Alejandro Garcia
3/11/2020
"I pledge my honor that I have abided by the stevens honor system"

Problem 1

The first step of the k-means algorithm is to initialize 10 random points for each of the RGB channels. This was easily done through the following

```
    r_points = [random.randint(0, 255) for i in range(k)]
    g_points = [random.randint(0, 255) for i in range(k)]
    b_points = [random.randint(0, 255) for i in range(k)]
```

The next step was to develop the clusters themselves. Which could be seen through the make_clusters function which takes the parameter of the random points of that RGB channel. This function runs until the old cluster equals the new cluster's points. From this function we take the clusters within the image and the cluster means.

```
def make_clusters(ran_points):
    cluster_old = []

    while not(cluster_old == ran_points):
        clusters = points_in_clusters(ran_points)
        cluster_means = set_mean(clusters)
        cluster_old = ran_points
        ran_points = cluster_means

    return clusters, cluster_means
```

Below is a demonstration of how to obtain those clusters. First, we sort all the points given. Then for each point p (ranging for 0 to 255 values) we find our closest cluster value. Then we set our index of the center cluster to that point p value.

```
def points_in_clusters(ran_points):
    clusters = [[] for i in range(len(ran_points))]

    ran_points.sort()

    bins = list(range(256))

    for p in range(len(bins)):
        min = 256
        idx = 0
        for i in range(len(ran_points)):
            if min > abs(ran_points[i] - bins[p]):
                min = abs(ran_points[i] - bins[p])
                idx = i
        clusters[idx] += [p]

    return clusters
```

Next the set_means function allows us to determine the cluster means. For each center point of our cluster set. We set those points to the mean points of that cluster and then return that point.

```
def set_mean(clusters):
    points = [0] * len(clusters)
    for i in range(len(clusters)):
        amount = len(clusters[i])
        sums = sum(clusters[i])
        if len(clusters[i]):
            points[i] = int(sums / amount)
        else:
            points[i] = 0

    return points
```

We do this for each RGB Channel

```
    #red

    clusters_r, r_mean = make_clusters(r_points)
    dict_r = set_cluster_def(clusters_r, r_mean)

    #green

    clusters_g, g_mean = make_clusters(g_points)
    dict_g = set_cluster_def(clusters_g, g_mean)

    #blue

    clusters_b, b_mean = make_clusters(b_points)
    dict_b = set_cluster_def(clusters_b, b_mean)
```

Finally we output to the image pixels and obtain the result using a dictionary from the cluster points and the cluster means. Then return and plot the image.

```
def set_cluster_def(clusters, cluster_means):
    dict = {}
    for i in range(len(clusters)):
        for j in clusters[i]:
            dict[j] = cluster_means[i]
    return dict
```

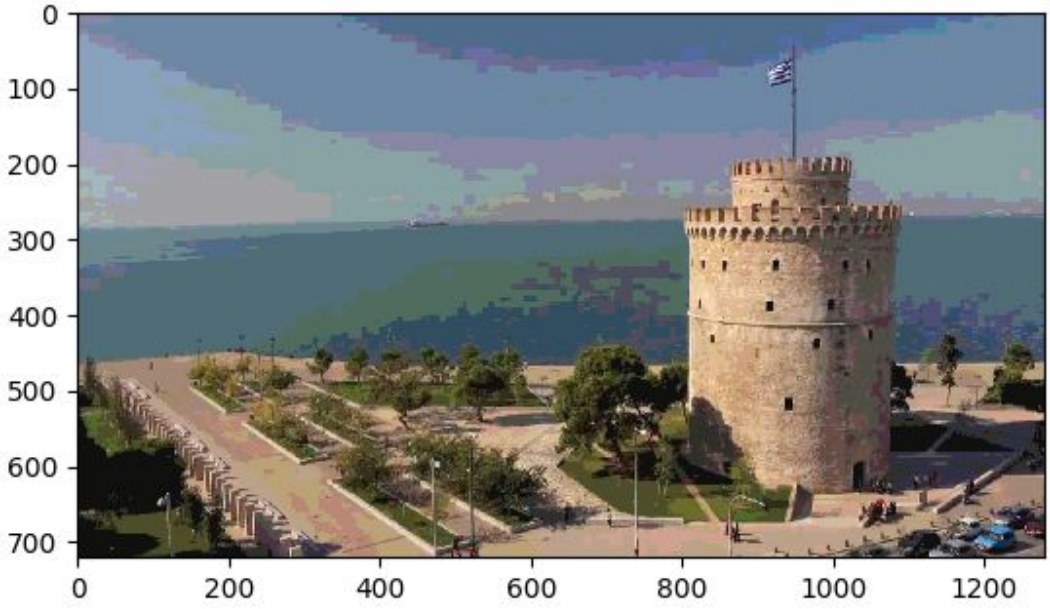
```
    for y in range(img_y):
        for x in range(img_x):
            r = int(img[y][x][0] * 255)
            g = int(img[y][x][1] * 255)
            b = int(img[y][x][2] * 255)
            img[y][x][0] = dict_r[r] / 255
            img[y][x][1] = dict_g[g] / 255
            img[y][x][2] = dict_b[b] / 255

    return img
```

The output results in the following image below. The Image looks similar, however, the image is segmented into k^3 pixel values. Since k is 10 the image has 1000 different pixel value combinations within the RGB channels.

Input:

Output:



Problem 2

Since the python implementation was too slow. The code was implemented in C++ for faster execution. The first step of the SLIC algorithm was to initialize the cluster centers on the image using S steps. Since the image was divided into 50 by 50 blocks. It was only reasonable that the centers start on S divided by two. A Point class was developed with the R, G, B, x, y values.

```
class Point {
public:
    int x, y;
    int r, g, b;
    Point(int x, int y, int r, int g, int b)
    : x(x), y(y), r(r), g(g), b(b)
    {}
    Point() : Point(0, 0, 0, 0, 0)
    {}
    Point(const Point &p) : x(p.x), y(p.y), r(p.r), g(p.g), b(p.b)
    {}
...
```

```
    Point *cluster_centers = new Point[K];
    int zz = 0;
    for (int y = S / 2; y < img_y; y += S)
        for (int x = S / 2; x < img_x; x += S, ++zz)
            cluster_centers[zz] = img->get_point(x, y);
```

Next, the gradient of the x and y of the RGB values were then obtained to find both the magnitude and the combined magnitude of the RGB image.

```
    unsigned char **rChan = init_2d<unsigned char>(img_x, img_y);
    unsigned char **gChan = init_2d<unsigned char>(img_x, img_y);
    unsigned char **bChan = init_2d<unsigned char>(img_x, img_y);
    for (int y = 0; y < img_y; ++y) {
        for (int x = 0; x < img_x; ++x) {
            Point p = img->get_point(x, y);
            rChan[y][x] = p.r;
            gChan[y][x] = p.g;
```

```
            bChan[y][x] = p.b;
        }
    }

    double **grad_rx, **grad_ry, **grad_gx, **grad_gy, **grad_bx,
**grad_by;
    gradient(rChan, grad_rx, grad_ry, img_x, img_y);
    gradient(gChan, grad_gx, grad_gy, img_x, img_y);
    gradient(bChan, grad_bx, grad_by, img_x, img_y);

    double **combined_grad = init_2d<double>(img_x, img_y);

    for(int y = 0; y < img_y; y++){
        for(int x = 0; x < img_x; x++){
            double r_mag = sqrt(pow(grad_rx[y][x], 2) + pow(grad_ry[y][x],
2));
            double g_mag = sqrt(pow(grad_gx[y][x], 2) + pow(grad_gy[y][x],
2));
            double b_mag = sqrt(pow(grad_bx[y][x], 2) + pow(grad_by[y][x],
2));
            combined_grad[y][x] = sqrt(pow(r_mag, 2) + pow(g_mag, 2) +
pow(b_mag, 2));
        }
    }
```

Then, in a 3 by 3 window, from each of the cluster centers the pixel with the lowest combined gradient was chosen

```
    for (int k = 0; k < K; ++k) {
        double min_grad = DBL_MAX;
        Point cluster = cluster_copy[k];
        if (cluster.x == 0 || cluster.y == 0)
            continue;
        for (int y = -1; y < 2; ++y) {
            for (int x = -1; x < 2; ++x) {
                if (combined_grad[cluster_copy[k].y + y][cluster_copy[k].x
+ x] < min_grad) {
```

```
                    Point p = img->get_point(cluster_copy[k].x + x,
cluster_copy[k].y + y);
                    cluster_centers[k].r = p.r;
                    cluster_centers[k].g = p.g;
                    cluster_centers[k].b = p.b;
                    cluster_centers[k].x = p.x;
                    cluster_centers[k].y = p.y;
                    min_grad = combined_grad[cluster_copy[k].y +
y][cluster_copy[k].x + x];
                }
            }
        }
    }
```

The pixels in a 2S by 2S region from each of the cluster centers were then assigned to find the smallest euclidean distance in the space. K-means was then applied in order to find the mean color and pixels with each cluster. This would be repeated until the residual error was greater than the threshold error.

```
    while (residual_error > error_threshold) {
        for (int k = 0; k < K; ++k) {
            Point cluster = cluster_centers[k];
            int x_center = cluster.x;
            int y_center = cluster.y;
            for (int y = -S; y <= S; y++) {
                for (int x = -S; x <= S; x++) {
                    if (y + cluster.y < 0 || x + cluster.x < 0 || y +
cluster.y >= img_y || x + cluster.x >= img_x)
                        continue;
                    Point pixel_i = img->get_point(cluster.x + x,
cluster.y + y);

                    double D = euclidean_distance(cluster, pixel_i, m, S);

                    int loc = pixel_i.y * img_x + pixel_i.x;

                    if (D < distance[loc]) {
```

```
                        distance[loc] = D;
                        label[loc] = k;
                    }
                }
            }
        }

        for (int k = 0; k < K; ++k) {
            Point new_center = Point();
            int total_in_cluster = 0;
            for(int j = 0; j < N; ++j) {
                Point pixel = img_pixels[j];
                int x = pixel.x;
                int y = pixel.y;
                if(label[y * img_x + x] == k) {
                    new_center += pixel;
                    total_in_cluster++;
                }
            }
            if (total_in_cluster > 0) {
                new_center /= total_in_cluster;
                Point tmp = cluster_centers[k] - new_center;
                double partial_error[5] = {(double)tmp.x, (double)tmp.y,
(double)tmp.r, (double)tmp.g, (double)tmp.b};
                double dot = dot_product(partial_error, partial_error, 5);
                residual_error += sqrt(dot);
                cluster_centers[k] = new_center;
            }
        }

        if (++iter > max_iter) {
            residual_error = error_threshold;
        }
    }
```

The euclidean distance was calculated as followed. Using the formula below

$$D = \sqrt{{d_c}^2 + \left(\frac{d_s}{S}\right)^2 m^2}.$$

```
double euclidean_distance(Point source, Point dest, int m, int S)
{
    double d_rgb = sqrt(pow(source.r - dest.r, 2) + pow(source.g - dest.g,
2) + pow(source.b - dest.b, 2));
    double d_xy = sqrt(pow((source.x - dest.x)/2.0, 2) + pow((source.y -
dest.y)/2.0, 2));
    return sqrt(pow(d_rgb, 2) + pow(d_xy / S, 2) * pow(m, 2));
}
```

The edges of each of the clusters were then calculated in order to segment the image. This was done by checking whether or not the the next values of the clusters were different as shown below

```
    for (int y = 0; y < img_y; ++y) {
        for (int x = 0; x < img_x; ++x) {
            int pos = y * img_x + x;
            int belongs_to = label[pos];
            if (x != 0 && y != 0 && x != img_x - 1 && y != img_y - 1) {
                if (label[pos + 1] != belongs_to || label[(y + 1) * img_x
+ x] != belongs_to)
                    edges[y][x] = 1;
            }
        }
    }
```

Finally the processing was finished by outputting the edges and the cluster means.

```
    for (int y = 0; y < img_y; ++y)
        for (int x = 0; x < img_x; ++x)
            color[y][x] = cluster_centers[label[y * img_x + x]];
```

```
    for (int y = 0, px = 0; y < img_y; ++y) {
        for (int x = 0; x < img_x; ++x, px+=3) {
            if (edges[y][x] == 1 || !y || !x || x == img_x - 1 || y ==
img_y - 1) {
                out[px+0] = 0;
                out[px+1] = 0;
                out[px+2] = 0;
                color[y][x] = Point();
            }
        }
    }
    for (int y = 0; y < img_y; ++y) {
        for (int x = 0; x < img_x; ++x) {
            if (edges[y][x] == -1) {
                Point p = replace_color(cluster_centers, label, x, y,
img_x, img_y);
                Point c = cluster_centers[label[y * img_x + x]];
                color[y][x] = p;
            }
        }
    }

    for (int y = 0, px = 0; y < img_y; ++y) {
        for (int x = 0; x < img_x; ++x, px += 3) {
            Point p = cluster_centers[label[y * img_x + x]];
            out[px+0] = color[y][x].r;
            out[px+1] = color[y][x].g;
            out[px+2] = color[y][x].b;
        }
    }


    stbi_write_png("slic_output.png", img_x, img_y, 3, out, 0);
```

The final output resulted in the following:

Input:

Output: