CSE 344 Computer Vision Assignment 2

Name : Arka Sarkar
Roll Number : 2018222

In [1]:
```python
#dependencies Required

import numpy as np
import cv2
import matplotlib.pyplot as plt
from IPython.display import display
from PIL import Image
import skfuzzy as fuzz
from sklearn.metrics import silhouette_score
from tqdm import tqdm
from skimage.segmentation import slic
from skimage.segmentation import mark_boundaries
from skimage.util import img_as_float
from skimage import io
from scipy.spatial.distance import cdist
import scipy.stats
import copy
import math
from scipy.integrate import quad
from sklearn.cluster import KMeans
from scipy import ndimage
```

Question 1

Write a program to implement a region segmentation algorithm using the fuzzy c-means algorithm on normalized 'RGBxy' data of an image. Merge stray (isolated) pixels (or very-small regions) to their surrounding regions. [3 marks]

In [2]:
```python
img = cv2.imread("Cap1.PNG")
fig=plt.figure(figsize=(6, 6))
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.show()
```

In [3]:
```python
#Making the Dataset

img = np.array(img)
img_data = np.zeros((img.shape[0], img.shape[1], 5))
for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        img_data[i,j,0:3] = img[i,j]/255
        img_data[i,j,3] = i/img.shape[0]
        img_data[i,j,4] =  j/img.shape[1]
img_data = img_data.reshape(((img.shape[0]*img.shape[1], 5))).T
```

Fuzzy C Means Clustering

Fuzzy clustering (also referred to as soft clustering or soft k-means) is a form of clustering in which each data point can belong to more than one cluster.
The FCM aims to minimize an objective function:

$$\underset{c}{\operatorname{argmin}} \sum_{i=1}^{n} \sum_{j=1}^{c} w_{ij}^{m} \times ||X_i - C_j||^2$$

where
X is the set of all data points
C is the cluster centers
m is the exponentiation factor

$$w_{ij}^{m} = \frac{1}{\sum_{k=1}^{c} \left(\frac{||X_i - C_j||}{||X_i - C_k||} \right)^{\frac{2}{m-1}}}$$

Reference : Wikipedia - Fuzzy clustering

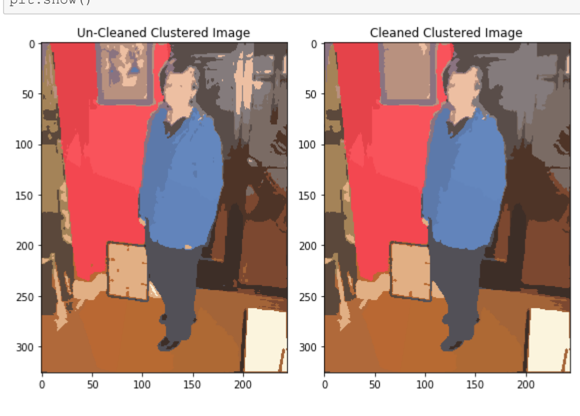
In [4]:
```python
#Applying FC Means
fcm = fuzz.cluster.cmeans(img_data, 25, 2, error=0.05, maxiter=1000, init=None)
```

Merge stray (isolated) pixels (or very small regions) to their surrounding regions.

To clean the image **binary_fill_holes** from scikit image was used on every binary cluster iteratively to fill small pixels regions to the surrounding regoin making the clustered image more smoother.

In [19]:
```python
def clean_image(img, plot = False):
    m,n = img.shape
    binary_maps = []
    if(plot):
        fig=plt.figure(figsize=(60, 60))
        columns = 5
        rows = len(np.unique(img))//5
        img = img + 1
        n = len(np.unique(img))
        for i in range(1,n+1):
            curr = copy.deepcopy(img)
            curr[curr != i] = 0
            curr[curr == i] = 1
            filled = ndimage.binary_fill_holes(curr).astype(int)
            idx = np.where(filled == 1)
            img[idx[0],idx[1]] = i
            fig.add_subplot(rows, columns, i)
            plt.imshow(filled.astype(int))
        fig.tight_layout()
        plt.show()

        return img-1
    img = img + 1
    n = len(np.unique(img))
    for i in range(1,n+1):
        curr = copy.deepcopy(img)
        curr[curr != i] = 0
        curr[curr == i] = 1
        filled = ndimage.binary_fill_holes(curr).astype(int)
        idx = np.where(filled == 1)
        img[idx[0],idx[1]] = i
    return img-1
```

In [6]:
```python
cluster_centers = fcm[0]
prob_matrix = fcm[1]
pred_matrix = np.argmax(prob_matrix, axis = 0)
pred_matrix = cluster_centers[pred_matrix]
clustered_image = pred_matrix[:,0:3]
clustered_image = clustered_image.reshape(((img.shape[0],img.shape[1], 3)))
clustered_image = clustered_image.astype('float32')
pred_matrix = np.argmax(prob_matrix, axis = 0)
cleaned_img = clean_image(np.squeeze(pred_matrix).reshape(img.shape[0],img.shape[1]))
cleaned_clus_img = cluster_centers[cleaned_img][:,:,0:3]
cleaned_clus_img = cleaned_clus_img.reshape(((img.shape[0],img.shape[1], 3)))
cleaned_clus_img = cleaned_clus_img.astype('float32')
fig=plt.figure(figsize=(8, 8))
columns = 2
rows = 1
fig.add_subplot(rows, columns, 1)
plt.imshow(cv2.cvtColor(clustered_image, cv2.COLOR_BGR2RGB))
plt.title("Un-Cleaned Clustered Image")
fig.add_subplot(rows, columns, 2)
plt.imshow(cv2.cvtColor(cleaned_clus_img, cv2.COLOR_BGR2RGB))
plt.title("Cleaned Clustered Image")
fig.tight_layout()
plt.show()
```



Question 2

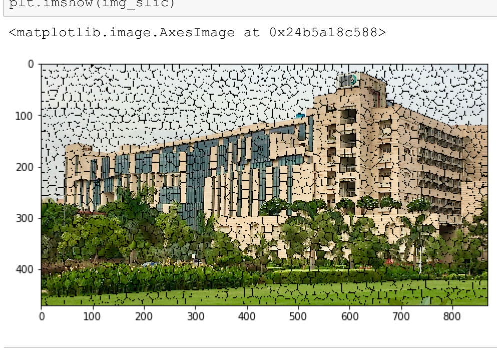
Write a program to obtain the spatial and contrast cues using SLIC superpixels of an image instead of pixels. [3 marks]

In [7]:
```python
img_path = '1558014721_E7jyWs_iiit_d.jpg'
image  = cv2.imread(img_path)
plt.figure(figsize = (8,8))
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
```

Out[7]: <matplotlib.image.AxesImage at 0x24b589bddc8>

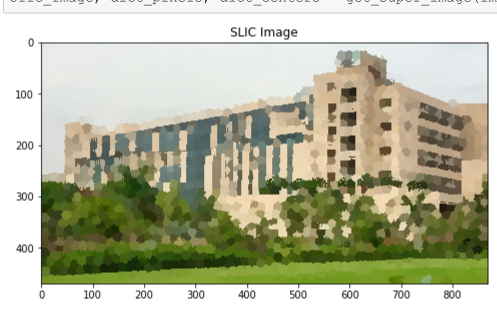
In [8]:
```python
img = cv2.imread(img_path)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
#Initialize the slic item, the average size of super pixels is 20 (default is 10), and the smoothing fa
ctor is 20
slic = cv2.ximgproc.createSuperpixelSLIC(img,region_size=15,ruler = 20.0)
slic.iterate(40)     #Number of iterations, the greater the better
mask_slic = slic.getLabelContourMask() #Get Mask, Super pixel edge Mask==1
label_slic = slic.getLabels()        #Get superpixel tags
number_slic = slic.getNumberOfSuperpixels()  #Get the number of super pixels
mask_inv_slic = cv2.bitwise_not(mask_slic)
img_slic = cv2.bitwise_and(img,img,mask =  mask_inv_slic) #Draw the superpixel boundary on the original
 image
plt.figure(figsize = (8,8))
plt.imshow(img_slic)
```

Out[8]: <matplotlib.image.AxesImage at 0x24b5a18c588>



In [9]:
```python
def get_super_image(image, segments):

    m,n = segments.shape

    dict_ = {}
    centers = {}

    for i in range(m):
        for j in range(n):
            if(segments[i,j] not in dict_):
                dict_[segments[i,j]] = []
                centers[segments[i,j]] = []
                dict_[segments[i,j]].append(image[i,j])
                centers[segments[i,j]].append(np.array([i,j]))
            else :
                dict_[segments[i,j]].append(image[i,j])
                centers[segments[i,j]].append(np.array([i,j]))
    for key in list(dict_.keys()):
        dict_[key] = np.mean(np.array(dict_[key]), 0).astype(int)
        centers[key] = np.mean(np.array(centers[key]), 0).astype(int)

    slic_image = np.zeros((image.shape[0],image.shape[1],3))

    for i in range(m):
        for j in range(n):
            slic_image[i,j] =  dict_[segments[i,j]]
    slic_image = slic_image.astype(int)
    plt.figure(figsize = (8,8))
    plt.imshow(slic_image)
    plt.title("SLIC Image")

    return slic_image, dict_, centers
```

In [10]:
```python
slic_image, dict_pixels, dict_centers = get_super_image(img,label_slic)
```



Contrast Cue

Contrast cue represents the **visual feature uniqueness** on the single or multiple images. Contrast is one of the most widely used cues for measuring saliency in single image saliency detection algorithms, since the contrast operator simulates the human visual receptive fields. This rule is also valid in the case of cluster-based method for the multiple images, while the difference is that contrast cue on the cluster-level better represents the global correspondence relationship than the pixel/patch level.
The contrast cue $w^c(k)$ of cluster C^k is defined using its feature contrast to all other clusters:

$$w^c(k) = \sum_{i=1, i \neq k}^{K} \left(\frac{n^i}{N} ||\nu^k - \mu^i||_2 \right)$$

where a L2 norm is used to compute the distance on the feature space, n^i represents the pixel number of cluster C^i , and N denotes the pixel number of all images.

Spatial Cue

In human visual system, the regions near the image center draw more attention than the other regions. When the distance between the object and the image center increases, the attention gain is depreciating. This scenario is known as **'central bias rule'** in single image saliency detection. We extend this concept to the cluster-based method, which measures a global spatial distribution of the cluster.
The spatial cue $w^s(k)$ of cluster C^k is defined as:

$$w^s(k) = \frac{1}{n^k} \sum_{j=1}^{M} \sum_{i=1}^{N_j} [N(||z_i^k - o^j||^2 | 0, \sigma^2) . \delta(b(p_i^j) - C^k)]$$

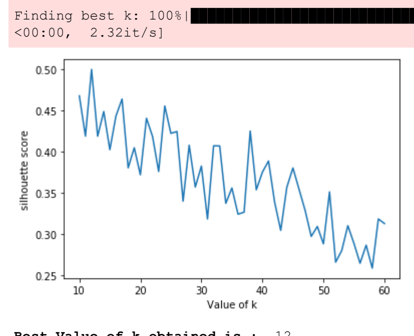
where $\delta(.)$ is the Kronecker delta function, o^j denotes the center of image I^j, and Gaussian kernel $N(\cdot)$ computes the Euclidean distance between pixel z_i^j and the image center o^j, the variance σ^2 is the normalized radius of images. And the normalization coefficient n^k is the pixel number of cluster C^k . Different from the single image model, our spatial cue w^s represents the location prior on the cluster-level, which is a global central bias on the multiple images. The same as the contrast cue, the spatial cue is also **valid on both single and multiple images**.

Reference : Cluster-Based Co-Saliency Detection

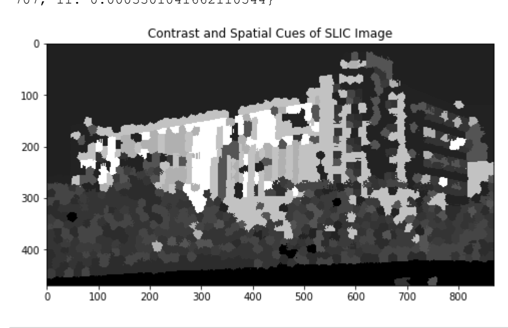
In [11]:
```python
def contrast_spatial_cue(slic_image, dict_pixels, label_slic, dict_centers):
    m,n = label_slic.shape

    data = np.zeros((len(list(dict_pixels.keys())), 3))
    l = sorted(list(dict_pixels.keys()))
    for i in range(len(l)):
        data[i] = dict_pixels[l[i]]

    data = np.float32(data)/255

    s_scores = []
    k = 5
    s_max = 0
    for k_i in tqdm(range(10,61), position = 0, desc = "Finding best k"):
        criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 200, 0.2)
        _, labels, (centers) = cv2.kmeans(data, k_i, None, criteria, 200, cv2.KMEANS_RANDOM_CENTERS )
        ss = silhouette_score(data, np.squeeze(labels))
        s_scores.append(ss)
        if(ss > s_max):
            s_max = ss
            k = k_i

    plt.plot([i for i in range(10,61)], s_scores)
    plt.xlabel("Value of k")
    plt.ylabel("silhouette score")
    plt.show()

    print("\033[1mBest Value of k obtained is : \033[0m",k)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 200, 0.02)
    _, labels, (centers) = cv2.kmeans(data, k, None, criteria, 200, cv2.KMEANS_RANDOM_CENTERS )
    pred_matrix = np.squeeze(labels)
    cluster_dist = cdist(centers,centers, 'euclidean')
    clustered_img = pred_matrix[label_slic]
    n_i = {}
    for i in np.unique(clustered_img):
        n_i[i] = np.count_nonzero(clustered_img == i)
    print("\033[1mNumber of Image Pixels per cluster : \033[0m",n_i)

    contrast_cues = {}
    for i in range(k):
        val = 0
        for j in range(k):
            if(i!=j):
                val+=(n_i[j]/(clustered_img.shape[0]*clustered_img.shape[1]))*cluster_dist[i,j]
        contrast_cues[i] = val
    print("\033[1mContrast Cues : \033[0m",contrast_cues)

    center_data = np.zeros((len(list(dict_centers.keys())), 2))
    l = sorted(list(dict_centers.keys()))
    for i in range(len(l)):
        a,b = dict_centers[l[i]]
        center_data[i,0] = a
        center_data[i,1] = b
    slic_image_center = dict_centers[label_slic[m//2,n//2]].reshape((1,2))
    slic_image_center[0,0] = slic_image_center[0,0]
    slic_image_center[0,1] = slic_image_center[0,1]
    slic_center_dist = cdist(center_data,slic_image_center,'euclidean')
    slic_center_dist_var = np.std(slic_center_dist)

    spatail_cues = {}
    for c in range(k):
        val = 0
        for i in range(len(slic_center_dist)):
            if(pred_matrix[i] == c):
                val+=scipy.stats.norm(0, slic_center_dist_var).pdf(slic_center_dist[i,0])
        val = val/np.count_nonzero(pred_matrix == c)
        spatail_cues[c] = val
    print("\033[1mSpatial Cues : \033[0m",spatail_cues)

    final_image = np.zeros((m,n))
    for i in range(m):
        for j in range(n):
            idx = clustered_img[i,j]
            val = spatail_cues[idx]*contrast_cues[idx]

            final_image[i,j] = val
    plt.figure(figsize = (8,8))
    plt.imshow(final_image, cmap = 'gray')
    plt.title("Contrast and Spatial Cues of SLIC Image")
    return contrast_cues, spatail_cues,clustered_img, final_image
```

In [12]:
```python
contrast_cues, spatail_cues,clustered_img, final_image = contrast_spatial_cue(slic_image, dict_pixels,
label_slic, dict_centers)
```

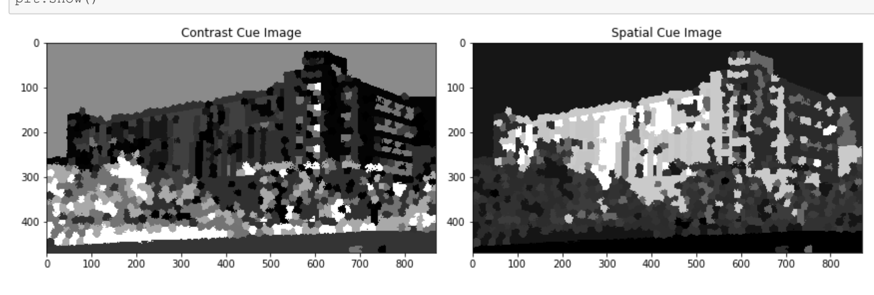
```
Finding best k: 100%|██████████████████████████████████████████| 51/51 [00:21
<00:00,  2.32it/s]
```




```
Best Value of k obtained is :  12
Number of Image Pixels per cluster :  {0: 30850, 1: 18227, 2: 53983, 3: 31794, 4: 40286, 5: 23412, 6:
 17714, 7: 25440, 8: 13962, 9: 23613, 10: 25970, 11: 103649}
Contrast Cues :  {0: 0.5001077553295481, 1: 0.5745287569404574, 2: 0.4952804724575096, 3: 0.635933366
5869334, 4: 0.4431685246076661, 5: 0.5724946495057592, 6: 0.5130627406302538, 7: 0.737204830960124,
 8: 0.46519157909491693, 9: 0.4373689002792746, 10: 0.4947235316349006, 11: 0.6011630977697624}
Spatial Cues :  {0: 0.0001858025539258633, 1: 0.00046039491373016964, 2: 0.0014792389031498127, 3: 0.
0005066217487978924, 4: 0.00046611763317128694, 5: 0.00046691656879523156, 6: 0.001820281254824394,
 7: 0.0003240302224580659, 8: 0.0014447053618514187, 9: 0.0008471857270555645, 10: 0.0005815901929739
707, 11: 0.0003301041662110544}
```



In [13]:
```python
m,n = clustered_img.shape
contrast_cue_image = np.zeros((m,n))
spatial_cue_image = np.zeros((m,n))
for i in range(m):
    for j in range(n):
        contrast_cue_image[i,j] = contrast_cues[clustered_img[i,j]]
contrast_cue_image = (contrast_cue_image - np.min(contrast_cue_image))/(np.max(contrast_cue_image) - np
.min(contrast_cue_image))
for i in range(m):
    for j in range(n):
        spatial_cue_image[i,j] = spatail_cues[clustered_img[i,j]]

spatial_cue_image = (spatial_cue_image - np.min(spatial_cue_image))/(np.max(spatial_cue_image) - np.min
(spatial_cue_image))
columns = 2
rows = 1
fig=plt.figure(figsize=(12, 12))
fig.add_subplot(rows, columns, 1)
plt.imshow(contrast_cue_image, cmap = 'gray')
plt.title("Contrast Cue Image")
fig.add_subplot(rows, columns, 2)
plt.imshow(spatial_cue_image, cmap = 'gray')
plt.title("Spatial Cue Image")
fig.tight_layout()
plt.show()
```



Question 3

Implement the separation measure discussed in Sec III.B.1 of the following paper to obtain quality scores for the two cues obtained in Q2. Use these quality scores as weights while performing the weighted sum of the two cues for getting the final saliency cue. [4 marks]

In [14]:
```python
def otsu(img):

    min_cost = float('inf')
    threshold = 0

    for i in range(1,256):
        v0 = np.var(img[img < i], ddof = 1)
        w0 = len(img[img < i])
        v1 = np.var(img[img >= i], ddof = 1)
        w1 = len(img[img >= i])

        cost = w0*v0 + w1*v1
        if(cost < min_cost):
            min_cost = cost
            threshold = i
    return threshold

def select_foreground(img):
    m,n = img.shape

    m0 = int(0.15*m)
    n0 = int(0.15*n)
    c0 = 0
    c1 = 0
    for i in range(m//2 - m0, m//2 + m0):
        for j in range(n//2 - n0, n//2 + n0):
            if(img[i,j] == 0):
                c0 = c0 + 1
            else:
                c1 = c1 + 1
    if(c0 > c1):
        return 0
    else :
        return 1
```

Separation Measure

A high-quality saliency mapshould have well-separated foreground and background likelihoods like a ground-truth binary mask. Assuming distributions of these likelihoods to be of Gaussian in nature, we attempt to measure the separation between the two. Let $\mu_f(S)$,$\mu_b(S)$, $\sigma_f(S)$, and $\sigma_b(S)$ denote foreground mean, background mean,foreground standard deviation, and background standard deviation, respectively, computed based on the two likelihood distributions (obtained by Otsu thresholding). Let us denote $D_f(z;S)$ and $D_b(z;S)$as foreground and background Gaussian distributions, respectively, whereztakes saliency value ranging between 0 and 1. Specifically,

$$D_f(z;S) = \frac{e^{-(\frac{z-\mu_f(S)}{\sigma_f(S)})^2}}{\sigma_f(S)\sqrt{(2\pi)}} \text{ and } D_b(z;S) = \frac{e^{-(\frac{z-\mu_b(S)}{\sigma_b(S)})^2}}{\sigma_b(S)\sqrt{(2\pi)}},$$

It is clear that the less the two distributions overlap with each other, the better the saliency map is, i.e., the foreground and background are more likely to be separable. In order to calculate such overlap,it is needed to figure out the intersecting point z^*. It can be obtained by equating the two functions, i.e. $D_f(z;S) = D_b(z;S)$, which finally leads to

$$z^2(\frac{1}{\sigma_b^2} - \frac{1}{\sigma_f^2}) - 2z(\frac{\mu_b}{\sigma_b^2} - \frac{\mu_f}{\sigma_f^2}) + \frac{\mu_b^2}{\sigma_b^2} - \frac{\mu_f^2}{\sigma_f^2} + 2log(\frac{\sigma_b}{\sigma_f}) = 0$$

When we solve the above quadratic equation, we get

$$z^* = \frac{\mu_b\sigma_f^2 - \mu_f\sigma_b^2}{\sigma_f^2 - \sigma_b^2} \pm \frac{\sigma_f\sigma_b}{\sigma_f^2 - \sigma_b^2} \times ((\mu_f - \mu_b)^2 - 2(\sigma_f^2 - \sigma_b^2)(log(\sigma_b) - log(\sigma_f)))^{\frac{1}{2}}$$

Having obtained z^*, overlap $L(S)$ can now be computed as

$$L(s) = \int_{z=0}^{z=z^*} D_f(z;S)dz + \int_{z=z^*}^{z=1} D_b(z;S)dz$$

And finally, separation measure ϕ for saliency map S is calculated as

$$\phi(S) = \frac{1}{1 + log_{10}(1 + \gamma L(S))}$$

where γ is set as number of bins used for representing the two distributions.

Reference : Quality-Guided Fusion-Based Co-Saliency Estimation for Image Co-Segmentation and Colocalization

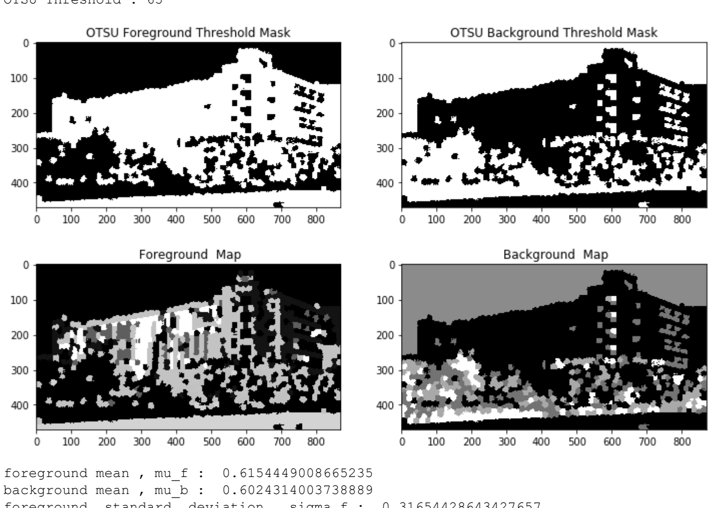
In [15]:
```python
def gaussian_distribution(x, mu, sigma):
    return (1/(sigma*math.sqrt(2*math.pi)))*np.exp(-(x/sigma - mu/sigma)**2)
```

In [16]:
```python
def separation_Measure(saliency_map):
    saliency_map = (saliency_map/np.max(saliency_map))*255
    thres = otsu(saliency_map.astype(int))
    print("OTSU Threshold :", thres)
    mask = copy.deepcopy(saliency_map)
    mask[mask < thres] = 0
    mask[mask >= thres] = 1
    fg = select_foreground(mask)
    if(fg == 1):
        foreground_mask = mask
        background_mask = 1 - foreground_mask
    else:
        foreground_mask = 1- mask
        background_mask = 1 - foreground_mask

    fig=plt.figure(figsize=(12, 12))
    columns = 2
    rows = 1
    fig.add_subplot(rows, columns, 1)
    plt.imshow(foreground_mask, cmap = 'gray')
    plt.title("OTSU Foreground Threshold Mask")
    fig.add_subplot(rows, columns, 2)
    plt.imshow(background_mask, cmap = 'gray')
    plt.title("OTSU Background Threshold Mask")
    plt.show()

    foreground_map = saliency_map*foreground_mask
    background_map = saliency_map*background_mask
    foreground_map = foreground_map/np.max(foreground_map)
    background_map = background_map/np.max(background_map)


    fig=plt.figure(figsize=(12, 12))
    columns = 2
    rows = 1
    fig.add_subplot(rows, columns, 1)
    plt.imshow(foreground_map, cmap = 'gray')
    plt.title(" Foreground  Map")
    fig.add_subplot(rows, columns, 2)
    plt.imshow(background_map, cmap = 'gray')
    plt.title(" Background  Map")
    plt.show()

    mu_f = np.mean(foreground_map[foreground_map > 0])
    sigma_f = np.std(foreground_map[foreground_map > 0])
    mu_b = np.mean(background_map[background_map > 0])
    sigma_b = np.std(background_map[background_map > 0])
    print("foreground mean , mu_f : ", mu_f)
    print("background mean , mu_b : ", mu_b)
    print("foreground  standard  deviation , sigma_f : ", sigma_f)
    print("background standard  deviation , sigma_b : ", sigma_b)

    fg_vals = foreground_map[foreground_map > 0].flatten()
    bg_vals = background_map[background_map > 0].flatten()
    fg_dist = scipy.stats.norm(mu_f, sigma_f**2).pdf(fg_vals)
    bg_dist = scipy.stats.norm(mu_b, sigma_b**2).pdf(bg_vals)

    z_star = (mu_b*sigma_f**2 - mu_f*sigma_b**2)/(sigma_f**2 - sigma_b**2)  +  (sigma_f*sigma_b/(sigma_
f**2 - sigma_b**2))*((mu_f - mu_b)**2 - 2*(sigma_f**2 - sigma_b**2)*(math.log(sigma_b) - math.log(sigma
_f)))**0.5
    print("z_star : ", z_star)
    L_s = quad(gaussian_distribution, 0, z_star, args=(mu_f,sigma_f))[0] + quad(gaussian_distribution,
z_star, 1, args=(mu_b,sigma_b))[0]
    print("L_s : ", L_s)
    gamma = 255
    phi_s = 1/(1 + math.log(1 + gamma*L_s,10))
    print("phi_s : ", phi_s)
    return phi_s
```

In [17]:
```python
print('\033[1mFor Contrast Cue \n  \033[0m')
sm_1 = separation_Measure(contrast_cue_image)
print('\n')
print('\033[1mFor Spatial Cue \n  \033[0m')
sm_2 = separation_Measure(spatial_cue_image)
```

For Contrast Cue

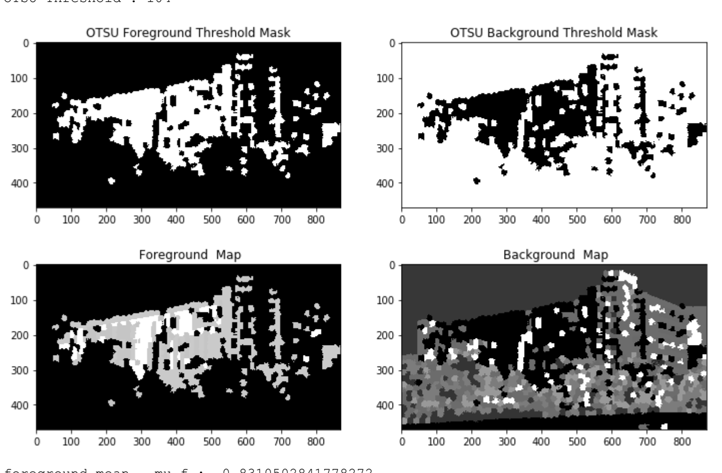
```
OTSU Threshold : 65
```




```
foreground mean , mu_f :  0.6154449008665235
background mean , mu_b :  0.6024314003738889
foreground  standard  deviation , sigma_f :  0.31654428643427657
background standard  deviation , sigma_b :  0.16304821932170072
z_star :  0.8170377079229726
L_s :  0.5969327805330965
phi_s :  0.3139412464280025
```

For Spatial Cue

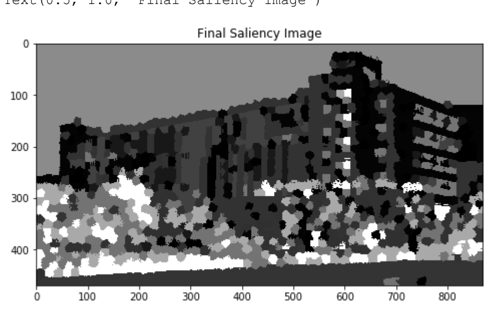
```
OTSU Threshold : 104
```




```
foreground mean , mu_f :  0.8310502841778272
background mean , mu_b :  0.40049340364985464
foreground  standard  deviation , sigma_f :  0.08659989638480675
background standard  deviation , sigma_b :  0.2199826233903447
z_star :  0.6717269028054917
L_s :  0.03195047572430588
phi_s :  0.5098668649735761
```

In [18]:
```python
final_saliency_image = np.zeros((m,n))
for i in range(m):
    for j in range(n):
        final_saliency_image[i,j] = sm_1*contrast_cues[clustered_img[i,j]] + sm_2*spatail_cues[clustere
d_img[i,j]]
plt.figure(figsize = (8,8))
plt.imshow(final_saliency_image, cmap = 'gray')
plt.title("Final Saliency Image")
```

Out[18]: Text(0.5, 1.0, 'Final Saliency Image')



$$---------------- END ----------------$$