CS 558-A Computer Vision Assignment 4

Problem 1 and 3

Source Code

```
np.random.seed(42)
image = cv.imread("/content/white-tower.png")
image = cv.cvtColor(image, cv.COLOR_BGR2RGB)
image.shape
def centroidDistance(x1, x2):
  distance = np.sqrt(np.sum((x1 - x2)**2))
  return distance
# **KMeans**
---
flattenedImage = image.reshape((-1, 3))
flattenedImage = np.float32(flattenedImage)
K = 10
clusters = [[]] * K
noOfCoordinates, colorsSize = flattenedImage.shape
noOfCoordinates, colorsSize
randomXYCoordinates = np.random.choice(noOfCoordinates, K, replace=False)
randomXYCoordinates
centroids = [None] * K
for i, j in enumerate(randomXYCoordinates):
  centroids[i] = flattenedImage[j]
def formClusters(K, flattenedImage, centroids):
  clusters = [[] for _ in range(K)]
  for i, j in enumerate(flattenedImage):
    distances = [centroidDistance(j, centroid) for centroid in centroids]
    centroidIndex = np.argmin(distances)
    clusters[centroidIndex].append(i)
```

```
 # print(clusters)
 return clusters
def newCentroids(K, flattenedImage, clusters):
 centroids = np.zeros((K, colorsSize))
 for i, j in enumerate(clusters):
  if (len(j) == 0):
    centroids[i] = 31
  else:
   clusterAvg = np.mean(flattenedImage[j], axis=0)
   centroids[i] = clusterAvg
 return centroids
def convergence(K, oldCentroids, newCentroids):
 distances = [centroidDistance(oldCentroids[i], newCentroids[i]) for i in range(K)]
 return sum(distances) == 0
def Kmeans(K, flattenedImage, centroids, iterations=10):
 for i in range(iterations):
  clusters = formClusters(K, flattenedImage, centroids)
  oldCentroids = centroids
  centroids = newCentroids(K, flattenedImage, clusters) # New Centroids
  isConvergence = convergence(K, oldCentroids, centroids)
  if (isConvergence):
   break

 # assigning cluster labels
 clusterLabels = np.empty(noOfCoordinates)
 for i, cluster in enumerate(clusters):
  for k in cluster:
   clusterLabels[k] = i
 return clusterLabels, centroids, clusters
clusterLabels, centroids, clusters = Kmeans(K, flattenedImage, centroids)
labels = clusterLabels.astype(int)
```

```
flattenLabels = labels.flatten()
centroids = np.uint8(centroids)
resultedImage = centroids[flattenLabels.flatten()]
resultedImage = resultedImage.reshape(image.shape)
plt.imshow(resultedImage)
plt.show()

# **Pixel Classification**
---
trainingImage = cv.imread("/content/sky_train.jpg")
trainingImageMask = cv.imread("/content/sky_train_mask.jpg")
trainingImage = cv.cvtColor(trainingImage, cv.COLOR_BGR2RGB)
trainingImageMask = cv.cvtColor(trainingImageMask, cv.COLOR_BGR2RGB)
K = 10
trainingImageReshaped = trainingImage.reshape((-1, 3))
trainingImageReshaped
trainingImageMaskReshaped = trainingImageMask.reshape((-1, 3))
trainingImageMaskReshaped
sky = []
non_sky = []
for i, j in enumerate(trainingImageReshaped):
  rM, gM, bM = trainingImageMaskReshaped[i]
  if(rM == 255 and gM == 255 and bM == 255):
    sky.append(trainingImageReshaped[i])
  else:
    non_sky.append(trainingImageReshaped[i])
sky = np.asarray(sky)
sky.shape
non_sky = np.asarray(non_sky)
non_sky.shape
# Sky Kmeans
```

```
clusters = [[]] * K
noOfCoordinates, colorsSize = sky.shape
noOfCoordinates, colorsSize
randomXYCoordinates = np.random.choice(noOfCoordinates, K, replace=False)
randomXYCoordinates
centroids = [None] * K
for i, j in enumerate(randomXYCoordinates):
  centroids[i] = sky[j]
centroids
sky_visual_words = Kmeans(K, sky, centroids, 15)
sky_visual_words = sky_visual_words[1]
# Non-Sky Kmeans
clusters = [[]] * K
noOfCoordinates, colorsSize = non_sky.shape
noOfCoordinates, colorsSize
randomXYCoordinates = np.random.choice(noOfCoordinates, K, replace=False)
randomXYCoordinates
centroids = [None] * K
for i, j in enumerate(randomXYCoordinates):
  centroids[i] = non_sky[j]
centroids
non_sky_visual_words = Kmeans(K, non_sky, centroids, 15)
non_sky_visual_words = non_sky_visual_words[1]
testImage1 = cv.imread("/content/sky_test1.jpg")
testImage1 = cv.cvtColor(testImage1, cv.COLOR_BGR2RGB)
testImage1Reshaped = testImage1.reshape((-1, 3))
testImage1Reshaped.shape
def findSky(testImage, sky_visual_words, num):
  sky_Pixel = []
  non_sky_Pixel = []
  for i, j in enumerate(testImage):
    r, g, b = testImage[i]
    for k, l in enumerate(sky_visual_words):
      rSky, gSky, bSky = sky_visual_words[k]
      if(abs(r-rSky) < num and abs(g-gSky) < num and abs(b-bSky) < num):
        # print(i,": Sky")
        sky_Pixel.append(i)
      else:
        # print(i,": Non-Sky")
        non_sky_Pixel.append(i)

  for i, j in enumerate(testImage):
    for k in sky_Pixel:
      if (i == k):
        testImage[i][0] = 255
        testImage[i][1] = 0
        testImage[i][2] = 0
  return testImage
skyImageOutput = findSky(testImage1Reshaped, sky_visual_words, 30)
resultedImage = skyImageOutput.reshape(testImage1.shape)
plt.imshow(resultedImage)
plt.show
testImage2 = cv.imread("/content/sky_test2.jpg")
testImage2 = cv.cvtColor(testImage2, cv.COLOR_BGR2RGB)
testImage2Reshaped = testImage2.reshape((-1, 3))
testImage2Reshaped.shape
skyImageOutput = findSky(testImage2Reshaped, sky_visual_words, 27)
resultedImage = skyImageOutput.reshape(testImage2.shape)
plt.imshow(resultedImage)
plt.show
testImage3 = cv.imread("/content/sky_test3.jpg")
testImage3 = cv.cvtColor(testImage3, cv.COLOR_BGR2RGB)
```

```
testImage3Reshaped = testImage3.reshape((-1, 3))
testImage3Reshaped.shape
skyImageOutput = findSky(testImage3Reshaped, sky_visual_words, 13)
resultedImage = skyImageOutput.reshape(testImage3.shape)
plt.imshow(resultedImage)
plt.show
testImage4 = cv.imread("/content/sky_test4.jpg")
testImage4 = cv.cvtColor(testImage4, cv.COLOR_BGR2RGB)
testImage4Reshaped = testImage4.reshape((-1, 3))
testImage4Reshaped.shape
skyImageOutput = findSky(testImage4Reshaped, sky_visual_words, 75)
resultedImage = skyImageOutput.reshape(testImage4.shape)
plt.imshow(resultedImage)
plt.show
```

Explanation

For Kmeans I got a list of 10 random unique centroids to start with and created an empty array of clusters too. Then I called the Kmeans function, where I formed the clusters first based on the centroids. Then I saved those centroids as old centroids. I then found the centroids of the clusters by calculating the avg of each clusters. Then I compared these new centroids with the old ones to see for convergence and if there was one, I'll break the loop and assign the lastest clusters with clusterLabels. After doing that, I converted the labels to integers and flattened the array. Then using the latest centroids and flattened labels, I created the image and reshaped it to its original form and printed.

For Pixel Classification, I created the Mask Image in Photoshop and then compared it pixel by pixel with the training image and wherever the pixels were white(the color I used for sky in my mask), I added it to the sky training set and rest to non_sky training set. Changed them to arrays and ran Kmeans on them separately and got the best centroids. These centroids will be the 10 visual words that we need. Then I created a findSky function that compares the testing image with the sky_visual_words and if its approximately equal, and identifies it as sky or not_sky. Then it colors all the sky pixels with RED color.

Output

Kmeans

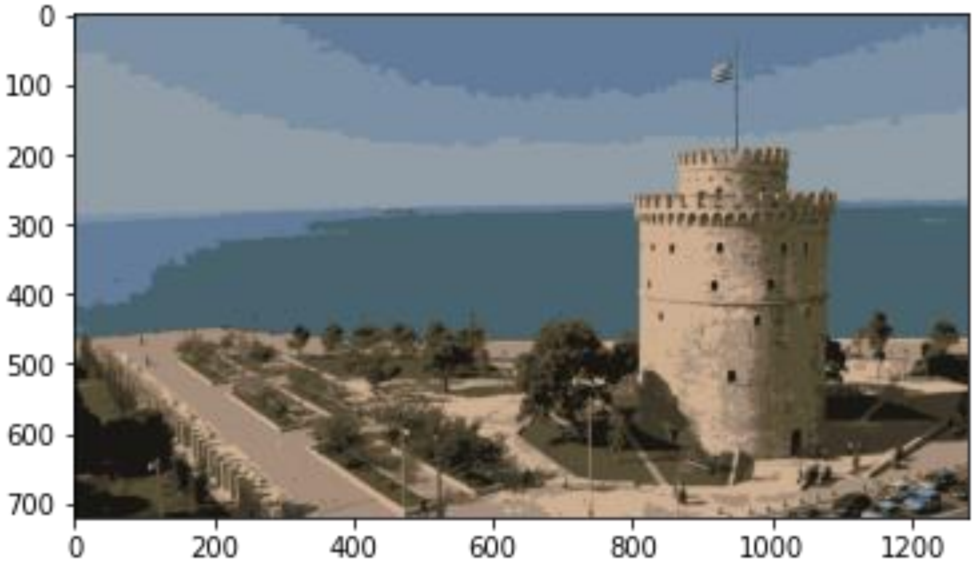
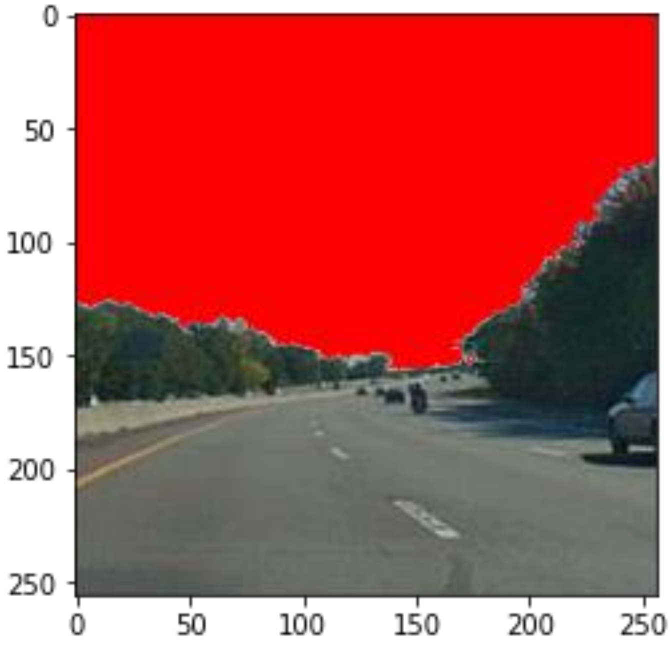
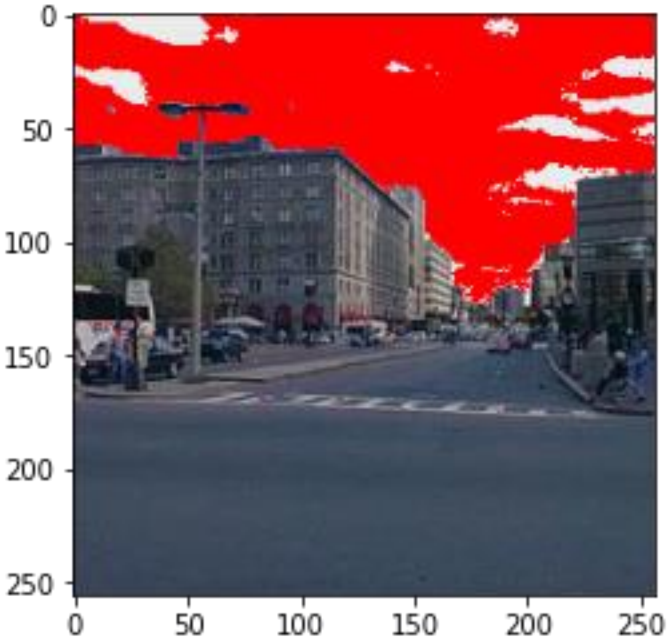


Image Segmentation

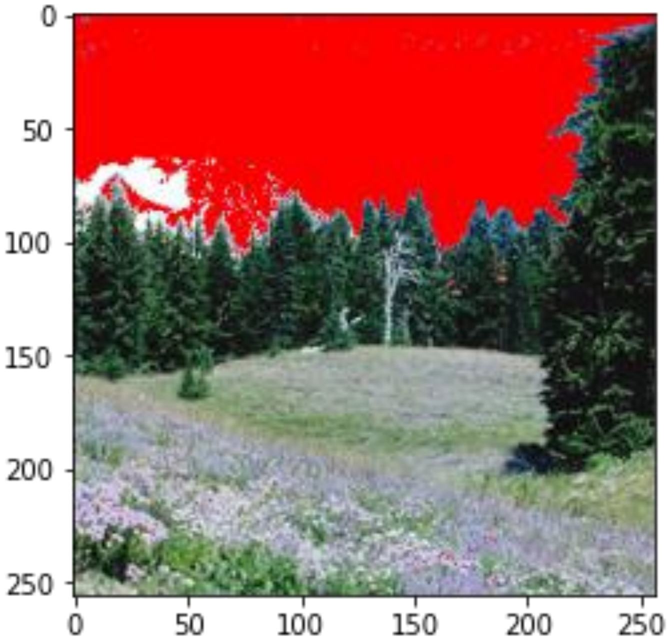
Test1



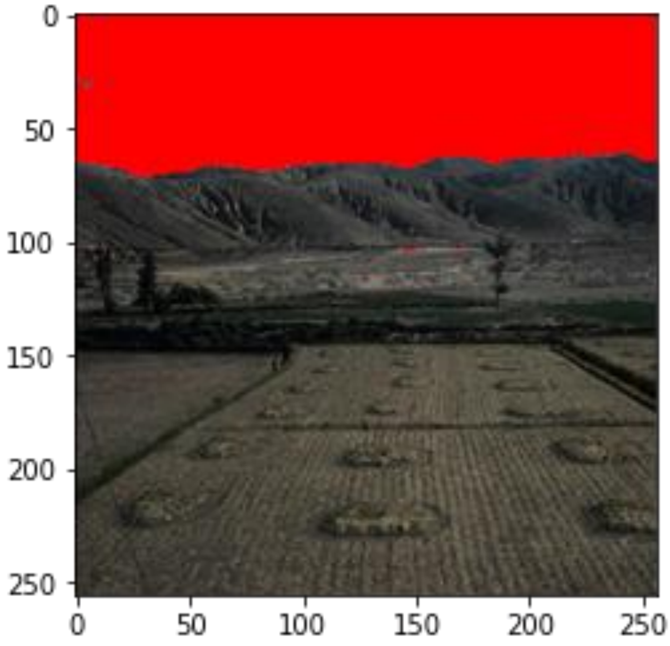
Test2


0
50
100
150
200
250
0
50
100
150
200
250

Test3

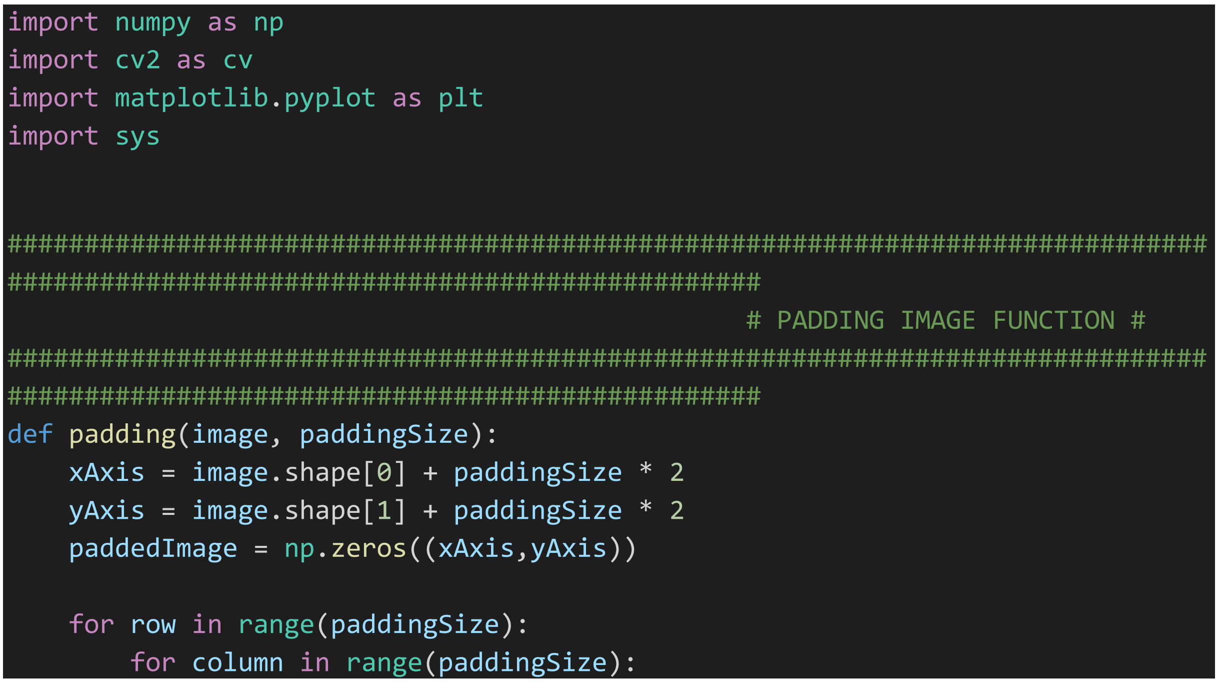

0
50
100
150
200
250
0
50
100
150
200
250

Test4



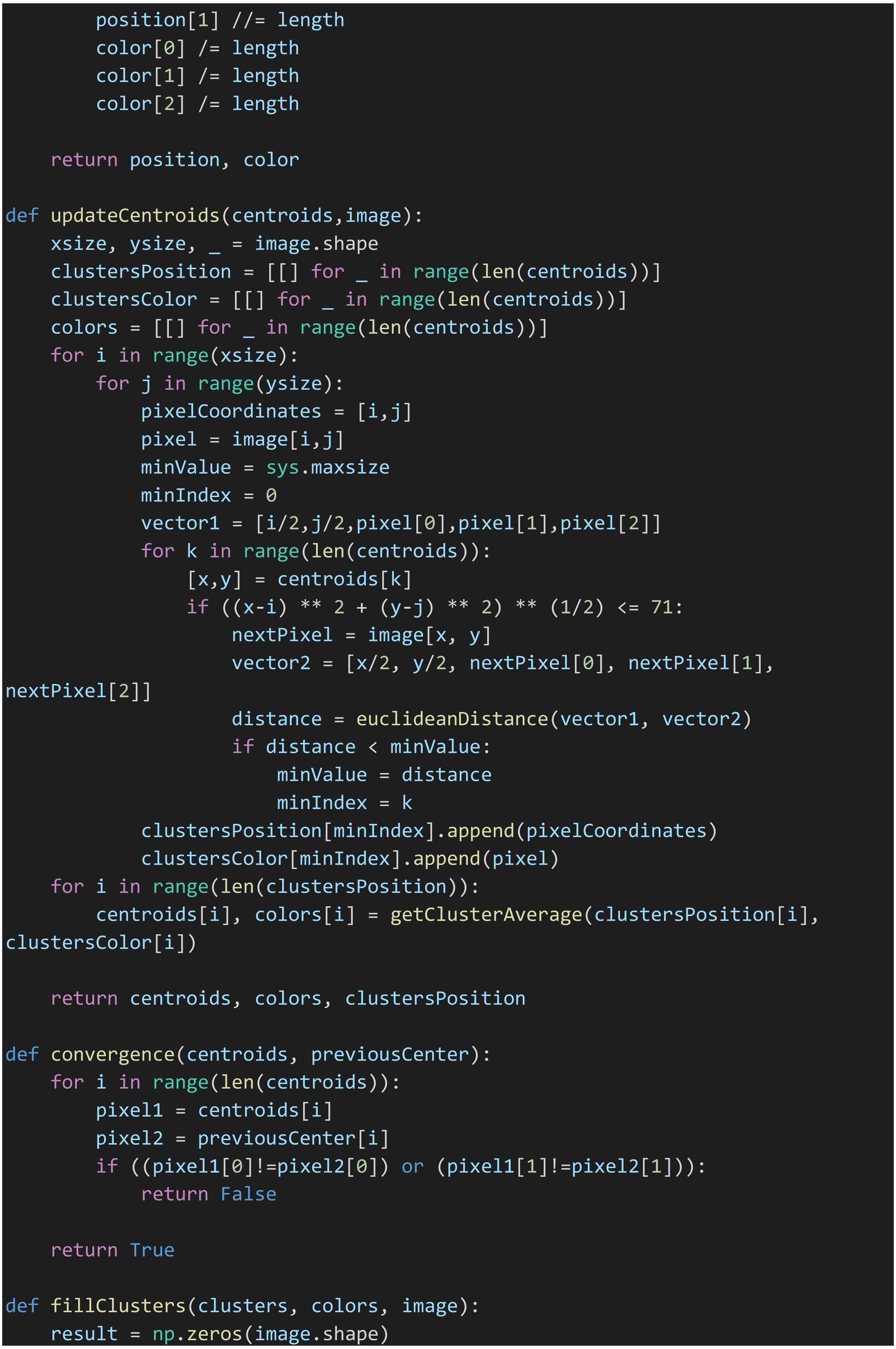
Problem 2

Source code

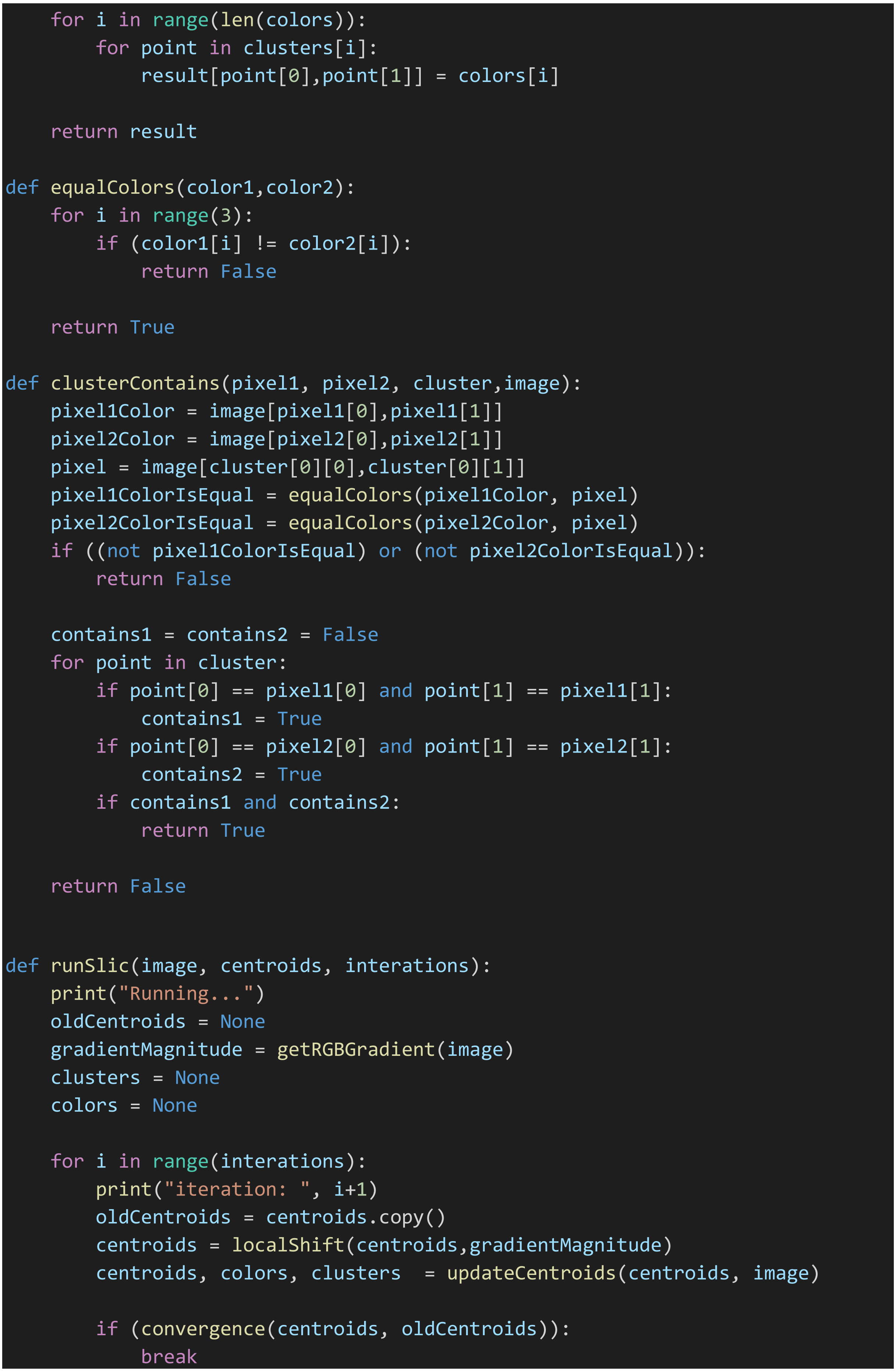


```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
import sys


##############################################################################
#################################################
                                                # PADDING IMAGE FUNCTION #
##############################################################################
#################################################
def padding(image, paddingSize):
    xAxis = image.shape[0] + paddingSize * 2
    yAxis = image.shape[1] + paddingSize * 2
    paddedImage = np.zeros((xAxis,yAxis))

    for row in range(paddingSize):
        for column in range(paddingSize):
            paddedImage[row][column] = image[0][0]
            paddedImage[row+xAxis-paddingSize][column] = image[image.shape[0]-
1][0]
            paddedImage[row+xAxis-paddingSize][column+yAxis-paddingSize] =
image[image.shape[0]-1][image.shape[1]-1]
            paddedImage[row][column+yAxis-paddingSize] =
image[0][image.shape[1]-1]

    for row in range(image.shape[0]):
        paddedImage[row+paddingSize][0:paddingSize] = image[row][0]
        paddedImage[row+paddingSize][paddingSize:yAxis-paddingSize] =
image[row]
        paddedImage[row+paddingSize][yAxis-paddingSize:yAxis] =
image[row][image.shape[1]-1]
    paddedImage = paddedImage.T

    for row in range(image.shape[1]):
        paddedImage[row+paddingSize][0:paddingSize] = image[0][row]
        paddedImage[row+paddingSize][xAxis-paddingSize-1:xAxis] =
image[image.shape[0]-1][row]
    paddedImage = paddedImage.T
    return paddedImage


def removePadding(image, paddingSize):
    xAxis = image.shape[0] - paddingSize * 2
    yAxis = image.shape[1] - paddingSize * 2
    nonPaddedImage = np.zeros((xAxis, yAxis))
    for row in range(xAxis):
        nonPaddedImage[row] = image[row +
paddingSize][paddingSize:paddingSize+yAxis]
    return nonPaddedImage

##############################################################################
#################################################
                                                        # CONVOLUTION #
##############################################################################
#################################################
def sobelConvolution(image, kernel):
    paddingXAxis = kernel.shape[0]//2
    paddingYAxis = kernel.shape[1]//2
    image = padding(image, paddingXAxis)
    xAxis, yAxis = image.shape
    paddedImage = np.copy(image)
    for row in range(paddingXAxis,xAxis-paddingXAxis):
        for column in range(paddingYAxis,yAxis-paddingYAxis):
            total = 0
            for i in range(-1 *  paddingXAxis, paddingXAxis + 1):
                for j in range(-1 * paddingYAxis , paddingYAxis + 1):
```

```python
                    total += kernel[paddingXAxis + i][paddingYAxis + j] * image[row - i][column - j]
            paddedImage[row][column] = total
    image = removePadding(image,paddingXAxis)
    paddedImage = removePadding(paddedImage,paddingXAxis)
    return paddedImage


def gaussianConvolution(image, sigma):
    padding = sigma * 3
    gaussian = [(sigma ** -1) * (2 * np.pi) ** (-1/2) * np.exp((-1/2) * (x/sigma)**2) for x in range(-1 * padding, padding+1)]
    gaussian = np.outer(gaussian, gaussian)
    image = sobelConvolution(image, gaussian)
    return image


###############################################################################################################################
                                                        # GRADIENT #
###############################################################################################################################
def gradientInfo(xGradient, yGradient, threshold):
    xAxis, yAxis = xGradient.shape
    magnitude = np.zeros(xGradient.shape)
    direction = np.zeros(xGradient.shape)
    for i in range(xAxis):
        for j in range(yAxis):
            distance =  (xGradient[i][j] ** 2 + yGradient[i][j] ** 2) ** .5
            if (distance > threshold):
                magnitude[i][j] = distance
    direction = np.arctan2(yGradient,xGradient) * 180 / np.pi
    return magnitude,direction

def maxValue(ret, magnitude, row, col, x, y):
    if ((magnitude[row + x][col + y] > magnitude[row][col]) or (magnitude[row - x][col -y ] > magnitude[row][col])):
        ret[row][col] = 0
    else:
        ret[row][col] = magnitude[row][col]

def nonMaxSuppression(magnitude, direction):
    padding(magnitude, 1)
    xAxis, yAxis = magnitude.shape
    ret = np.zeros(magnitude.shape)
    for row in range(1, xAxis-1):
        for col in range(1, yAxis-1):
            gradientDirection = direction[row][col]
            if ((gradientDirection > -22.5) and (gradientDirection <= 22.5) or
(gradientDirection > 157.5) and (gradientDirection <= -157.5)):
                maxValue(ret, magnitude, row, col, 1, 0)
            elif ((gradientDirection > 22.5) and (gradientDirection <= 67.5)
or (gradientDirection > -157.5) and (gradientDirection <= -112.5)):
                maxValue(ret, magnitude, row, col, 1, 1)
            elif ((gradientDirection > 67.5) and (gradientDirection < 112.5)
or (gradientDirection > -112.5) and (gradientDirection < -67.5)):
                maxValue(ret, magnitude, row, col, 1, 0)
            else:
                maxValue(ret, magnitude, row, col, 1, -1)
    removePadding(ret, 1)
    removePadding(magnitude, 1)
    return ret



##############################################################################
#################################################
                                                    # SLIC FUNCTIONS #
##############################################################################
#################################################
# DIVIDING THE IMAGE INTO 50x50 and initialize the centroids at the center of
the block
def divideAndInitializeCentroids(image):
    centroids = []
    xAxis, yAxis, colorChannels = image.shape
    xAxis = xAxis // 50
    yAxis = yAxis // 50
    for i in range(xAxis):
        for j in range(yAxis):
            x = 25 + i * 50
            y = 25 + j * 50
            centroids.append([x,y])
    return centroids

def getRGBGradient(image):
    # get all color channels
    colorChannels = [image[:,:,0],image[:,:,1],image[:,:,2]]
    xAxisKernal = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    yAxisKernal = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
    magnitudeArray = []
    for image in colorChannels:
        # gradient magnitude for each channel
        xgradient = sobelConvolution(image, xAxisKernal)
        ygradient = sobelConvolution(image, yAxisKernal)
        currentMag , _ = gradientInfo(xgradient,ygradient,0)
        magnitudeArray.append(currentMag)
    # total magnitude
    magnitude = (magnitudeArray[0] ** 2 + magnitudeArray[1] ** 2 +
magnitudeArray[2] ** 2) ** (1/2)
    return magnitude

# helper for local shift, find the smallest value in 3x3
def findMinIndex(chunk):
    minValue = np.min(chunk)
    if chunk[1,1] == minValue:
        return [0,0]
    for i in range(-1,2):
        for j in range(-1,2):
            if chunk[i+1][j+1] == minValue:
                return [i,j]
    return [0,0]

def localShift(centroids,magnitude):
    for i in range(len(centroids)):
        [x, y] = centroids[i]
        if ((x!=0) and (x!=len(magnitude)) and (y!=0) and
(y!=len(magnitude[0]))):
            chunk = magnitude[x-1:x+2, y-1:y+2]
            [shiftx, shifty] = findMinIndex(chunk)
            centroids[i] = [x + shiftx, y + shifty]

    return centroids

def euclideanDistance(vector1, vector2):
    total = 0
    for i in range(len(vector1)):
        total += (vector1[i] - vector2[i]) ** 2

    total = np.sqrt(total)
    return total

def getClusterAverage(clustersPosition, clustersColor):
    position = [0,0]
    color = [0,0,0]
    length = len(clustersPosition)
    if length != 0:
        for i in range(len(clustersPosition)):
            pixelIndex = clustersPosition[i]
            position[0] += pixelIndex[0]
            position[1] += pixelIndex[1]
            pixelColor = clustersColor[i]
            color[0] += pixelColor[0]
            color[1] += pixelColor[1]
            color[2] += pixelColor[2]
        position[0] //= length
```



```python
        position[1] //= length
        color[0] /= length
        color[1] /= length
        color[2] /= length

    return position, color

def updateCentroids(centroids,image):
    xsize, ysize, _ = image.shape
    clustersPosition = [[] for _ in range(len(centroids))]
    clustersColor = [[] for _ in range(len(centroids))]
    colors = [[] for _ in range(len(centroids))]
    for i in range(xsize):
        for j in range(ysize):
            pixelCoordinates = [i,j]
            pixel = image[i,j]
            minValue = sys.maxsize
            minIndex = 0
            vector1 = [i/2,j/2,pixel[0],pixel[1],pixel[2]]
            for k in range(len(centroids)):
                [x,y] = centroids[k]
                if ((x-i) ** 2 + (y-j) ** 2) ** (1/2) <= 71:
                    nextPixel = image[x, y]
                    vector2 = [x/2, y/2, nextPixel[0], nextPixel[1],
nextPixel[2]]
                    distance = euclideanDistance(vector1, vector2)
                    if distance < minValue:
                        minValue = distance
                        minIndex = k
            clustersPosition[minIndex].append(pixelCoordinates)
            clustersColor[minIndex].append(pixel)
    for i in range(len(clustersPosition)):
        centroids[i], colors[i] = getClusterAverage(clustersPosition[i],
clustersColor[i])

    return centroids, colors, clustersPosition

def convergence(centroids, previousCenter):
    for i in range(len(centroids)):
        pixel1 = centroids[i]
        pixel2 = previousCenter[i]
        if ((pixel1[0]!=pixel2[0]) or (pixel1[1]!=pixel2[1])):
            return False

    return True

def fillClusters(clusters, colors, image):
    result = np.zeros(image.shape)
```

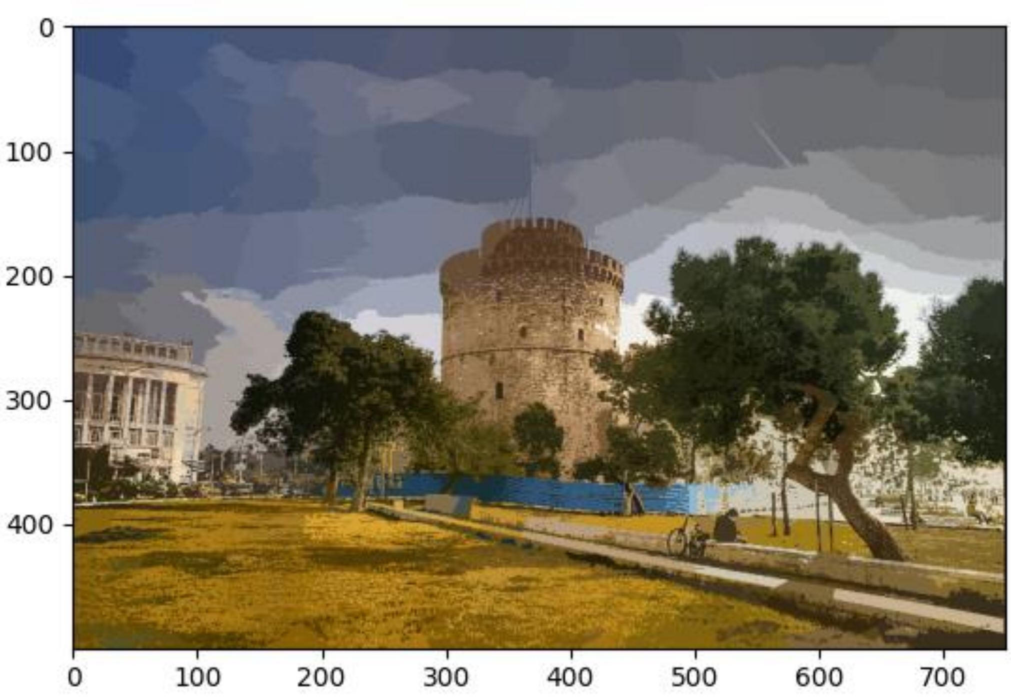


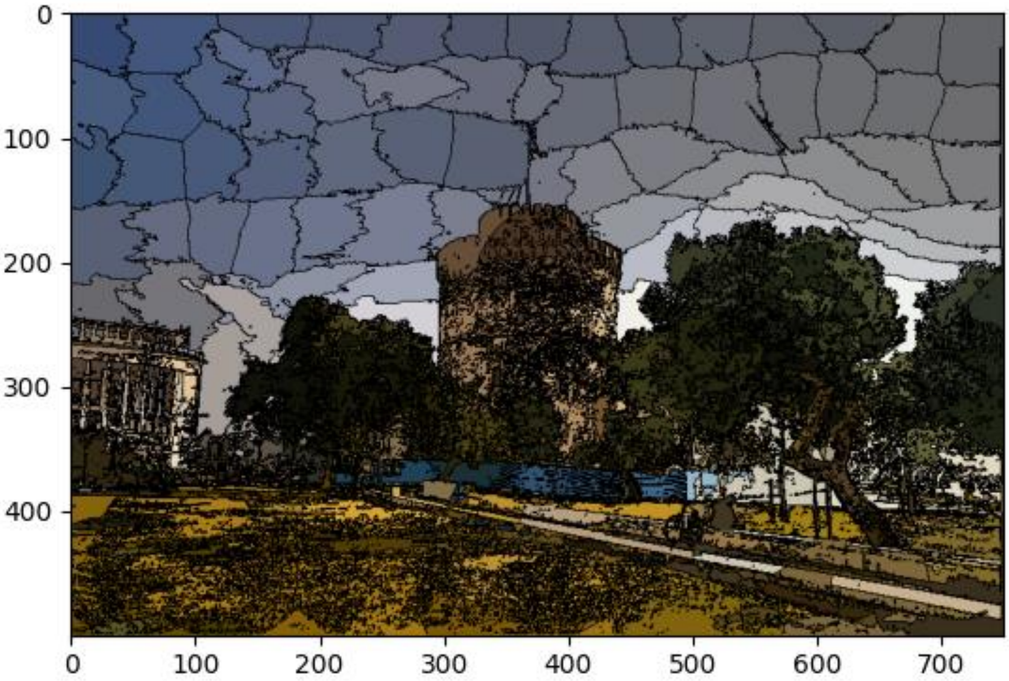
```python
    for i in range(len(colors)):
        for point in clusters[i]:
            result[point[0],point[1]] = colors[i]

    return result

def equalColors(color1,color2):
    for i in range(3):
        if (color1[i] != color2[i]):
            return False

    return True

def clusterContains(pixel1, pixel2, cluster,image):
    pixel1Color = image[pixel1[0],pixel1[1]]
    pixel2Color = image[pixel2[0],pixel2[1]]
    pixel = image[cluster[0][0],cluster[0][1]]
    pixel1ColorIsEqual = equalColors(pixel1Color, pixel)
    pixel2ColorIsEqual = equalColors(pixel2Color, pixel)
    if ((not pixel1ColorIsEqual) or (not pixel2ColorIsEqual)):
        return False

    contains1 = contains2 = False
    for point in cluster:
        if point[0] == pixel1[0] and point[1] == pixel1[1]:
            contains1 = True
        if point[0] == pixel2[0] and point[1] == pixel2[1]:
            contains2 = True
        if contains1 and contains2:
            return True

    return False


def runSlic(image, centroids, interations):
    print("Running...")
    oldCentroids = None
    gradientMagnitude = getRGBGradient(image)
    clusters = None
    colors = None

    for i in range(interations):
        print("iteration: ", i+1)
        oldCentroids = centroids.copy()
        centroids = localShift(centroids,gradientMagnitude)
        centroids, colors, clusters  = updateCentroids(centroids, image)

        if (convergence(centroids, oldCentroids)):
            break
        i +=  1
    print("Almost done...")
    slicImage = fillClusters(clusters, colors, image)

    return slicImage, clusters

def slicResultWithBorders(slicImage, clusters):
    xAxis, yAxis, colorChannels = slicImage.shape
    slicImageWithBorders = np.zeros(slicImage.shape)

    for i in clusters:
        for j in i:
            if ((j[0] < xAxis - 1) and (j[1] < yAxis - 1)):
                if clusterContains([j[0]+1, j[1]], [j[0], j[1]+1], i,
slicImage):
                    slicImageWithBorders[j[0], j[1]] = slicImage[j[0], j[1]]
                else:
                    slicImageWithBorders[j[0], j[1]] = [0, 0, 0]
            else:
                slicImageWithBorders[j[0], j[1]] = slicImage[j[0], j[1]]

    slicImageWithBorders /= 255

    return slicImageWithBorders


##############################################################################
#################################################
                                                    # MAIN FUNCTION #
##############################################################################
#################################################
image = cv.imread("wt_slic.png")
image = cv.cvtColor(image, cv.COLOR_BGR2RGB)
imageSlic = image.astype('float32')

centroids = divideAndInitializeCentroids(imageSlic)

slicResult, clusters = runSlic(imageSlic, centroids, 3)
plt.imshow(slicResult / 255)
plt.show()

slicResultsWithBorders = slicResultWithBorders(slicResult, clusters)
plt.imshow(slicResultsWithBorders)
plt.show()
print("Done!")
```

Explanation

Here, I created a function for splitting the image into 50x50 and assigned the centre of those blocks as the initial centroids. Then I passed those centroids and the image to the slic function, that returns the slic image without the borders and the latest clusters. After that I add borders to those clusters by passing the clusters and the slic image without the borders in to a function that adds borders to those clusters and returns that image. I then print that image.

Output




0
100
200
300
400
0
100
200
300
400
500
600
700