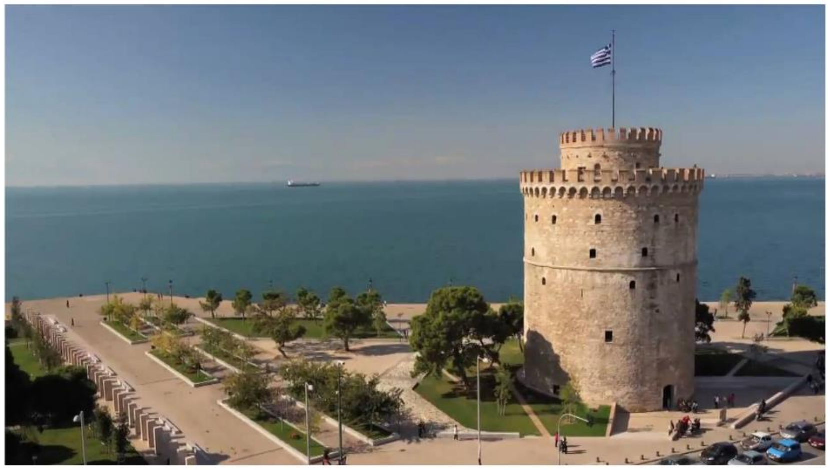
John A. V. Cinquegrana
4/16/2020
CS 558: Homework #2

Computer Vision Homework #2

Problem #1: *K-means*

```
clear;

%Problem Number 1
tower = imread( 'white-tower', 'png' );
imshow( tower )
```



Here is the original input image for the program.

```
%k-means segmentation
k = 10
clusters = random_centers( tower, k );
```

k is the number of different clusters that we will be looking for in this program. We initialize an array of those clusters to random points in the image by randomly choosing x and y coordinates. We then validate that none of the clusters are the same point.

```
%Change the image into an array of RGBPoints
W = size(tower, 2);
H = size(tower, 1);
points(H, W) = RGBPoint( tower(H,W,1), tower(H,W,2), tower(H,W,3) );


for y = 1:H
    for x = 1:W
        points(y, x) = RGBPoint( tower(y,x,1), tower(y,x,2), tower(y,x,3) );
    end
end
```

```
%Clean up the environment
clear tower;
```

I change the image from a matrix of pixels into a matrix of point objects. This way of accessing fits much better into an OOP model and allows each pixel to keep track of which cluster they are in. The object, **RGBPoint**, also has several useful functions.

```
%Begin the while loop
iterate = true;
while( iterate )

%Find out which cluster each point is in and add that point to the working
%average of that cluster
cluster_amount = zeros(k, 1);
cluster_average = zeros(k, 3);
for y = 1:H
    for x = 1:W
        points(y, x) = points(y, x).find_cluster( clusters );
        c = points(y, x).getcluster();
        cluster_average(c, 1) = (cluster_average(c, 1) * cluster_amount(c) +
points(y,x).getr()) / (cluster_amount(c) + 1);
        cluster_average(c, 2) = (cluster_average(c, 2) * cluster_amount(c) +
points(y,x).getg()) / (cluster_amount(c) + 1);
        cluster_average(c, 3) = (cluster_average(c, 3) * cluster_amount(c) +
points(y,x).getb()) / (cluster_amount(c) + 1);
        cluster_amount( c ) = cluster_amount(c) + 1;
    end
end
```

The variables of *cluster_amount* and *cluster_average* hold the amount of points belonging to each cluster and the average RGB values of each cluster, respectively. The **RGBPoint**.find_cluster function takes in an array of cluster points and assigns the closest cluster to the point. After we find out which cluster is closest to the point, we update the average RGB value of that cluster, and increment the amount of points that are contained in that cluster. After this line in the code, all points have been filtered into their respective cluster groups for this iteration of K-means.

```
%Create usable points out of the averages of each cluster
new_clusters(k) = RGBPoint();
for i = 1:k
    new_clusters(i) = RGBPoint( round( cluster_average(i, 1)),  round(
cluster_average(i, 2)),  round( cluster_average(i, 3)) );
end
```

The *new_clusters* variable is used to store the values of the cluster centers for the next iteration of K-means. We take the average RGB values of each cluster, round them off to an integer value, and initialize an **RGBPoint** to hold those values. We store the clusters as an **RGBPoint** so that we can make efficient use of the class's methods for manipulation of our data.

```
 thresh = 150; %If RGB-Distance between two points are below this threshold, we
consider them the same point.
 [iterate, distance_array] = same_clusters( clusters, new_clusters, thresh );
 iterate = ~iterate;
 disp( "Distances between SSD of old clusters averages and new ones" );
 disp( distance_array );

 clusters = new_clusters;

 end %End of while loop
```

At the end of our K-means iteration we calculate the distance between each of the cluster centers in *clusters* and in *new_clusters*. This is equivalent to calculating how far each of the centers have moved in RGB space. If any of the centers have moved more than *thresh* distance, than we consider them different points, otherwise we consider them congruent. If no single center moved more than *thresh*, than we break out of the while loop. (Note that I don't take the square root in my distance calculation, so the true threshold value is closer to the root of the *thresh* variable).

```
clear distance_array cluster_amount cluster_average i iterate new_clusters thresh

disp( "These are the resulting clusters of the k-means algorithm." )
for i = 1:k
    clusters(i).show()
end
image = zeros( H, W, 3, 'uint8');
for y = 1:H
    for x = 1:W
        c = points(y, x).getcluster();
        image(y, x, 1) = uint8(clusters(c).getr());
        image(y, x, 2) = uint8(clusters(c).getg());
        image(y, x, 3) = uint8(clusters(c).getb());
    end
end

imshow( image );
clear k clusters points c clusters H i image k points W x y
```

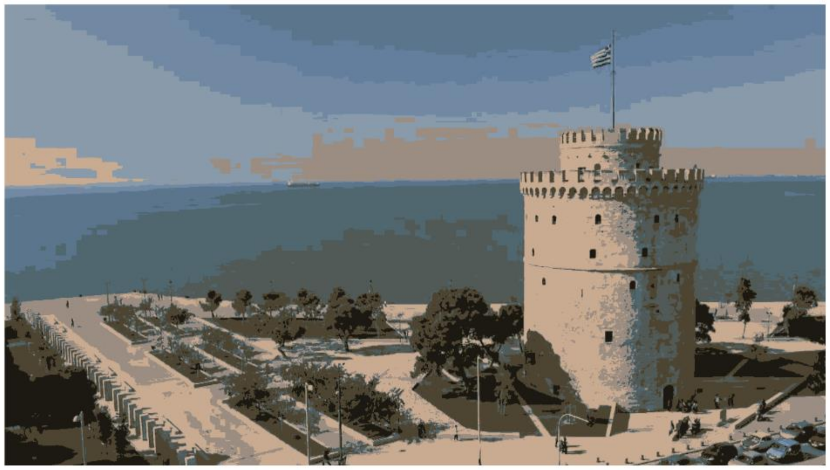
The result of displaying all of the separate clusters that resulted from the K-means algorithm is shown to the right.

```
These are the resulting
clusters of the k-means
algorithm.
R:114, G:137, B:163, C:0
R:77, G:69, B:49, C:0
R:91, G:117, B:141, C:0
```

After printing the clusters, we create the resulting image. Instead of using the RGB value of the point, we use the **RGB** value of the cluster that the point belongs to, as is required of the **K**-means algorithm.

The resulting image from the algorithm is shown below.

```
R:82, G:90, B:87, C:0
R:28, G:25, B:18, C:0
R:155, G:141, B:129, C:0
R:100, G:126, B:158, C:0
R:137, G:155, B:170, C:0
R:200, G:171, B:144, C:0
R:81, G:103, B:111, C:0
```



John A. V. Cinquegrana
4/16/2020
CS 558: Homework #2

Problem #2: SLIC algorithm

```
%%This is problem 2
clear;

SLIC = imread( 'wt_slic.png', 'png' );
imshow( SLIC );
```

Here is the input image.

```
S = 50; %The distance between the beginning superpixel groups
W = size(SLIC, 2);
H = size(SLIC, 1);

%Change the image into the point form we used before
points(W, H) = RGBPoint( SLIC(H,W,1), SLIC(H,W,2), SLIC(H,W,3) );
for y = 1:H
    for x = 1:W
        points(x, y) = RGBPoint( SLIC(y,x,1), SLIC(y,x,2), SLIC(y,x,3) );
    end
end

clear SLIC
```

Just as we did in the first problem, instead of dealing with a matrix of 8-bit integers, we read the image into an array of **RGBPoint** objects.

```
%Initialize the first centers of the superpixels to be every 50 pixels
num_super = ceil((H-S/2)/S) * ceil((W-S/2)/S)
super_pixels( num_super ) = Point();
```

```
new_super_pixels( num_super ) = FullPoint();
index = 1;
for y = S/2:S:H
    for x = S/2:S:W
        super_pixels(index) = Point(x, y);
        index = index + 1;
    end
end

clear index
```

```
num_super = 150
```

The above value (num_super) is the result of a mathematical calculation to find out the amount of superpixels in the image. We calculate this amount so we can allocate space for the necessary arrays beforehand to save efficiency. We initialize the superpixel centers into the *super_pixel* array as **Point** objects that hold only x and y values.

```
%Check the gradient around each superpixel, and move to the minimum in a
%3x3 area
gradient = zeros(3, 3);
for i = 1:size(super_pixels, 2)
    %gradient 2,2 is where our pixel will be
    px = super_pixels(i).x;
    py = super_pixels(i).y;
    for x = -1:+1
        for y = -1:+1
            if px-x > 1 && px-x < W && py-y > 1 && py-y < H
                gradient(x+2, y+2) = ...
                    points(px-x-1, py).RGB_distance( points(px-x+1, py) ) + ...
                    points(px, py-y-1).RGB_distance( points(px, py-y+1));
            end
        end
    end
    %Gradient has been calculated for each point around the superpixel in a
    %3x3 window
    minx = 1;
    miny = 1;
    min_distance = gradient(1, 1);
    for x = 1:3
        for y = 1:3
            if gradient(x, y) < min_distance
                min_distance = gradient(x, y);
                minx = x;
                miny = y;
            end
            end
        end
        %minx and miny point to the least value in gradient
        new_super_pixels(i) = points( px+minx-2, py+miny-2).to_full_point(px+minx-2,
 py+miny-2).set_cluster(i);
    end

    clear gradient min_distance minx miny
    super_pixels = new_super_pixels;
```

We iterate over each and every superpixel. We find the gradient of each point in a 3 by 3 window around that superpixel, and then move the superpixel to the position of the pixel with the minimum gradient value. Here we also change the form of the superpixel centers to **FullPoint** objects. The **FullPoint** object contains RGBxy values and has methods for calculating distance in five dimensions.

```
    %All the superpixels are now set
    %Run k-means in a 100 by 100 neighboorhood around each superpixel

    iterate = true;
    while( iterate )
        distance = Inf( W, H ); %accessed by a y,x pair. Stores distance to nearest
 superpixel.
        %Initialized to inf because initialliy, it's closest to no superpizel.

        for i = 1:num_super
            %We do this for each superpixel
            px = super_pixels(i).x;
            py = super_pixels(i).y;
            %We search in a 2S by 2S square for associated pixels
            for x = px-S:px+S
                for y = py-S:py+S
                    %Make sure the pixel we're testing is within image
                    %boundaries
                    if x > 0 && x <= W && y > 0 && y <= H
                        dis = super_pixels(i).full_distance( points(x, y), x, y );
                        if dis < distance(x, y)
                            %Set the cluster and distance to the new center
                            distance(x, y) = dis;
                            points(x,y) = points(x,y).set_cluster(i);
                        end
                    end
                end
            end
        end
```

John A. V. Cinquegrana
4/16/2020
CS 558: Homework #2
For each superpixel we search the neighborhood of the center for pixels that may belong to it. The *distance* matrix contains the distance from each **RGBPoint** in *points* to the nearest superpixel cluster. We use this because in the SLIC algorithm we iterate superpixel by superpixel instead of by point to point; therefore, we cannot compare a single point to each superpixel at once. Since the superpixel each point belongs to can change in later iterations of the for loop, we cannot calculate the superpixel averages in this run over the image.

```
     sup_amount = zeros(num_super, 1); %Holds the amount of pixels contained within 
each superpixel.
     sup_average = zeros(num_super, 5); %Holds the average of R, G, B, x, y for each 
superpixel

     for y = 1:H
         for x = 1:W
             %Find the averages of each superpixel so we can move them
             c = points(x, y).getcluster();
             sup_average(c, 1) = (sup_average(c, 1) * sup_amount(c) + 
points(x,y).getr()) / (sup_amount(c) + 1);
             sup_average(c, 2) = (sup_average(c, 2) * sup_amount(c) + 
points(x,y).getg()) / (sup_amount(c) + 1);
             sup_average(c, 3) = (sup_average(c, 3) * sup_amount(c) + 
points(x,y).getb()) / (sup_amount(c) + 1);
             sup_average(c, 4) = (sup_average(c, 4) * sup_amount(c) + x) / 
(sup_amount(c) + 1);
             sup_average(c, 5) = (sup_average(c, 5) * sup_amount(c) + y) / 
(sup_amount(c) + 1);
             sup_amount( c ) = sup_amount(c) + 1;
         end
     end
```

We iterate over each point in the image. We average that points RGBxy values into the corresponding superpixel averages in the *sup_average* variable. We use the *sup_amount* variable to store the amount of points in each superpixel so that we can accurately average in each value without knowing, at the time, how many points will be in each superpixel.

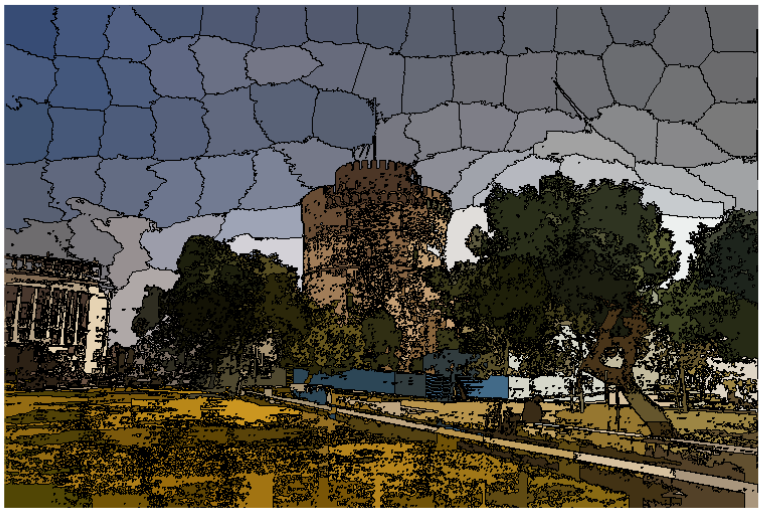
```
     %Create usable data out of the averages of each superpixel
     new_super_pixels( num_super ) = FullPoint();
     distance_arr = zeros( num_super, 1);
     for i = 1:num_super
         new_super_pixels(i) = FullPoint(...
             round( sup_average(i, 1)),...
             round( sup_average(i, 2)),...
             round( sup_average(i, 3)),...
             round( sup_average(i, 4)),...
```

```
            round( sup_average(i, 5))     );
        distance_arr( i ) = new_super_pixels(i).distance( super_pixels(i) );
    end
    %Determine whether or not to iterate
    thresh = 200;
    if max( distance_arr ) < thresh
        iterate = false;
    end
```

We populate the *new_super_pixels* array. This array contains, as **FullPoint** objects, what the new center of each superpixel will be in the next iteration. Each object in *new_super_pixels* contains RGBxy values. In each iteration we also add a corresponding value to the *distance_arr* array. This array contains the distance each superpixel center moved in five-dimensional RGBxy space. If any superpixel has there center move more than *thresh* distance, than we run the while loop again with the *super_pixels* values replaced by the *new_super_pixels* values. (Note that I don't take the square root in my distance calculation, so the true threshold value is closer to the root of the *thresh* variable).

```
    %Set superpixels for the next iteration
    super_pixels = new_super_pixels;

end %End of k-means iteration

clear distance distance_arr min_distance new_super_pixels sup_amount sup_average

%super_pixels contains the final versions of super_pixels
%Each pixel in the points array has its cluster set accordingly

%Building the resulting image again
image = zeros( H, W, 3, 'uint8');
for y = 1:H
    for x = 1:W
        c = points(x, y).getcluster();
        image(y, x, 1) = uint8(super_pixels(c).getr());
        image(y, x, 2) = uint8(super_pixels(c).getg());
        image(y, x, 3) = uint8(super_pixels(c).getb());
    end
end

%add the black lines between each cluster boundary
for y = 3:H
    for x = 3:W
        if points(x,y).getcluster() ~= points(x-1,y).getcluster() ||...
            points(x,y).getcluster() ~= points(x,y-1).getcluster()
            %If a cluster has changed, make this pixel black
            image(y,x, 1) = uint8(0);
            image(y,x, 2) = uint8(0);
            image(y,x, 3) = uint8(0);
        end
    end
end
imshow( image );
disp( "SLIC Algorithm has finished" );
```

Once outside the while loop, we prepare the final image. We read the point array back into an 8-bit integer matrix, the MatLab format for images. After that we iterate over each pixel in the image we created and, if it is in a different superpixel than its immediate neighbors, we paint it black to show superpixel boundaries. After showing the final image we print an output line to show that our program is completed.



```
SLIC Algorithm has finished
```

Additional Notes

Object/Method	*Description*
FullPoint. *distance*	This function does not just calculate the 5-dimensional distance between two FullPoints in RGBxy space. It divides the xy values by 2 in order to make the RGB values hold more weight in the SLIC algorithm.
RGBPoint	This object stores each of its RGB and cluster value as 8-bit unsigned integers. Therefore, if the you try and run k-means with more than 255 clusters, or a large enough image in SLIC that needs over 255 superpixels, than the program will crash.

John A. V. Cinquegrana
4/16/2020
CS 558: Homework #2

RGBPoint. *find_cluster()*	This method, when given an array of RGBPoints, compares the distance, in RGB space, from itself to each of the RGBPoints. It then sets the cluster value of itself to the index of the closest point in the array. This is the method I use to find out in which cluster a point belongs to in the k-means algorithm.
FullPoint. *distance()* And ***RGBPoint.*** *RGB_distance()*	These two methods each, when given a point, computes the distance between the object point, and the parameter point, in either RGBxy or RGB space depending on class. Neither of these functions calculate true distance since they never use a square root function. Since distance values are only ever compared to other distance values, the same exact logical conclusion is still reached. It is worth remembering, however, that the distance values in my program are so large because the are essentially the algebraic square of the true distance values.