Sean Carda
PID: A59009786
ECE 253 – Image Processing
December 8, 2021
Homework 4

Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind.

By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.

Problem 1:
Code:

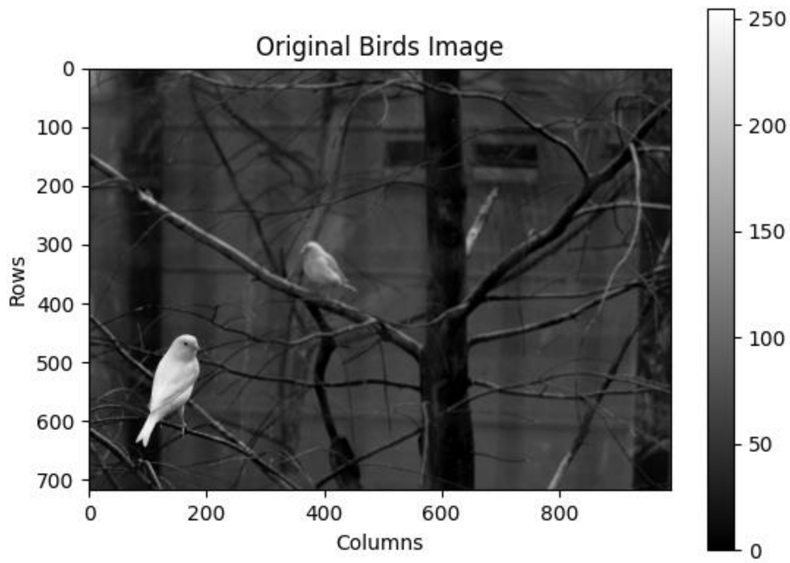
```python
#-------------------------------------------------
# Sean Carda
# ECE 253 - Image Processing
# Dr. Mohan Trivedi
# Homework 3
#-------------------------------------------------

#---------------------------------
# Global Imports.
import cv2
import numpy as np
import scipy.ndimage as ndi
import matplotlib.pyplot as plt
#---------------------------------
# Suppress scientific notation representation for numpy arrays.
np.set_printoptions(suppress=True)


#--------------------------------------------------------------------------------------------------
# Cross Correlation Filter.

# First, read in the first birds image and the template we will use to identify the features
# of interest in the image.
print('Loading images...')
birds = cv2.cvtColor(cv2.imread('birds1.jpeg'), cv2.COLOR_BGR2GRAY)
birds_2 = cv2.cvtColor(cv2.imread('birds2.jpeg'), cv2.COLOR_BGR2GRAY)
template = cv2.cvtColor(cv2.imread('template.jpeg'), cv2.COLOR_BGR2GRAY)
print('Done!')

# Define a method for showing images which will reduce the lines of code needed
# to show images.
def imshow(im, title, fig_num, xlabel, ylabel):
    plt.figure(fig_num)
    plt.imshow(im, cmap='gray')
    plt.colorbar()
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

# Show the original image and template.
imshow(birds, 'Original Birds 1 Image', 1, 'Columns', 'Rows')
imshow(birds_2, 'Original Birds 2 Image', 2, 'Columns', 'Rows')
imshow(template, 'Template for Matching', 3, 'Columns', 'Rows')

# Compute convolution between the birds image and the template.
print('Cross correlating...')
matching = ndi.convolve(np.float64(birds), np.float64(template))
```

```python
imshow(matching, 'Birds 1 Image Convolved with Template', 4, 'Columns', 'Rows')
print('Done!')
plt.show()

#-----------------------------------------------------------------------------------------------
# Normalized Cross Correlation Filter.

# Define a function that accepts an image and a template and computes the normalized
# cross-correlated image.
def normalized_cc(im, template):
    # Grab the sizes of the image and template.
    im_size = im.shape
    t_size = template.shape

    # Pad the image with the size of the template.
    im_pad = np.pad(im, (int(t_size[0] / 2) + 1, int(t_size[1] / 2)), mode='symmetric')

    # Instantiate an output image.
    output = np.zeros(im_size)

    # Calculate the mean of the template and the term of the normalization contributed by the
    # template.
    t_avg = np.mean(template)
    t_offset = template - t_avg
    norm_coeff_t = np.sum((template - t_avg)**2)
    for r in range(0, im_size[0]):
        for c in range(0, im_size[1]):
            # Grab the image windowed by the template.
            im_window = im_pad[r:r+t_size[0], c:c+t_size[1]]

            # Compute the average of the image in the current window.
            im_avg = np.mean(im_window)

            # Using the calculated means, compute the cross correlation of the image in the current
            # window.
            im_offset = (im_window - im_avg)
            cross_corr = np.sum(im_offset * t_offset)

            # Compute the normalized coefficients of the image and the template at the current
            # window.
            norm_coeff_im = np.sum(im_offset**2)
            norm_coeff = norm_coeff_t * norm_coeff_im

            # Comptue the normalized cross-correlation.
            norm_cc = cross_corr / np.sqrt(norm_coeff)
            output[r, c] = norm_cc

    # Return.
    print('Done!')
    return output


# Accepts an image and a point in the image and places a box of given dimension around that
# point in the image.
def identify(im, point, dim):
    # Define an output image which will be boxed.
    size = im.shape
    im_boxed = np.zeros([size[0], size[1], 3])
    im_boxed[:, :, 0] = np.copy(im)
    im_boxed[:, :, 1] = np.copy(im)
    im_boxed[:, :, 2] = np.copy(im)

    # Calculate the height and width of the box.
    width = int(dim[1] / 2)
    height = int(dim[0] / 2)

    # Create points.
    point_1 = (point[1] - width, point[0] - height)
    point_2 = (point[1] + width, point[0] + height)

    # Draw the rectangle.
    cv2.rectangle(im_boxed, point_1, point_2, color=(255, 0, 0), thickness=3)

    # Return.
    return np.uint8(im_boxed)

# Compute the normalized cross correlation on the first birds image.
print('Computing normalized cross correlation for birds 1...')
birds_norm_cc = normalized_cc(np.float64(birds), np.float64(template))
imshow(birds_norm_cc, 'Birds Normalized Cross Correlation', 1, 'Columns', 'Rows')

# Place a box around the point of strongest intensity.
point = np.unravel_index(np.argmax(birds_norm_cc), birds_norm_cc.shape)
print(point)
matched_image = identify(birds, point, template.shape)
imshow(matched_image, 'Birds 1 with Identified Template Match', 2, 'Columns', 'Rows')

# Compute the normalized cross correlation on the second birds image.
print('Computing normalized cross correlation for birds 2...')
birds_norm_cc = normalized_cc(np.float64(birds_2), np.float64(template))
imshow(birds_norm_cc, 'Birds 2 Normalized Cross Correlation', 3, 'Columns', 'Rows')

# Place a box around the point of strongest intensity.
point = np.unravel_index(np.argmax(birds_norm_cc), birds_norm_cc.shape)
print(point)
matched_image = identify(birds_2, point, template.shape)
imshow(matched_image, 'Birds 2 with Identified Template Match', 4, 'Columns', 'Rows')
```

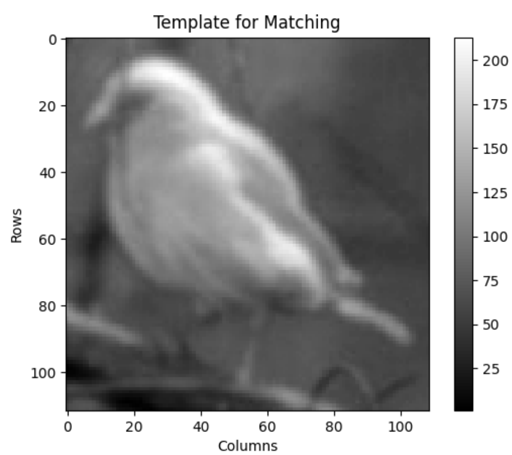
Output:

Cross Correlation:

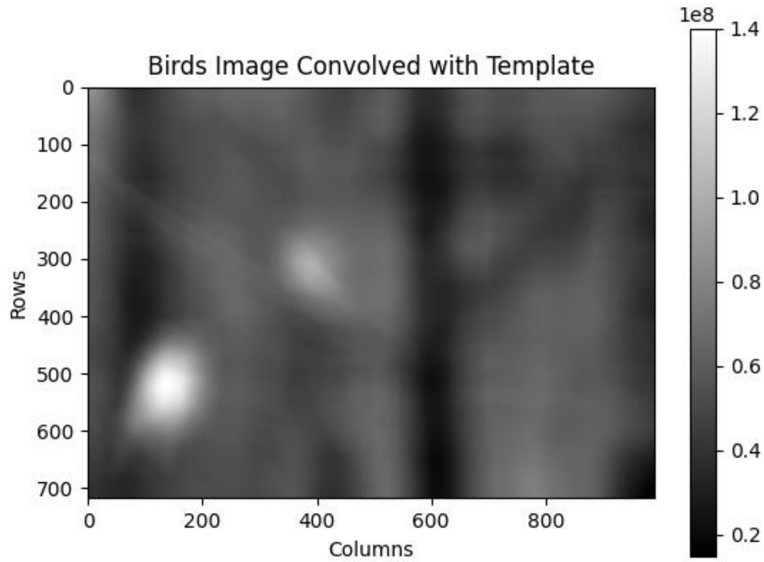
Original Birds 1 Image:



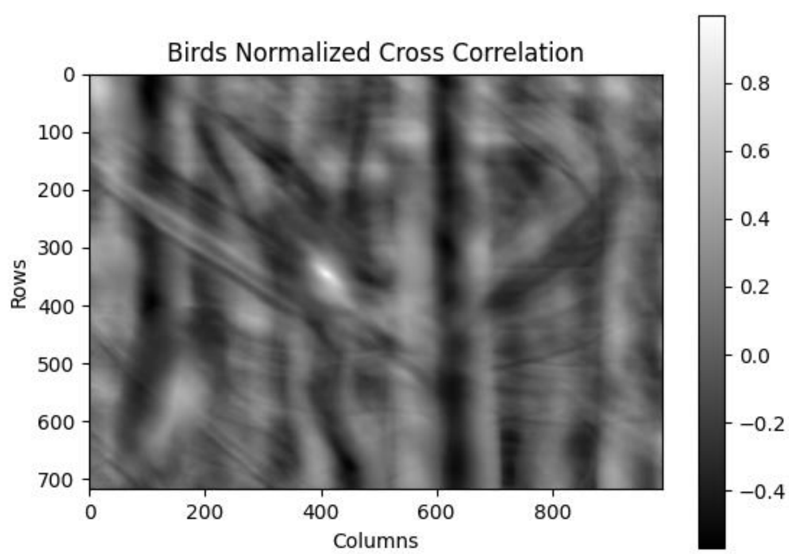
Template Image:



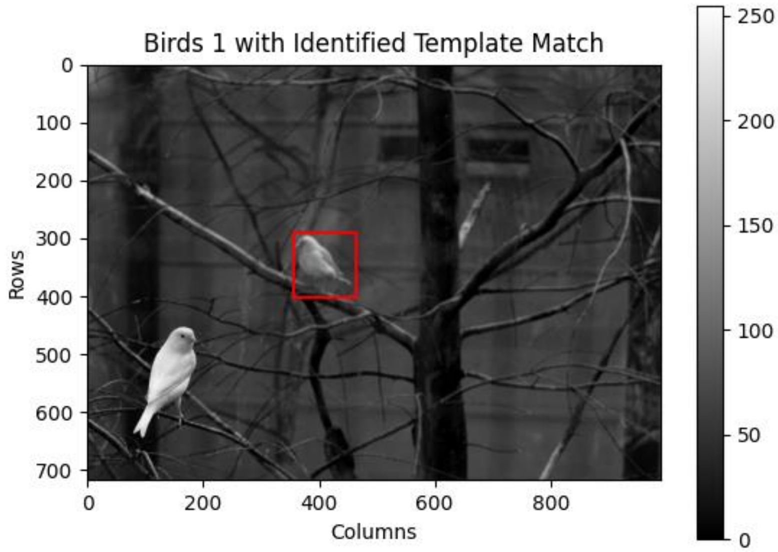
Cross Correlation of Birds 1 with Template:



Normalized Cross Correlation:
Birds 1:

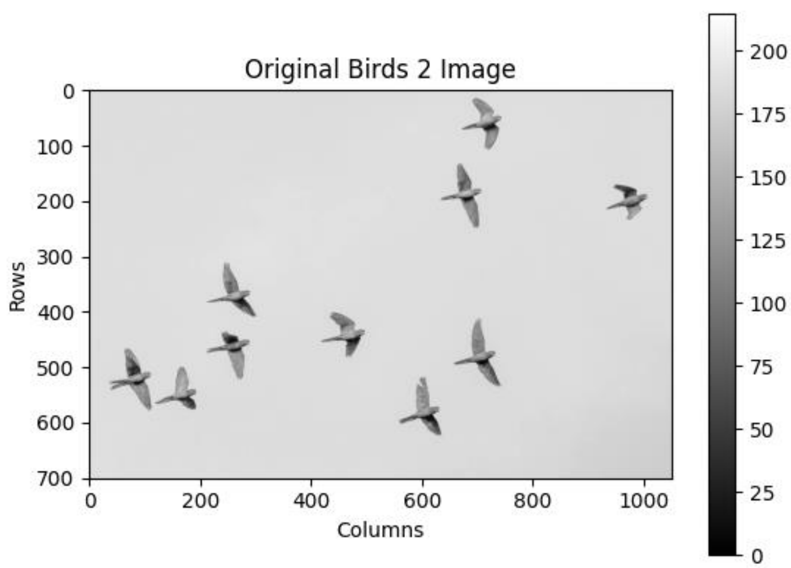


Birds 1 Matched:

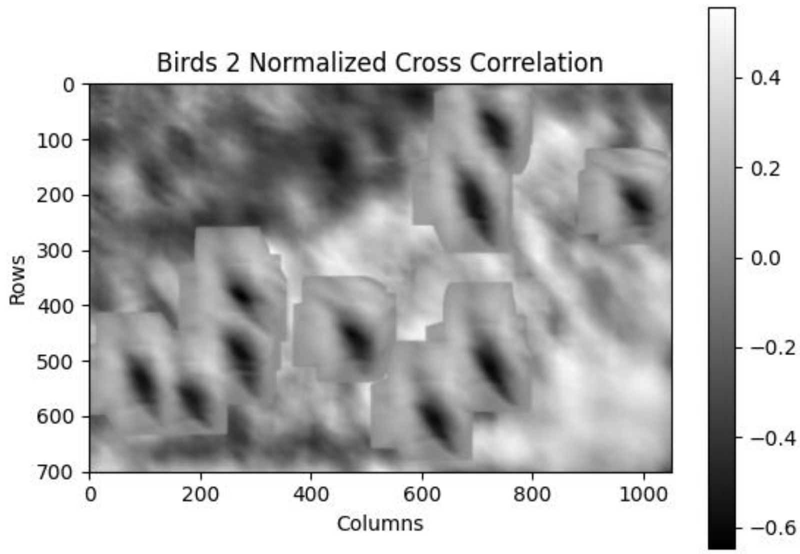


Here, it is evident that the template has found a match in the Birds 1 image. This is simple due to the fact that the bird’s orientation and lightning in the image strongly matches with the bird in the template.

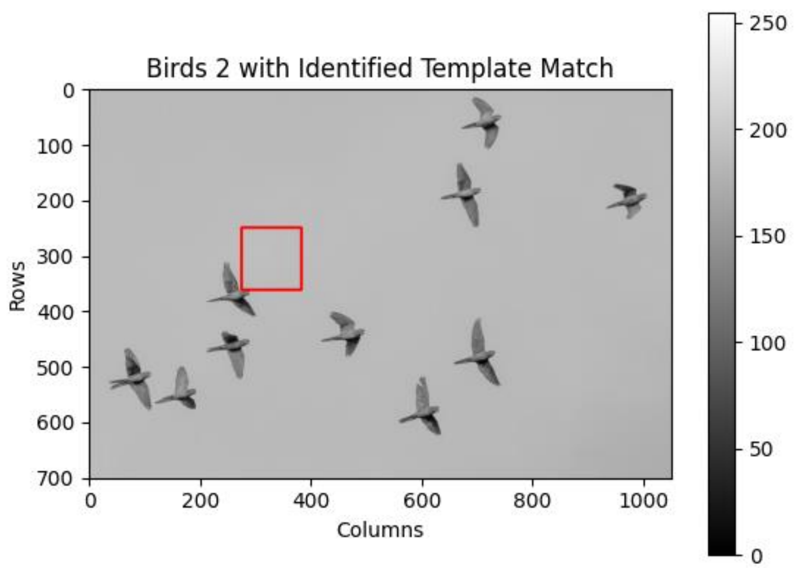
Original Birds 2 Image:



Birds 2 NCC:



Birds 2 Matched:



Here, it is clear that none of the birds have been identified in the image. This is due to the fact that the lightning and orientation of all the birds in the image do not match up with any components of the template image. It's possible that if one of the birds were facing the other direction, we would have a stronger correlation between the image and the template.

Problem 2

Code:

```python
#------------------------------------------------
# Sean Carda
# ECE 253 - Image Processing
# Dr. Mohan Trivedi
# Homework 3
#------------------------------------------------

#--------------------------------
# Global Imports.
import cv2
import numpy as np
import scipy.ndimage as ndi
import matplotlib.pyplot as plt
#--------------------------------
# Suppress scientific notation representation for numpy arrays.
np.set_printoptions(suppress=True)

#-----------------------------------------------------------------------------------------------
# HOUGH TRANSFORM ON TEST IMAGE.

# Method which accepts an image and computes the Hough transform.
def HT(im):
    # Maximum angle for the transform.
    t_max = 90

    # Generate all possible theta values.
    theta = np.int64(np.round(np.linspace(-t_max, t_max, 2 * t_max + 1)))

    # Calculate the maximum distance between opposite corners of the image.
    size = im.shape
    D = np.int64(np.floor(np.sqrt((size[0]**2) + (size[1]**2))))
    rho = np.int64(np.round(np.linspace(-D, D, 2 * D + 1)))

    # Instantiate an output image.
    output = np.zeros([len(rho), len(theta)])

    for r in range(0, size[0]):
        for c in range(0, size[1]):
            if im[r, c] >= 1:
                for t in theta:
                    p = (im[r, c] * c * np.cos(np.deg2rad(t))) + (im[r, c] * r * np.sin(np.deg2rad(t)))
                    p = np.int64(p + np.max(rho))
                    output[p, t + t_max] += 1
        if (r % 50) == 0:
            print('Progress: ' + str(round(100 * r / size[0], 2)) + '%...')

    # Return the transform and the established theta and rho values.
    return [output, rho, theta]

# Method which accepts an image and a list of rho and theta parameters and draws the corresponding
# x-y lines on the image. Parameters = [list of theta, list of rho]
def draw_lines(params, im, width, round):
    # Create a new image which will overlay the lines on top of the image.
    size = im.shape
    output = np.zeros([size[0], size[1], 3])
    output[:, :, 0] = im
    output[:, :, 1] = im
    output[:, :, 2] = im

    # For every parameter combination, draw the corresponding line.
    for combo in params:
        # Grab theta and rho
        theta = combo[0]
        rho = combo[1]

        # Compute the new line parameters.
        if theta == 0:
            theta = 0.01
        elif theta == 90:
            theta = 89.99
        elif theta == -90:
            theta = -89.99

        b = rho / (np.sin(np.deg2rad(theta)))
        if round is True:
            m = np.round(-1 / (np.tan(np.deg2rad(theta))))
        else:
            m = -1 / (np.tan(np.deg2rad(theta)))

        # Compute the two points needed to generate the line
        point_1 = ((-999, int(np.round((-999 * m) + b))))
        point_2 = ((999, int(np.round((999 * m) + b))))

        # Draw the line onto the image.
        cv2.line(output, point_1, point_2, (255, 0, 0), width, lineType=cv2.LINE_AA)

    # Return the image.
    return output


# Method which accepts a thresholded transform and the respective rho and theta values and
# returns the theta-row indices of the transformed image.
def get_params(im, rho, theta):
    # Return the indices at which the transform is nonzero.
    list = np.transpose(np.nonzero(im))
```

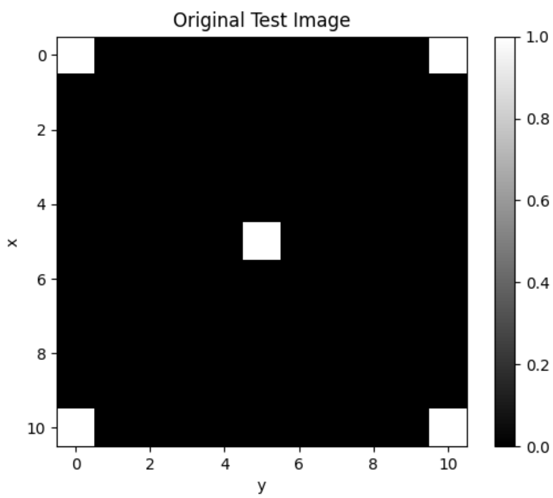
```python

    # Augment the list by the theta values by the maximum theta value.
    list[:, 1] = list[:, 1] - np.max(theta)
    list[:, 0] = list[:, 0] - np.max(rho)

    # Return the flipped array so that theta is an index [~, 0] and rho is at [~, 1]
    list_flip = np.flip(list, 1)
    return list_flip


#-----------------------------------------------------------------------------------------------------
# HOUGH TRANSFORM ON TEST IMAGE.
# First, generate an 11 x 11 image with 5 1's located in the center and in the four corners.
test_im = np.zeros([11, 11])
size = test_im.shape
points = [[0, 0], [0, size[1] - 1], [int(size[0] / 2), int(size[1] / 2)], [size[0] - 1, 0], [size[0] - 1, size[1] - 1]]
for point in points:
    test_im[point[0], point[1]] = 1

# Calculate the Hough transform of the test image.
[transform, rho, theta] = HT(test_im)

# Threshold the transformed image and look for intersections greater than 2.
transform_thresh = np.where(transform > 2, transform, 0)

# Draw the lines specified by the transform.
params = get_params(transform_thresh, rho, theta)
print(params)
test_lines = draw_lines(params, test_im, 1, True)

plt.figure(1)
plt.imshow(test_im, cmap='gray')
plt.title('Original Test Image')
plt.xlabel('y')
plt.ylabel('x')
plt.colorbar()

plt.figure(2)
plt.imshow(transform, extent=[theta[0], theta[len(theta) - 1], rho[len(rho) - 1], rho[0]], cmap='gray')
plt.title('Hough Transform on Test Image')
plt.xlabel('theta (degrees)')
plt.ylabel('p')
plt.colorbar()

plt.figure(3)
plt.plot(3, 5, 1, 9, color='white', linewidth=1)
plt.imshow(test_lines, cmap='gray')
plt.title('Test Image with Lines')
plt.xlabel('y')
plt.ylabel('x')
plt.colorbar()
plt.show()

#-----------------------------------------------------------------------------------------------------
# HOUGH TRANSFORM ON LANE IMAGE III

# Read in the lane image.
lane = cv2.imread('lane.png')
thresh = 240
binary_lane = cv2.Canny(lane[:, :, 0], 175, thresh, apertureSize=3, L2gradient=True) / 255


plt.figure(4)
plt.imshow(lane, cmap='gray')
plt.title('Original Lane Image')
plt.xlabel('y')
plt.ylabel('x')
plt.colorbar()

plt.figure(5)
plt.imshow(binary_lane, cmap='gray')
plt.title('Original Lane Image')
plt.xlabel('y')
plt.ylabel('x')
plt.colorbar()
plt.show()

# Compute the Hough transform of the edged image.
[transform, rho, theta] = HT(np.uint64(binary_lane))

# Threshold the transformed image and look for intersections greater than 2.
transform_thresh = np.where(transform > (0.75 * np.max(transform)), transform, 0)

# Draw the lines specified by the transform.
params = get_params(transform_thresh, rho, theta)
print('Parameters for analysis:')
print(params)
print('Drawing Lines...')
lane_lines = draw_lines(params, lane[:, :, 0], 3, False)

plt.figure(4)
plt.imshow(lane, cmap='gray')
plt.title('Original Lane Image')
plt.xlabel('y')
plt.ylabel('x')
plt.colorbar()
```

```python

plt.figure(5)
plt.imshow(binary_lane, cmap='gray')
plt.title('Edged Lane Image')
plt.xlabel('y')
plt.ylabel('x')
plt.colorbar()


plt.figure(6, figsize=(10, 20))
plt.imshow(transform, extent=[theta[0], theta[len(theta) - 1], rho[len(rho) - 1], rho[0]], aspect='auto', cmap='gray')
plt.title('Hough Transform on Lane Image')
plt.xlabel('theta (degrees)')
plt.ylabel('p')
plt.colorbar()

plt.figure(8)
plt.imshow(np.uint8(lane_lines), cmap='gray')
plt.title('Lane Image with Lines')
plt.xlabel('y')
plt.ylabel('x')
plt.colorbar()
plt.show()


#-----------------------------------------------------------------------------------------------------
# HOUGH TRANSFORM ON LANE IMAGE IV

# Set a theta range for which to preserve lane edges.
theta_n_low = -57 + 90
theta_n_high = -50 + 90
theta_p_low = 50 + 90
theta_p_high = 57 + 90

# Threshold the thresholded image again, but only preserving the ranges of theta specified.
t_size = transform_thresh.shape
theta_thresh = np.zeros(transform_thresh.shape)
for i in range(t_size[1]):
    if (i > theta_n_low and i < theta_n_high) or (i > theta_p_low and i < theta_p_high):
        theta_thresh[:, i] = transform_thresh[:, i]


# Draw the lines specified by the transform.
params = get_params(theta_thresh, rho, theta)
print('Parameters for analysis:')
print(params)
print('Drawing Lines...')
new_lane_lines = draw_lines(params, lane[:, :, 0], 3, False)

# Show the new lined image.
plt.figure(1)
plt.imshow(np.uint8(new_lane_lines), cmap='gray')
plt.title('New Lane Image with Lines')
plt.xlabel('y')
plt.ylabel('x')
plt.colorbar()
plt.show()
```

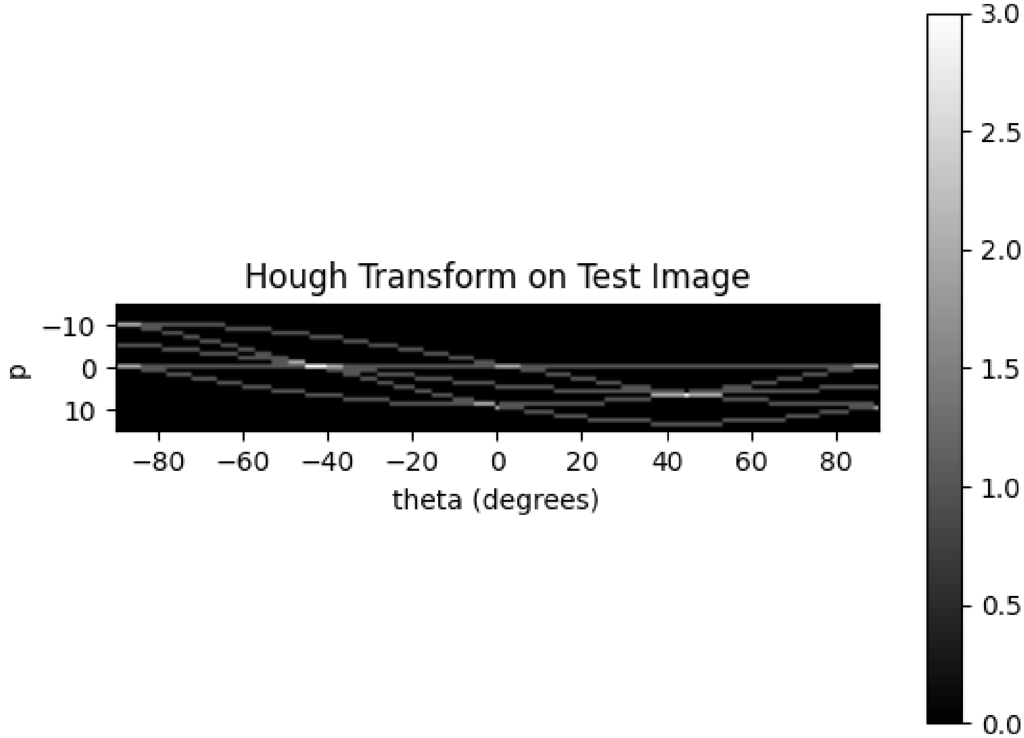
Output:

Test Image:

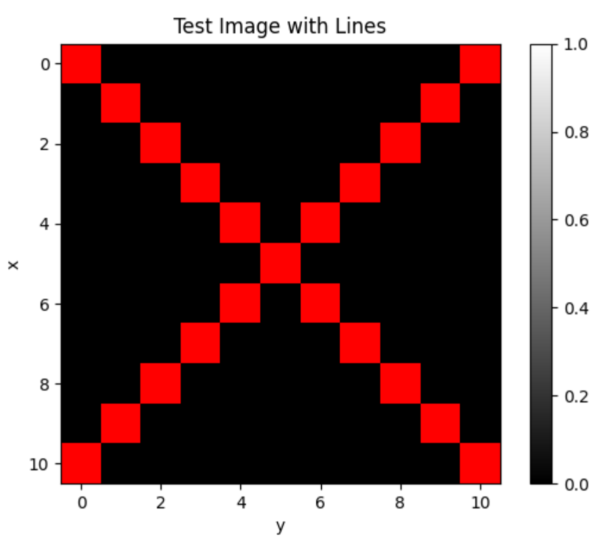
Original Image:



Hough Transform of Test Image:

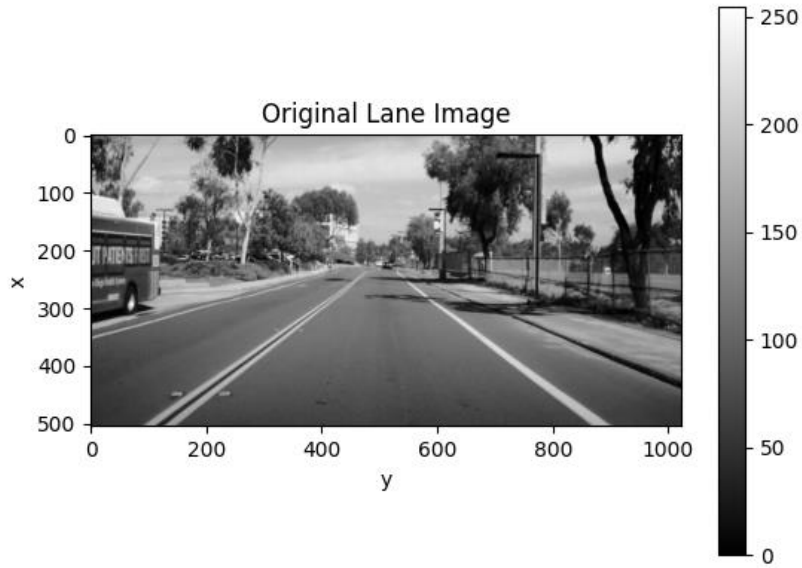


Lines Overlayed on Test Image:

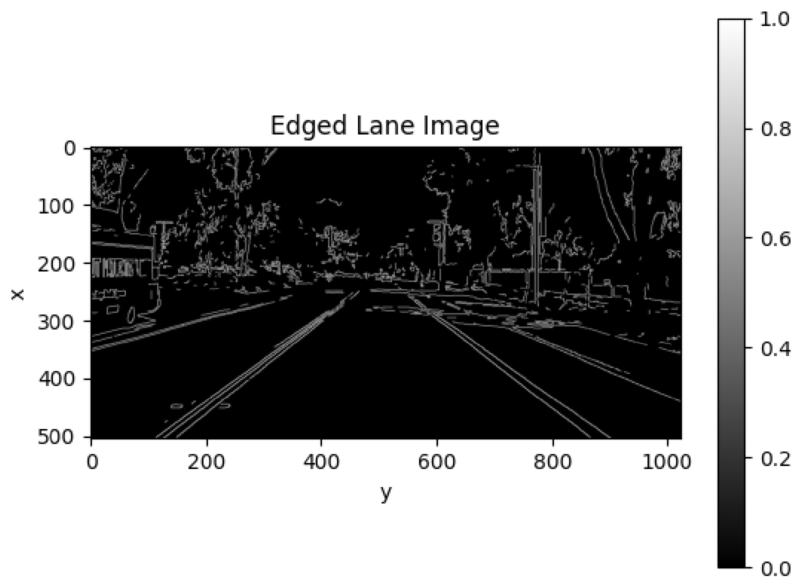


Lanes Image:

Original Image:



Binary Edge Image:



Hough Transform of Edge Image:

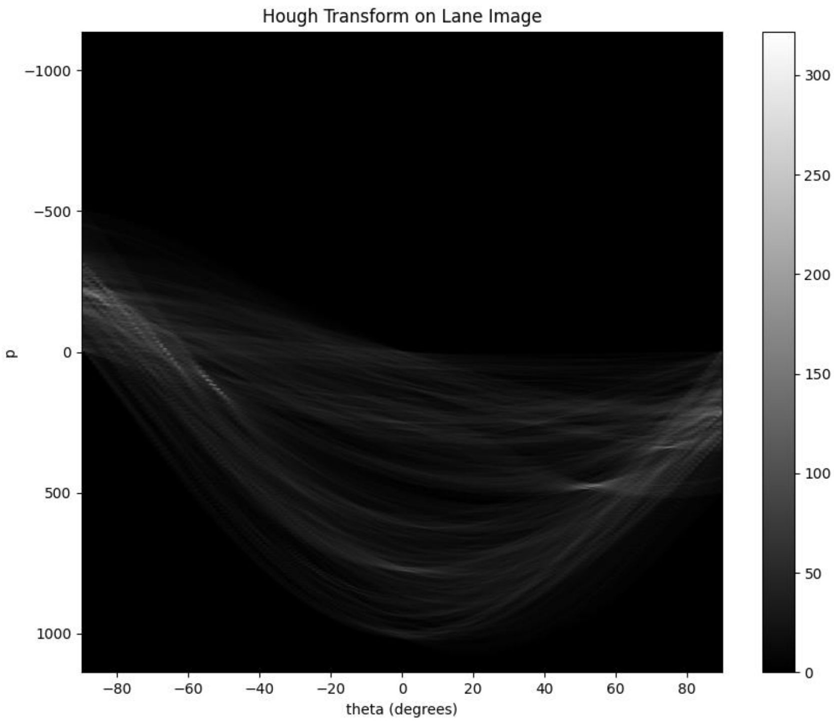
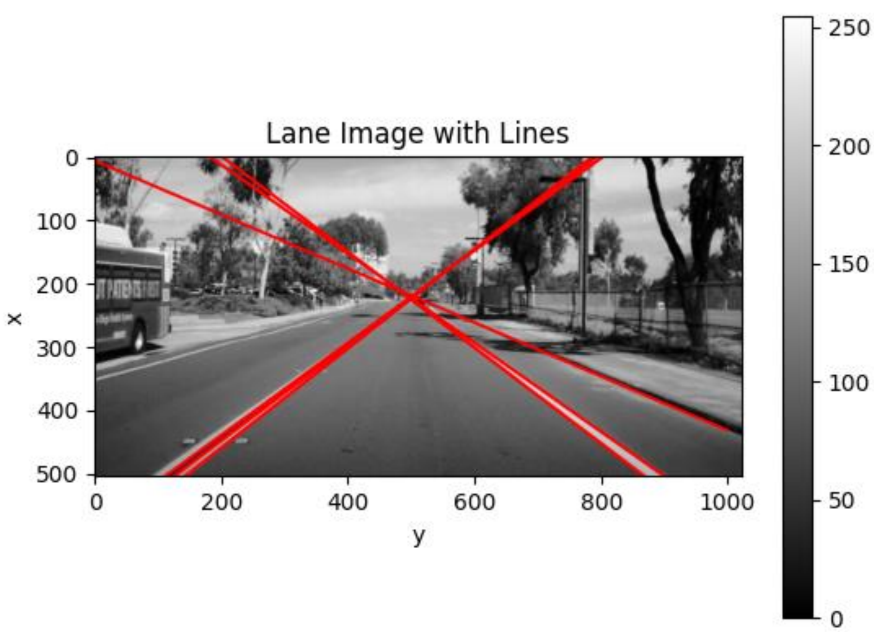


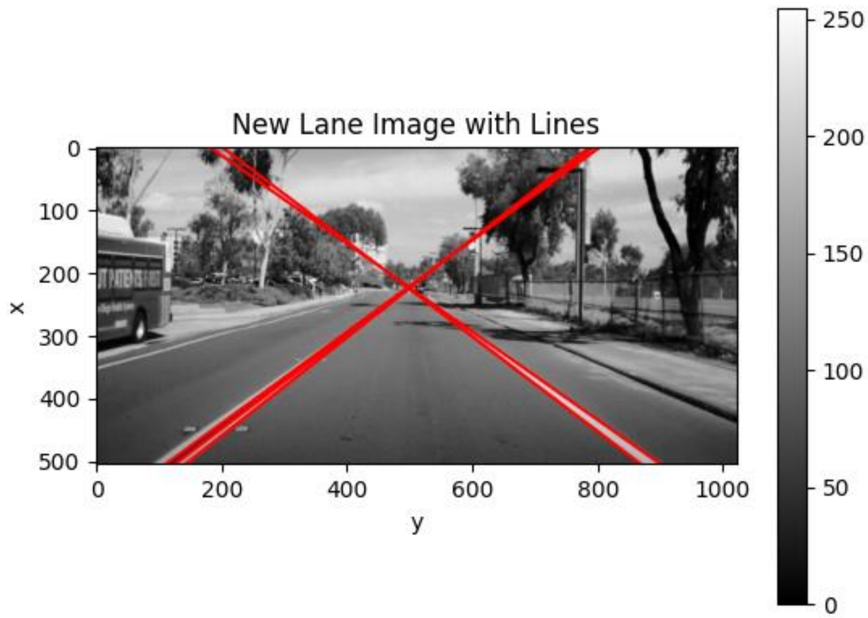
Image with Lines at 75% Threshold of HT:



Threshold values:

$$-57° \leq \theta \leq -50°$$
And
$$50° \leq \theta \leq 57°$$

Image with Lines at the Theta Threshold Above:



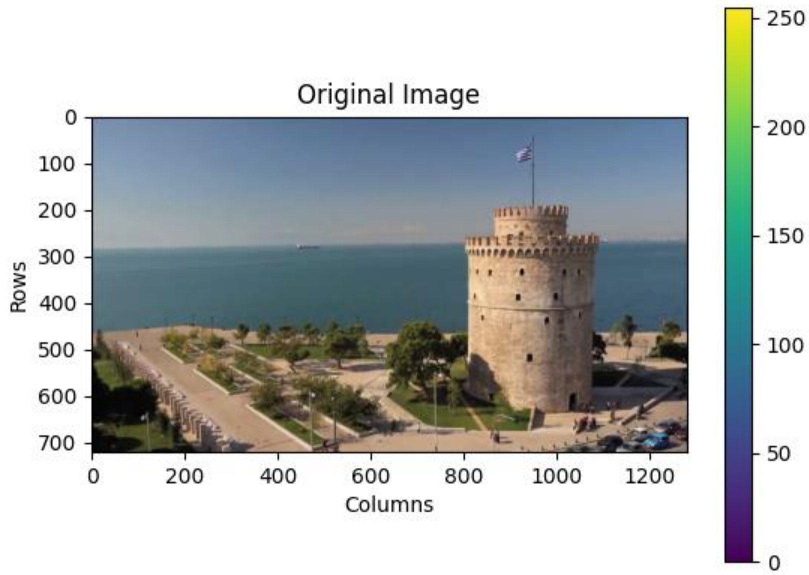
Problem 3:

Code:

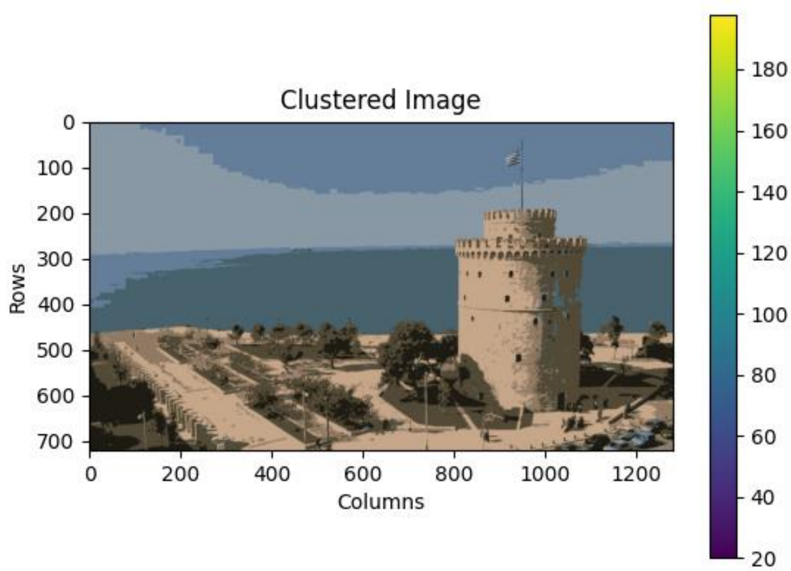
```python
#-----------------------------------------------
# Sean Carda
# ECE 253 - Image Processing
# Dr. Mohan Trivedi
# Homework 3
#-----------------------------------------------

#--------------------------------
# Global Imports.
import cv2
import numpy as np
import scipy.ndimage as ndi
import matplotlib.pyplot as plt
#--------------------------------
# Suppress scientific notation representation for numpy arrays.
np.set_printoptions(suppress=True)


# Method to create a dataset on which to cluster our data.
def createDataset(im):
    # Grab the size of the image.
    [N, M, D] = im.shape

    # Seperate the image into it's layered components.
    red = im[:, :, 2]
    green = im[:, :, 1]
    blue = im[:, :, 0]

    # Reshape the vectors to N*M x 1.
    new_shape = (N * M, 1)
    red_v = np.reshape(red, new_shape)
    green_v = np.reshape(green, new_shape)
    blue_v = np.reshape(blue, new_shape)

    # Concatonate the new vectors into a dataset matrix.
    features = np.zeros([N * M, 3])
    features[:, 0] = blue_v[:, 0]
    features[:, 1] = green_v[:, 0]
    features[:, 2] = red_v[:, 0]

    # Return.
    return features


# Method to perform K-means segmentation on a given dataset.
def kMeansCluster(features, centers):
    # Grab the length of the features and the length of the center matrices.
    f_size = features.shape
    q = f_size[0]
    m_size = centers.shape
    k = m_size[0]

    # Instantiate a new cluster matrix which will store the pixels matching the center.
    clusters = {}
    difference = {}
    for i in range(k):
        clusters[i] = None
        difference[i] = None

    # Instantiate a set of new k-mean centers.
    new_centers = centers

    # Perform the k-means clustering by looping through the dataset and comparing
    # pixels with k-mean centers.
    iterations = 70
    for iter in range(iterations):
        clusters = {}
        difference = {}
        for j in range(k):
            clusters[j] = None
            difference[j] = None

        # Compute the difference between the features and the centers.
        for j in range(k):
            difference[j] = (np.linalg.norm(features - new_centers[j, :], axis=1))**2

        # Generate a matrix from the calculated norms.
        diff_mat = np.zeros([q, k])
        for j in range(k):
            diff_mat[:, j] = difference[j]

        # Compute the argmin of the rows of the difference matrix.
        indices = np.argmin(diff_mat, axis=1)

        # For the calculated indices, generate an array of the indices for a specified minimum
        # index. Then, using those indices, find the features in those locations.
        for j in range(k):
            index_list = np.argwhere(indices == j)
            clusters[j] = np.array(features[index_list, :])
            if clusters[j].size != 0:
                new_centers[j, :] = np.mean(clusters[j], axis=0)
            else:
                new_centers[j, :] = np.random.randint(0, 255, size=(1, 3))

        print(new_centers)
```

```python

    # Return the new calculated centers and clusters.
    return [clusters, new_centers, indices]


# Method to map values in the old array to values calculated by k-means clustering.
def mapValues(im, idx, centers):
    # Grab the various dimensions of the given objects.
    [N, M, D] = im.shape
    k_size = centers.shape
    k = k_size[0]

    # Replace the values of the old image with the k-mean centers. For every k,
    # find the values in the idx array that match the current minimum index. Find the
    # center that matches this index and replace the rows of the output with this value.
    output = np.zeros([N * M, 3])
    for i in range(k):
        index_list = np.argwhere(idx == i)
        if index_list.size != 0:
            output[index_list, :] = centers[i, :]

    # Extract color components.
    red = np.reshape(output[:, 0], (N, M))
    green = np.reshape(output[:, 1], (N, M))
    blue = np.reshape(output[:, 2], (N, M))

    # Generate the image from the extracted color components.
    new_im = np.zeros(im.shape)
    new_im[:, :, 0] = blue
    new_im[:, :, 1] = green
    new_im[:, :, 2] = red

    # Return the image.
    return new_im


# Method to generate a k x 3 set of random k-mean centers.
def generateMeans(k):
    # Instantiate a random set of k means (centers).
    return np.random.randint(20, 200, size=(k, 3))


# Read in an image.
file_path = r"white-tower.png"
#A = cv2.cvtColor(cv2.imread(file_path), cv2.COLOR_BGR2RGB)
A = cv2.imread(file_path)
size = A.shape

# Generate the features of A.
print('Generating the image features...')
features = createDataset(np.int64(A))
print('Done!')

# Generate a random set of k-mean centers.
k = 7
print('\nGenerating Random k-means...')
k_means = generateMeans(k)
print(k_means)
print('Done!')

# Compute the k-means cluster.
print('\nCalculating k-mean clusters...')
[clusters, centers, idx] = kMeansCluster(features, k_means)
print('Done!')

# Display the k-mean centers.
print('K-mean Centers Produced After Clustering')
print(centers)

# Map the old image values to the new k-mean centers.
print('\nGenerating the new image...')
clustered_image = mapValues(A, idx, centers)
print('Done!')
print('')

# Show the two images.
plt.figure(1)
plt.imshow(cv2.cvtColor(np.uint8(A), cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.xlabel('Columns')
plt.ylabel('Rows')
plt.colorbar()

plt.figure(2)
plt.imshow(np.uint8(clustered_image))
plt.title('Clustered Image')
plt.xlabel('Columns')
plt.ylabel('Rows')
plt.colorbar()
plt.show()

```

Output:
Input Image:



Segmented Image:



K-mean Centers:

```
K-mean Centers Produced After Clustering
[[153 125 100]
 [100 122 145]
 [165 152 137]
 [109  98  72]
 [138 167 198]
 [ 20  27  30]
 [ 54  74  82]]
```

Problem 4:

Output and Questions:

1:

```
        #-----------------------------------------------------
        # IMPLEMENTED CONV BLOCK 2
        self.conv_block2 = nn.Sequential(
            nn.Conv2d(c1, c2, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(c2, c2, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, stride=2, ceil_mode=True),
        )
        #-----------------------------------------------------


        self.conv_block3 = nn.Sequential(
            nn.Conv2d(c2, c3, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(c3, c3, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(c3, c3, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, stride=2, ceil_mode=True),
        )

        #-----------------------------------------------------
        # IMPLEMENTED CONV BLOCK 4
        self.conv_block4 = nn.Sequential(
            nn.Conv2d(c3, c4, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(c4, c4, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(c4, c4, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, stride=2, ceil_mode=True),
        )
        #-----------------------------------------------------
```

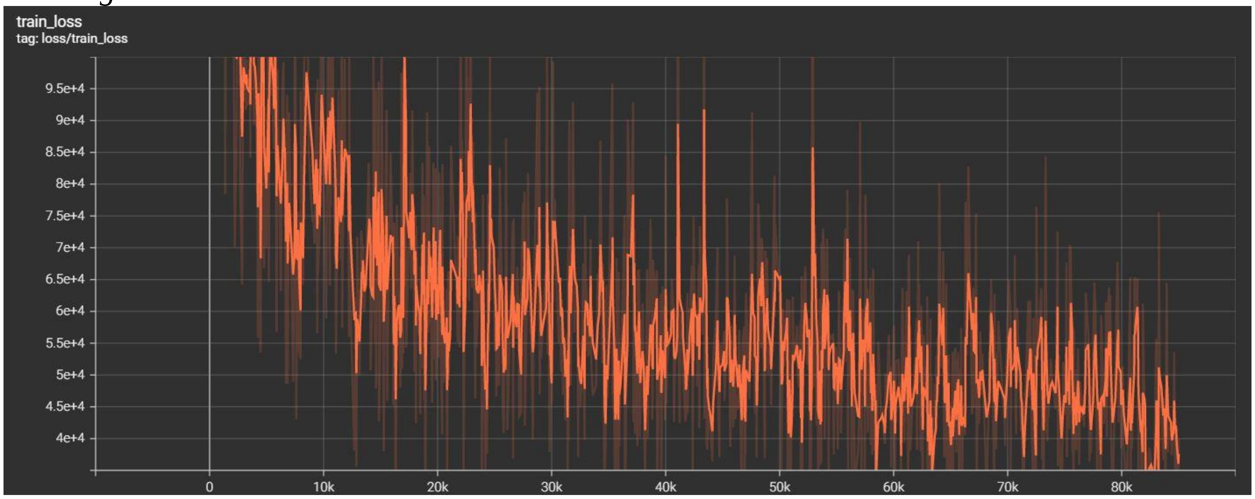
Above is the code that was written in order to complete the network model. Here, the structure of the network is a series of convolutional blocks defined as a series of convolutions followed by activations. At the end of that sequence of convolutions and activations, the activation layers are pooled.

2:

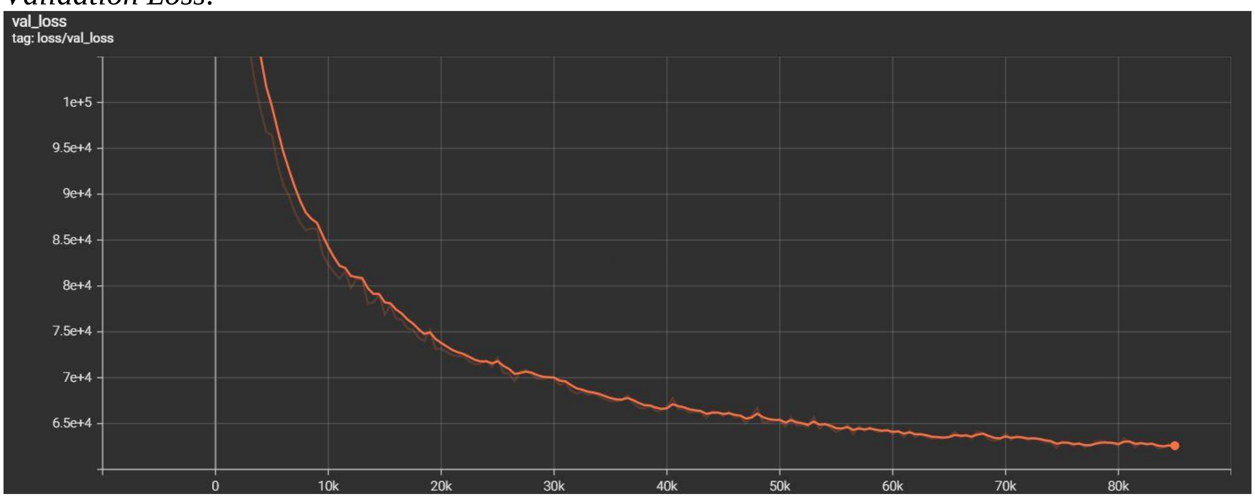
Here, we are **not** using predefined weights for our model, and we are training the model ourselves. This appears to be the case due to the structure of the *utils.py* script, in the "get_upsampling_weight" method. In this method, we can see that new weights are being calculated from incoming data.

3:

Training Loss:



Validation Loss:



4:

The metrics used by the original paper are

1. *Pixel accuracy*
2. *Mean pixel accuracy*
3. *Mean IU*
4. *Frequency-weighted IU*

Below are the results of validating the model after training:

```
Overall Acc:     0.9127122486125301
Mean Acc :       0.6298571089239537
FreqW Acc :      0.847739033039004
Mean IoU :       0.5393143651087131
0 0.9548651652445043
1 0.677273897919775
2 0.845158462670853
3 0.414655117508439
4 0.3486531279763397
5 0.24138339127602687
6 0.3013780707010186
7 0.4027415409241566
8 0.8585626209329092
9 0.5291734553865135
10 0.8817832670736081
11 0.5433616143546321
12 0.21213563688292777
13 0.8510924409864606
14 0.500840784996439
15 0.48907569632830455
16 0.4194971818451498
17 0.22319979157063552
18 0.5521416724868552
```

Here, we are measuring the

1. *Overall accuracy*
2. *Mean accuracy*
3. *Frequency-weighted accuracy*
4. *Mean IoU*

Stronger vs Weaker Classes	
IoU Above 0.5	*IoU Below 0.5*
Road	Wall
Sidewalk	Fence
Building	Pole
Vegetation	Traffic Light
Terrain	Traffic Sign
Sky	Rider
Person	Bus
Car	Train
Truck	Motorcycle
Bicycle	

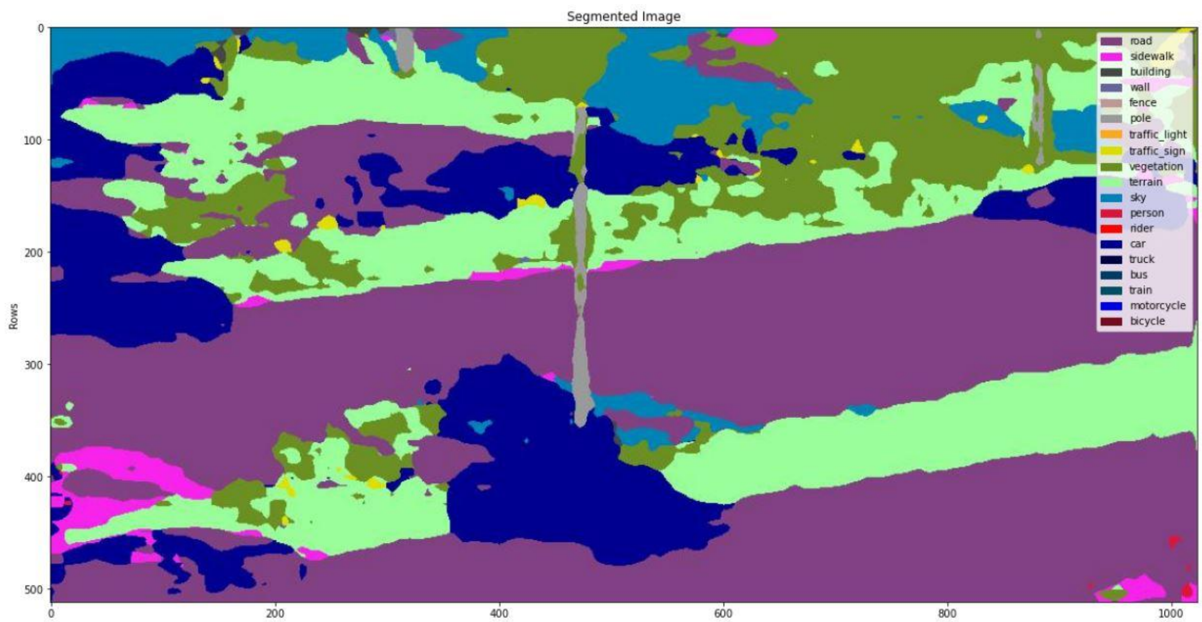
6:

Code:

```
#-------------------------------
# Sean Carda
# ECE 253 - Image Processing
# Homework 4 - Problem 4.6
# 12/8/21
#-------------------------------
import cv2
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as clr
import matplotlib.patches as ptch

# Function to plot the segmented image.
def plot(im, labels, item_list, legend_on, fignum, title):
    plt.figure(fignum, figsize=(20, 50))
    plt.imshow(cv2.cvtColor(im, cv2.COLOR_BGR2RGB))
    plt.title(title)
    plt.xlabel('Columns')
    plt.ylabel('Rows')
    if legend_on:
        plt.legend(handles=item_list)

# Read in the original image.
A_original = cv2.imread('test_image.jpg')
A_segment = cv2.imread('test_segment.jpg')

# List the colors used for segmentation.
colors = [  # [  0,   0,   0],
    [128, 64, 128],
    [244, 35, 232],
    [70, 70, 70],
    [102, 102, 156],
    [190, 153, 153],
    [153, 153, 153],
    [250, 170, 30],
    [220, 220, 0],
    [107, 142, 35],
    [152, 251, 152],
    [0, 130, 180],
    [220, 20, 60],
    [255, 0, 0],
    [0, 0, 142],
    [0, 0, 70],
    [0, 60, 100],
    [0, 80, 100],
    [0, 0, 230],
    [119, 11, 32],
]

# List the labels.
class_names = [
    "road",
    "sidewalk",
    "building",
    "wall",
    "fence",
    "pole",
    "traffic_light",
    "traffic_sign",
    "vegetation",
    "terrain",
    "sky",
    "person",
    "rider",
    "car",
    "truck",
    "bus",
    "train",
    "motorcycle",
    "bicycle",
]

item_list = []
for i in range(len(class_names)):
    current_color = colors[i]
    current_color_tuple = (current_color[0]/255, current_color[1]/255, current_color[2]/255)
    item_list.append(ptch.Patch(color=current_color_tuple, label=class_names[i]))


plot(A_original, None, None, False, 1, 'Original Image')
plot(A_segment, class_names, item_list, True, 2, 'Segmented Image')
plt.show()
```

Output:
Original Test Image:

Segmented Classified Image:



It is difficult to determine whether or not this output looks acceptable. If we were only interested in obtaining the identity of the regions in the original image or count how many of any one object appears in the original image, then this could be sufficient. However, the segmentation seems to have a difficult time connecting the same object for a test image. For example, some of the cars are not complete, and some have "road" identified inside of them, which is clearly not the case in the original. The classifier also seems to have some difficulty in correctly identifying certain objects, such as sky (which is not contained in the original, but the model has identified it anyway). This is interesting as “sky” was one of the labels that performed very well.

7:

Something that could have an effect, that is slightly unrelated to the actual convolutional network is perhaps segmenting the images beforehand with something like k-means clustering. With all the dynamic objects that occur in an image, having to learn so many different variations on a local level could be complicated. We saw this during training – that even after 85k iterations, the losses of the images were still immense. Narrowing the color levels to something more recognizable could lessen the level of variety required. An adverse effect of this is that the number of objects that can be classified might be less depending on the value of k used (if using k-means clustering).

A simpler option could be to utilize fewer class labels. Using fewer labels might reduce the training error by giving the model less to identify. As we could see from the validation, there were several labels that were weakly identified by the model. If we threw those out, we might have a better time predicting the labels that were often identified by the model.

Problem 5:

Code:

```python
#------------------------------------------------
# Sean Carda
# ECE 253 - Image Processing
# Dr. Mohan Trivedi
# Homework 3
#------------------------------------------------

#--------------------------------
# Global Imports.
import cv2
import numpy as np
import scipy.ndimage as ndi
import matplotlib.pyplot as plt
#--------------------------------
# Suppress scientific notation representation for numpy arrays.
np.set_printoptions(suppress=True)

# ------------------------------------------------------------------------------------------
# This filter will load an image, perform canny edge operation on it, and replace the white
# pixels with colored pixels.
#-------------------------------------------------------------------------------------------
file_path = 'lane.png'
A = cv2.imread(file_path)

# Compute the canny edge of the image.
A_edge = 255 * cv2.Canny(A, 100, 200, L2gradient=True)
edge_size = A_edge.shape

# Layer the edged image.
output = np.zeros([edge_size[0], edge_size[1], 3])
output[:, :, 0] = A_edge
output[:, :, 1] = A_edge
output[:, :, 2] = A_edge

# For every columns, replace it with a colored pixel column.
for i in range(edge_size[1]):
    new_pixel = np.random.randint(1, 255, size=(1, 3))
    output[:, i, :] = np.where(output[:, i] > 0, new_pixel, 0)

# Show the image.
plt.figure(1)
plt.imshow(np.uint8(output))
plt.title('Colorful Lanes!')
plt.xlabel('Columns')
plt.ylabel('Rows')
plt.colorbar()
plt.show()
```

Output: