Vincent Lee   I pledge my honor that I have abided by the stevens honor system.

Driver.py runs the program parameterized. It takes in the image you want to apply k means to, the number of centers k and the image you want to apply SLIC to.

```
driver.py
import os

if __name__ == "__main__":
    os.system("python main.py cs558s18_hw3_data/white-tower.png 10 cs558s18_hw3_data/wt_slic.png")
```

Main.py calls kmeans and slic to the parameters.

```
main.py
import imageio
import matplotlib.pyplot as plt
import sys

import kmeans
import slic

if __name__ == "__main__":
    if len(sys.argv) != 4:
        print('Useage: python main.py kmeansImage kCenters slicImage')
        exit(1)
    imageKmeans = imageio.imread(sys.argv[1]).astype('float32')
    k = int(sys.argv[2])
    imageSlic = imageio.imread(sys.argv[3]).astype('float32')
    kmeansResult = kmeans.kmeans(k,imageKmeans)
    plt.imshow(kmeansResult)
    plt.show()
    slicResult = slic.slic(imageSlic)
    plt.imshow(slicResult)
    plt.show()
```

Kmeans.py is where all the kmeans computation is done. kmeans, finds the centers and the clusters for each corresponding center and fills the clusters with the center value which is the average of the cluster. I divide by 255 because pyplot wants values 0 to 1

```
def kmeans(k,image):
    print('Start Kmeans')
    centers, clusters_index = findCenters(k,image)
    ret = numpy.zeros(image.shape)
    for i in range(k):
        for pixel_loc in clusters_index[i]:
            # divide by 255 because plt shows 0 to 255
            ret[pixel_loc[0]][pixel_loc[1]][0] = centers[i][0] / 255
            ret[pixel_loc[0]][pixel_loc[1]][1] = centers[i][1] / 255
            ret[pixel_loc[0]][pixel_loc[1]][2] = centers[i][2] / 255
    print('End Kmeans')
    return ret
```

findCenters, initializes k random centers of pixel values (we are doing kmeans on color). Then it iterates through the whole image and does the euclidean distance between each pixel color and each center's color. The closest center gets that particular pixel in its cluster. Then it updates the center by getting the average value of each cluster. We keep recomputing the new centers and the new clusters until the clusters and centers no longer change.

```python
def findCenters(k,image):
    centers = []
    clusters_color = None
    clusters_index = None
    previousSet = None
    xsize, ysize, _ = image.shape
    # randomly find k centers
    for _ in range(k):
        xrandom = randint(0,xsize-1)
        yrandom = randint(0,ysize-1)
        centers.append(image[xrandom][yrandom])
    # if it is the first iteration or the centers have not updated
    while not previousSet or updated(centers,previousSet):
        clusters_color = [[] for _ in range(k)]
        clusters_index = [[] for _ in range(k)]
        for i in range(xsize):
            for j in range(ysize):
                pixel = image[i][j]
                min_index = 0
                min_val = sys.maxsize
                # find the center the pixel is closest to
                for index in range(k):
                    distance = euclideanDistance(pixel,centers[index])
                    if distance < min_val:
                        min_val = distance
                        min_index = index
                clusters_color[min_index].append(pixel)
                clusters_index[min_index].append([i,j])
        previousSet = centers
        for i in range(k):
            centers[i] = clusterAverage(clusters_color[i])
    return centers, clusters_index
```

The other functions are helpers. Euclidean distance gets the euclidean by pixel values, clusterAverage, takes the average pixel value from a group of pixels, and updated checks if the centers have moved since the last iteration.

```python
def euclideanDistance(pixel1,pixel2):
    R = (pixel1[0] - pixel2[0]) ** 2
    G = (pixel1[1] - pixel2[1]) ** 2
    B = (pixel1[2] - pixel2[2]) ** 2
    distance = (R + G + B) ** (1/2)
    return distance

def clusterAverage(cluster):
    total = [0,0,0]
    length = len(cluster)
    # in case one of the clusters is empty
    if length != 0:
        for i in range(length):
            total[0] += cluster[i][0]
            total[1] += cluster[i][1]
            total[2] += cluster[i][2]
        total[0] = total[0] / length
        total[1] = total[1] / length
        total[2] = total[2] / length
    return total

def updated(centers, nextSet):
    for i in range(len(centers)):
        if centers[i] != nextSet[i]:
            return True
    return False
```

Here is the result of kmeans with 10 as k.

Slic.py is where slic is done and it utilizes functions from other homework assignments such as gradient, padding, etc.

The slic function contains the general algorithm which is to create centers in 50x50 blocks of the image. Then iterate at most 3 times. Doing localshift and clustering to recompute the center and clusters. It then fills in the clusters the same color and draws a border around the different colors.

```python
# added prints for debug/ let you know what stage you are in
# average runtime is about 4 minutes
def slic(image):
    print ('beginning slic')
    centers = initialCenters(image)
    previousCenters = None
    print ('getting color channel magnitudes')
    gradientMagnitude = getRGBGradient(image)
    iterations = 0
    clusters = None
    colors = None
    print('finished getting color channel magnitudes')
    # run for three iterations or if it centers have converged
    while iterations != 3:
        print('iteration: ', iterations + 1)
        previousCenters = centers.copy()
        centers = localShift(centers,gradientMagnitude)
        centers, colors, clusters  = updateCentroids(centers,image)
        if converge(centers,previousCenters):
            break
        iterations +=  1
    print('coloring in now')
    ret = fillClusters(clusters,colors,image)
    plt.imshow(ret/255)
    plt.show()
    ret = drawBorders(ret)
    ret /= 255
    return ret
```

initialCenters initializes centers within 50 x 50 blocks of the image using basic math and knowledge of image shape.

```python
def initialCenters(image):
    # centers are 50 apart starting at 25,25
    centers = []
    xsize, ysize, _ = image.shape
    xblocks = xsize // 50
    yblocks = ysize // 50
    for i in range(xblocks):
        for j in range(yblocks):
            x = 25 + i * 50
            y = 25 + j * 50
            centers.append([x,y])
    return centers
```

getRGBGradient gets the gradient within the three color channels red, green, and blue and does the l2 norm of them.

```python
def getRGBGradient(image):
    # get all color channels
    colorChannels = [image[:,:,0],image[:,:,1],image[:,:,2]]
    sobelx = numpy.array([[-1,0,1],[-2,0,2],[-1,0,1]])
    sobely = numpy.array([[1,2,1],[0,0,0],[-1,-2,-1]])
    magnitudeArray = []
    for image in colorChannels:
        # gradient magnitude for each channel
        xgradient = convolution.convulve2d(image,sobelx)
        ygradient = convolution.convulve2d(image,sobely)
        currentMag , _ = gradient.gradientInfo(xgradient,ygradient,0)
        magnitudeArray.append(currentMag)
    # total magnitude
    magnitude = (magnitudeArray[0] ** 2 + magnitudeArray[1] ** 2 + magnitudeArray[2] ** 2) ** (1/2)
    return magnitude
```

localShift tries to find the minimum value in the gradient space around a 3x3 grid of the center to move the center there. It utilizes a helper function findMinIndex which finds where the smallest value in the gradient space 3x3 grid is.

```python
# helper for local shift, find the smallest value in 3x3
def findMinIndex(chunk):
    minValue = numpy.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(centers,magnitude):
    for i in range(len(centers)):
        [x,y] = centers[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]
            # move the center to the smallest value in a 3x3 around it
            [shiftx, shifty] = findMinIndex(chunk)
            centers[i] = [x + shiftx, y + shifty]
    return centers
```

updateCentroids finds the pixels which correspond to nearest centroid. It only looks for pixels with euclidean distance of 71 indices away from it. It finds the closest centroid by doing the l2 norm of the colors and indices where the indices are halved.

```python
def updateCentroids(centers,image):
    xsize, ysize, _ = image.shape
    clustersPosition = [[] for _ in range(len(centers))]
    clustersColor = [[] for _ in range(len(centers))]
    colors = [[] for _ in range(len(centers))]
    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(centers)):
                [x,y] = centers[k]
                # check if pixel is within 71 pixels of the target center
                if ((x-i) ** 2 + (y-j) ** 2) ** (1/2) <= 71:
                    nextPixel = image[x,y]
                    # divide pixel distances by 2
                    vector2 = [x/2,y/2,nextPixel[0],nextPixel[1],nextPixel[2]]
                    distance = euclideanDistance(vector1,vector2)
                    # minimum distance
                    if distance < minValue:
                        minValue = distance
                        minIndex = k
            clustersPosition[minIndex].append(pixelCoordinates)
            clustersColor[minIndex].append(pixel)
    for i in range(len(clustersPosition)):
        centers[i], colors[i] = getClusterAverage(clustersPosition[i], clustersColor[i])
    return centers,  colors, clustersPosition
```

EuclideanDistance is a helper function that does the l2 norm of a list. getClusterAverage gets the average index and the average color of the pixels within a cluster.

```
def euclideanDistance(vector1,vector2):
    total = 0
    for i in range(len(vector1)):
        total += (vector1[i] - vector2[i]) ** 2
    return total ** (1/2)

# average of the cluster position and color
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
        position[1] //= length
        color[0] /= length
        color[1] /= length
        color[2] /= length
    return position, color
```

Converge checks if the cluster centers have changed by their index.

```
# check if the centers have not changed aka converged
def converge(centers,previousCenter):
    for i in range(len(centers)):
        pixel1 = centers[i]
        pixel2 = previousCenter[i]
        if pixel1[0]!=pixel2[0] or pixel1[1]!=pixel2[1]:
            return False
    return True
```

colorCenters is a helper that is not currently used but is meant to help with understanding if the solution makes sense as it shows where the centers are and therefore how many colors should be around it.

```
# to color the centeroids
def colorCenters(image,centers):
    xsize, ysize, _ = image.shape
    for center in centers:
        center[0] = int(center[0])
        center[1] = int(center[1])
        if center[0]!=0 and center[0]!=xsize and center[1]!= 0 and center[1]!=ysize:
            image[center[0]-1:center[0]+2,center[1]-1:center[1]+2] = [255,0,0]
    return image
```

fillClusters iterates through the clusters and fills the pixels in each color with their average value.

```python
# fill in the clusters with color
def fillClusters(clusters, colors, image):
    ret = numpy.zeros(image.shape)
    for i in range(len(colors)):
        for point in clusters[i]:
            ret[point[0],point[1]] = colors[i]
    return ret
```

drawBorders looks through the pixels in the clusters and calls clusterContains which checks if the target pixel value is the same color as the ones to the right and below if not it means they are in different clusters and colors the pixel black. If they are the same color it ensures they are in the same cluster by checking if the pixel index on the right and below are in the same cluster.

```python
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]]
    if not equalColors(pixel1Color,pixel) or not equalColors(pixel2Color,pixel):
        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 drawBorders(image,clusters):
    xsize, ysize, _ = image.shape
    ret = numpy.zeros(image.shape)
    for cluster in clusters:
        for pixel in cluster:
            if pixel[0] < xsize - 1 and pixel[1] < ysize - 1:
                if clusterContains([pixel[0]+1,pixel[1]],[pixel[0],pixel[1]+1],cluster,image):
                    ret[pixel[0],pixel[1]] = image[pixel[0],pixel[1]]
                else:
                    ret[pixel[0],pixel[1]] = [0,0,0]
            else:
                ret[pixel[0],pixel[1]] = image[pixel[0],pixel[1]]
    return ret
```

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

Here are the results of slic. Note that when you draw the cluster centers and the borders it makes sense why there are a lot of borders in certain patches. The centers are close together and therefore the values around it can either be one or the other leading to a lot of black lines/borders. There are a lot more borders/ issues with segmentation in high textured/ detailed areas.

Slic with no borders

Slic with centers:

Slic with borders:

Slick with borders and centers: