BILKENT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

CS484
HOMEWORK 3

BERAT BİÇER
21503050

Dec. 16, 2018
ANKARA, TURKEY

Required Fields

Description of the parameters used for superpixel segmentation
We used MATLAB implementation of superpixel segmentation, with the following prototype:

[L,NumLabels] = superpixels(A,N) where *L* is the label matrix of type double corresponding to superpixel ids and *NumLabels* is the actual number of superpixels computed in the image. Input *A* is the RGB image that we want to find superpixels on, and *N* is the number of superpixels we wish to acquire. Note that MATLAB uses SLIC0 as the default SLIC algorithm. Final function call we used is as follows:

```
[labels, count] = superpixels(tmp, in);
```

Segmentation results for all images using the superpixel segmentation algorithm
We used *N = 250* for this part and SLIC0 for segmentation algorithm as described in previous section.

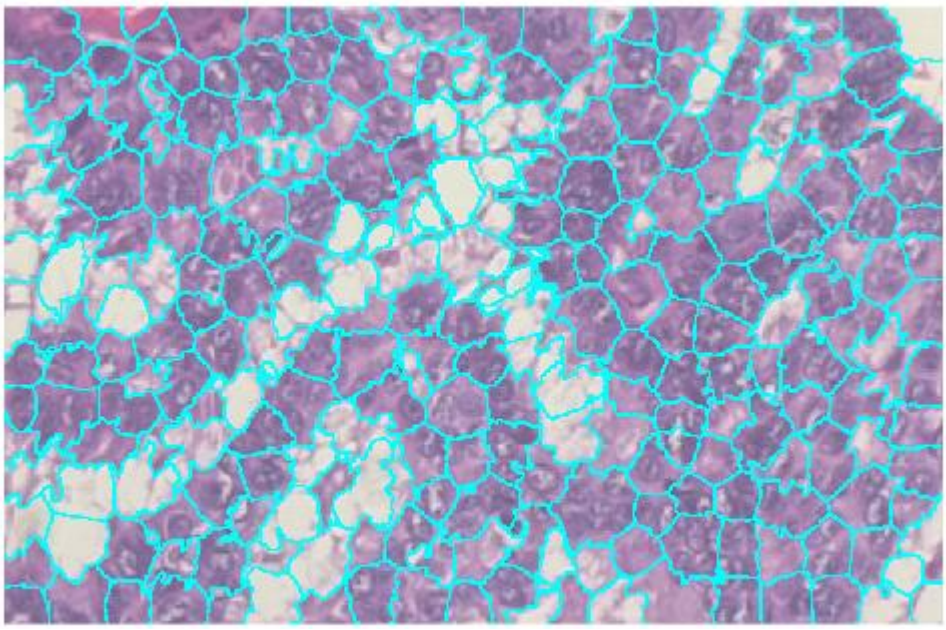

Figure 1: Superpixels for image 01

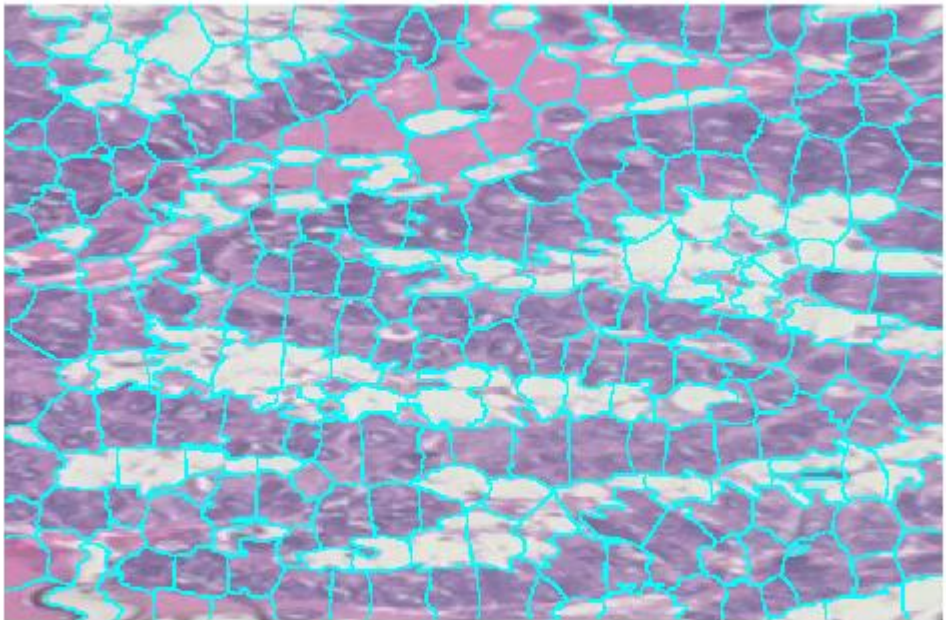

Figure 2: Superpixels for image 02

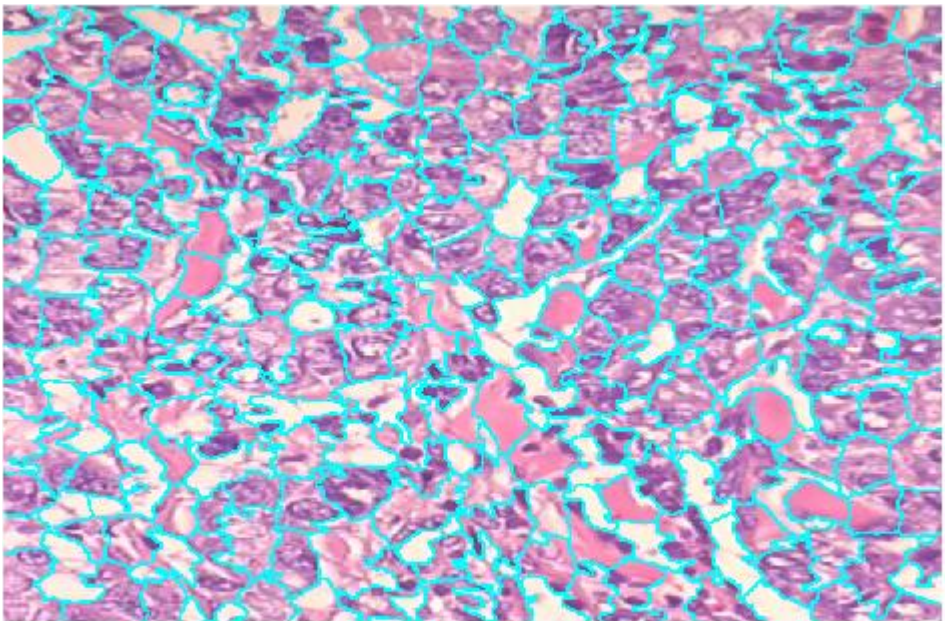

Figure 3: Superpixels for image 03

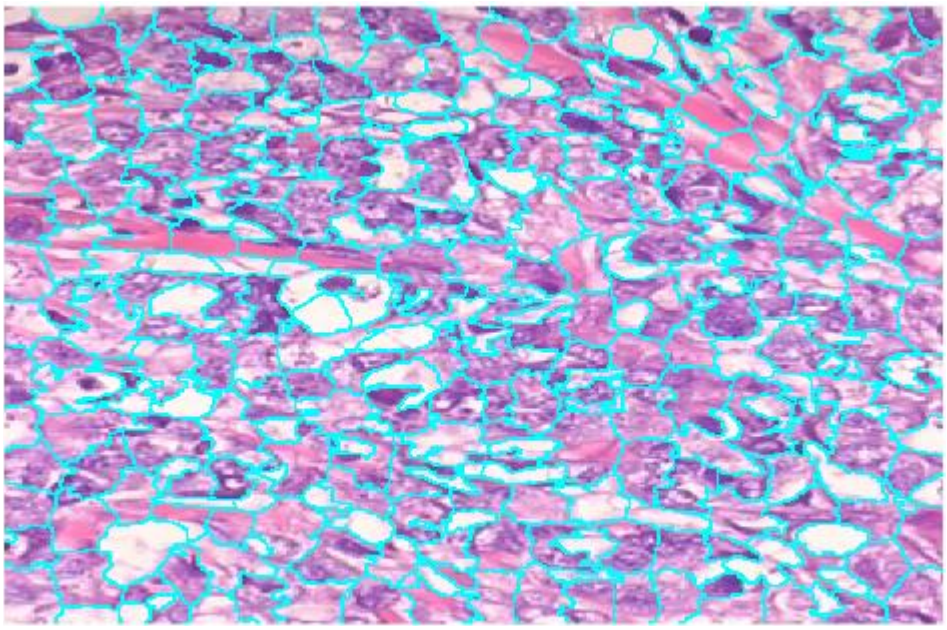

Figure 4: Superpixels for image 04

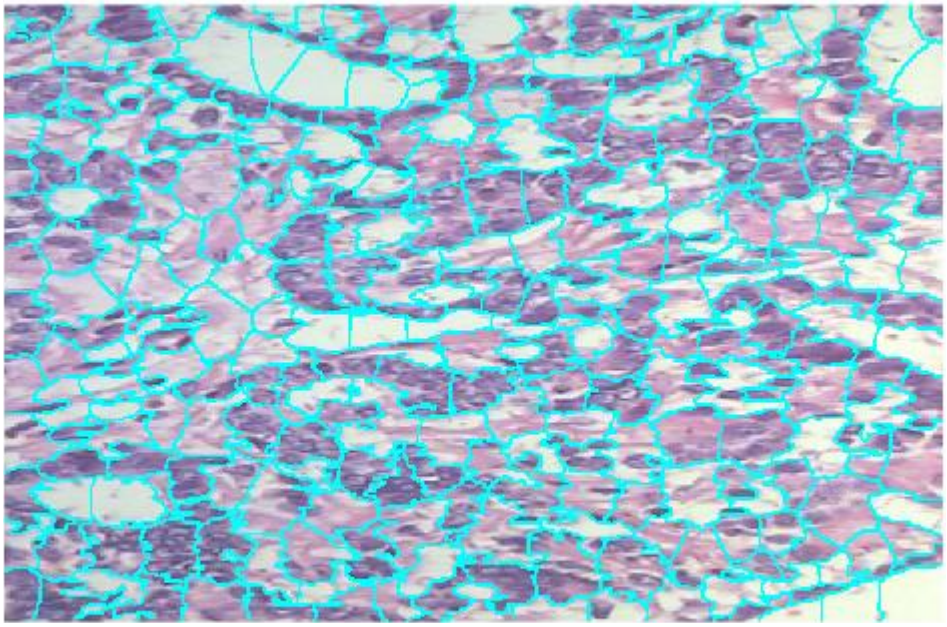

Figure 5: Superpixels for image 05

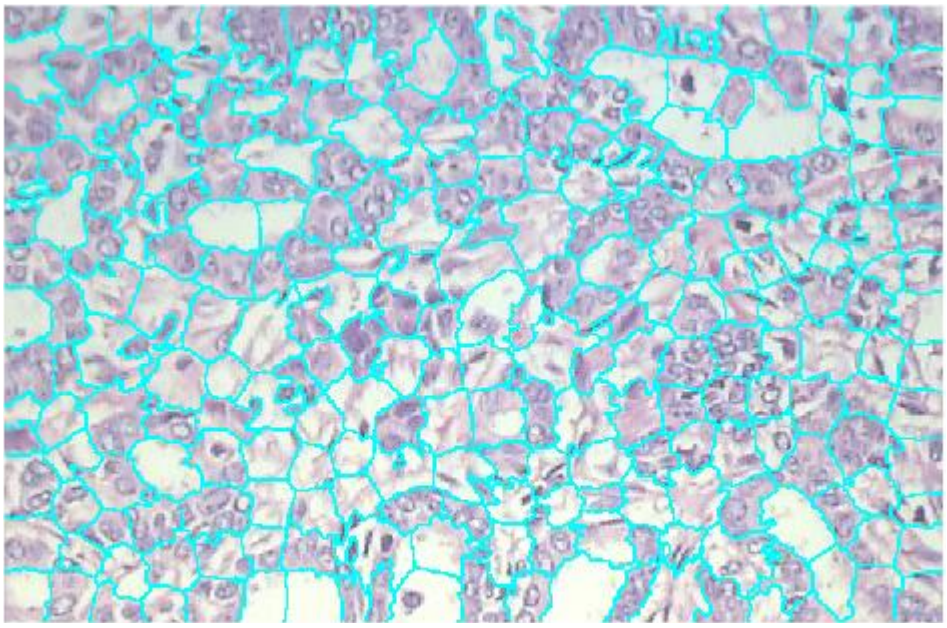

Figure 6: Superpixels for image 06

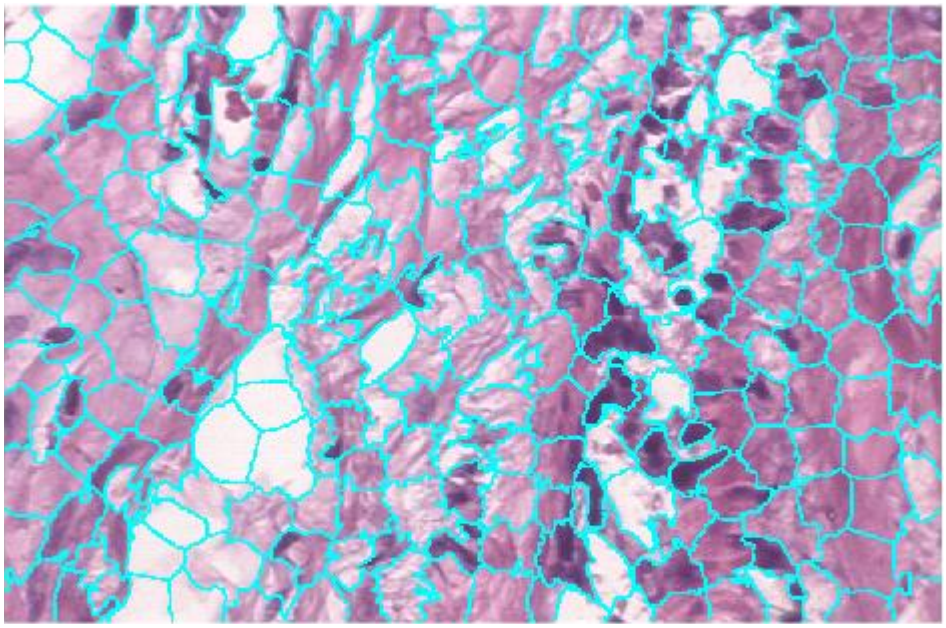

Figure 7: Superpixels for image 07

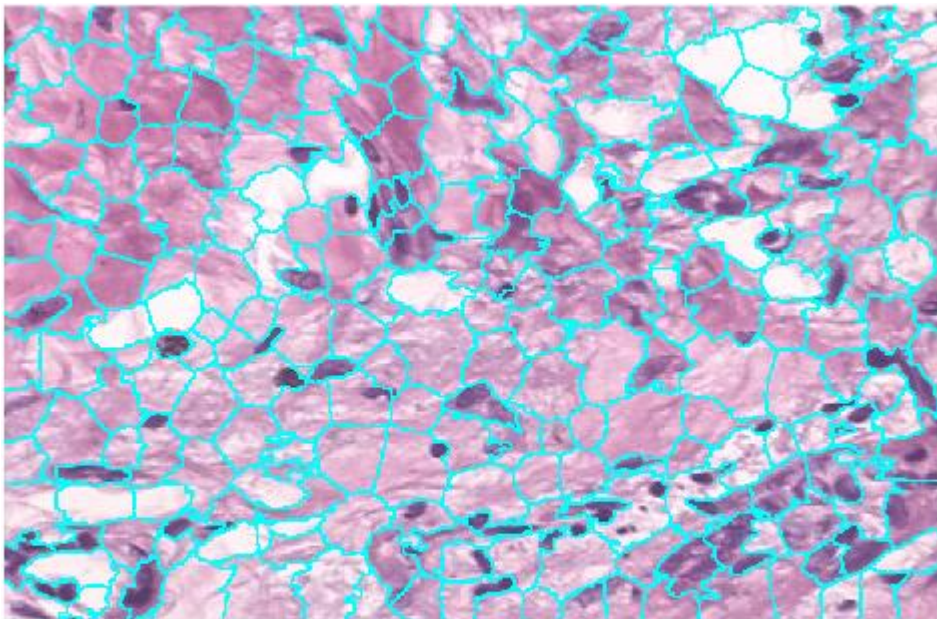

Figure 8: Superpixels for image 08

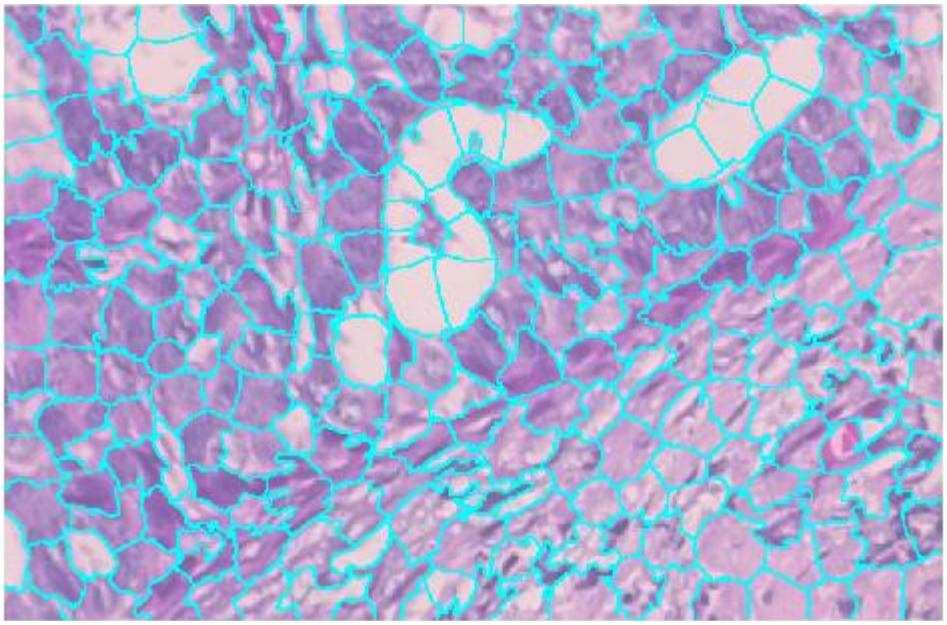

Figure 9: Superpixels for image 09

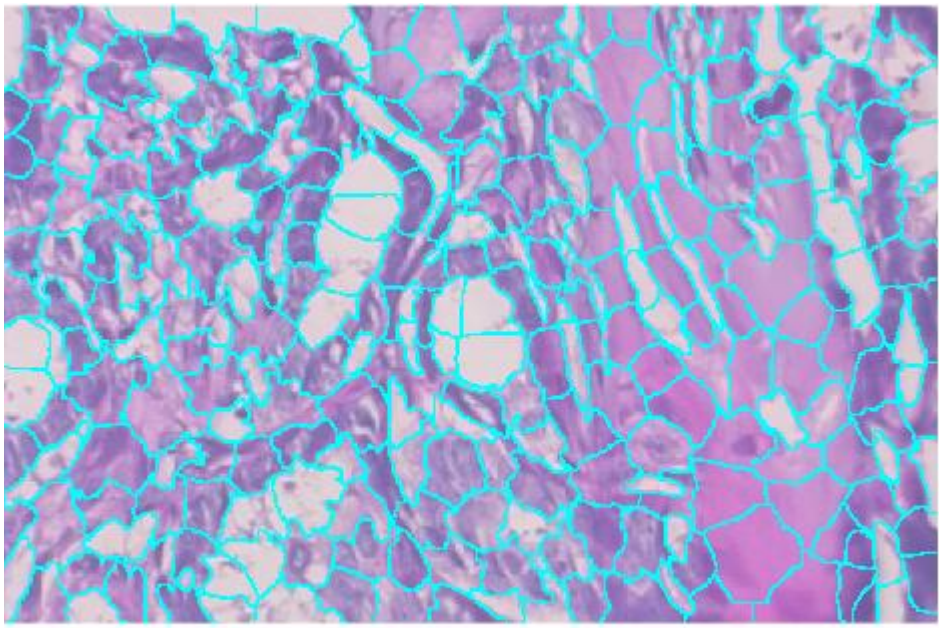

Figure 10: Superpixels for image 10

We observed that as the number of superpixels increase (for example as high as 500), they get flatter but segmentation becomes more accurate. Compared to small a number of superpixels (as 100), segmentation performance decreases but superpixels become larger, which makes it easier to compute representations for them. Therefore, we decided to use a value between the edge cases, which is 250 to get a good segmentation and a reasonable computing time. We now give examples of superpixel segmentation for $N = 100$ and $N = 500$ to solidy our reasoning.

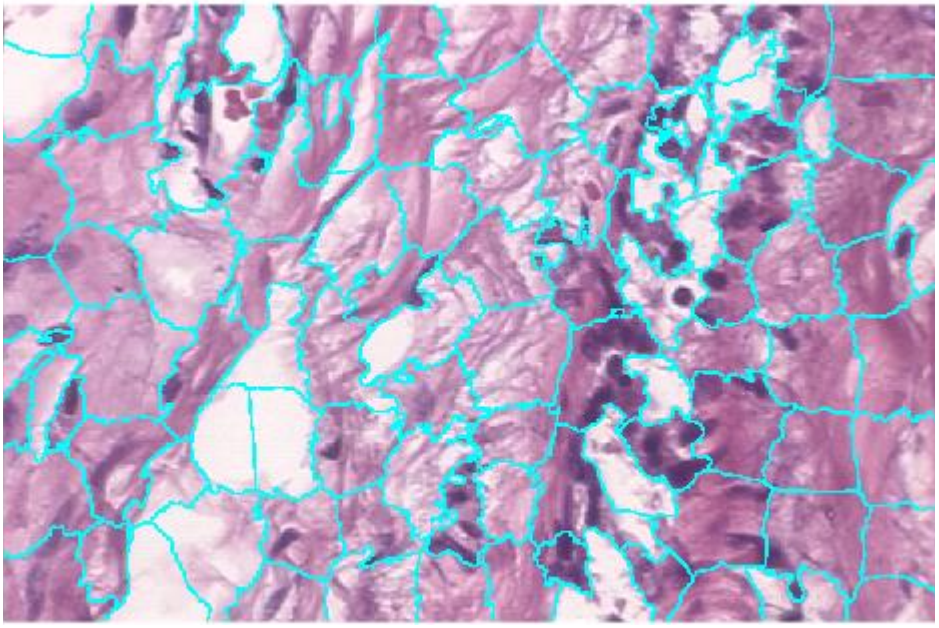

Figure 11: Superpixels for image 07, N = 100

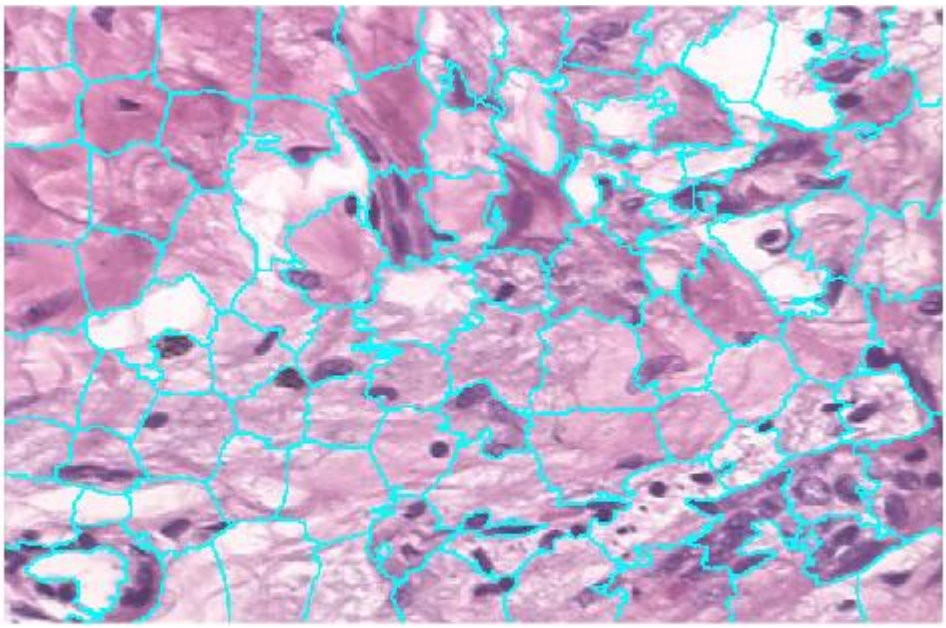

Figure 12: Superpixels for image 08, N = 100

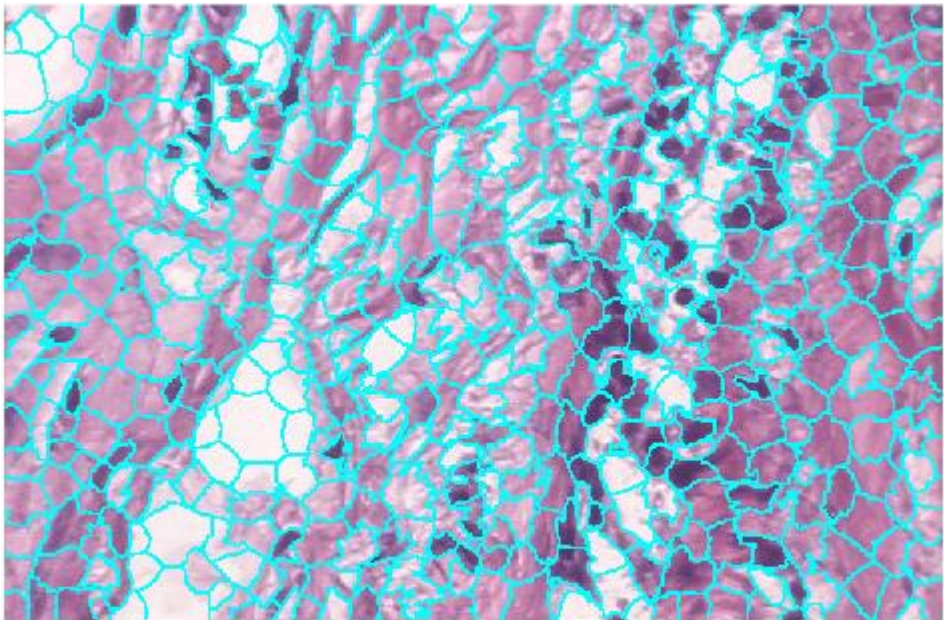

Figure 13: Superpixels for image 07, N = 500

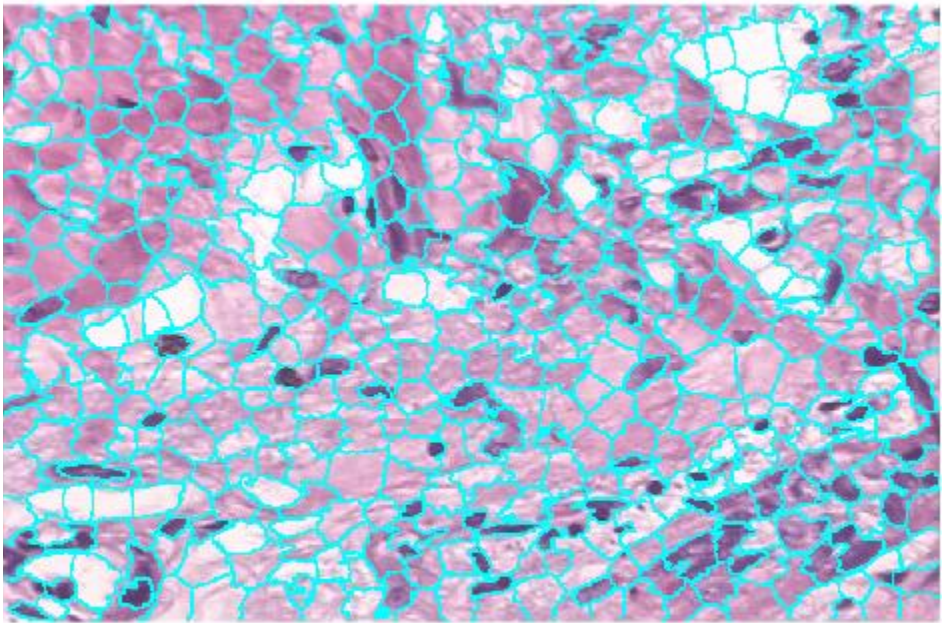

Figure 14: Superpixels for image 08, N = 500

Description of the parameters used for Gabor texture feature extraction
We first created a Gabor filter bank using *gabor* function from MATLAB as follows:

```
gabor_filter_bank = gabor([10 15 20 25], [0 45 90 135]);
```

This function gets 2 vectors as input, first one containing wavelengths and second one containing orientations (in degrees), and creates a gabor filter for each combination of (wavelength, orientation). The output is a *1x16 gabor* vector where each element is a gabor filter.

We then applied each gabor filter acquired before to input images using MATLAB function *imgaborfilt* as follows:

```
[mag, phase] = imgaborfilt(tmp, gabor_filter_bank);
```

The function takes 2 inputs, first one the image which we wish to apply gabor filters on and second one the *1x16 gabor* vector we created before. The output is a 2-vector, first one is the magnitudes of gabor responses and the second one is the phase information of the response.

Gabor texture feature examples
In this section, we display gabor filter responses for each image. Every plot in the graphs correspond to image response to one of 16 gabor filters used, and which gabor filter the response belong to is written above the plot. Note that each tuple above the plots is in the form (wavelength, orientation).

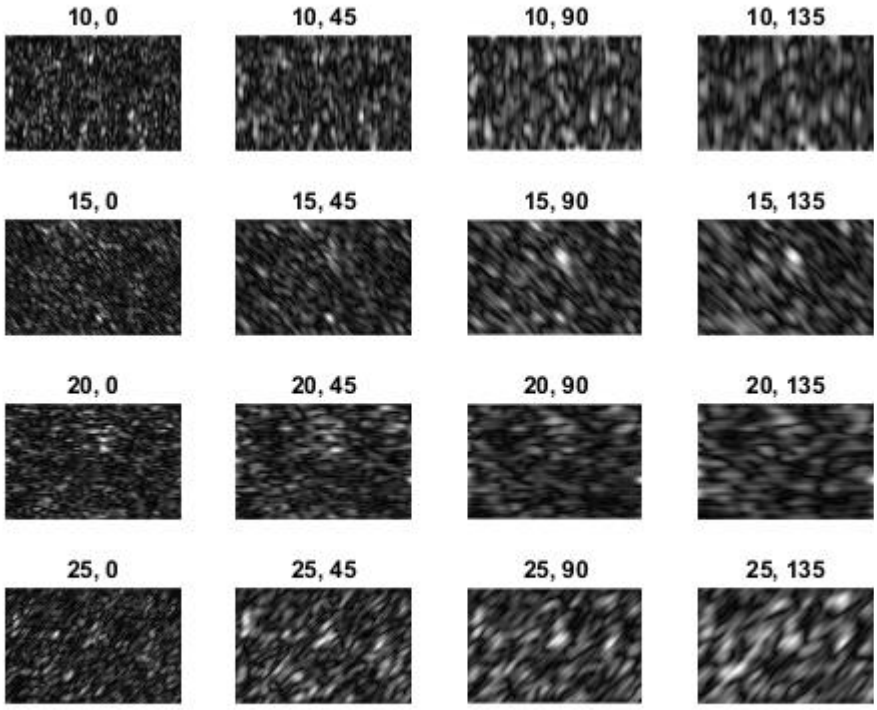

Figure 15: Gabor filter responses for image 01

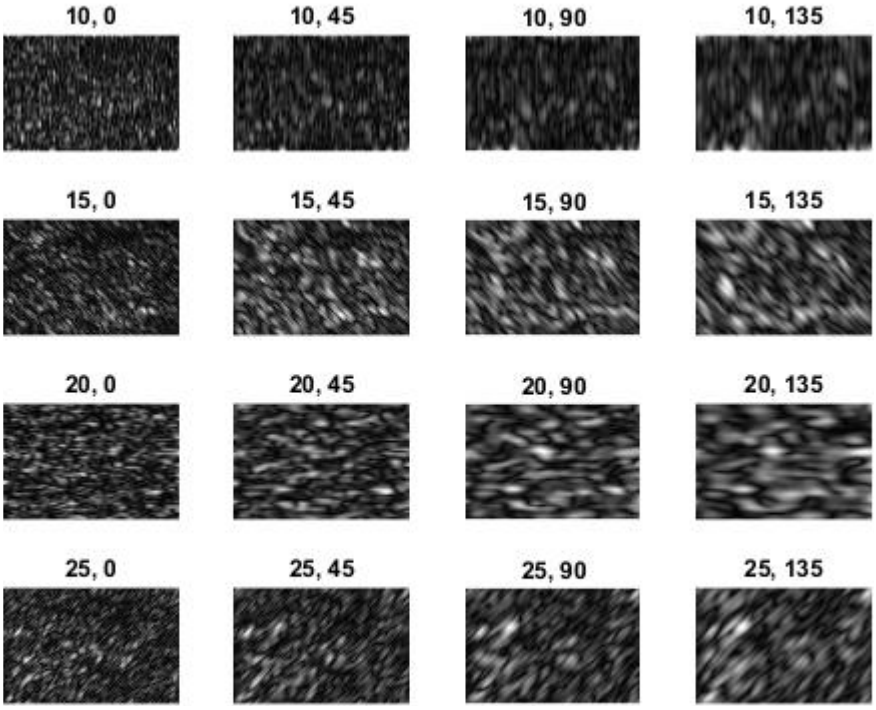

Figure 16: Gabor filter response for image 02

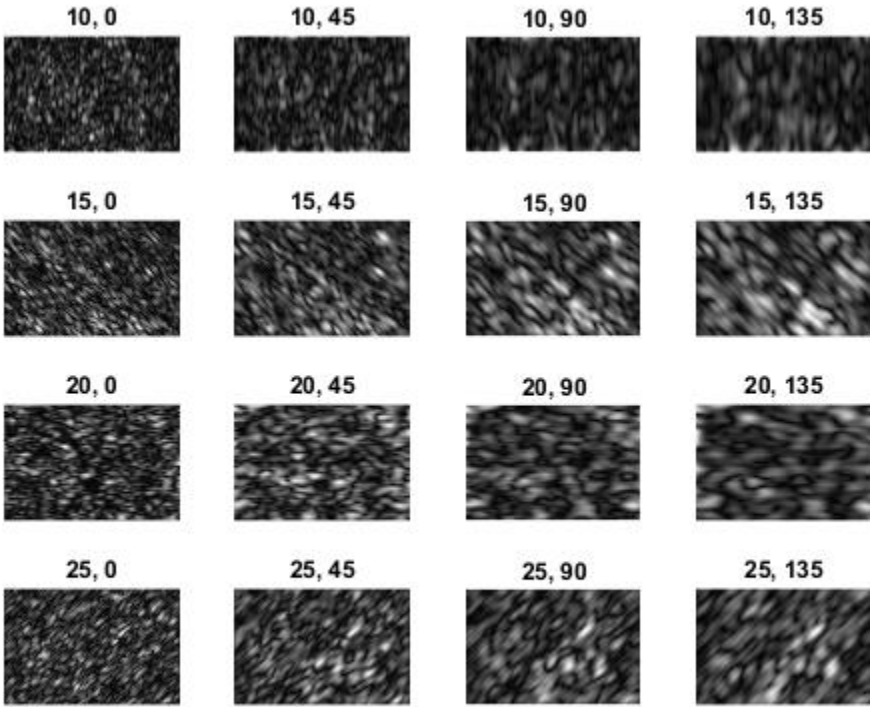

Figure 17: Gabor filter response for image 03

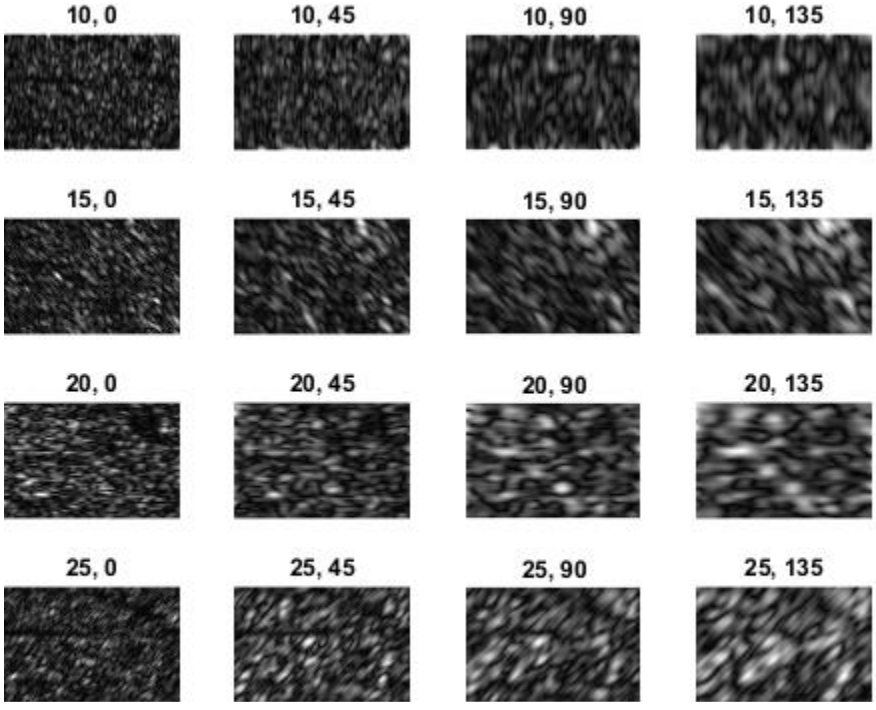

Figure 18: Gabor filter response for image 04

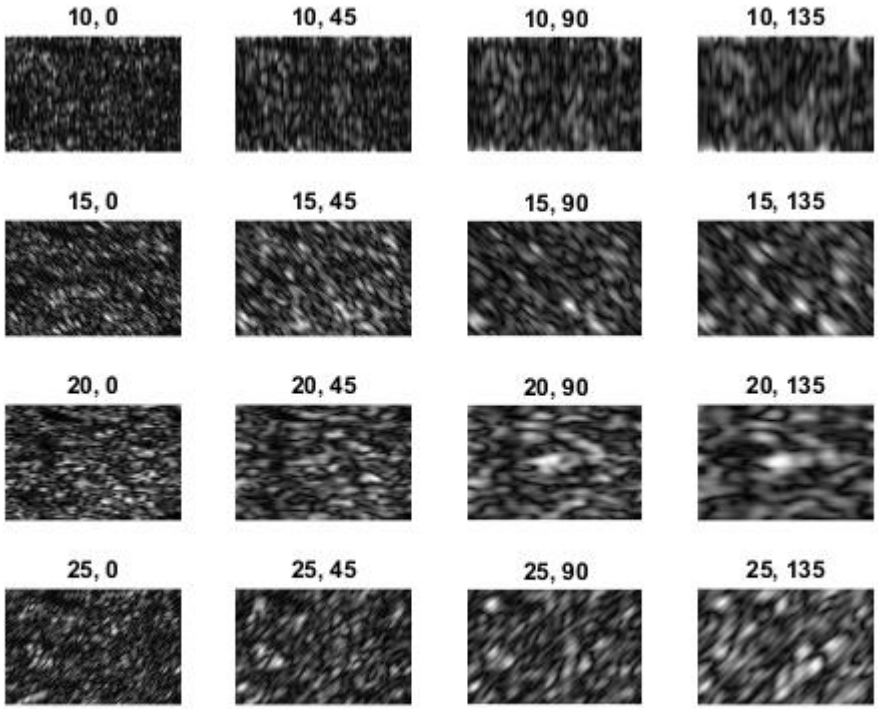

Figure 19: Gabor filter response for image 05

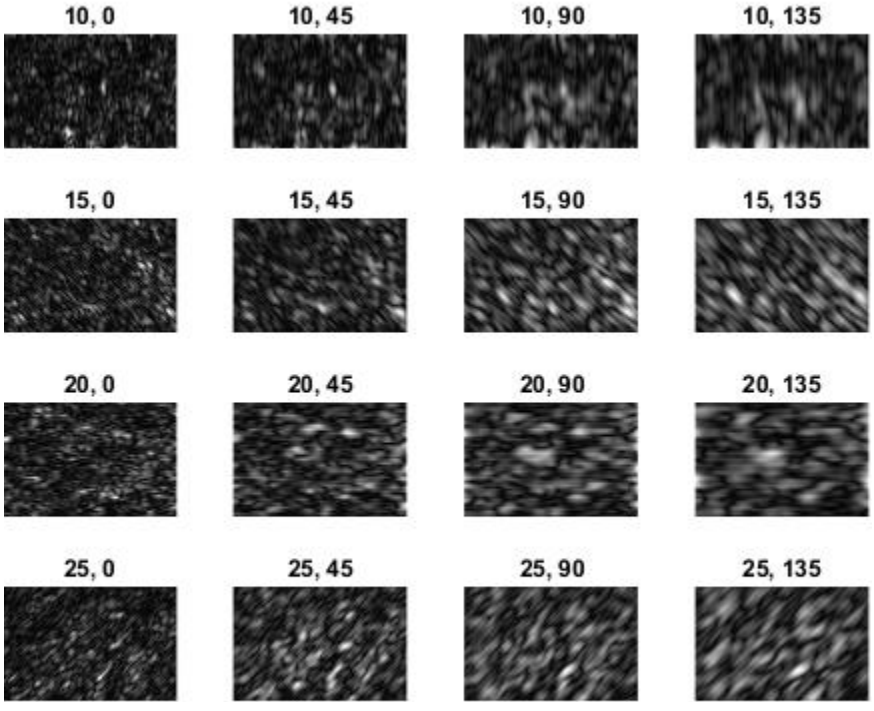

Figure 20: Gabor filter response for image 06

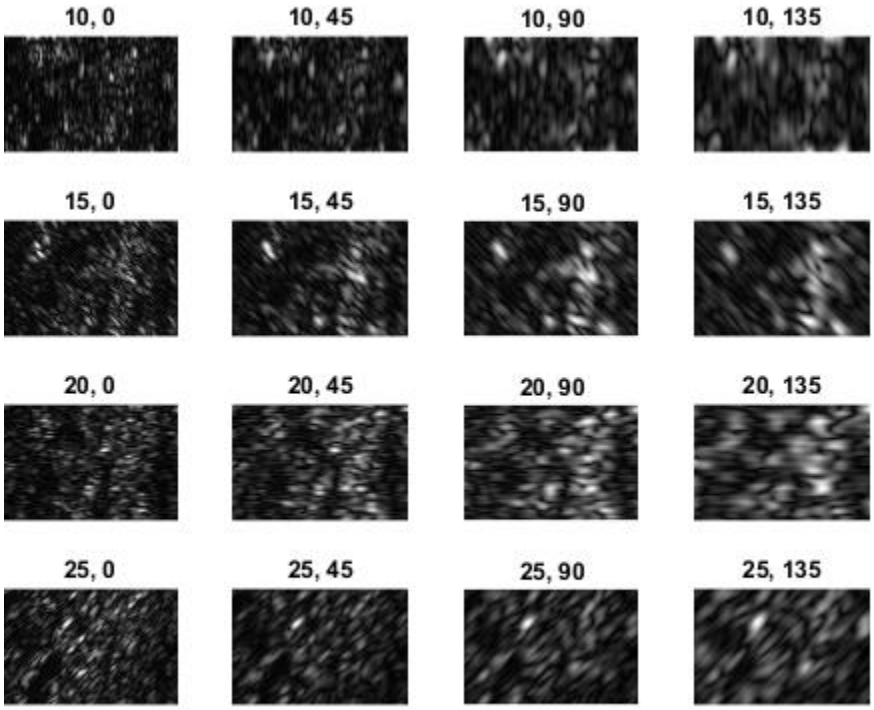

Figure 21: Gabor filter response for image 07

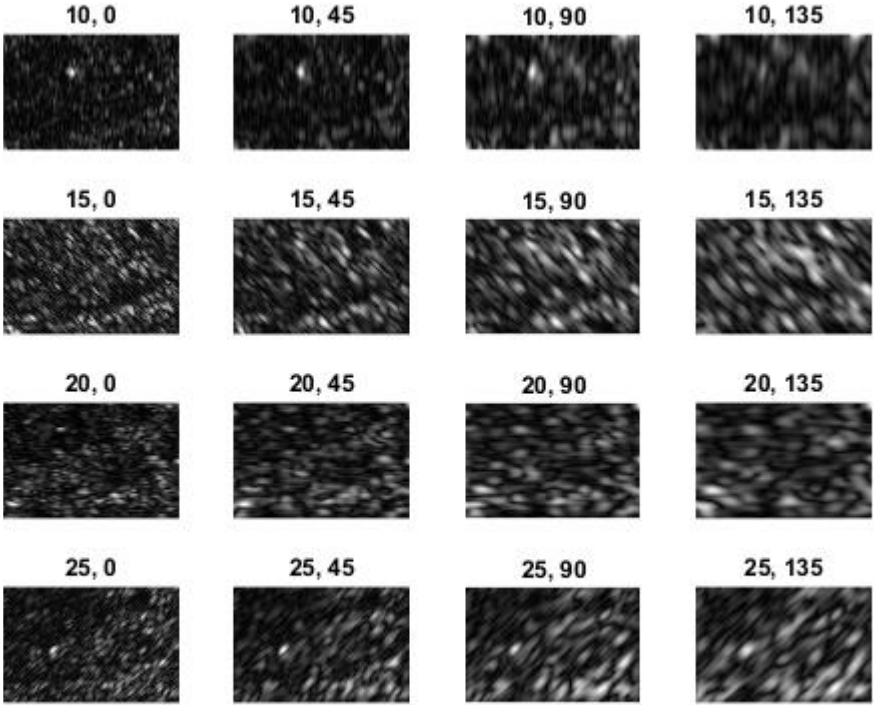

Figure 22: Gabor filter response for image 08

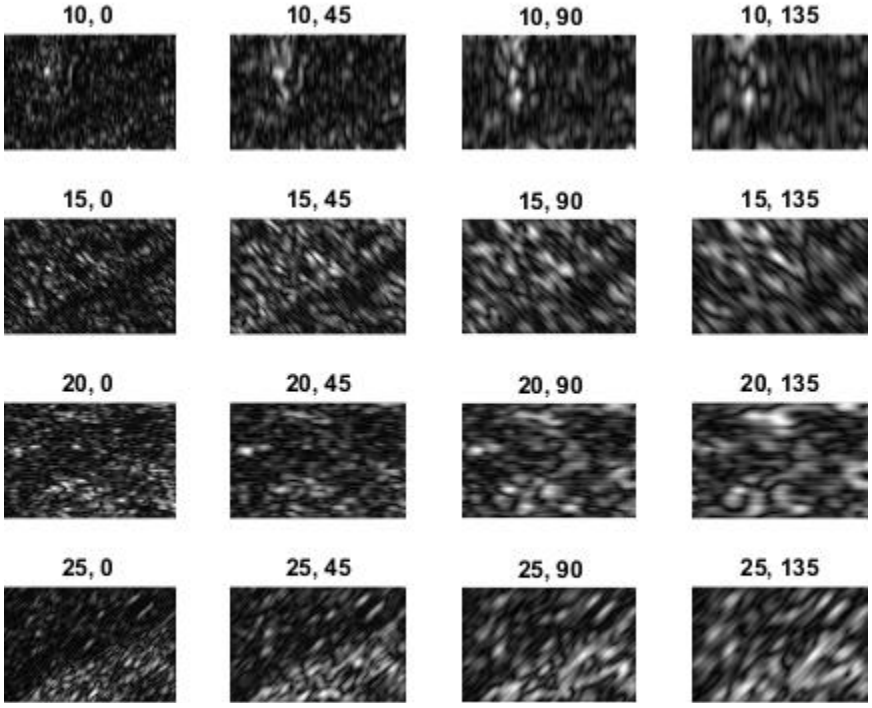

Figure 23: Gabor filter response for image 09

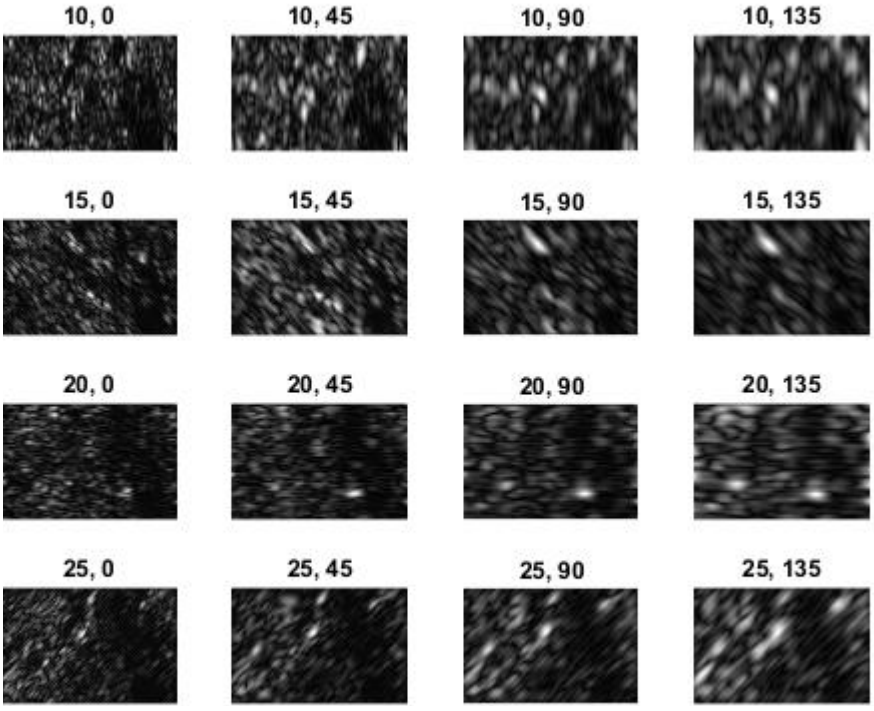

Figure 24: Gabor filter response for image 10

Clustering / segmentation results for all images

In this section, we display k-means clustering results with false color overlays over original images. Each pair of image contains 2 images: left one is the false coloring output using gabor and LAB representations alone (part 4), right one is the false coloring output using first and second neighbour representations concatenated to the representations in part 4 (which is the representations we acquire at the end of part 5).

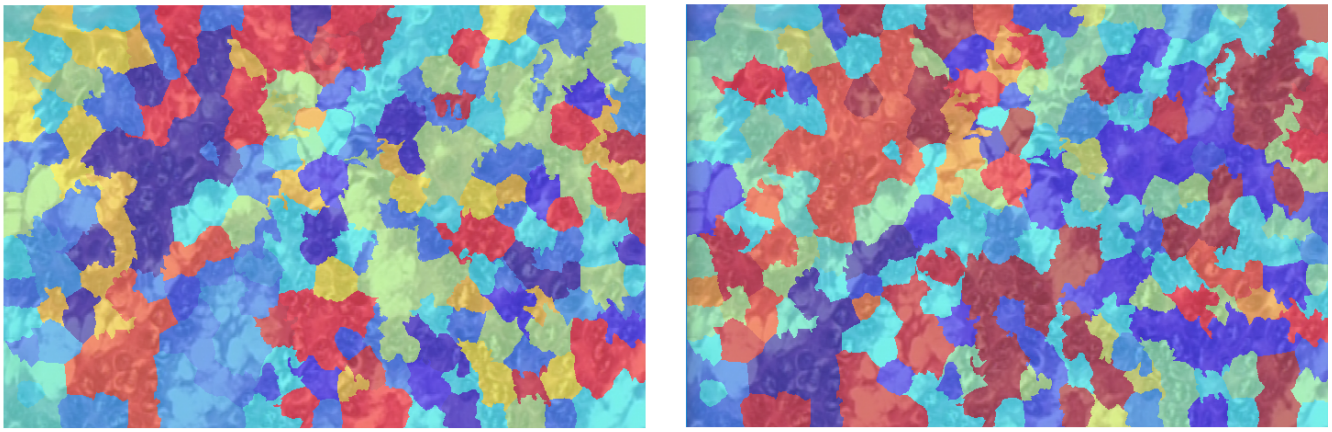

Figure 25: False coloring overlays for image 01

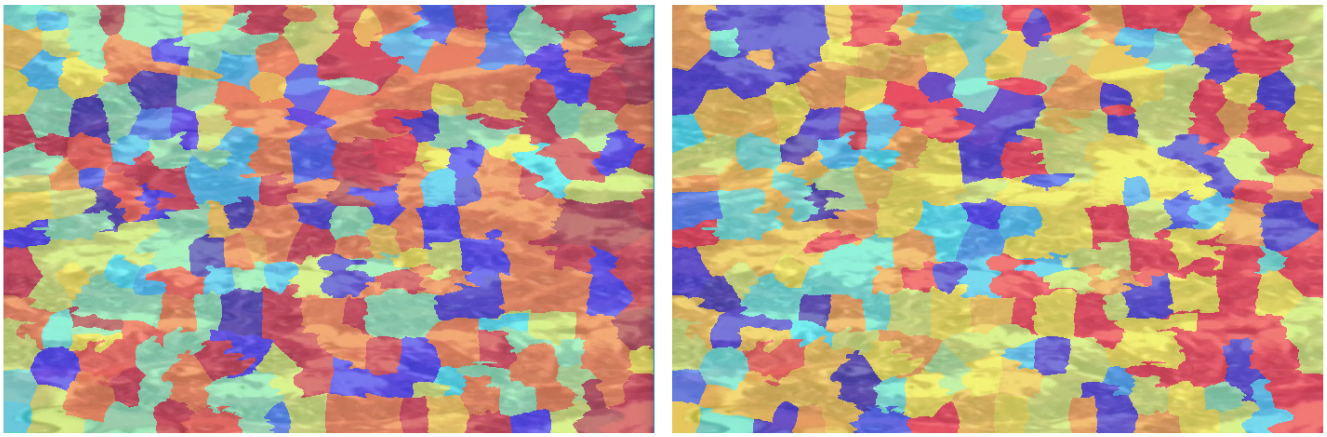

Figure 26: False coloring overlays for image 02

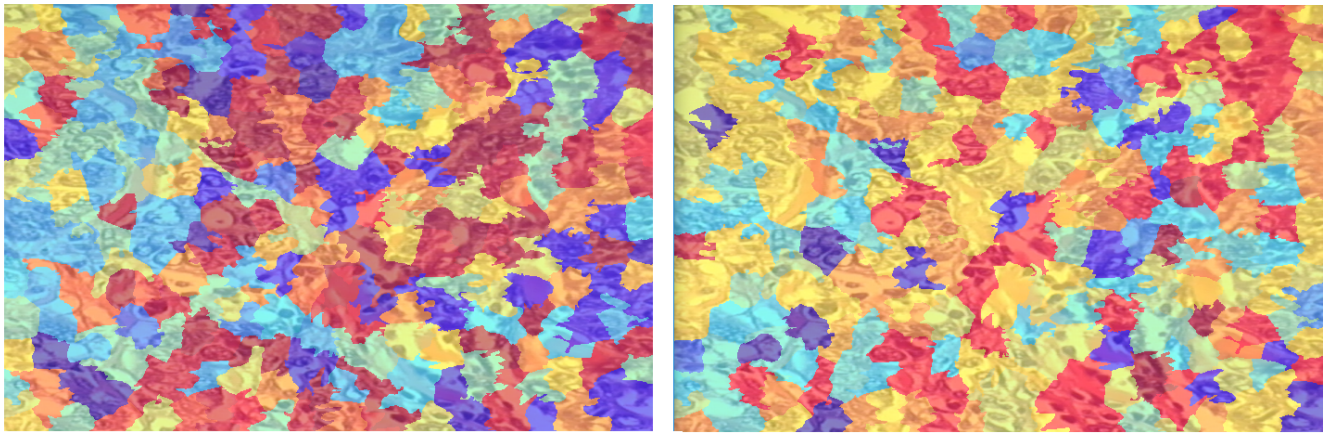

Figure 27: False coloring overlays for image 03

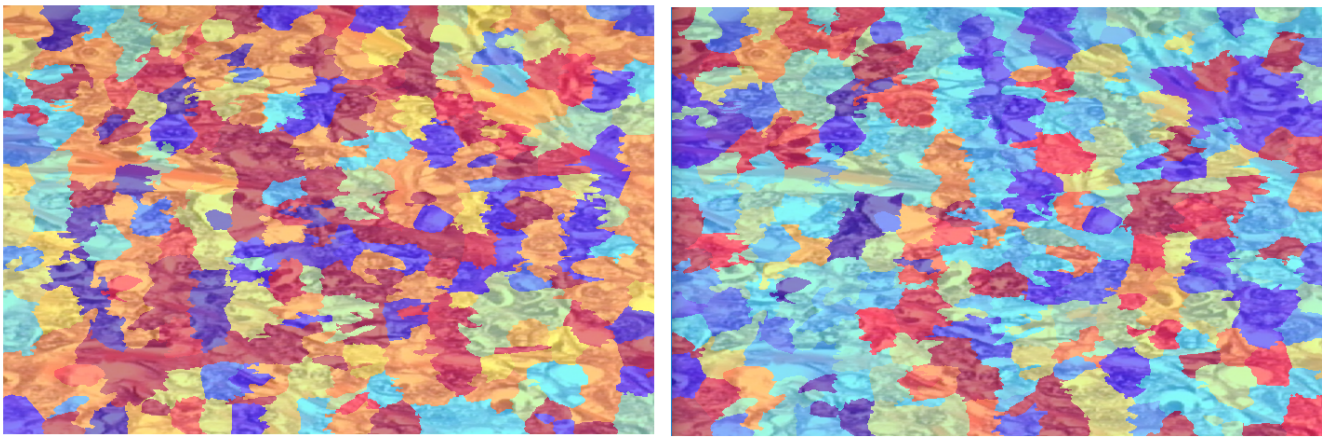

Figure 28: False coloring overlays for image 04

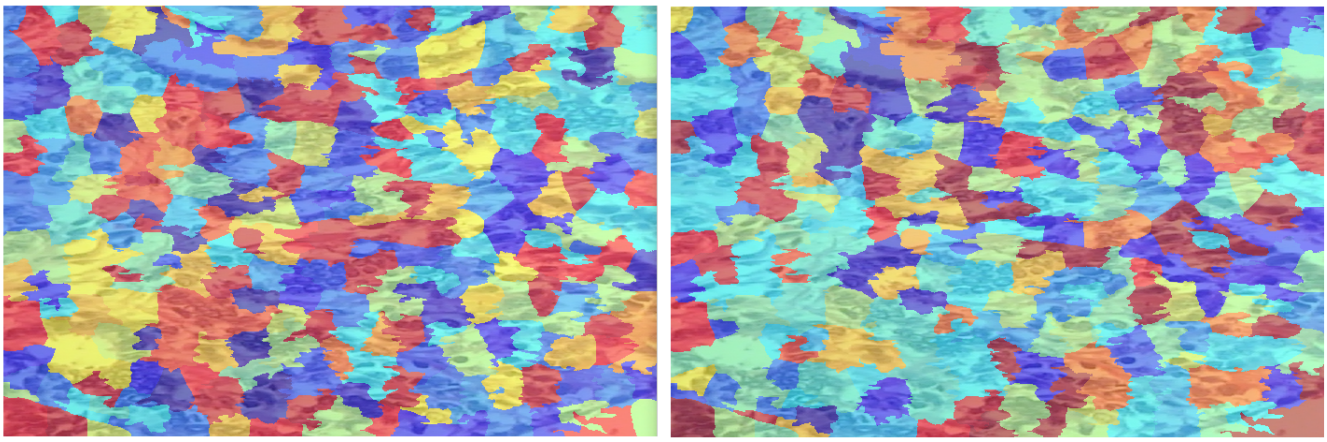

Figure 29: False coloring overlays for image 05

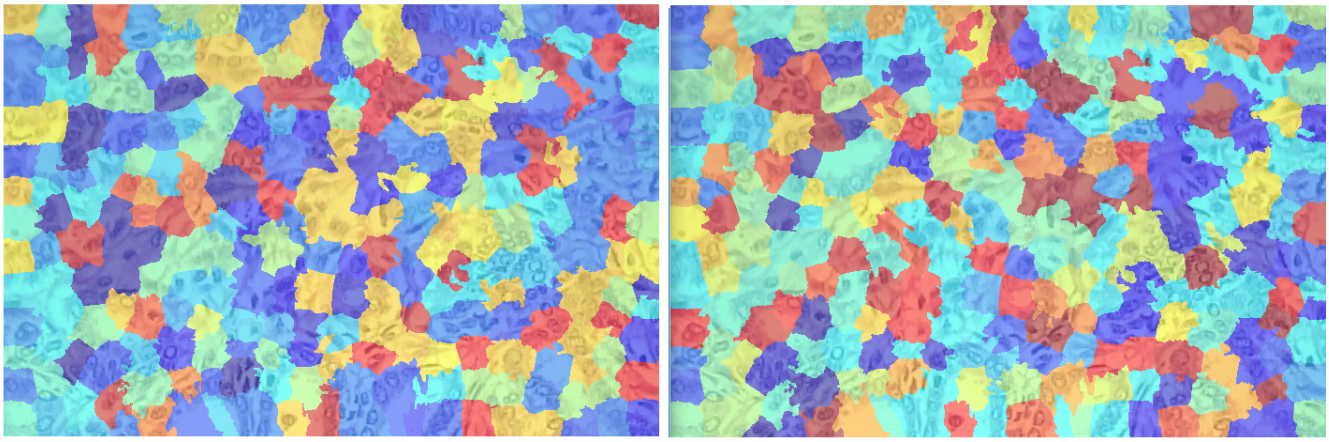

Figure 30: False coloring overlays for image 06

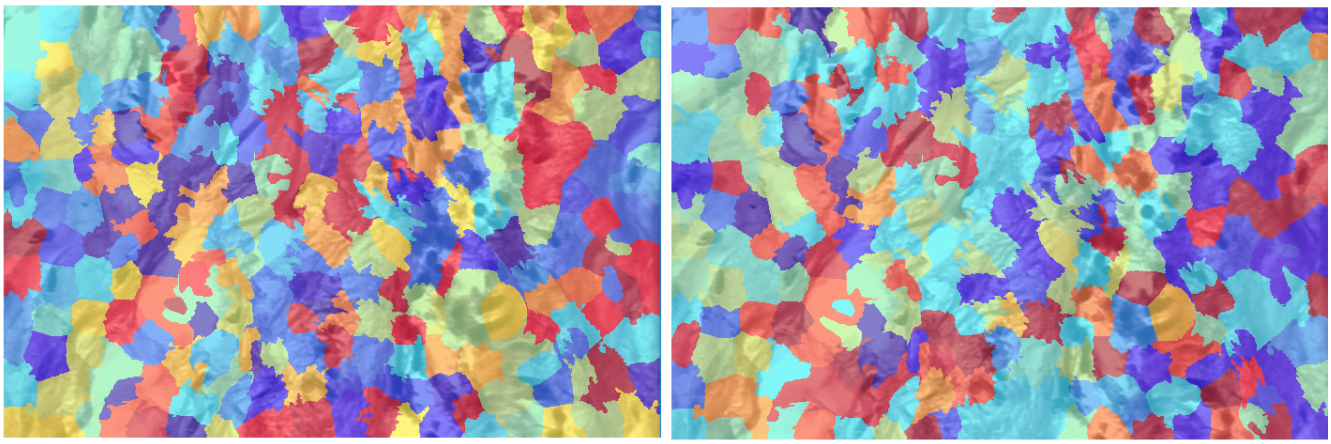

Figure 31: False coloring overlays for image 07

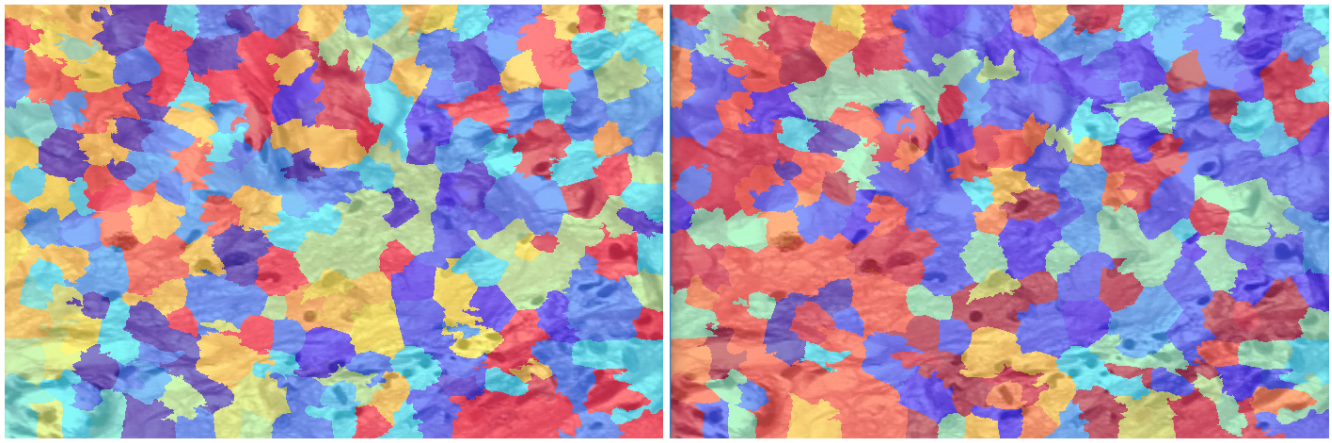

Figure 32: False coloring overlays for image 08

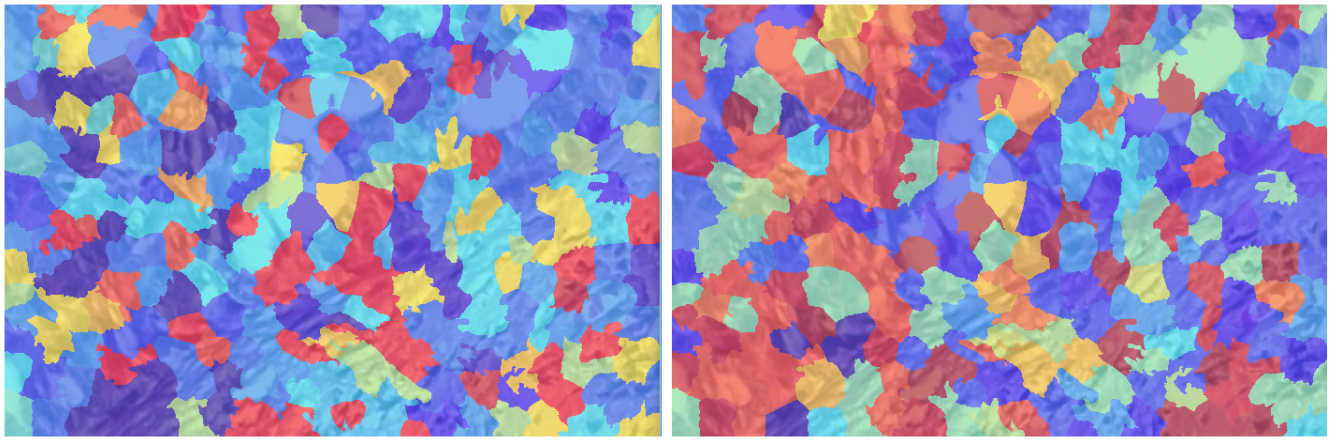

Figure 33: False coloring overlays for image 09

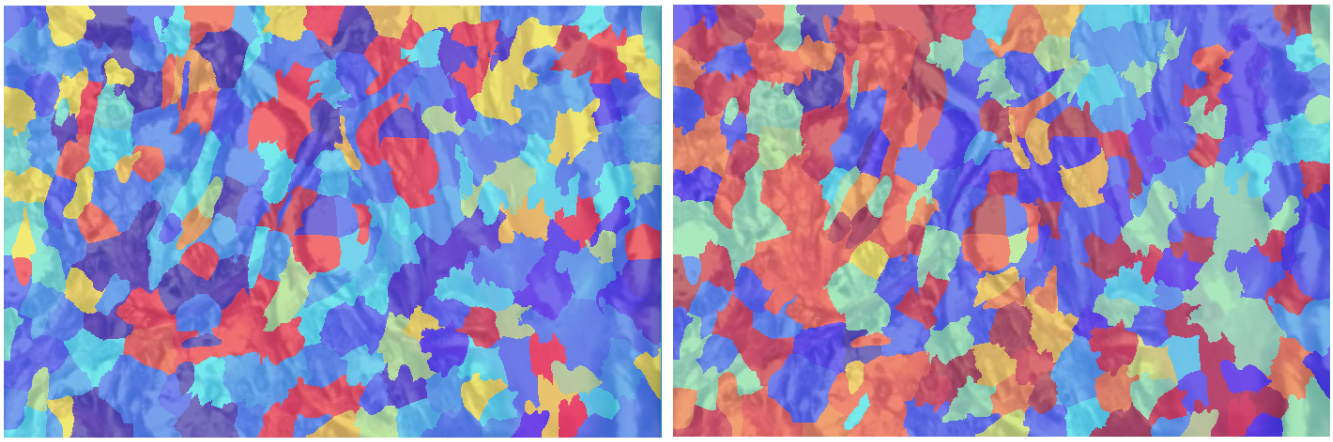

Figure 34: False coloring overlays for image 10

Discussion

First issue was to determine how MATLAB implemented LAB representation. Articles mentioned in the assignment suggests that for L values interval is [0, 100], for A and B values interval is [-128, 127]. However, a quick research on the internet suggests 4 alternatives for A & B intervals exist: [-128, 127], [-110, 110], [-100, 100], and another that is adaptive to pixel values. Also, it is not clear which interval MATLAB implementation uses and a random assumption for a fixed interval resulted in NaN values in the normalized LAB representations; because when all values for a bucket is 0 across all images (column values), standard deviation for that column becomes zero and normalization results in a division by zero. Cause of this issue is that we normalize each bucket value across all superpixels.

To prevent this, we decided to find the maximum and minimum L, A, and B values for all input images. Then, we defined each bucket *i* to hold quantities in the following interval:

```
a_min + a_bucket_size * i <= a_k < a_min + a_bucket_size * (i + 1)
b_min + b_bucket_size * i <= b_k < b_min + b_bucket_size * (i + 1)
l_min + l_bucket_size * i <= l_k < l_min + l_bucket_size * (i + 1)
```

Here, quantaties `a_k`, `b_k`, and `l_k` correspond to LAB values for a given pixel. Note that bucket sizes are constant for all dataset and computed as follows:

```
a_bucket_size = (a_max - a_min) / 8;
b_bucket_size = (b_max - b_min) / 8;
l_bucket_size = (l_max - l_min) / 8;
```

Compared to previous approach where we define bucket intervals to be predefined, we minimize the accumulation of pixels in a small subset of buckets and distribute the placement evenly, which resulted in a better representation for superpixels.

Another issue was to determine first and secondary neighbours of of each superpixel. One problem we faced was to determine a threshold for a superpixel that is marked as a neighbour if a percentage of its pixels are in neighbourhood distance to the given superpixel. We can visualize this as follows:

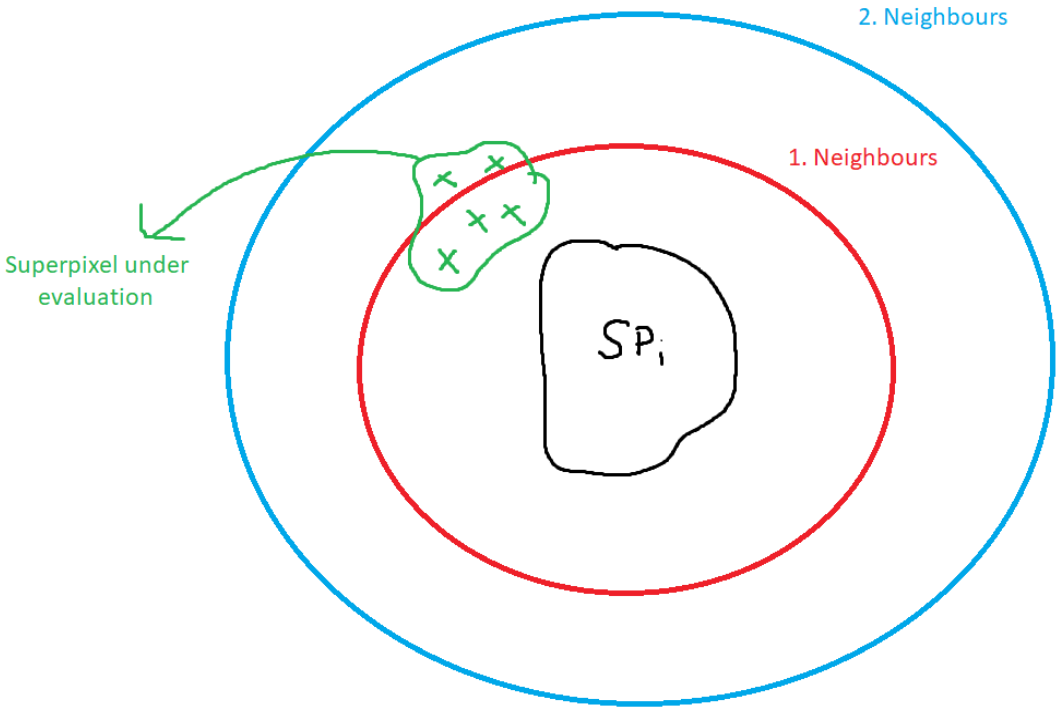

Figure 35: First and secondary neighbourhood circles of a superpixel

To determine a good threshold, we experimented with 3 values: 75 %, 50 %, and 25%. The results are as follows:

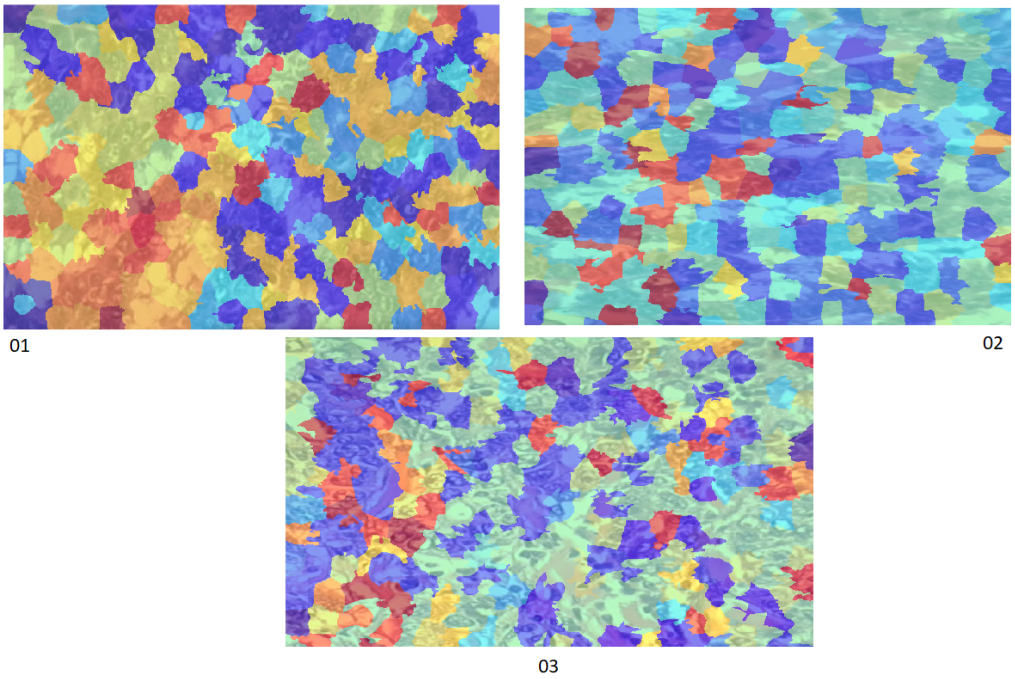

Figure 36: False Coloring Using Neighbourhood Representations, Threshold = 75 %; Top Left: image 01, Top Right: image 02, Bottom: image 03

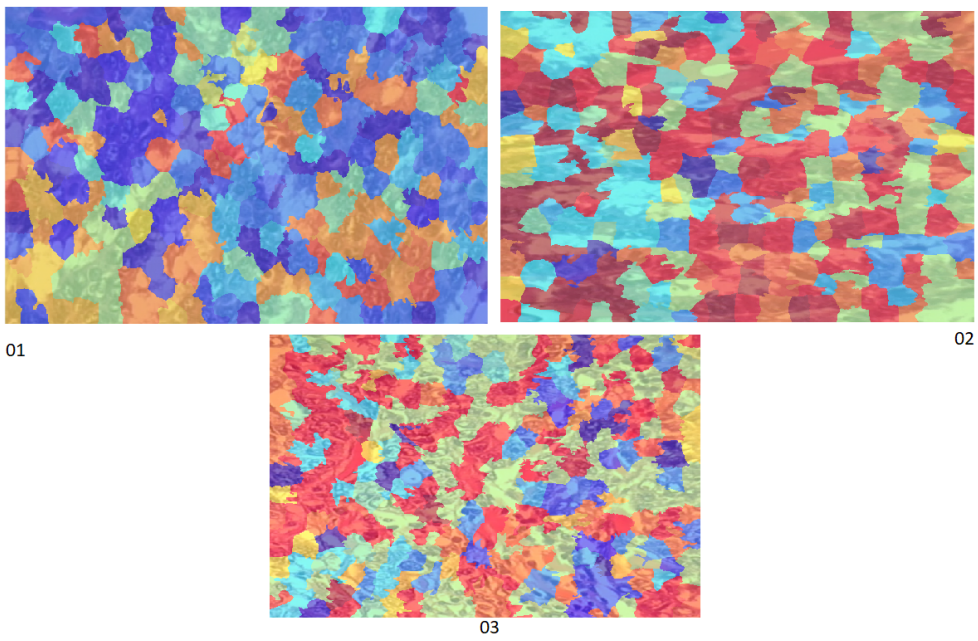

Figure 37: False Coloring Using Neighbourhood Representations, Threshold = 50 %; Top Left: image 01, Top Right: image 02, Bottom: image 03

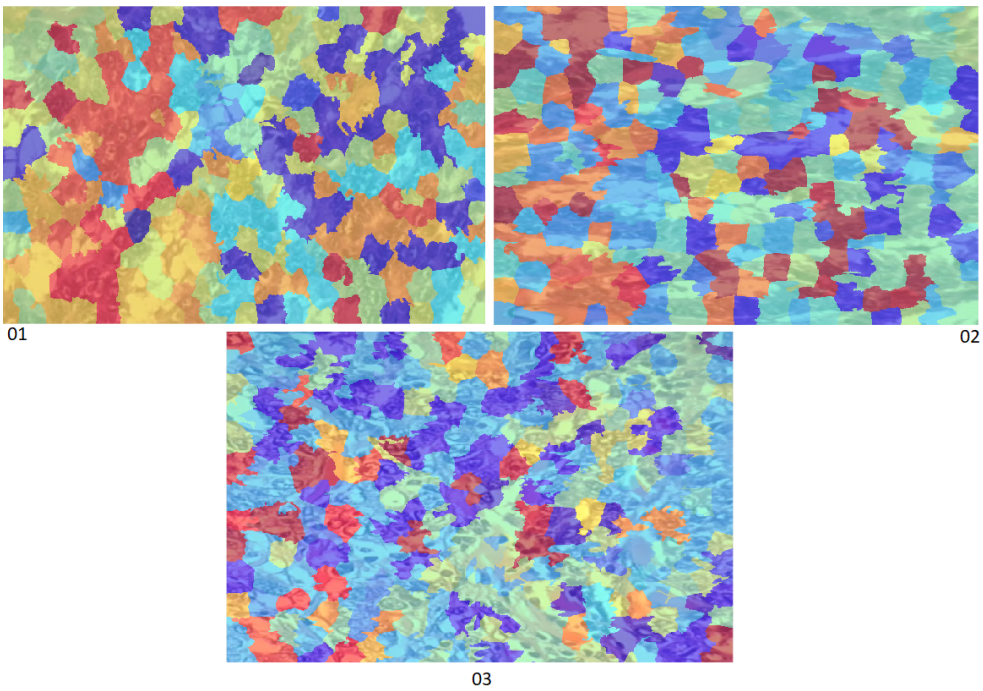

Figure 38: False Coloring Using Neighbourhood Representations, Threshold = 25 %; Top Left: image 01, Top Right: image 02, Bottom: image 03

From these results, we see that low (25%) and high (75%) threshold values show inconsistent behavior: in some cases segmentation performance improves while in others, it declines. For example, regions in image 02 are move visible when threshold is 50% compared to when 75%. Similarly, regions in image 03 are more connected when threshold is 25% compared to 50%. For these reasons, we decided to use a threshold in between, 50 %. By doing so, we managed to capture local information around a superpixel while acquiring a significant number of neighboring superpixels for each superpixel. Final outputs for part 5 shown previously are calculated using this threshold value.

Another problem related to neighborhood was to determine expected size of each superpixel. Since MATLAB does not provide this information, we decided to calculate it by hand. First option was to use the center of mass (CoM) of each superpixel and accept the length of longest line passing through CoM and the corners of bounding rectangle of the superpixel as the radius. This option was easy to compute, however, when the superpixel was not convex, CoM was outside the superpixel and radius computed resulted in faulty neighborhood representations. Next, we decided to take the mid point of the bounding rectangle as the center of the superpixel and one half of Euclidean norm of sides of the bounding rectangle as the radius. Since this method ensures that center point will be inside the superpixel, the results were more accurate, and we decided to use this approach.

Second issue was to choose Gabor filter wavelengths. Since wavelengths determine how large the texture needs to be to get a strong filter response, we decided to experiment with 3 different sets of gabor filters: [2 4 6 8], [4 8 12 16], [10 15 20 25]. The outputs of [10 15 20 25] are already shown previously (false colors). We now give example outputs using these filters, using false coloring for part 4 (left) and part 5 (right).

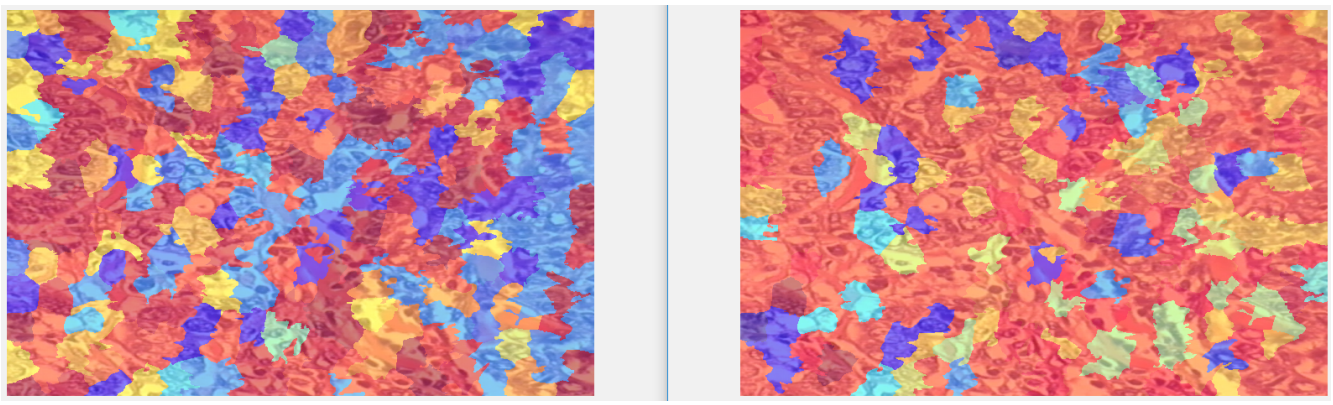

Figure 39: Gabor wavelengths = [2 4 6 8], image 03

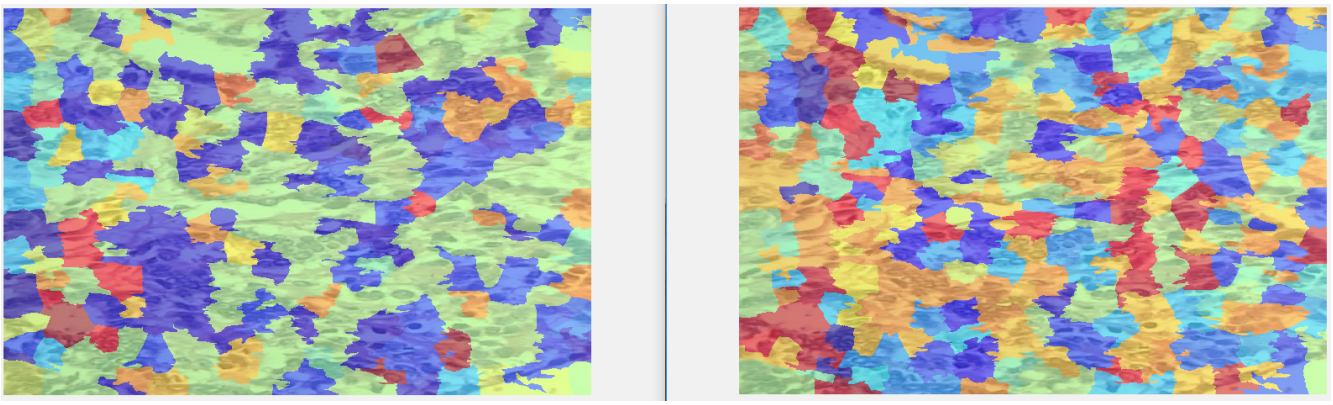

Figure 40: Gabor wavelengths = [2 4 6 8], image 05

We observe that this wavelength vector show inconsistent behavior: in one case, the segmentation spawns almost all image whereas in the other we see a decrease in performance after neighboring superpixel representations are used. Also, we'd like to think that identifying individual cells &| cell groups are important for this task. Therefore, we decided not to use this wavelength for gabor filters. Note that since wavelengths are small, filter response is strong in entire image. Thus, gabor response does not provide useful information here.

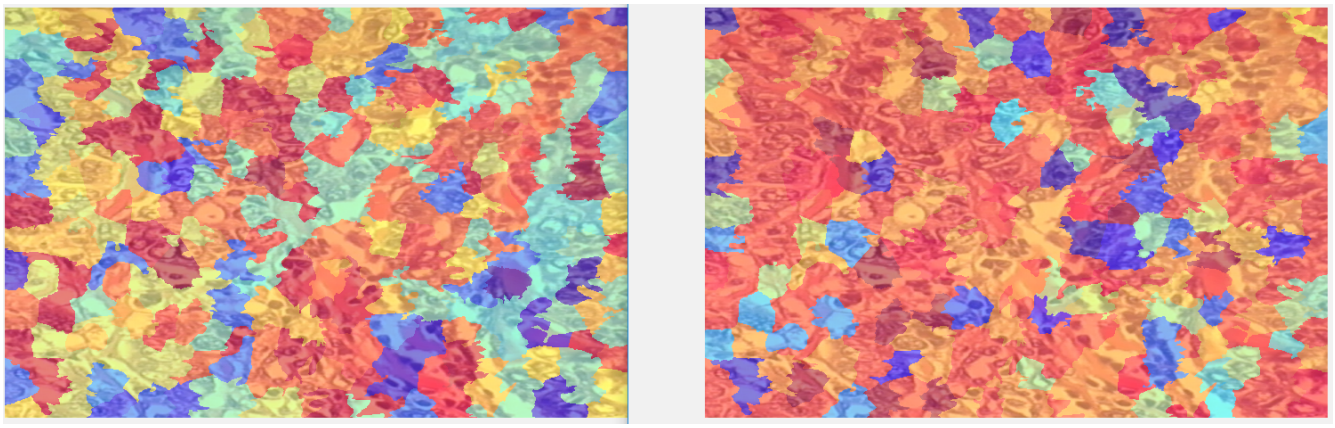

Figure 41: Gabor wavelengths = [4 8 12 16], image 03

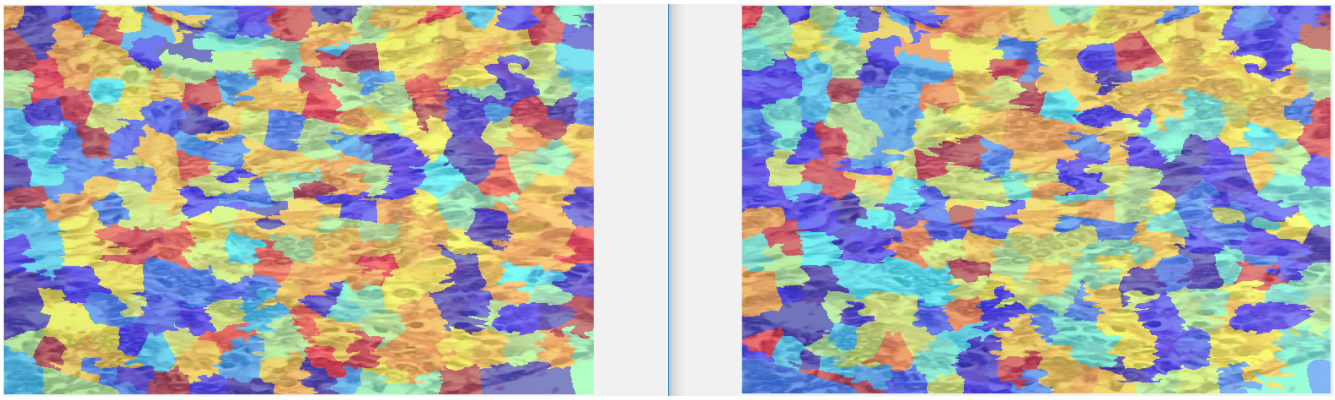

Figure 42: Gabor wavelengths = [4 8 12 16], image 05

Unlike the previous case, here we see some detailed grouping of the cells. Comparing image 03 shows that the information we obtained from gabor response is more informative than the previous case. However, if we look at [10 15 20 25] output, we clearly see that segmentation performance is better than all three. We are able to detect details that are ignored by the previous cases. For example, gabor responses belonging to bottom right corner of image 03 vary across the three cases, and [10 15 20 25] gives the most informative segmentation. This result can also be confirmed by human observation.

Experiments show that [10 15 20 25] is the better choice among the three we proposed. Thus, we decided to use it in the previous parts.

Last problem was to select cluster count for k-means algorithm. We observed that using large values (ex. 100) results in a noisy representation: similar superpixels observed in the input are assigned to different clusters. We suspect that, since cluster count is large, small differences between similar superpixels become significant when assigning a cluster label. Moreover, using a small cluster count (ex. 10) result in poor differentiation among superpixels, that region with different color histograms are assigned the same label. Experiments show that for N = 250, 20-30 clusters give satisfactory results. These experiments can be easily repeated by changing the input to the MATLAB script. Finally, we decided to use 20 as cluster count. Note that for a different superpixel count, cluster count to be used might vary. We now show effect of different cluster counts on the output image. Note that we assumed

that local differences are significant for the purpose of this assignment, which implies that cluster count should be high.

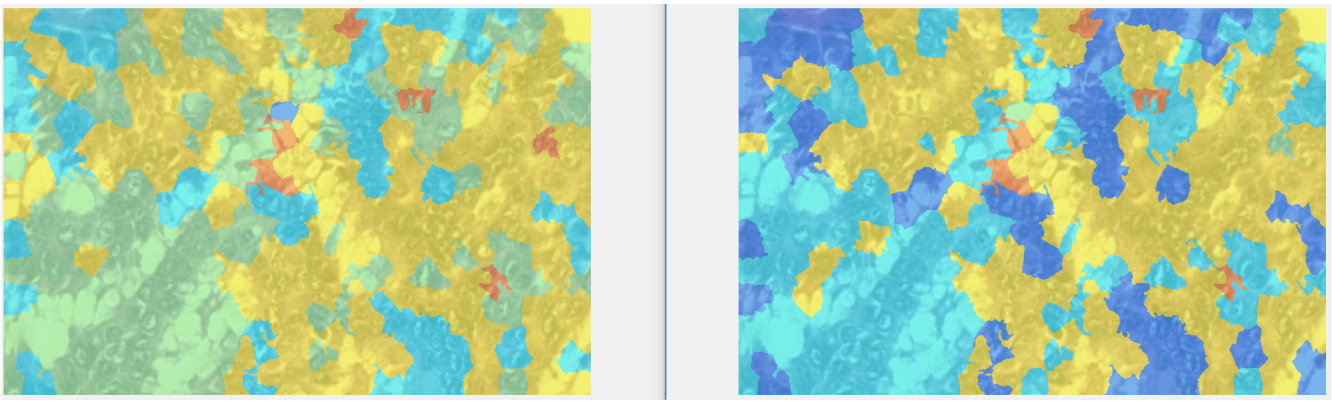

Figure 43: False color overlay of k-means output on image 01, k = 5

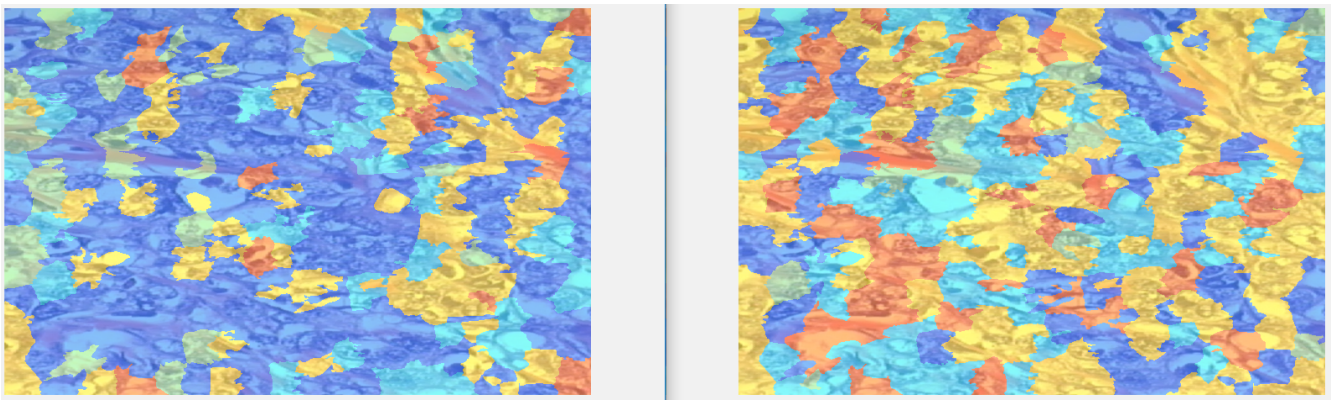

Figure 44: False color overlay of k-means output on image 04, k = 5

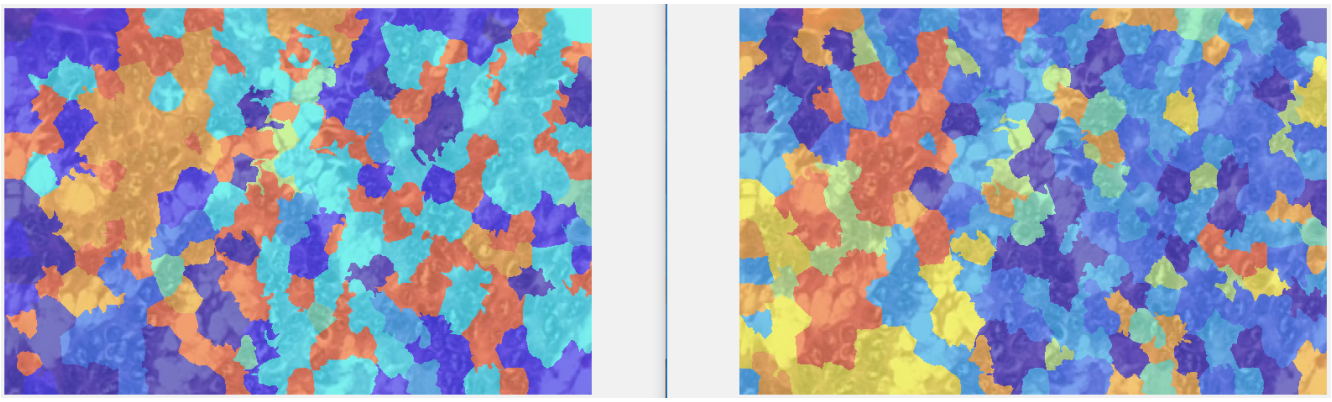

Figure 45: False color overlay of k-means output on image 01, k = 10

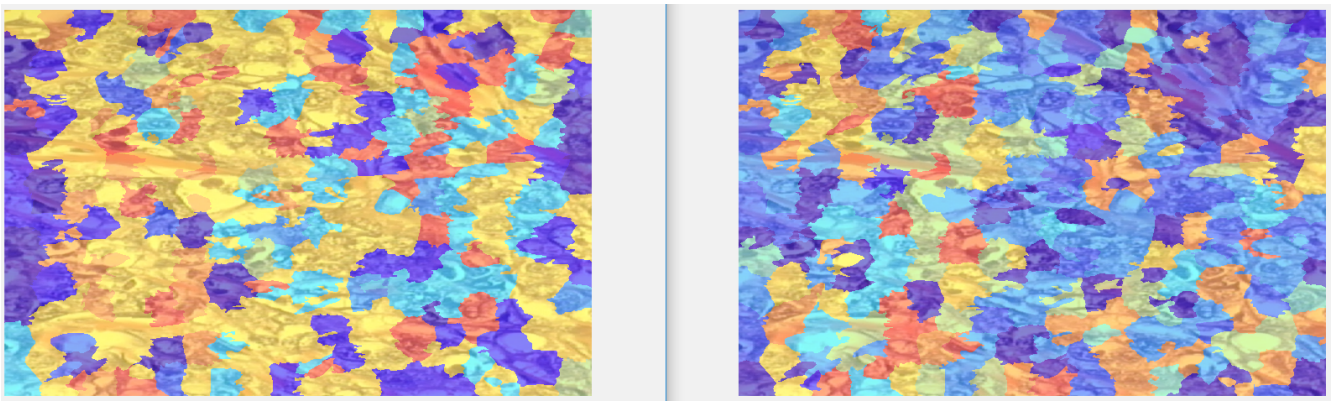

Figure 46: False color overlay of k-means output on image 04, k = 10