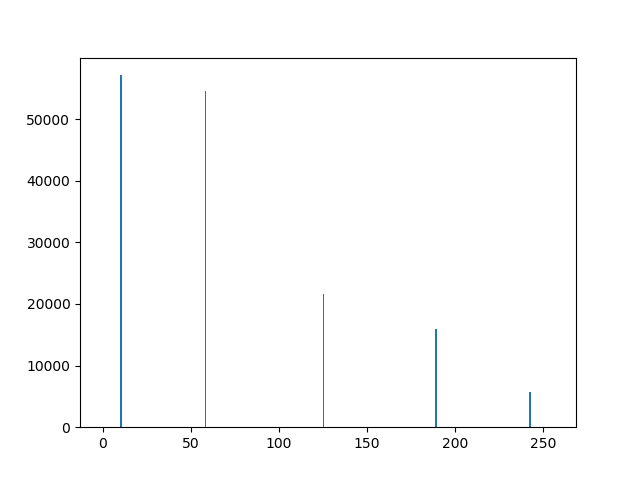
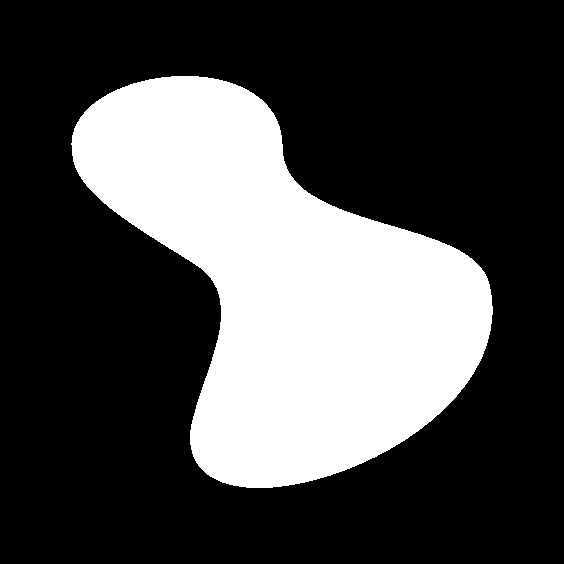


Figure : K-Means Clustering on image 1060a with K = 5 histogram

Two additional sets of images are appended to this report. These images are meant as additional examples illustrating the functionality of the two image segmentation algorithms created for this assignment. The first, titled “Basic Global Thresholding” contains dual image sets containing an original image followed by the result of the Basic Global Thresholding algorithm being applied to it. The second, titled “K-Means Clustering” contains triple image sets containing an original image, followed by two results. Both results are the output of my K-Means Clustering algorithm being performed on the original image, with the former using a cluster size of 3 and the latter using a cluster size of 5.

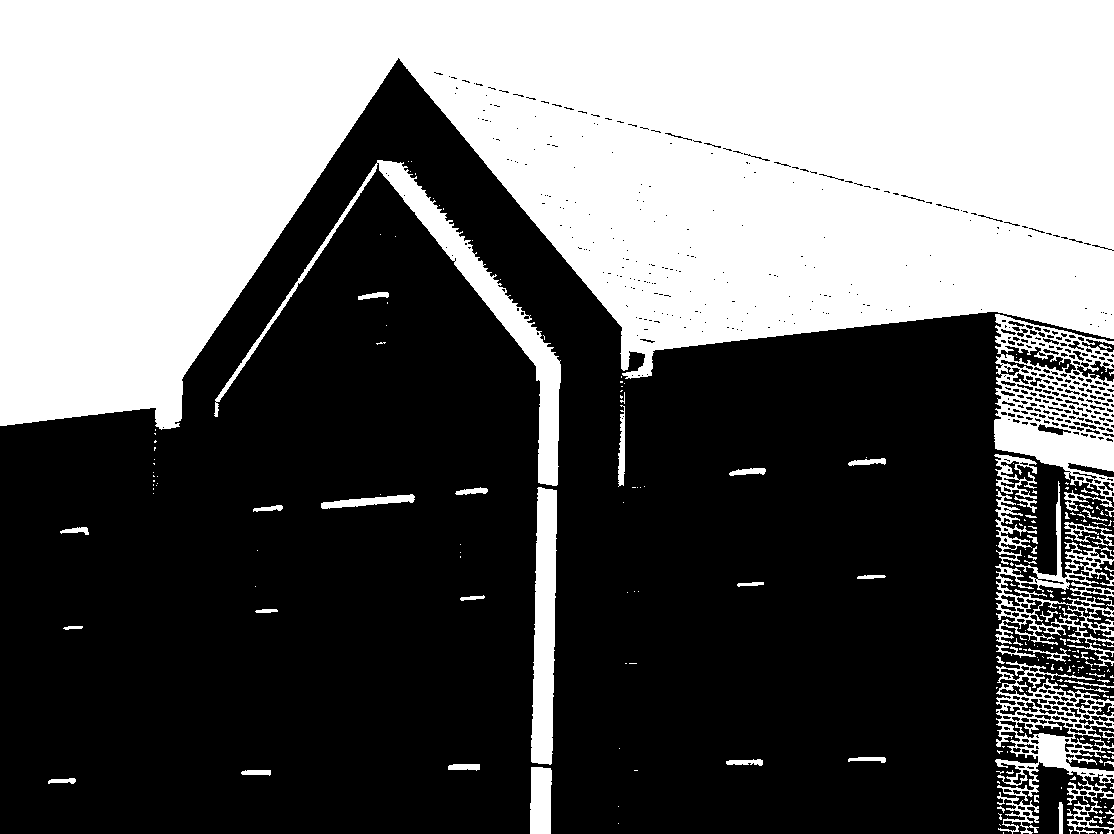
Basic Global Thresholding

Image 1001a:



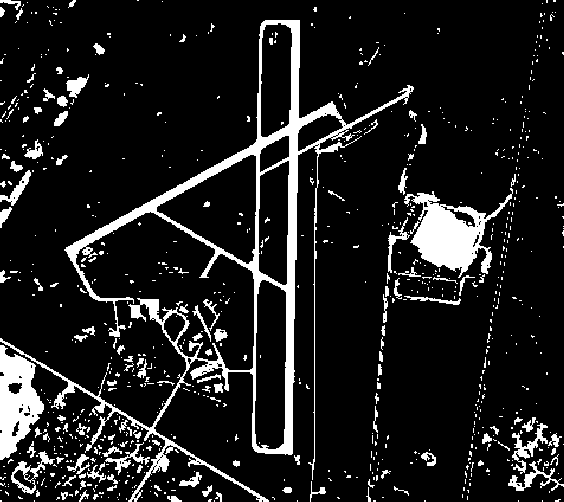
Original Basic Global Thresholding

Image 1022a:



Original Basic Global Thresholding

Image 1034a:



Original Basic Global Thresholding

K-Means Clustering

Image 1022a:

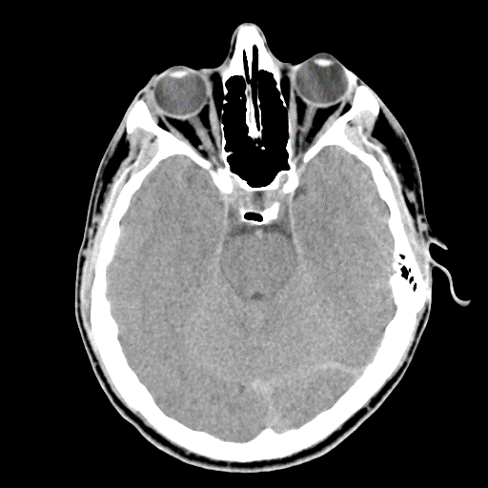


Original



K = 3 K = 5

Image 1026a:



Original

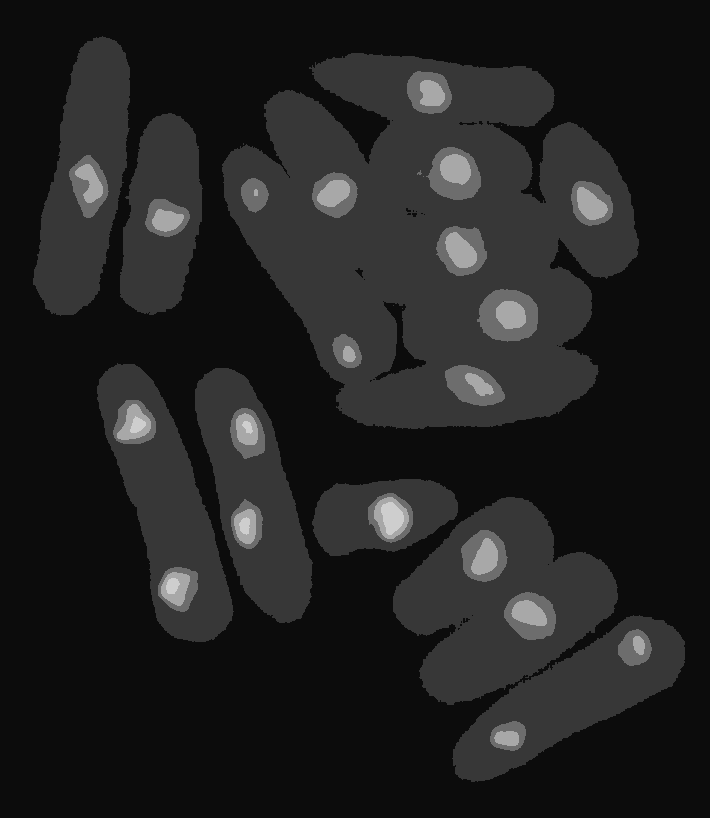
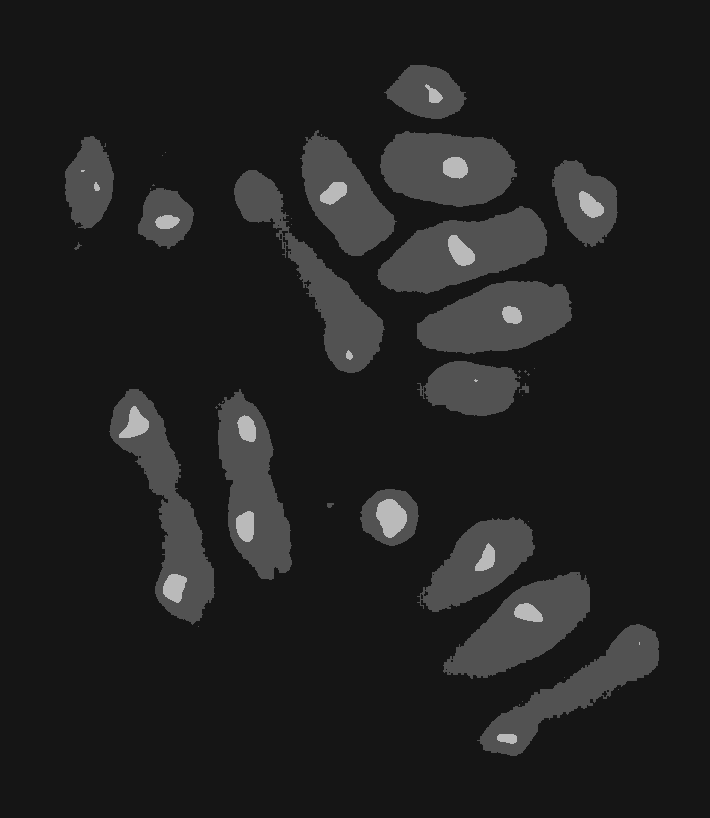


K = 3 K = 5

Image 1043a:



Original



K = 3 K = 5