Tuğberk Dikmen

21802480

CS-484

Homework-3

Bilkent University

24.05.2023

Part 1)

1. **Image Size:** The scale and quantity of superpixels produced depend on the size of the input image. Smaller images might need fewer superpixels to prevent over-segmentation, while larger images might need more superpixels to capture fine details.

2. **Superpixel Size:** The average size of the superpixels is determined by this option. It regulates the segmentation's level of granularity. Larger superpixel sizes produce rougher segmentation while smaller superpixel sizes produce more fine segmentation.

3. **Compactness:** The superpixels' conformity to the borders of the image is controlled by compactness. Superpixels are compelled to align themselves more closely with edges when the compactness value is larger, creating compact and regular-shaped regions. Superpixels can spread across more regions and form more erratic shapes when compactness values are lower.

4. **Spatial Proximity:** To ensure connectivity and spatial coherence, superpixels might be affected by their spatial proximity. The neighborhood size or connectivity limits utilized during superpixel synthesis are determined by spatial proximity criteria. They can designate whether superpixels can have up to 8 connections or just 4 connections.

5. **Color Space:** The effectiveness and look of superpixel segmentation can be affected by the color space selection. The color spaces RGB, Lab, HSV, and Luv are frequently used. Each color space has unique qualities and can be better suited for particular image kinds or applications.

6. **Iterations:** Iterative optimization is frequently used in superpixel segmentation methods. The number of iterations controls how many times the superpixel boundaries have been adjusted by the algorithm to produce the desired segmentation.

Part 2)

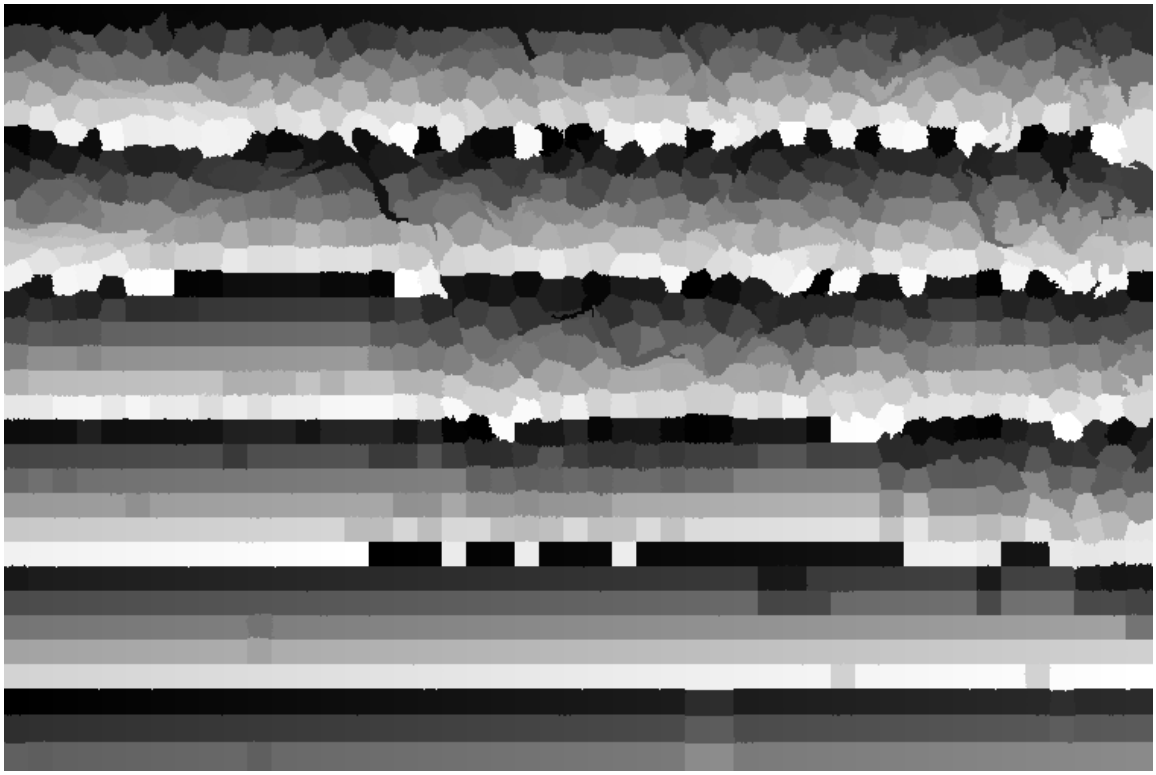


Image 1

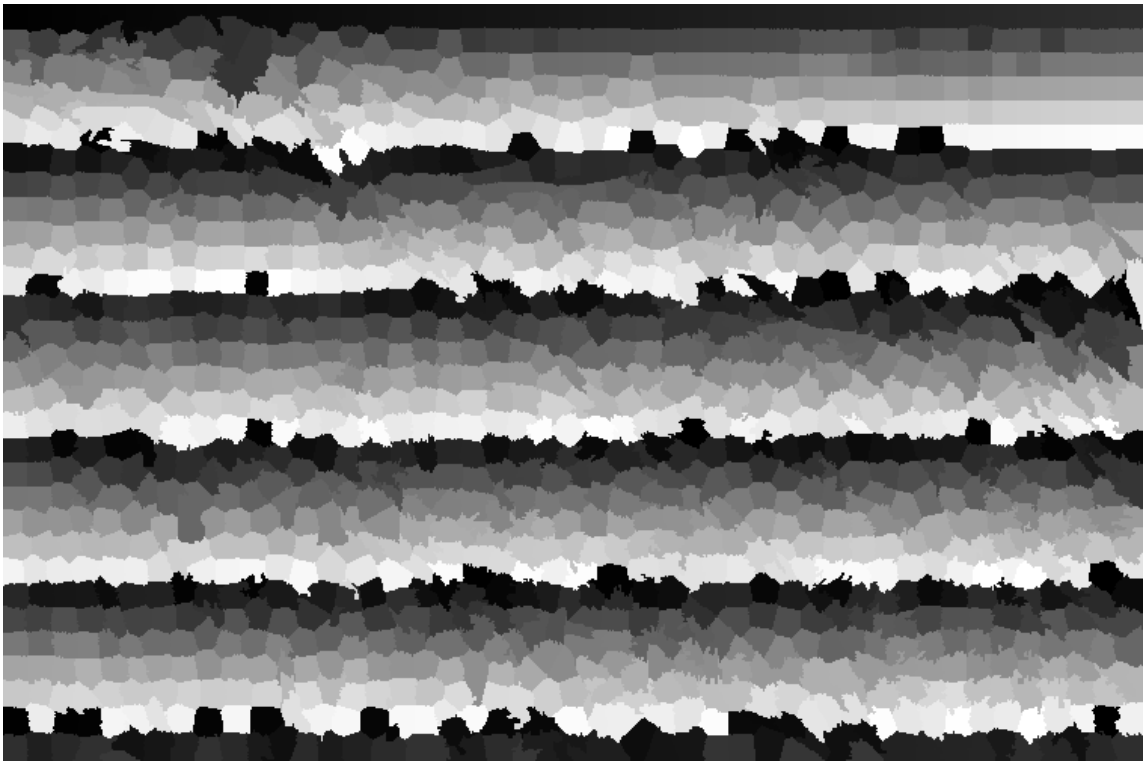


Image 2

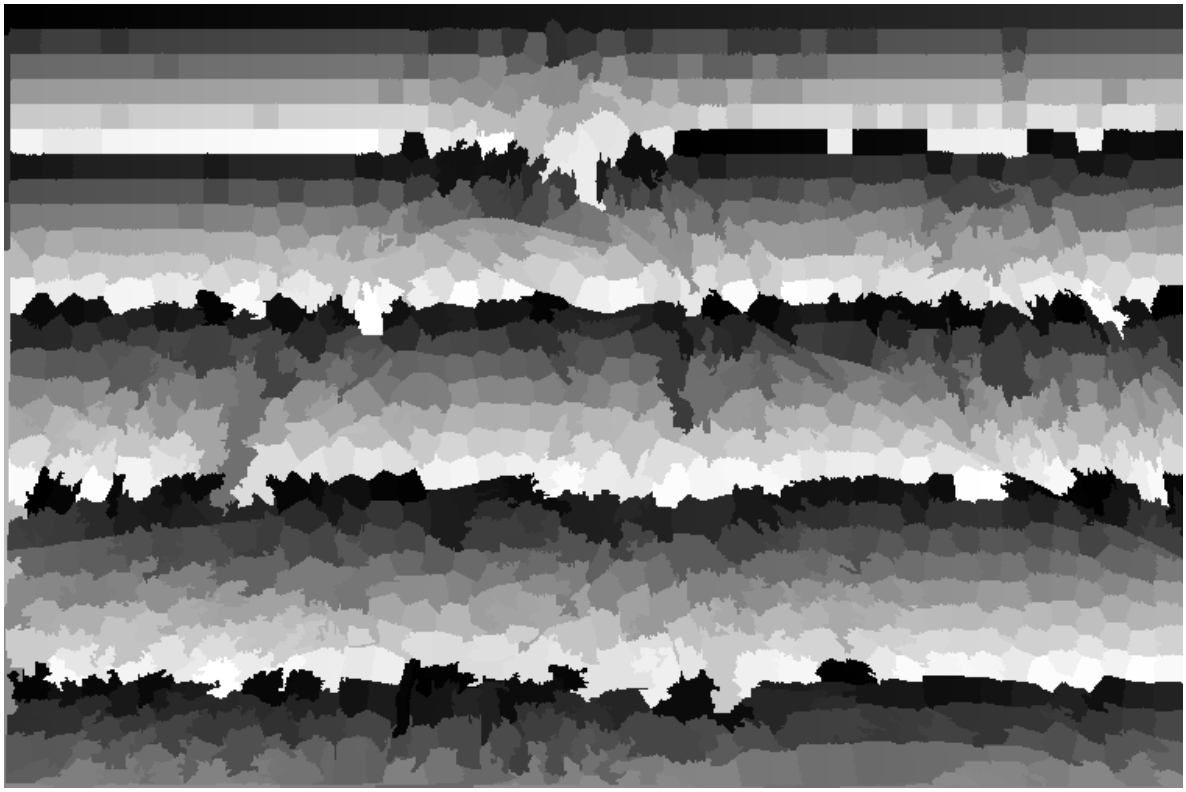


Image 3

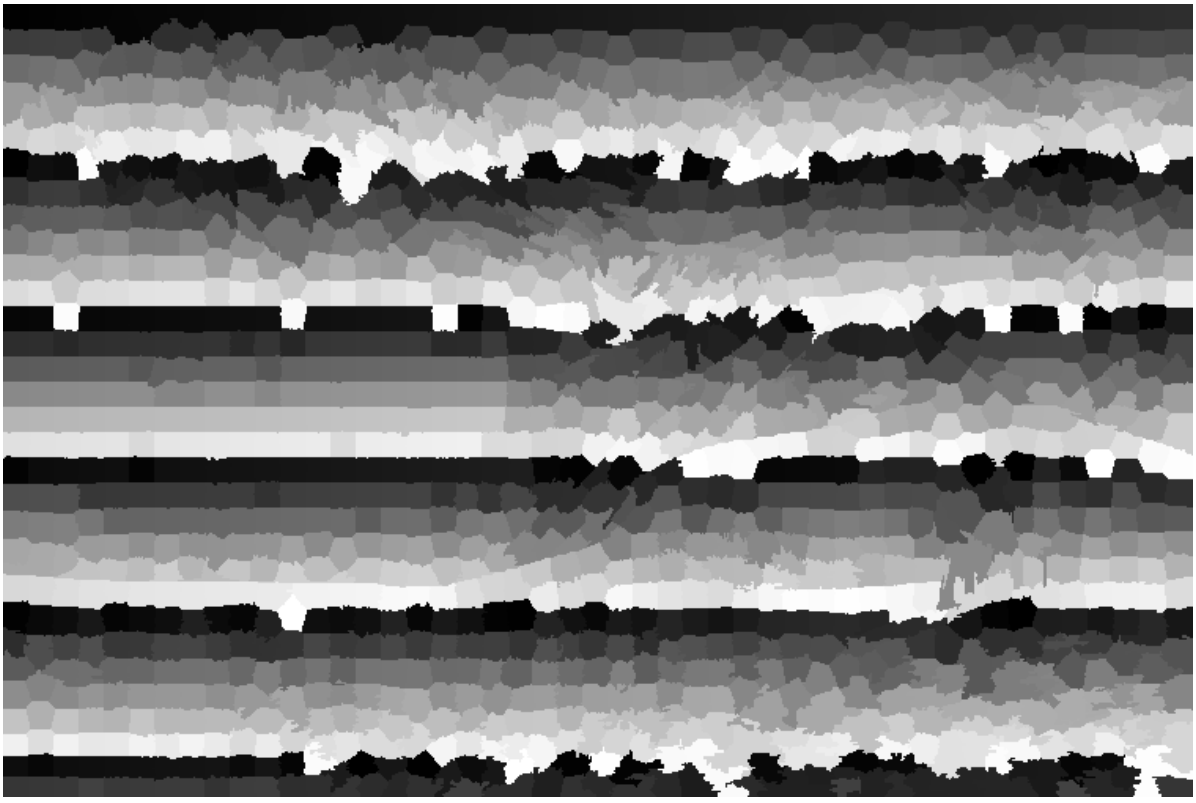


Image 4

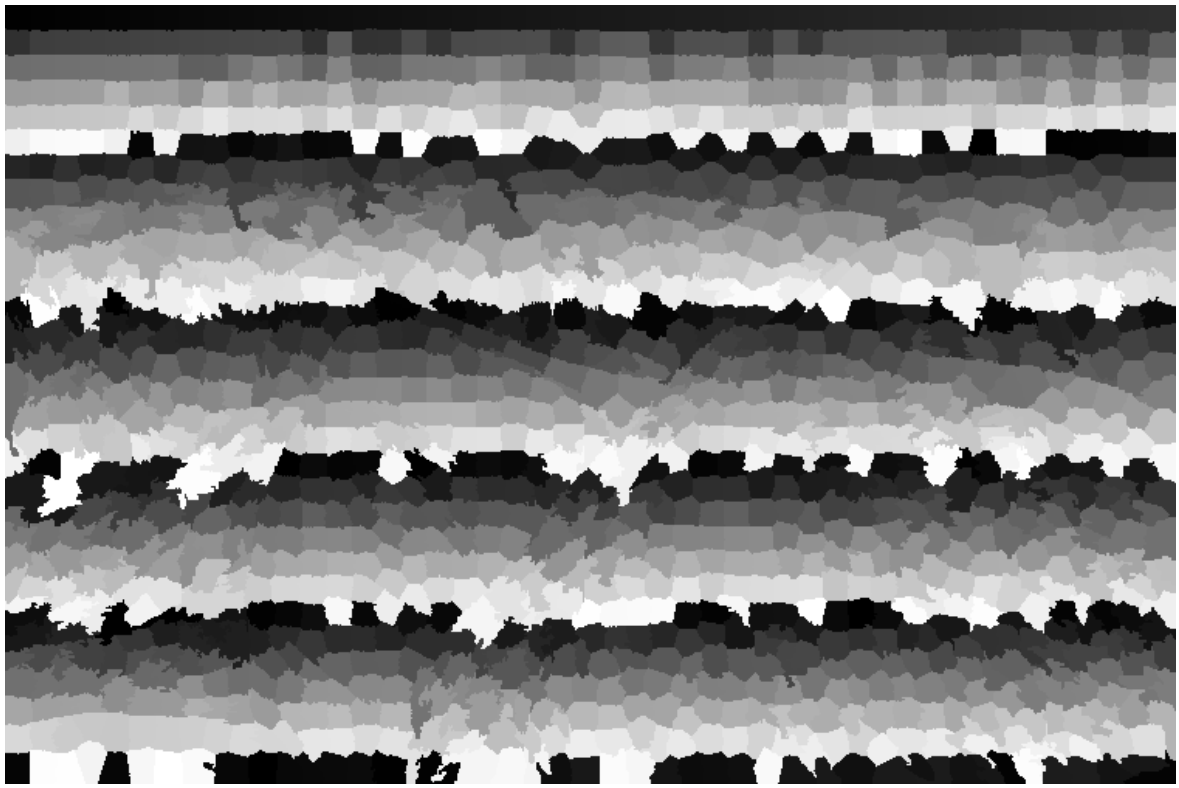


Image 5

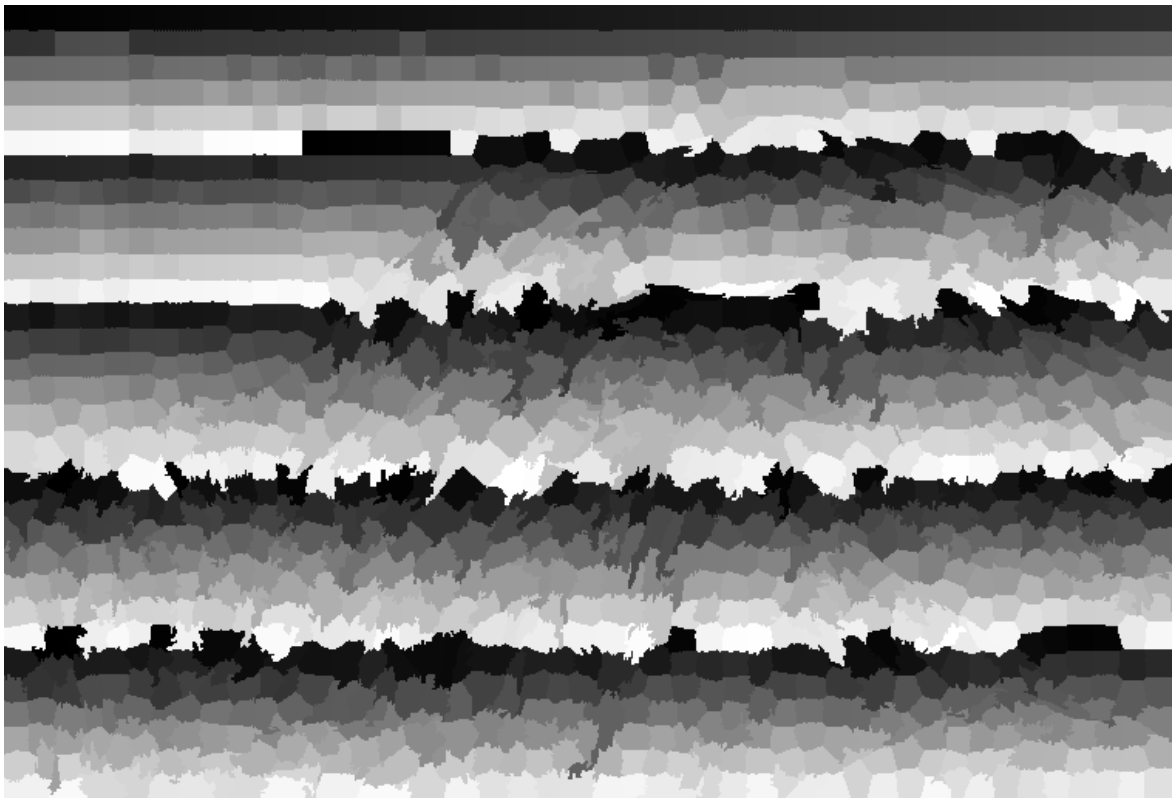


Image 6

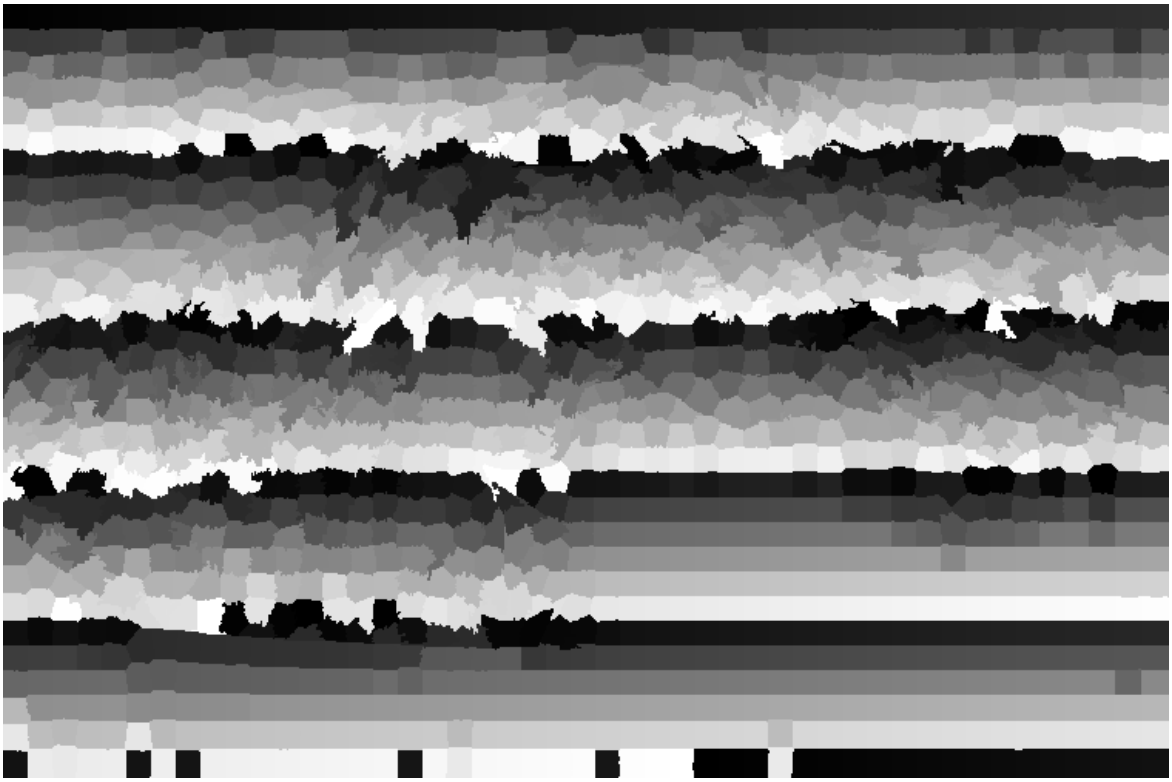


Image 7

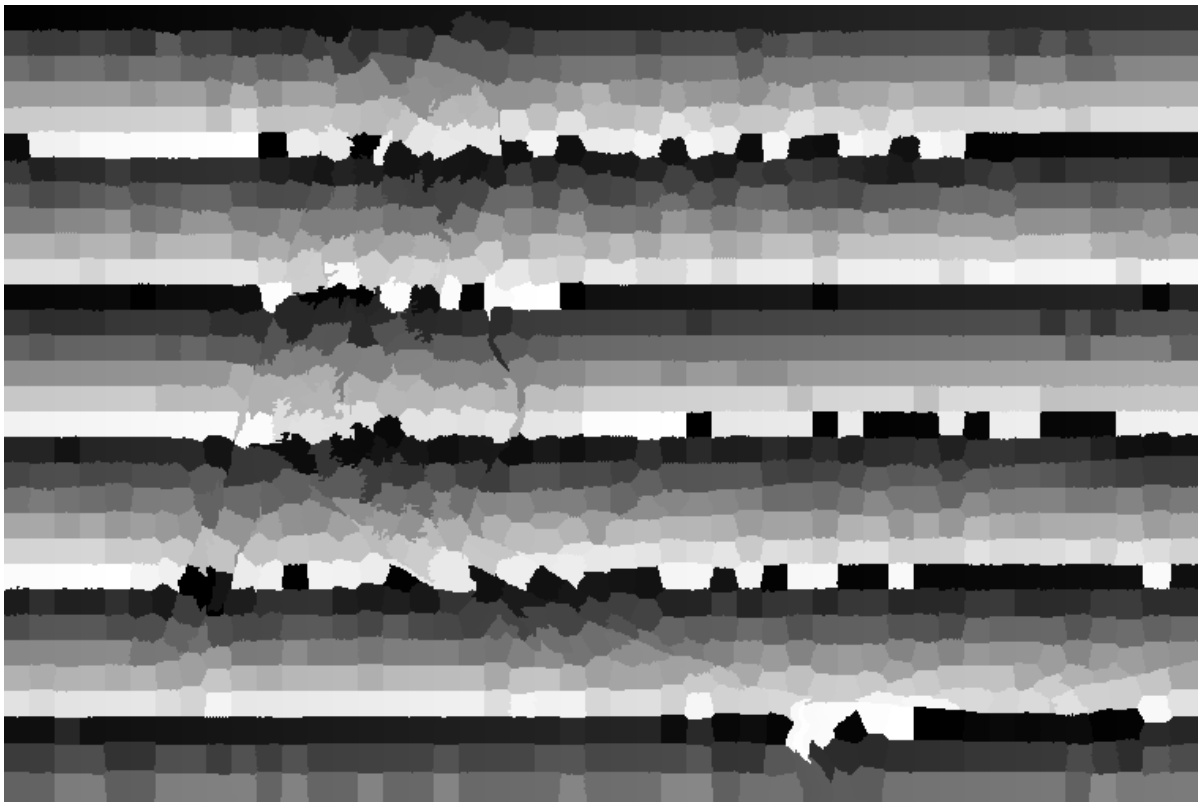


Image 8

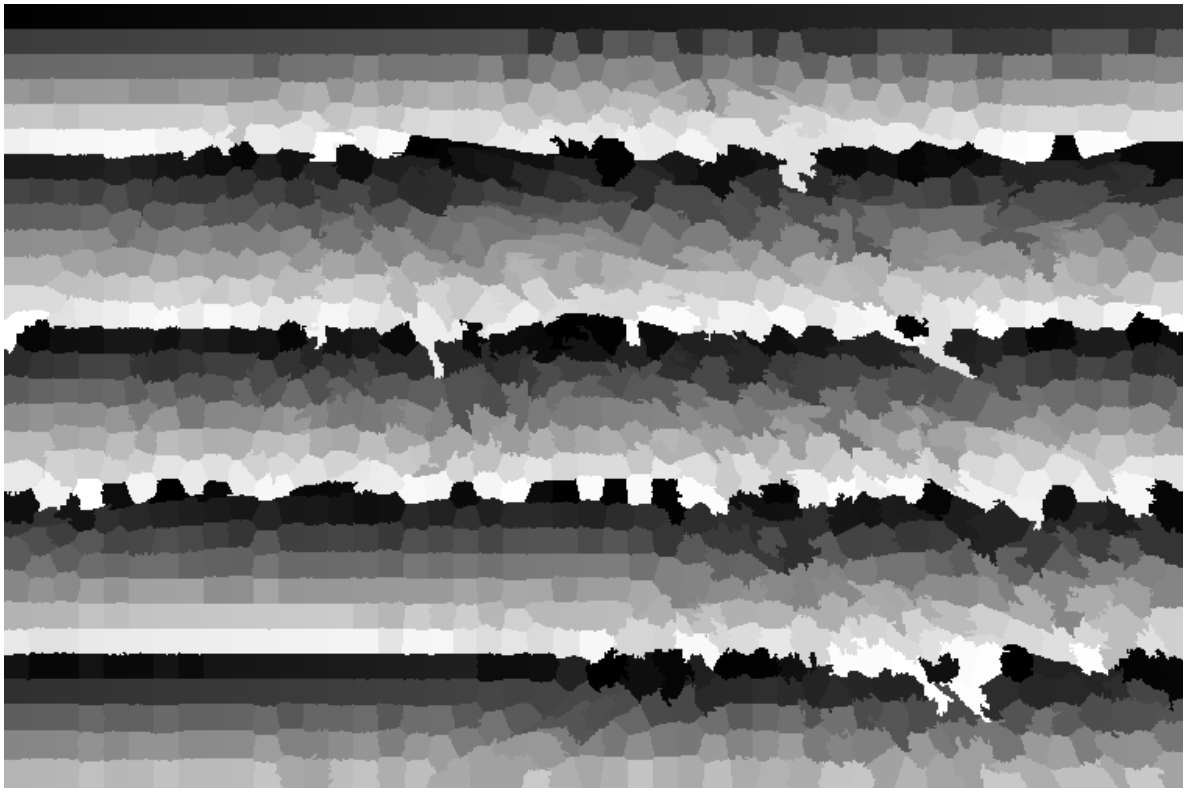


Image 9

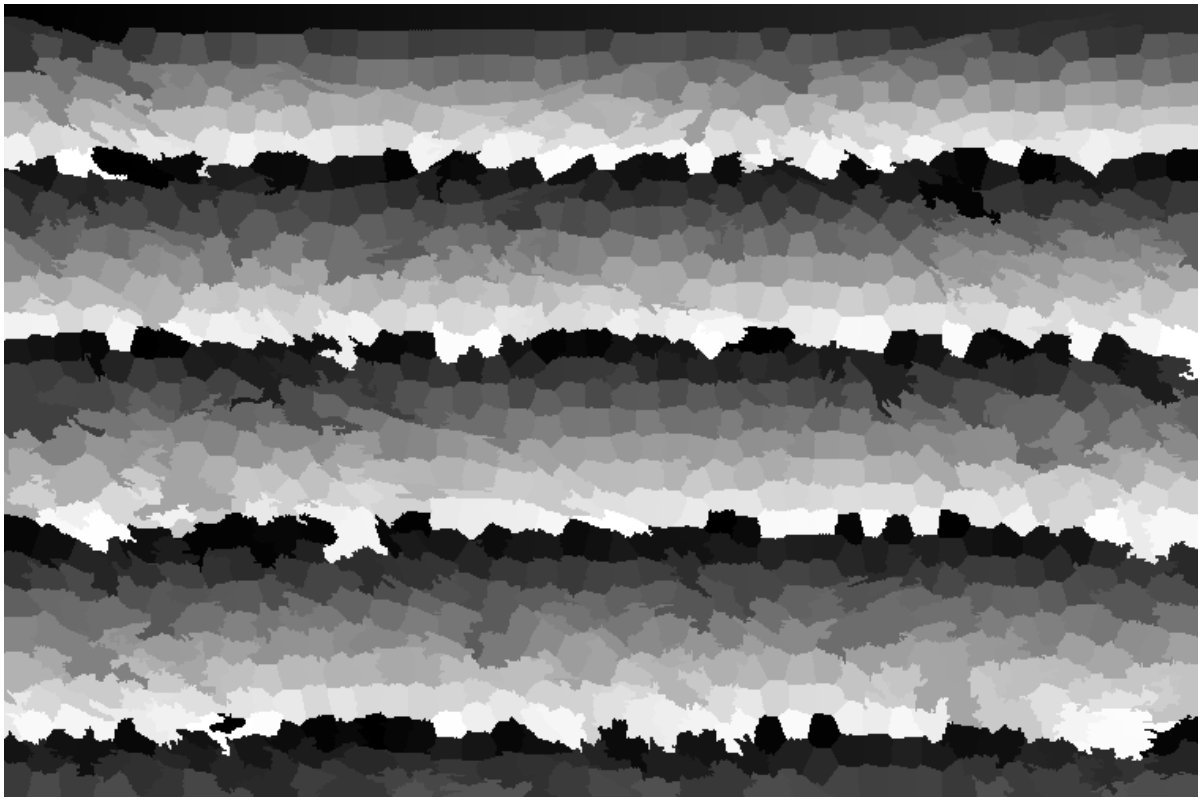


Image 10

Part 3)

1. **Frequency:** The quantity of oscillations within a specific spatial extent is determined by the Gabor filter's frequency. Lower frequencies capture broader patterns and larger texture elements, whereas higher frequencies capture finer details and smaller texture patterns.

2. **Orientation:** The sinusoidal wave's orientation inside the filter is determined by the Gabor filter's orientation. Gabor filters can be used in a variety of orientations, including horizontal, vertical, diagonal, etc., to capture texture information at various orientations. You can find texture features that are rotation-invariant by taking into account various orientations.

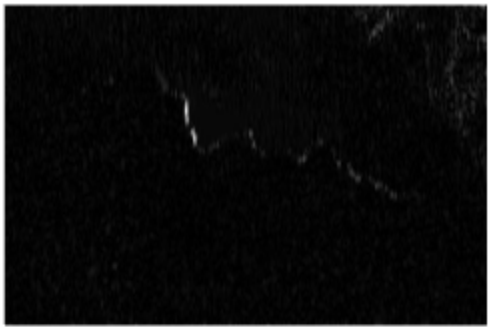
3. **Spatial Aspect Ratio:** The Gabor filter's shape is determined by the spatial aspect ratio or ellipticity. It establishes the filter's width and height in relation to one another. A Gabor filter with a spatial aspect ratio of 1 is circular, while values lower than 1 and higher than 1 provide extended filters along the horizontal and vertical axes, respectively. This parameter's adjustment enables the capture of texture features with various aspect ratios.

4. **Kernel Size:** The kernel size describes the Gabor filter's spatial coverage. It establishes the dimensions of the receptive field that the Gabor filter will use to study the texture. A lower kernel size catches local details but may be noise-sensitive, whereas a larger kernel size captures texture information over a greater region but may lose finer details.

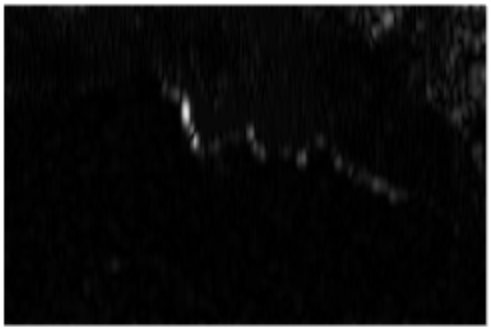
5. **Phase Offset:** The sinusoidal wave's phase shift inside the Gabor filter is specified by the phase offset parameter. Phase congruency and phase-independent features are two separate types of texture information that can be extracted using various phase offsets.

Part 4)

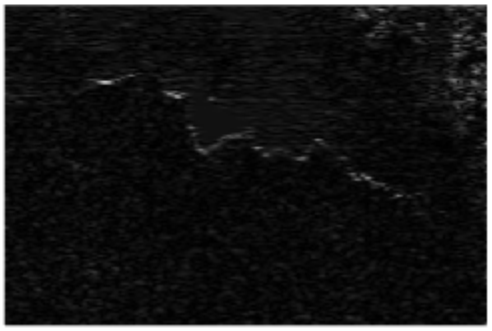
Orientation=0, Wavelength=4



Orientation=0, Wavelength=8



Orientation=90, Wavelength=4



Orientation=90, Wavelength=8

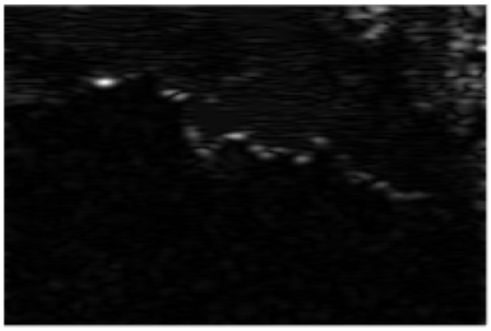


Image 1

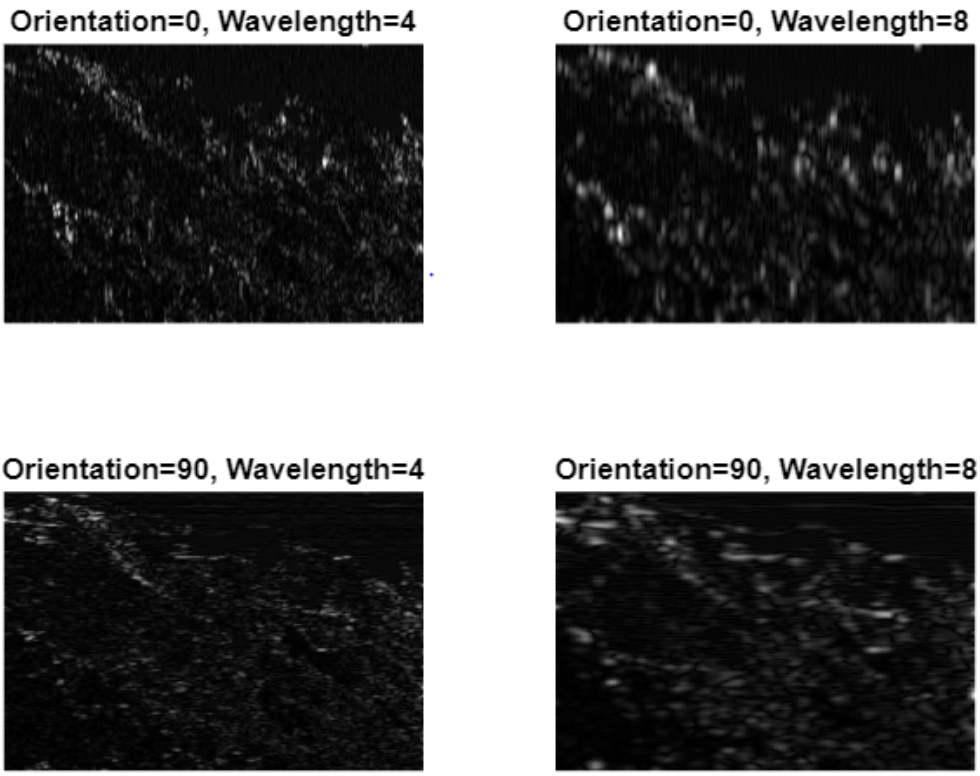


Image 2

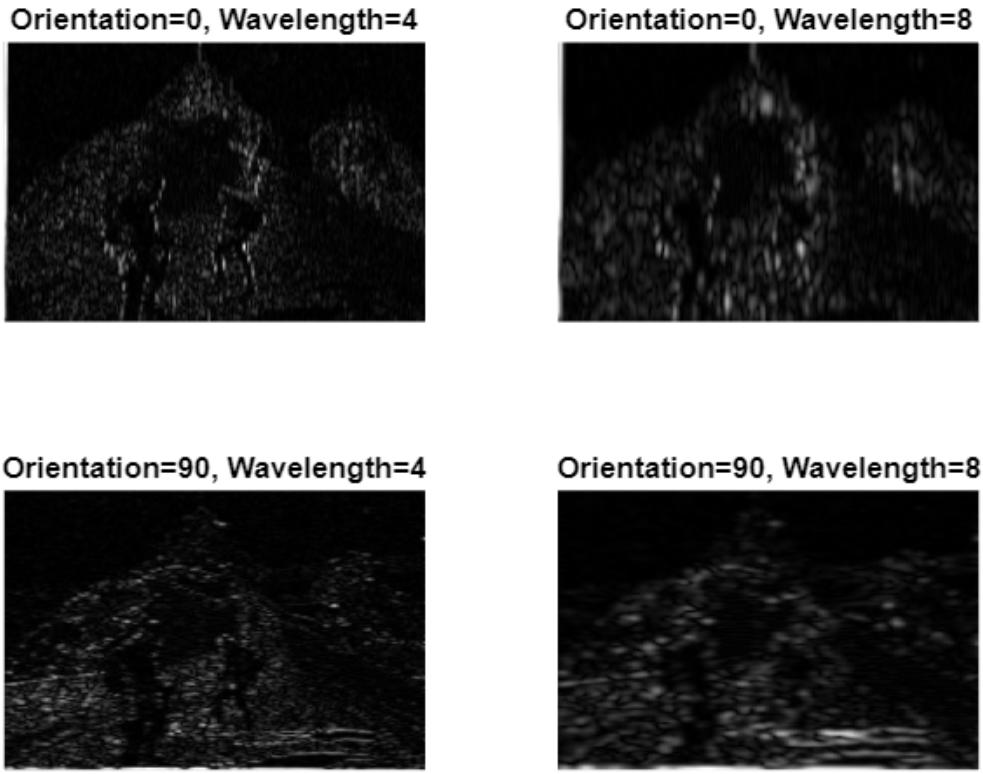


Image 3

Part 5)

Image 1 Clustered

Image 1 Contextual Clustered

Image 2 Clustered

Image 2 Contextual Clustered

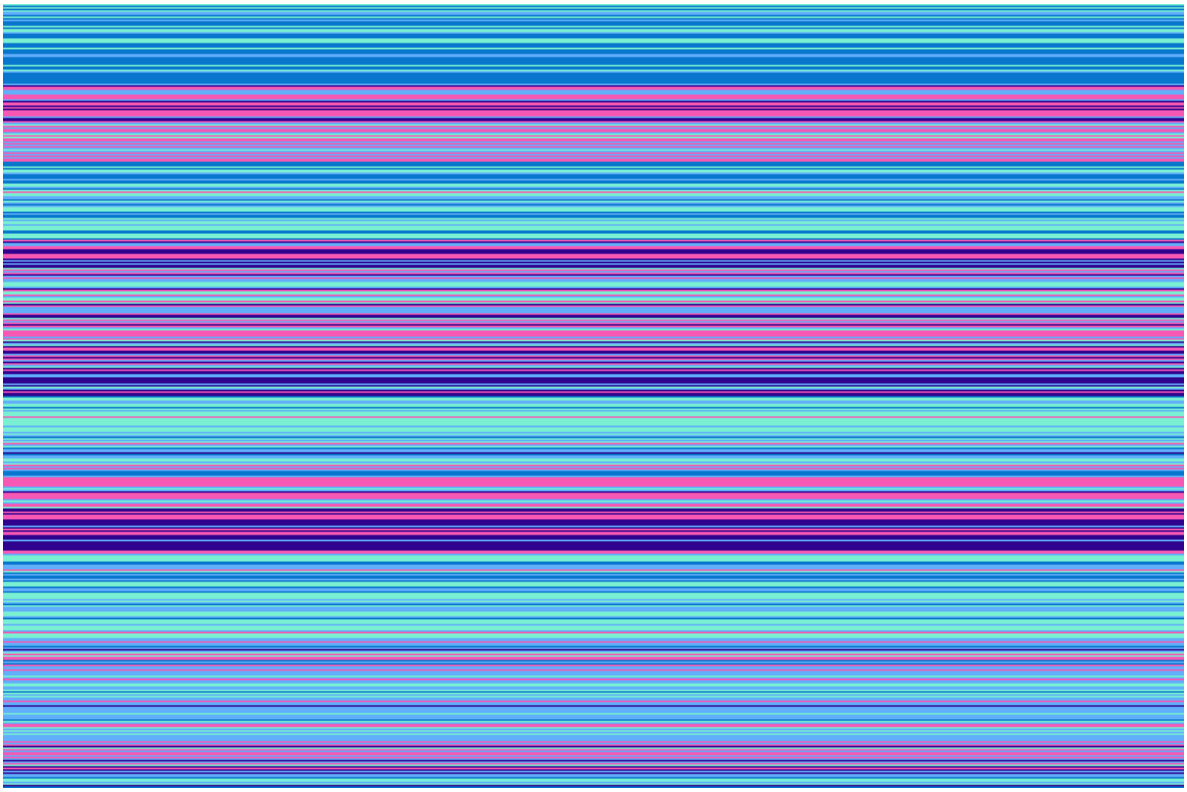


Image 3 Clustered

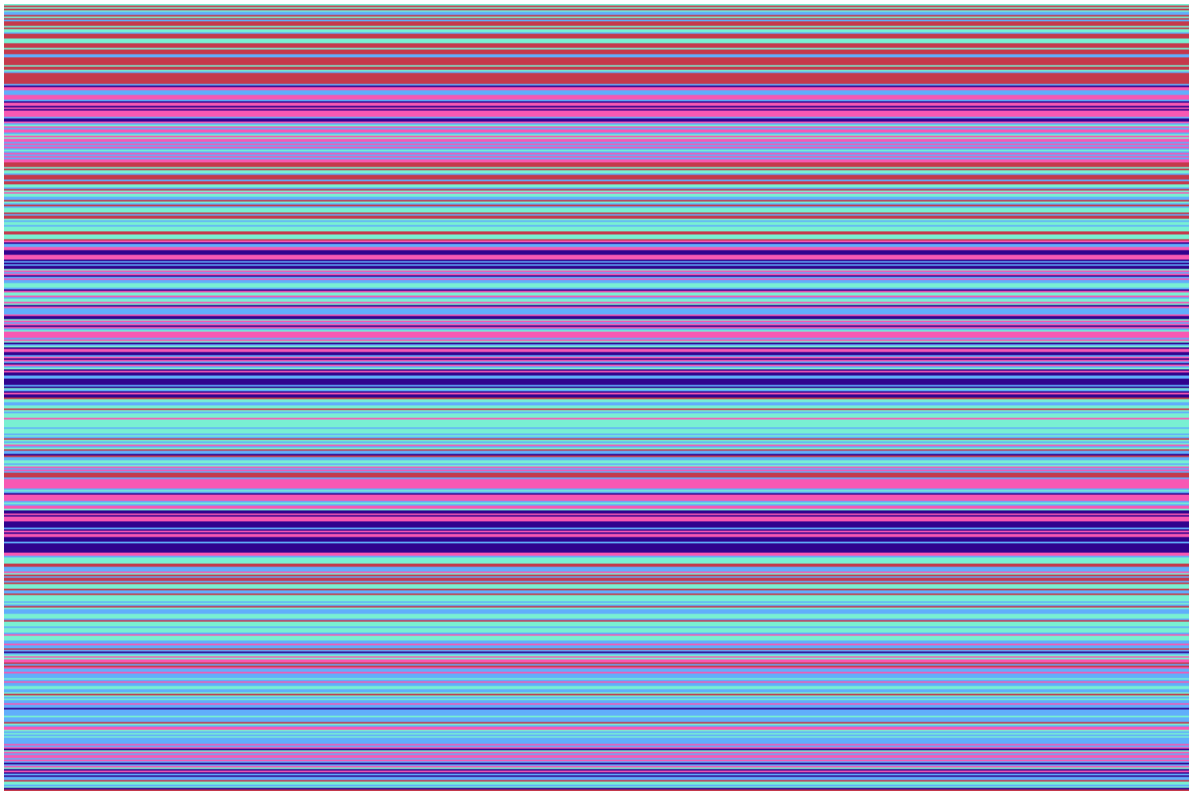


Image 3 Contextual Clustered

Image 4 Clustered

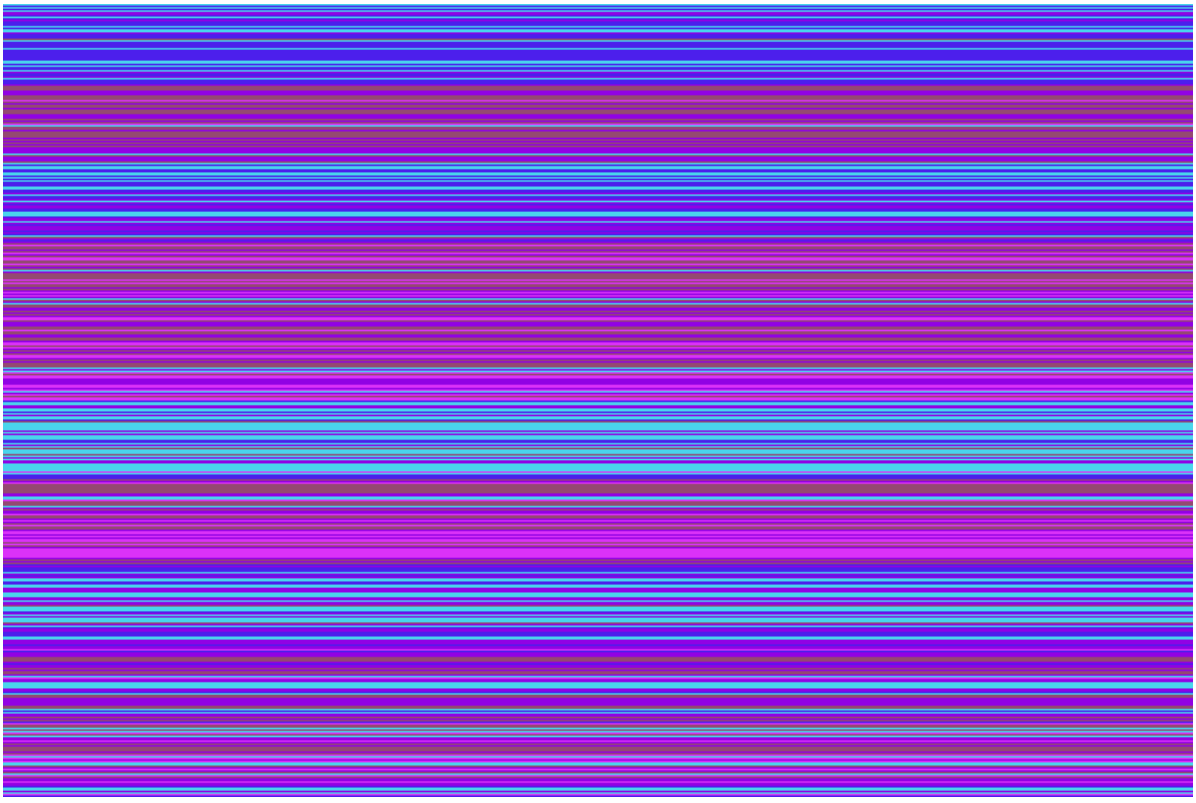


Image 4 Contextual Clustered

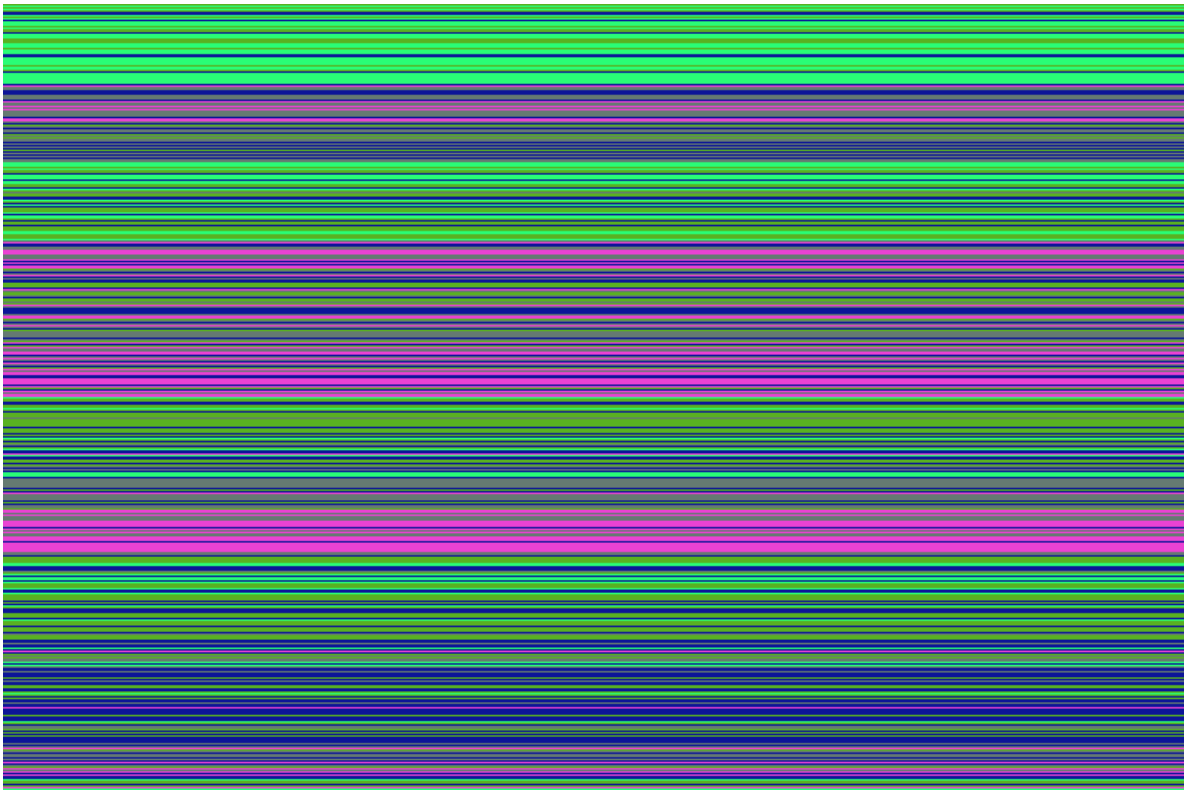


Image 5 Clustered

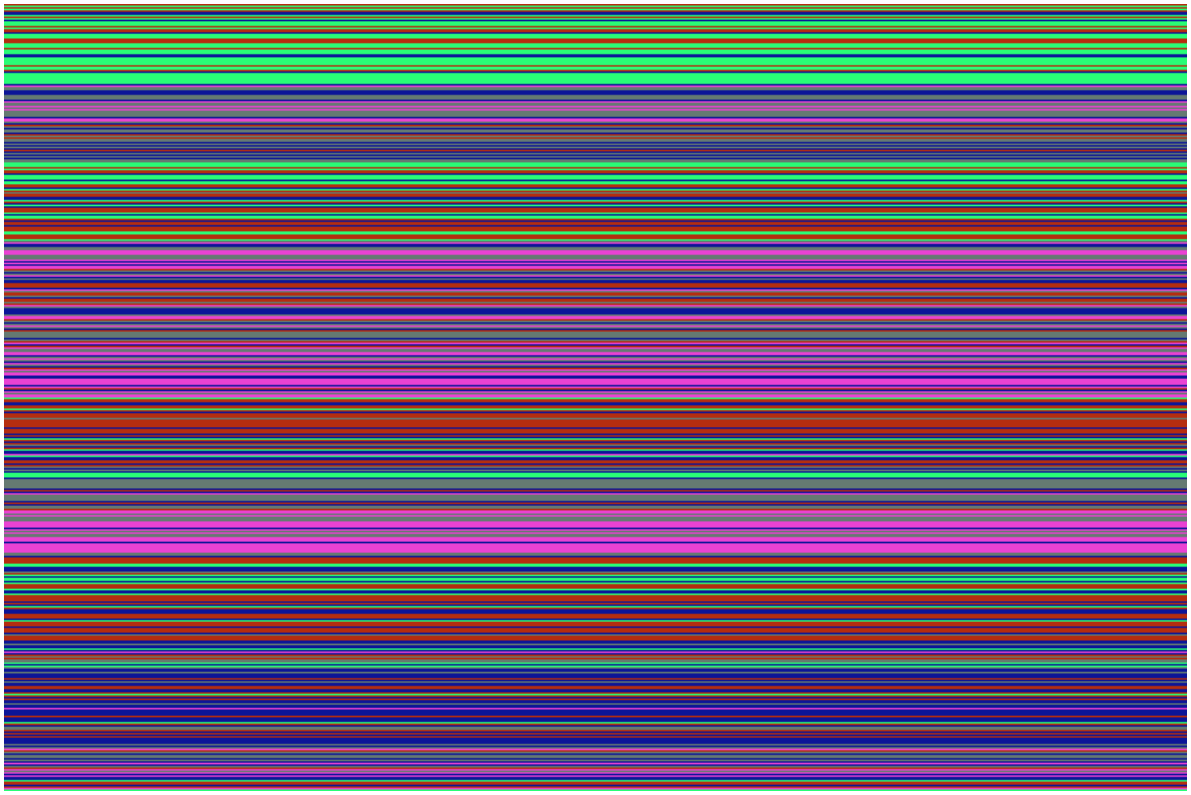


Image 5 Contextual Clustered

Image 6 Clustered

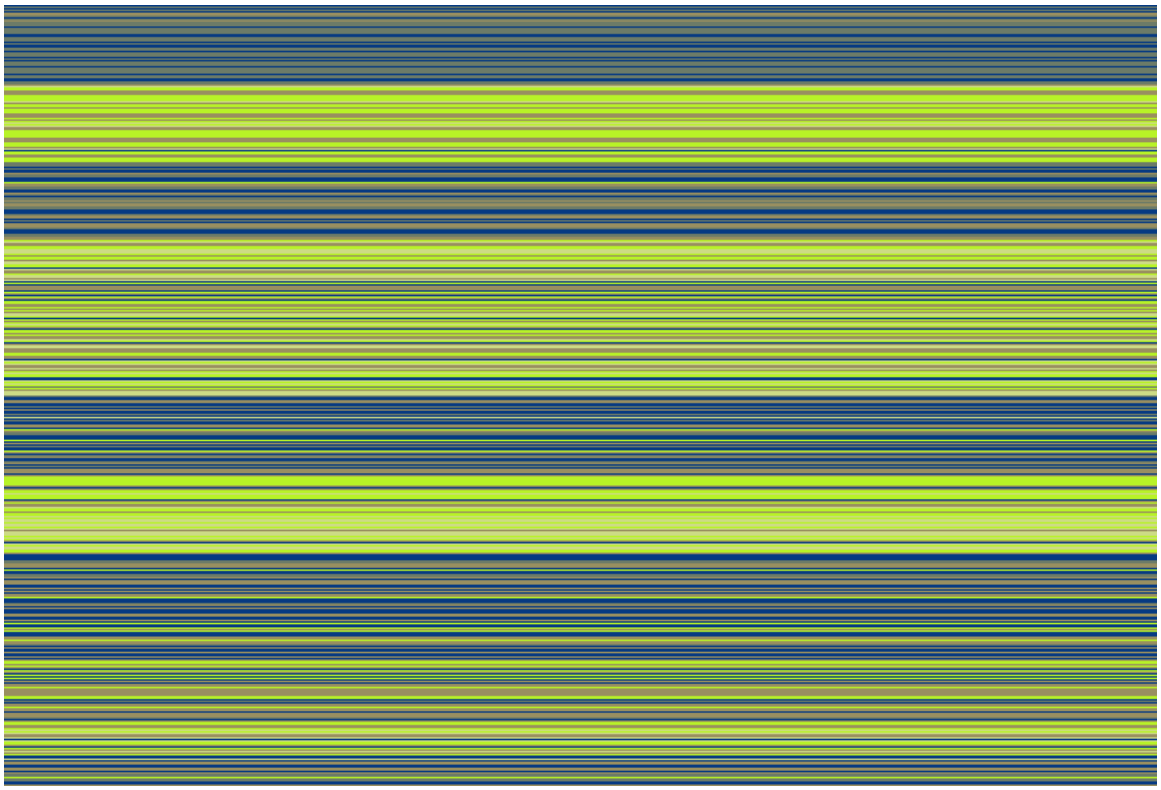


Image 6 Contextual Clustered

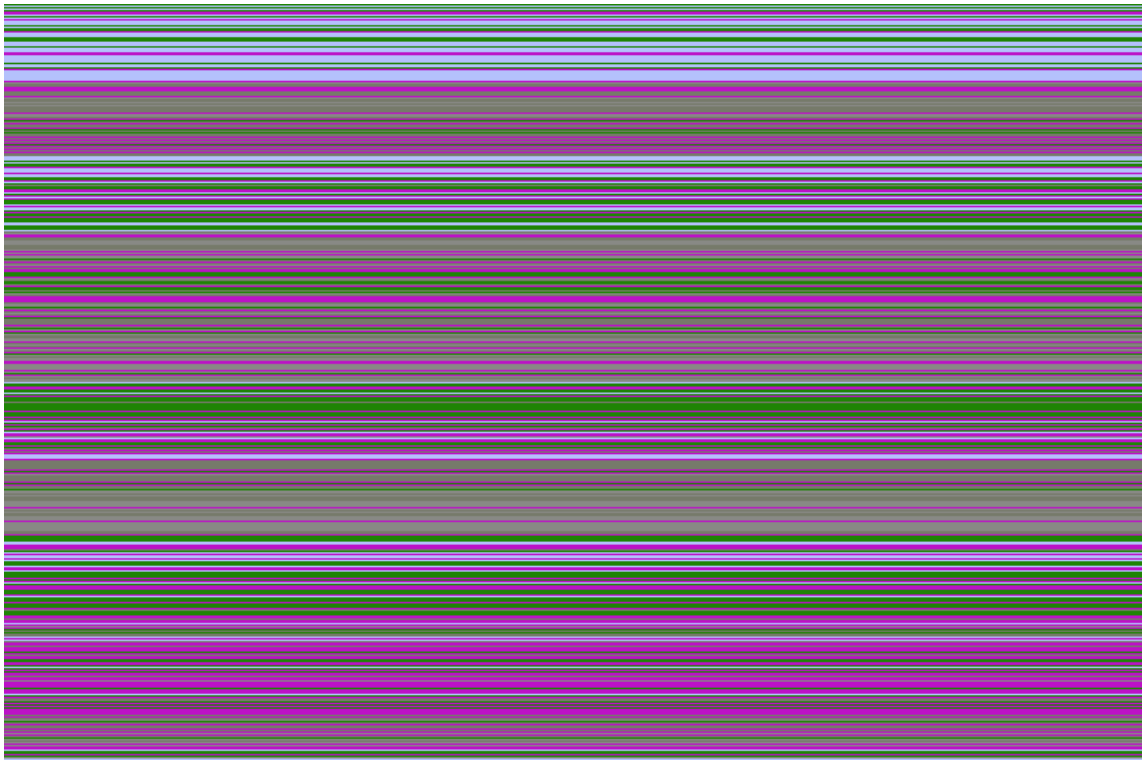


Image 7 Clustered

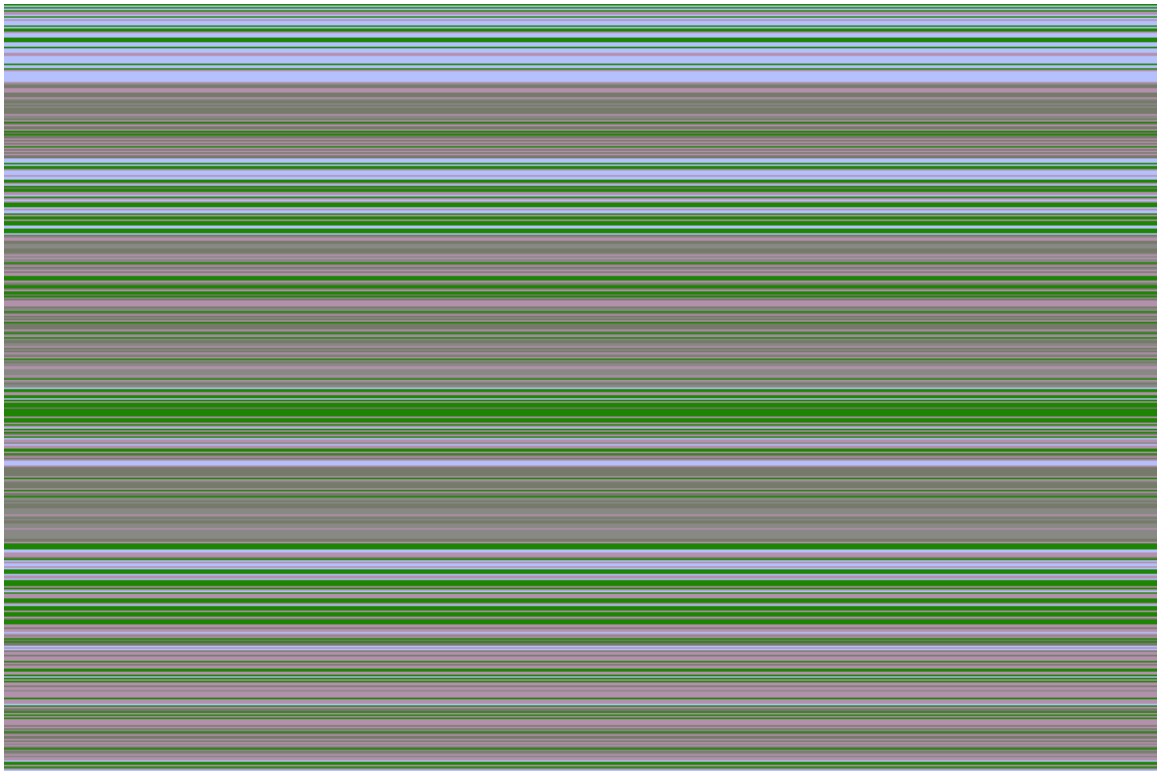


Image 7 Contextual Clustered

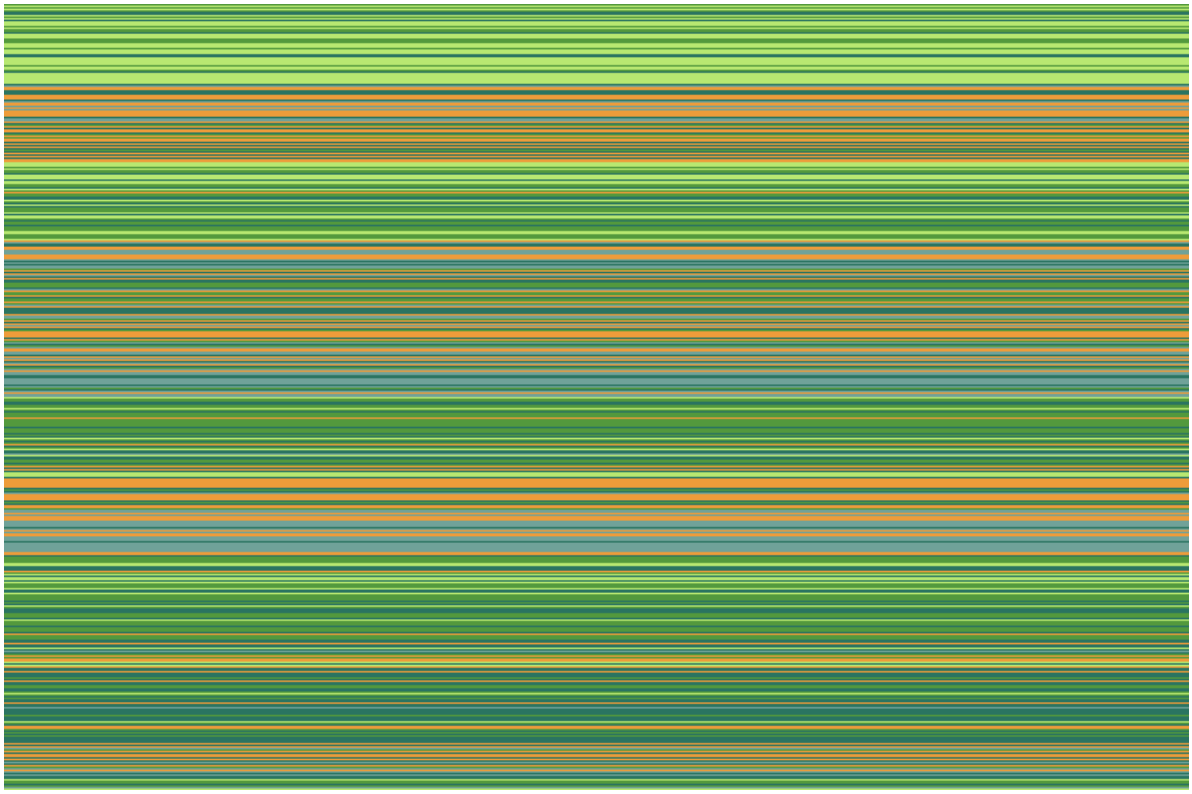


Image 8 Clustered

Image 8 Contextual Clustered

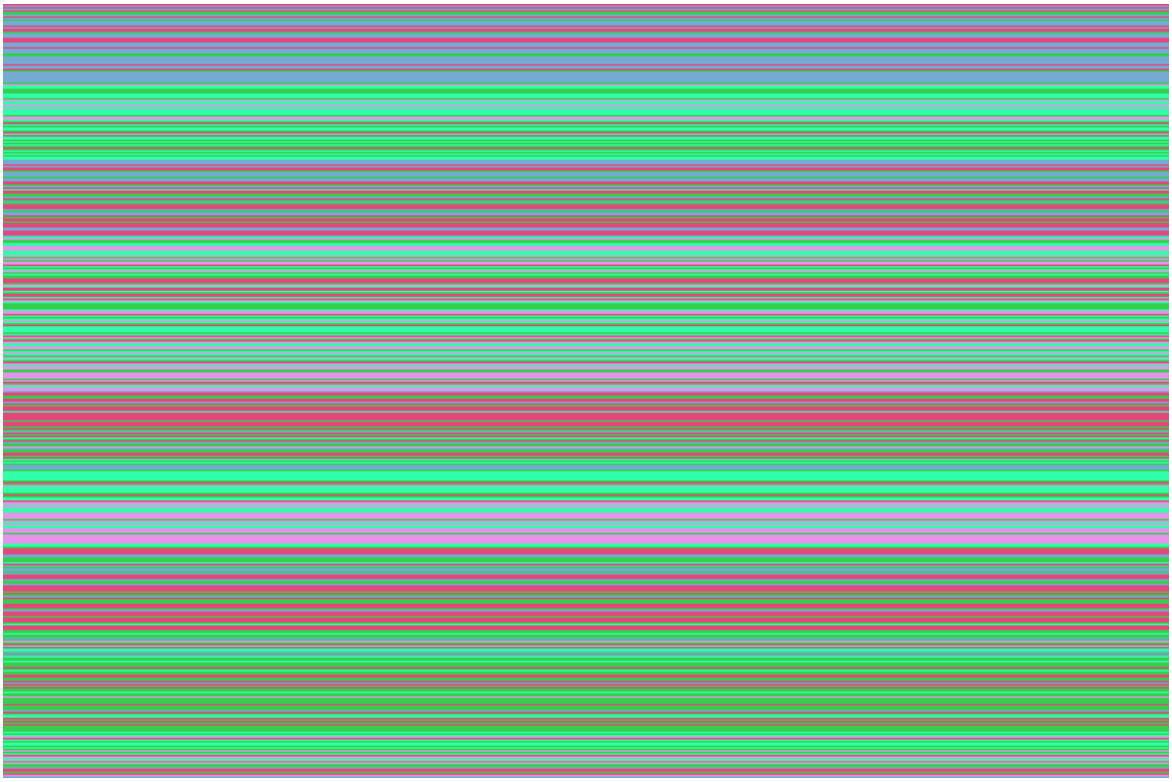


Image 9 Clustered

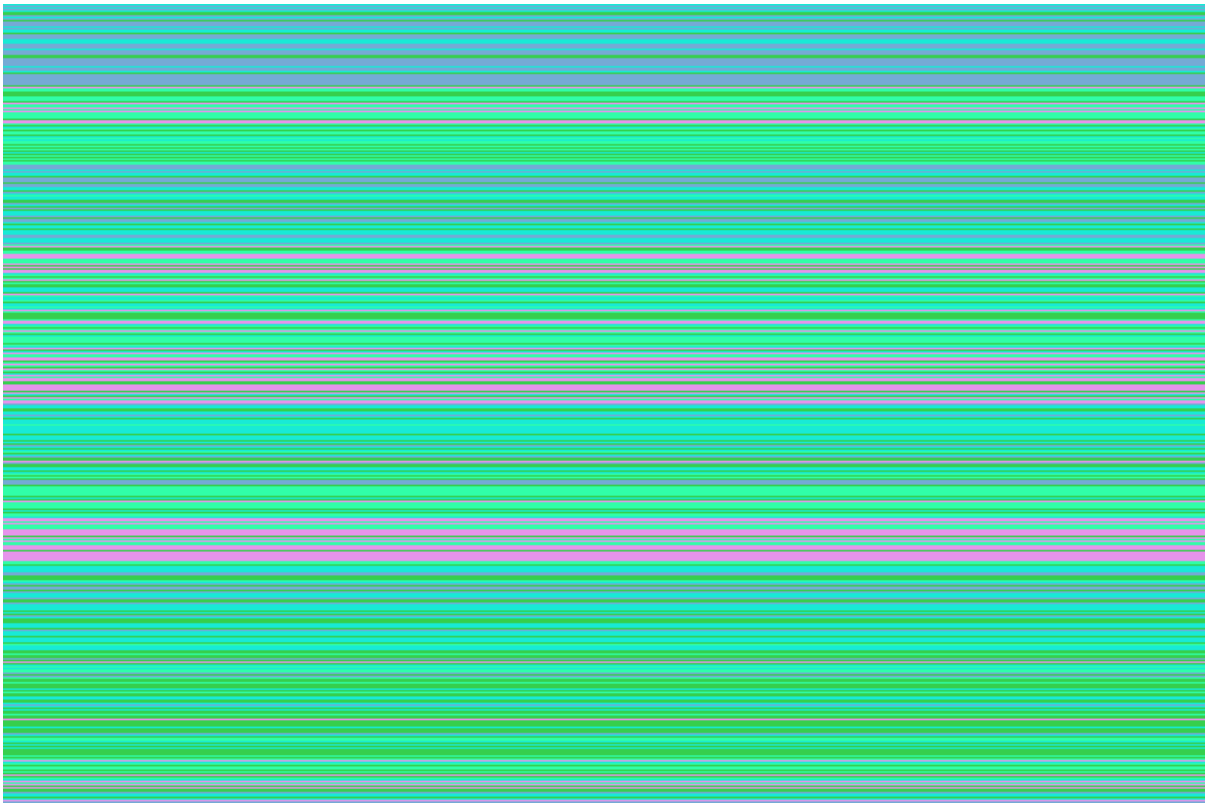


Image 9 Contextual Clustered

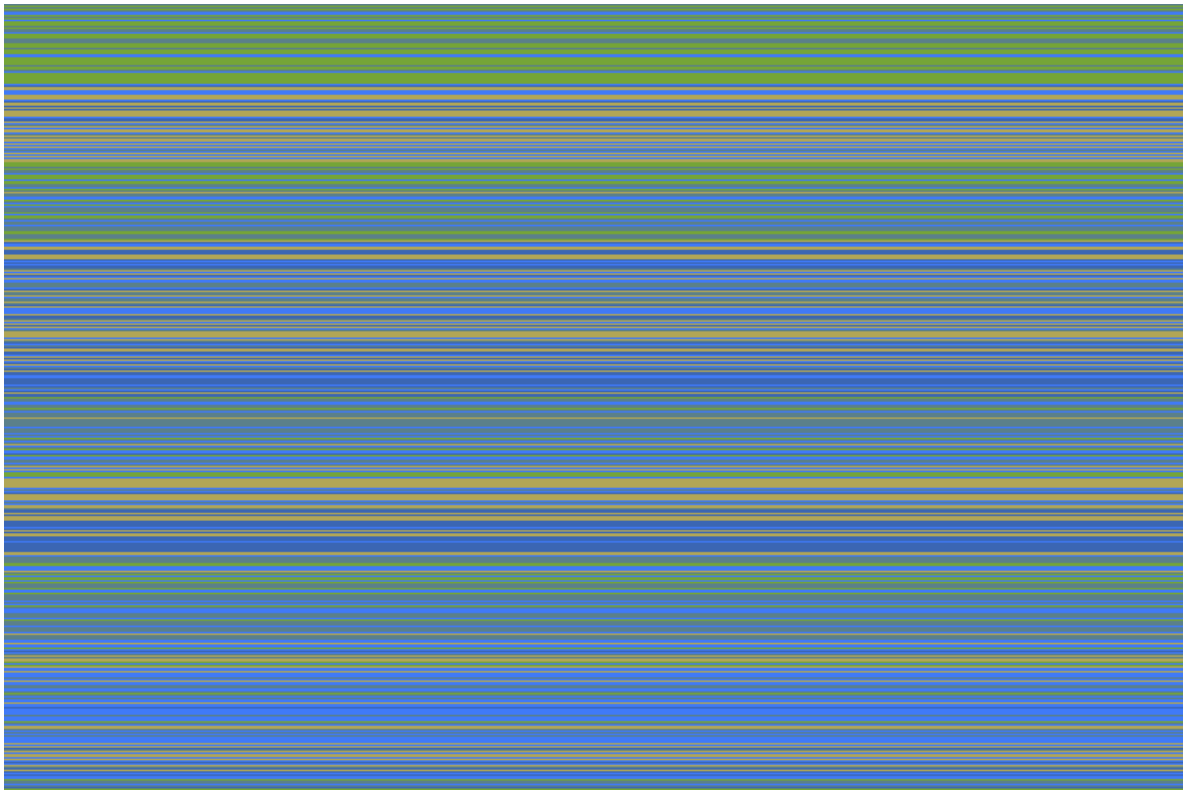


Image 10 Clustered

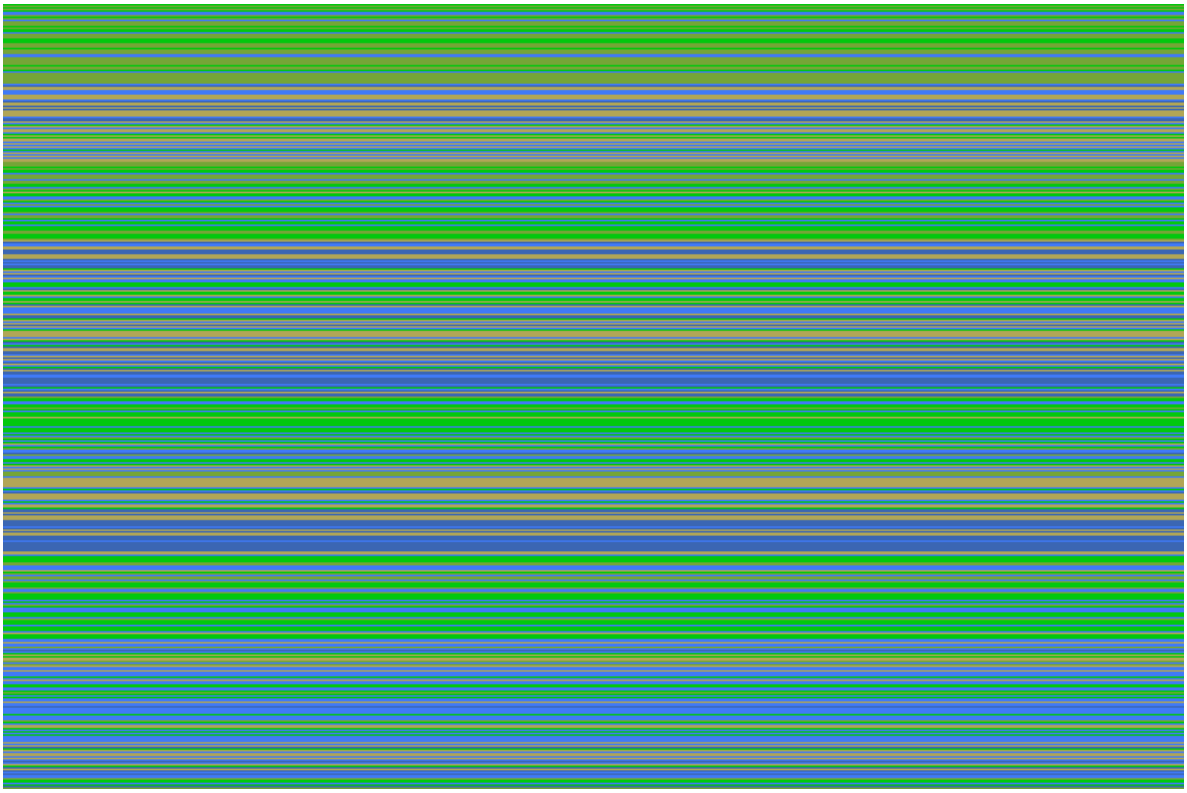


Image 10 Contextual Clustered

Part 6)

1. Program

part1.py > obtain_superpixels

```python
import cv2
import numpy as np
from skimage.segmentation import slic

def obtain_superpixels(image, num_segments, compactness):
    #Convert image to RGB
    if len(image.shape) == 2:
        image = cv2.cvtColor(image, cv2.COLOR_GRAY2RGB)
    elif image.shape[2] == 4:
        image = cv2.cvtColor(image, cv2.COLOR_BGRA2RGB)

    #SLIC superpixel algorithm
    labels = slic(image, n_segments=num_segments, compactness=compactness)

    return labels

def save_label_image(labels, output_path):
    #conv grayscale
    cv2.imwrite(output_path, labels.astype(np.uint8))

# Parameters for SLIC superpixels
num_segments = 1500  #desired superpixels
compactness = 10  #compactness

#the dataset
dataset_path = 'input'
output_path = 'label output'

for i in range(1, 11):
    image_path = f'{dataset_path}/{i}.jpg'  #image file
    output_image_path = f'{output_path}/label{i}.png'  #output file
    try:
        image = cv2.imread(image_path)

        if image is None:
            raise Exception(f"Failed to load image: {image_path}")

        # Obtain superpixels
        labels = obtain_superpixels(image, num_segments, compactness)

        # Save image
        save_label_image(labels, output_image_path)
    except Exception as e:
        print(f"Error processing image {image_path}: {str(e)}")
```

2. Program

```
def gaborconvolve(image, filters):
    # Preprocess image
    image = img_as_float(image)

    # Compute Gabor filter responses
    responses = []
    for kernel in filters:
        response = convolve2d(image, kernel, mode='same')
        responses.append(response)

    return responses
```

```
def compute_gabor_features(image, labels, filters):
    num_superpixels = np.max(labels) + 1
    num_filters = len(filters)

    # Compute Gabor filter responses for each pixel
    responses = gaborconvolve(image, filters)

    # Compute Gabor features for each superpixel
    features = np.zeros((num_superpixels, num_filters))
    for i in range(num_superpixels):
        indices = np.where(labels == i)
        for j in range(num_filters):
            response = responses[j]
            feature = np.mean(response[indices])
            features[i, j] = feature

    return features
```

```
# Generate Gabor filters
filters = []
for scale in scales:
    for orientation in orientations:
        wavelength = 2.0 / scale
        theta = orientation * np.pi / 180.0
        kernel = cv2.getGaborKernel((0, 0), wavelength, theta, 10.0, 0.5, 0, ktype=cv2.CV_64F)
        filters.append(kernel)
```

3. Program

```
# Generate pseudo-color representation for each image
for i in range(1, 11):
    image_path = f'{images_path}/{i}.jpg'  # image
    label_path = f'{labels_path}/label{i}.png' # label image
    output_image_path = f'{output_path}/image{i}_clustered.png'  # output image
    output_image_path_labels = f'{output_path_labels}/image{i}_clustered_labels.png'  # output image

    # Load orj image
    image = cv2.imread(image_path)

    # Resize the labels
    resized_labels = cv2.resize(labels.reshape(-1, 1), (image.shape[1], image.shape[0]), interpolation=cv2.INTER_NEAREST)

    # Assign pseudo-colors to pixels based on the cluster labels
    colors = [np.random.randint(0, 255, 3) for _ in range(num_clusters)]
    for label, color in zip(np.unique(resized_labels), colors):
        mask = resized_labels == label  # Create a boolean mask for pixels belonging to the current cluster
        image[mask] = color

    cv2.imwrite(output_image_path, image)

    # Load label image
    labelImages = cv2.imread(label_path)

    # Resize the labels
    resized_labels = cv2.resize(labels.reshape(-1, 1), (labelImages.shape[1], labelImages.shape[0]), interpolation=cv2.INTER_NEAREST)

    #pseudo-colors to pixels based on the cluster labels
    colors = [np.random.randint(0, 255, 3) for _ in range(num_clusters)]
    for label, color in zip(np.unique(resized_labels), colors):
        mask = resized_labels == label  # Create a boolean mask for pixels belonging to the current cluster
        labelImages[mask] = color

    cv2.imwrite(output_image_path_labels, labelImages)
```

4. Program

```
# first-level neighbors
num_superpixels = labels.max() + 1
num_features = features.shape[1]
first_level_features = np.zeros((num_superpixels, num_features))
for label in range(num_superpixels):
    mask = resized_labels == label  # mask for pixels belonging to the current superpixel
    indices = np.where(mask)[0]  # indices of the pixels belonging to the current superpixel
    if indices.size > 0:
        first_level_indices = np.unique(labels[indices])
        first_level_features[label] = np.mean(features[np.isin(labels, first_level_indices)], axis=0)

# second-level neighbors
second_level_features = np.zeros((num_superpixels, num_features))
for label in range(num_superpixels):
    first_level_indices = np.unique(labels[np.where(resized_labels == label)[0]])
    second_level_indices = np.unique(labels[np.isin(labels, first_level_indices)])
    second_level_features[label] = np.mean(features[np.isin(labels, second_level_indices)], axis=0)

# average feature vectors
superpixel_features = features
first_level_average_features = first_level_features[labels]
second_level_average_features = second_level_features[labels]
```

Part 7)

[1] “Sklearn.cluster.kmeans,” scikit, https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html (accessed May 24, 2023).

[2] “Module: Util¶,” Module: util - skimage v0.20.0 docs, https://scikit-image.org/docs/stable/api/skimage.util.html (accessed May 24, 2023).

[3] “Module: Segmentation¶,” Module: segmentation - skimage v0.20.0 docs, https://scikit-image.org/docs/stable/api/skimage.segmentation.html (accessed May 24, 2023).

[4] Ucalyptus, “UCALYPTUS/Gaborconv2d: Gabor convolutional layer.open to contributions. link to paper https://github.com/ucalyptus/gaborconv2d/blob/master/gabor.pdf,” GitHub, https://github.com/ucalyptus/GaborConv2D/tree/master (accessed May 24, 2023).

[5] “A,” Apply Gabor filter or filter bank to 2-D image - MATLAB, https://www.mathworks.com/help/images/ref/imgaborfilt.html (accessed May 24, 2023).

Part 8)

In this research, considerable effort went into choosing parameter values that would produce accurate superpixel segmentation and texture feature extraction results. 1500 superpixels (num_segments) were selected as the optimal number to balance computational effectiveness and segmentation quality. This setting gave a good amount of segmentation detail without taxing the computer's processing power. Additionally, a compactness value of 10 was used to preserve the information within each superpixel while keeping a respectable level of border conformance. This setting prevented excessive over smoothing or detail loss while producing visually pleasing and well-defined superpixels.

The quantity of scales and orientations was vital in capturing the variety of texture patterns during the extraction of Gabor texture features. To achieve an adequate representation of textures at various degrees of detail and directions, four scales and four orientations were used. This choice established a balance between capturing extensive variations in texture patterns and staying away from overly sophisticated computational methods. These values were selected with the project's goal of covering a broad variety of texture qualities without sacrificing computing efficiency in mind.

The K-means clustering algorithm's cluster count was a crucial factor to take into account while identifying texture patterns. To find a compromise between capturing various texture patterns and preventing overfitting or mistaking noise for discrete patterns, five clusters were chosen. This decision reduced the possibility of oversimplification or excessive complexity in the clustering process while still allowing for the identification of relevant texture groups.

Overall, this study aimed to get the best outcomes in both the superpixel segmentation and texture feature extraction processes by the careful selection of parameter values. The study attempted to achieve an ideal balance between computing efficiency, segmentation accuracy, texture representation, and pattern detection by carefully balancing the trade-offs associated with different parameter values.