```py
# Apply k-means segmentation on white-tower.png with k=10.
# The distance function should only consider the RGB color channels and ignore pixel coordinates.
# Randomly pick 10 RGB triplets from the existing pixels as initial seeds and run to convergence.
# After k-means has converged, represent each cluster with the average RGB value of its members,
# creating an image as in slide 18 of Week 6.
from PIL import Image, ImageDraw
import numpy as np
import random
import math#imported libraries
# 1. Randomly initialize the cluster centers, c_1, ..., c_K
# 2. Given cluster centers, determine points in each cluster
# - For each point p, find the closest c_i. Put p into cluster i
# 3. Given points in each cluster, solve for c_i
# - Set c_i to be the mean of points in cluster i
# 4. If c_i have changed, repeat Step 2
im = Image.open("white-tower.png")
rgb_im = im.convert('RGB')#given data

def euclideanDist(pointA, pointB):
    # for each dimenstion(RGB) in A and B square the difference
    # finally take the square root of the sum of those values and return it
    d = 0.0
    for index in range(3):
        d += (float(pointA[index])-float(pointB[index]))**2
    d = math.sqrt(d)
    return d

def checker(listA, listB):
```

```
    # in this method I iterate through the elements of each list
    # if any are not equal I return False, otherwise I return True
    for i in range(len(listA)):
        if (listA[i] != listB[i]):
            return False
    return True


def kmeans(img0, k):
    # here the pix list contains the tuples containing each pixel's rgb values
    pix = []
    # I iterate through the given image (converted to a 2darray) and add the elements(RGB-values) to pix
    for row in np.array(img0):
        for cell in row:
            pix.append(list(cell))
    # here the cluster centers are randomly initialized from 10 random points
    centers = random.sample(pix, k)
    print(centers)
    # here I define clusters
    clusters = [[]]*k
    print(clusters)
    # this keeps track of the distance everything is from a given pixel
    dist = []
    # the index used later
    i = 0
    # these values are used between iterations in my loop
    avgR = 0
    avgG = 0
    avgB = 0
    # used to break out of the loop
```

```
converged = False
while(not converged):
    # here I set the old centers equal to our current ones
    oldcenters = []
    for center in centers:
        oldcenters.append(center.copy())
    ####this code block fills each cluster with the pixels closest to the working center
    # clear for our new clusters
    clusters = [[],[],[],[],[],[],[],[],[],[]]
    # iterate through pixels
    for pixel in pix:
        # clear dist
        dist = []
        # fill the dist with each point's distance from each cluster center
        for center in centers:
            dist.append(euclideanDist(pixel,center))
        # fill the clusters with the k pixels with the least distances from its center
        for index in range(k):
            if dist[index] == min(dist):
                clusters[index].append(pixel)
                break
    # print results of iteration
    for cluster in clusters:
        print(len(cluster),end=' ')
    print()
    print(centers)
    ####this code block updates the centers
    # here the mean rgb values of each cluster are found and set as the new centers
    i = 0
```

```
        for cluster in clusters:
            avgR = 0
            avgG = 0
            avgB = 0
            for point in cluster:
                avgR += point[0]
                avgG += point[1]
                avgB += point[2]
            centers[i][0] = avgR//len(cluster)
            centers[i][1] = avgG//len(cluster)
            centers[i][2] = avgB//len(cluster)
            i += 1
        # here we check if our new centers equal our old ones if so we exit the loop else we repeat
        converged = checker(oldcenters,centers)
    return (centers,clusters)

# calls my k-means function storing the clusters and centers
tup = kmeans(rgb_im,10)
cs = tup[0]
cl = tup[1]
print(cs)

####this code block gets the mean rgb values of each cluster storing them in t
def mean(clustr):
    m = [0,0,0]
    for i in range(3):
        for ele in clustr:
            m[i] += ele[i]
        m[i] = m[i]//len(clustr)
```

```
    return m
# calls mean with my clusters to get the rgb-values to be set in our new image
t = []
for n in range(10):
    t.append(mean(cl[n]))



####this code block will create a new image where the rgb values of each cluster are set to the cluster's avg
imf = Image.open("white-tower.png")
vals = imf.load()
# the dimensions of the image are stored here
width, height = im.size
distance = []
for x in range(width):
    for y in range(height):
        # here we get the distances between each center and our current pixel storing them in dist
        distance = []
        for c in cs:
            distance.append(euclideanDist(vals[x,y],c))
        # finally we find the closest cluster center
        # use that to determine the value in t containing the color
        # and set the pixel's color in our new image to that color in t
        for index in range(10):
            if distance[index] == min(distance):
                vals[x,y] = (t[index][0],t[index][1],t[index][2])
                break
        if(x%100==0 and y%100==0):
            print("%d,%d"%(x,y))
```

```
# here we save the image and are finished

imf.save("segmentation.png")

#Image.close(img1)
```

```py
# Problem 2: SLIC. (65 points) Apply a variant of the SLIC algorithm to wt_slic.png, by implementing
# the following steps:
# 1. Divide the image in blocks of 50x50 pixels and initialize a centroid at the center of each
# block.
# 2. Compute the magnitude of the gradient in each of the RGB channels and use the square root
# of the sum of squares of the three magnitudes as the combined gradient magnitude. Move
# the centroids to the position with the smallest gradient magnitude in 3x3 windows centered
# on the initial centroids.
# 3. Apply k-means in the 5D space of x,y,R,G,B. Use the Euclidean distance in this space,
# but divide x and y by 2.
# 4. After convergence, display the output image as in slide 41 of week 6: color pixels that touch
# two different clusters black and the remaining pixels by the average RGB value of their
# cluster.

from PIL import Image, ImageDraw
import numpy as np
import random
import math

im = Image.open("wt_slic.png")
rgb_im = im.convert('RGB')

height,width = im.size

def euclideanDist(pointA, pointB):
    #for each dimenstion(xyRGB) in A and B square the difference
    #finally take the square root of the sum of those values and return it
    d = 0.0
```

```
    for index in range(5):
        if(index<=1):
            d += (0.5*float(pointA[index])-0.5*float(pointB[index]))**2
        else:
            d += (float(pointA[index])-float(pointB[index]))**2
    d = math.sqrt(d)
    return d


def lowestGradPos(img1,xval,yval):
    ###this method finds the lowest gradient possible for the pixel in this image
    gradPos = []
    for i in range(xval-1,xval+2):
        for j in range(yval-1,yval+2):#so this is the 3x3 window.
            #magnitude of gradient in R
            mgR = (int(img1[i+1][j][0])-int(img1[i-1][j][0]))**2 + (int(img1[i][j+1][0])-int(img1[i][j-1][0]))**2
            #magnitude of gradient in G
            mgG = (int(img1[i+1][j][1])-int(img1[i-1][j][1]))**2 + (int(img1[i][j+1][1])-int(img1[i][j-1][1]))**2
            #magnitude of gradient in B
            mgB = (int(img1[i+1][j][2])-int(img1[i-1][j][2]))**2 + (int(img1[i][j+1][2])-int(img1[i][j-1][2]))**2
            gradPos.append((i,j,math.sqrt(mgR+mgG+mgB)))
    # here I am able to sort the list of tuples by the final element in ascending order and take the first
entry
    tup = sorted(gradPos, key=lambda e: e[2])[0]
    x,y = tup[0],tup[1]
    return (x,y,img1[x][y][0],img1[x][y][1],img1[x][y][2])# I return the x,y coordinates and rgb values of the
found pixel


# here I use the rgb gradients to get the initial cluster centers as requested
img0 = np.array(rgb_im)
```

```
CC_Is = []
# using these loops I start at position 25,25 and get the initial cluster center for every 50x50 segment
for x in range(25, width, 50):
    for y in range(25, height, 50):
        CC_Is.append((x,y,img0[x][y][0],img0[x][y][1],img0[x][y][2]))
CC_Is = [lowestGradPos(img0,c[0],c[1]) for c in CC_Is]
# here I create a grid the size of the image to store distances of each pixel to a given cluster center
# I set the highest distances to infinity and fill my array with that
di = np.ndarray((width,height))
highestPosDist = np.inf
for row in range(width):
    for col in range(height):
        di[row][col] = highestPosDist

# this is a simular array for the labels/clusters each pixel belongs to
# each pixel is initally at the cluster dubbed -1 (so None)
li = np.zeros((width,height))
for row in range(width):
    for col in range(height):
        li[row][col] = -1

###This method calculates the residual error by using the euclidean distance with respect to x and y between 2 centers
def residualError(oldc, newc):
    re = 0
    for a,b in zip(oldc,newc):
        for index in range(0,2):
            re +=abs(a[index]-b[index])
    return re/300
```

```
###This recursive function employs the bulk of my SLIC algo
def assignment(oldcenters, img1, iters, dis, lis):
    ###in this code block I define and initialize a list containing each cluster as a list of pixels
    ###I also update my arrays dis and lis
    clusters = []
    for index in range(len(oldcenters)):
        clusters.append([])
    num = 0
    for center in oldcenters:
        for xpos in range(max(0,int(center[0]-50)),min(int(center[0]+50),int(width))):
            for ypos in range(max(0,int(center[1]-50)),min(int(center[1]+50),int(height))):
                pix = (xpos,ypos,img1[xpos][ypos][0],img1[xpos][ypos][1],img1[xpos][ypos][2])
                D = euclideanDist(center,pix)
                if(D<=dis[xpos][ypos]):
                    dis[xpos][ypos] = D
                    lis[xpos][ypos] = num
                clusters[int(lis[xpos][ypos])].append(pix)
        num += 1
    num = 0



    print('iteration %d'%iters)
    ###In this code block I fill the new centers with the avg values of the pixels in its corresponding cluster
    newcenters = [(0,0,0,0,0)]*len(oldcenters)
    for cluster in clusters:
        for pixel in cluster:
            newcenters[num] = tuple([a+b/len(cluster) for a,b in zip(newcenters[num],pixel)])
        num += 1
```

```
    # I set err = to the residual error using my old and new cluster centers
    err = abs(residualError(oldcenters,newcenters))
    print(err)
    # Here I see if I have met the threshold/exceeded the limit of iterations
    # if not I recursively call this function, otherwise I return the centers clusters and labels
    if err > 0.01 and iters < 10:
        return assignment(newcenters, img1, iters+1, dis, lis)
    return (newcenters,clusters,lis)

# I call and store the results of my SLIC algo function
tup = assignment(CC_Is, img0, 0, di, li)
cs = tup[0]
cl = tup[1]
li = tup[2]

####this code block gets the mean rgb values of each cluster storing them in t
def mean(clustr):
    m = [0,0,0]
    for i in range(2,5):
        for ele in clustr:
            m[i-2] += ele[i]
        m[i-2] = m[i-2]//len(clustr)
    return m

# same block as in K-means
t = []
for n in cl:
    t.append(mean(n))
```

```
####this code block will create a new image where the rgb values of each cluster are set to the cluster's avg
imf = Image.open("wt_slic.png")
vals = imf.load()


# same as in last q
distance = []
num = 0
for l in cl:
    for ele in l:
        vals[ele[1],ele[0]] = (t[num][0],t[num][1],t[num][2])
    num+=1


# here is where I check if each pixel is an edge pixel or not by comparing it's label to it's adjacent pixels' labels
def check(grid, x, y):
    retval = 0
    if(x>0 and x<len(grid)-1):
        if(y>0 and y<len(grid[0])-1):
            for i in range(-1,2):
                for j in range(-1,2):
                    if not (grid[x+i][y+j] == grid[x][y]):
                        return True
            return False
    return True


print(li)


# I save what I had before outlining the area around clusters
```

```
imf.save("slic_segmentation2.png")


# Here I set the pixels that are on the edges of clusters to black using my function above
for xp in range(len(li)):
    for yp in range(len(li[xp])):
        if(check(li,xp,yp)):
            vals[yp,xp] = (0,0,0)

# finally I save my result
imf.save("slic_segmentation.png")
```

Problem 1:

For K-means I set the number of clusters, k, to ten. I initialized the cluster centers from a random sample of pixels in the image. I filled placed each point in the image in a cluster with the lowest Euclidean distance using RGB values to that pixel. I set my centers to be the mean of points in their respective clusters. I repeated this process until the clusters stopped changing. Finally for each cluster I found the mean RGB values and set the pixel values within that cluster to their corresponding means.

Problem 2:

I divided the image in blocks of 50x50 pixels and initialize a centroid at the center of each block. I computed the magnitude of the gradient in each of the RGB channels and used the square root of the sum of squares of the three magnitudes as the combined gradient magnitude. I Moved the centroids to the position with the smallest gradient magnitude in 3x3 windows centered on the initial centroids. I Applied k-means in the 5D space of x, y, R, G, B. Using the Euclidean distance in this space, but divided x and y by 2. When k-means converged, I saved the output image with pixels that touch two different clusters colored black and the remaining pixels by the average RGB value of their cluster like before.