Date: 21/04/2021

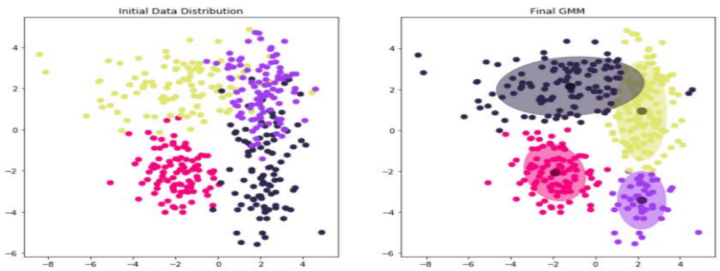
CS 6643 - Computer Vision, Spring 2021 **Prof.James Fishbaugh**

HOMEWORK II: GAUSSIAN MIXTURE MODELING, DEFORMABLE CONTOUR SEGMENTATION, PARALLEL STEREO CAMERAS

Pruthviraj R Patil, Computer Science, MS.
New York University, Tandon School of Engineering
Email: prp7650@nyu.edu

1. Gaussian Mixture Modeling

The company you work for has a client who insists that all delivered code is extensively documented. Your boss has an implementation of Gaussian mixture modeling that she wants to integrate into the final product, but it does not contain a single comment describing the functionality of the code. It is well known around the office that you are knowledgeable about the theory of Gaussian mixture modelling, so you have been given the job to thoroughly document the uncommented code.



Part A.

Starting from the attached code for Gaussian mixture modeling, comment and document the code to bring it up to the high standards of the client. Be thorough, and reference relevant equations and the EM algorithm. Include the commented code in the appendix.

SOLUTION:

(Code from where the commenting provision was given)

```
#########################################################################
# START COMMENTING HERE
#########################################################################

#initializing number of Gaussians
k = 4
```

```
#assign number of observations per Gaussiann
num_per_class = np.zeros((k,1), np.float32)

# iterate over all the data points to find number of data points
assigned per Gaussian
for c in range(0, len(all_data)):
    cur_class = classes[c]
    num_per_class[cur_class] += 1

# initialize pi (mixing weights) values as the fraction of data to
utilize them in EM algorithm
piks = num_per_class/len(all_data)

#initialize the dictionary to assign the cluster number to the
centroids
means = dict()
#extract centroids and asssign them to their cluster number in the
dictionary "means"
for c in range(0, k):
    means[c] = centroids[c]

#initialize the covariance dictionary to map cluster number with
empty matrix of
#covariance that can be modified later
covariances = dict()
for c in range(0, k):
    covariances[c] = np.array([[0,0], [0,0]], np.float32)


#for all the Gaussian centers, calculate their covariance with the
data assigned
#to their respective clusters
for c in range(0, k):
    for d in range(0, len(all_data)):
        #the observation that only belongs to the corresponding
cluster is considered to contribute to the covariance matrix of the
respective centroid
        if (classes[d] != c):
            continue
```

```
        #find the difference between the data point and its cluster
centroid
        diff = (all_data[d] - means[c])
        #convert it into column matrix
        diff.shape = (len(all_data[d]), 1)
        #calculate the covariance by taking the dott product of the
difference matrix and it's transpose
        #add the covariance to the total covariance matrix to
aggregate
        covariances[c] += diff.dot(diff.T)
    #divide the covariances with number of data points to normalize
    covariances[c] /= (num_per_class[c]-1)

fig1 = plt.figure(figsize = (7,7))
splot = plt.subplot(1, 1, 1)

#print data from every gaussian
for i in range(0, len(all_data)):
    plt.scatter(all_data[i,0], all_data[i,1], s=50, c='k')

#print the initial surrondings of the data with respect to their
cluster centroids
for i in range(0, k):
    cur_mean = means[i]
    plt.scatter(cur_mean[0], cur_mean[1], s=120, c='r')
    plt.title('Initialization for GMM Fitting')

    # calculate the eigenvalues and eigenvectors using covariance
matrix per gaussian
    v, w = np.linalg.eigh(covariances[i])
    #eigen vectors are calculated because they illustrate the
principle line of force where the data is pointing towards
    #whereas the eigen values are the coefficients attached to the
eigen vectors
    v = 2. * np.sqrt(2.) * np.sqrt(v)
    u = w[0] / np.linalg.norm(w[0])

    angle = np.arctan(u[1] / u[0])
```

```
    angle = 180. * angle / np.pi
    ell = mpl.patches.Ellipse(cur_mean, v[0], v[1], 180. + angle,
color=colors[i])
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.5)
    splot.add_artist(ell)
```


```
#initialize the number of iterations the process to be run
iters = 1
max_iters = 200
#initialize the log likely hood to a random value
prev_log_likelihood = 1e6
#initialize the binary counter "Converged". If this is toggled true
in the future, then the process stops
converged = False
convergence_epsilon = 1e-3

#initialize Gaussian probability matrix which illustrates the
```

```
probability of every element to belong to the each and every cluster
#this has the dimensions of (Number of data points, number of
clusters)
gammaks = np.zeros((len(all_data),k), np.float32)

while iters < max_iters and not converged:
    #iterate over all the data and find the probability of their
belonging to every Gaussian and store them in the Gammak matrix
    for d in range(0, len(all_data)):
        for c in range(0, k):
            gammaks[d,c] = Gaussian_ND(all_data[d], means[c],
covariances[c])*piks[c]

    #iterate over all the probabilities arrays of every data point
and normalize them make their sum equivalant to 1 (normalized)
    for d in range(0, len(all_data)):
        gammaks[d,:] /= np.sum(gammaks[d,:])

    #intialize Nk that determines the number of data points assigned
to the particular cluster
    Nk = np.zeros((k,1), np.float)
    for c in range(0,k):
        Nk[c] = np.sum(gammaks[:,c])

    #initlaize the dictionaries the new/updated covariance matrices,
new means(centroids)
    new_means = dict()
    new_covariances = dict()
    for i in range(0, k):
        new_means[i] = np.array((0,0), np.float32)
        new_covariances[i] = np.array([[0,0], [0,0]], np.float32)

    #iterate over all the clusters to access their corresponding
elements
    for c in range(0,k):
        for d in range(0, len(all_data)):
            #calculate the new means using the Nk (calculated number
of elements in the respective clusters) and gammak probabilities
matrix
```

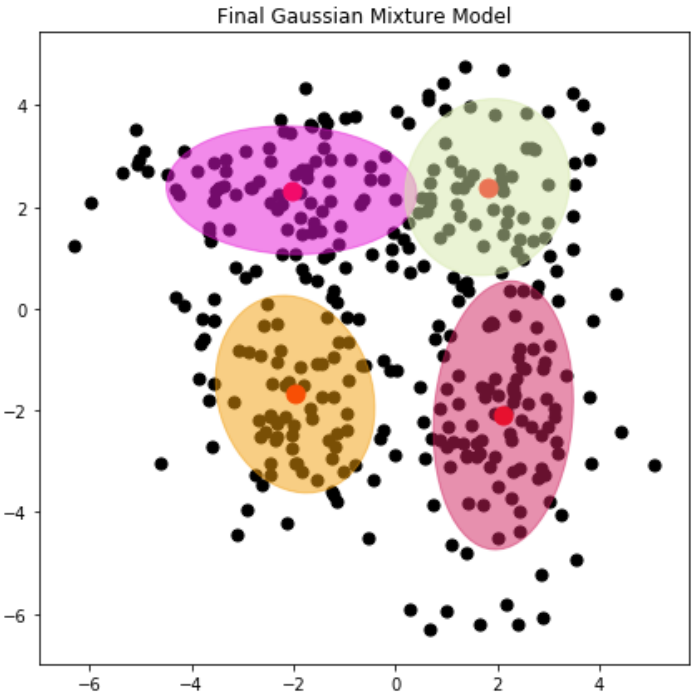
```
            new_means[c] += (1/Nk[c]) * gammaks[d,c]*all_data[d]
            #rewrite the same process of calculating the covariance
that is done above
            diff = (all_data[d] - means[c])
            diff.shape = (len(all_data[d]), 1)
            new_covariances[c] +=
(1/Nk[c])*gammaks[d,c]*diff.dot(diff.T)

    #Pick new mixing weights for updating using EM algorithm
    new_piks = np.zeros((k,1))
    for c in range(0,k):
        new_piks[c] = Nk[c]/len(all_data)

    means = new_means
    covariances = new_covariances
    piks = new_piks

    #initlaize log likely hood = 0
    log_likelihood = 0

    #iterating over all the data, calculate the updated proability of
each data point to belong to every cluster using their covariance
matrices, and means(centroids)
    for d in range(0, len(all_data)):
        k_sum = 0
        for c in range(0, k):
            k_sum += piks[c] * Gaussian_ND(all_data[d], means[c],
covariances[c])
        log_likelihood += -np.log(k_sum)

    #print the updation information evert epoch
    iteration_info = 'Iteration %0.3d: Log likelihood = %0.3f'
%(iters, log_likelihood)
    print(iteration_info)

    if (np.abs(log_likelihood - prev_log_likelihood) <
convergence_epsilon):
        print('   *** Optimization has converged ***')
        converged = True
```

```
    prev_log_likelihood = log_likelihood

    iters += 1
```

```
Iteration 029: Log likelihood = 1557.263
Iteration 030: Log likelihood = 1557.260
Iteration 031: Log likelihood = 1557.257
Iteration 032: Log likelihood = 1557.255
Iteration 033: Log likelihood = 1557.253
Iteration 034: Log likelihood = 1557.251
Iteration 035: Log likelihood = 1557.250
Iteration 036: Log likelihood = 1557.248
Iteration 037: Log likelihood = 1557.247
Iteration 038: Log likelihood = 1557.246
Iteration 039: Log likelihood = 1557.245
   *** Optimization has converged ***
```

```
fig1 = plt.figure(figsize = (7,7))
splot = plt.subplot(1, 1, 1)

#print data from every gaussian
for i in range(0, len(all_data)):
    plt.scatter(all_data[i,0], all_data[i,1], s=50, c='k')

#print the final surrondings of the data with respect to their
cluster centroids
for i in range(0, k):

    cur_mean = means[i]
    plt.scatter(cur_mean[0], cur_mean[1], s=120, c='r')
    plt.title('Final Gaussian Mixture Model')

    # calculate the eigenvalues and eigenvectors using covariance
matrix per gaussian
    v, w = np.linalg.eigh(covariances[i])
    #eigen vectors are calculated because they illustrate the
principle line of force where the data is pointing towards
    #whereas the eigen values are the coefficients attached to the
```

```
eigen vectors
    v = 2. * np.sqrt(2.) * np.sqrt(v)
    u = w[0] / np.linalg.norm(w[0])

    angle = np.arctan(u[1] / u[0])
    angle = 180. * angle / np.pi
    ell = mpl.patches.Ellipse(cur_mean, v[0], v[1], 180. + angle,
color=colors[i])
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.5)
    splot.add_artist(ell)
```



Part B.

The client also wants the option to have a hard clustering/labeling. In your report, describe a method (of your choice) to use the output of the Gaussian mixture model to assign hard class labels to every data point. Implement your method and show and discuss the results on random data generated by the code (by showing a plot of data points colored by class/cluster like the figure above).

SOLUTION:

The hard clustering is nothing but direct assignment of the cluster number to the observation. This can be done by harnessing the Gaussian matrix (‘gammak”) that contains the probabilities of every data point belonging to every cluster. Hence, taking the probability array of every data point, we assign that particular cluster number that has max probability of having that data point to that particular data point.

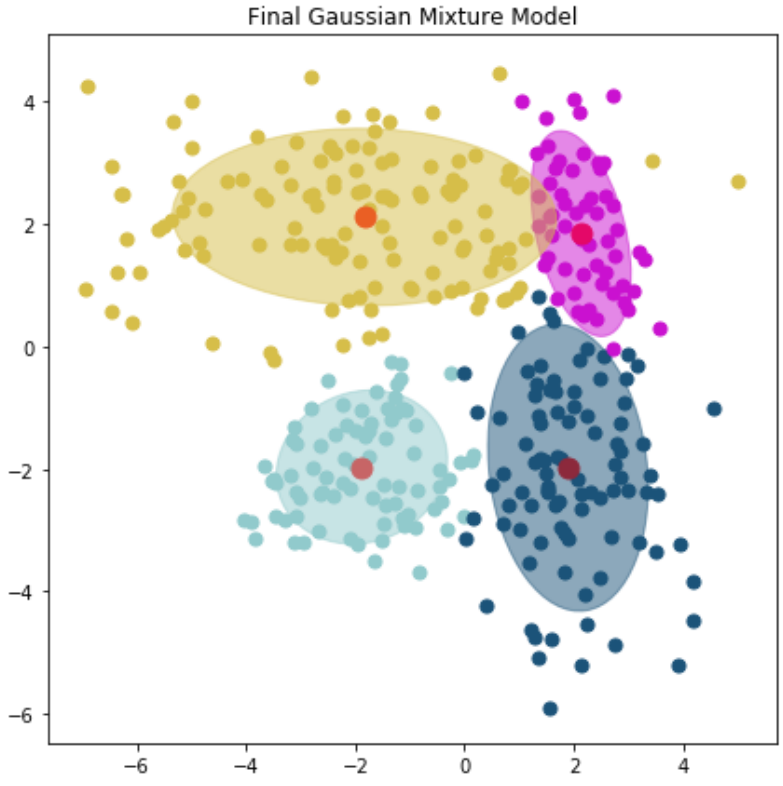
PYTHON CODE with proper comments:

```
def find_cluster_elements(cluster_number, classes, all_data):
    #status: complete
    #take the particular class and seggregate data point with respect
to that class
    seggregated_data_x=[]
    seggregated_data_y=[]
    for i in range(len(classes)):
        if(classes[i]==cluster_number):
            seggregated_data_x.append(all_data[i][0])
            seggregated_data_y.append(all_data[i][1])
    return seggregated_data_x, seggregated_data_y

#---------------------------------------------------------------------#
#---------------------------------------------------------------------#

#Gammaks is the gaussians that consider of data points and their
probabilities of th
class_numbers=[]
for cluster_probabilities in gammaks:
    max_val, max_idx = max((val, idx) for (idx, val) in
enumerate(cluster_probabilities))
    class_numbers.append(max_idx)

#find the number of elements in every gaussian
counts=np.unique(class_numbers, return_counts=True)

# Random colors for the clusters
colors = ["#"+''.join([random.choice('0123456789ABCDEF') for j in
range(6)]) for i in range(gt_k)]
```

```
seggregated_data_x=[]
seggregated_data_y=[]

for class_number in range(4):
    #gather data for every class number
    x, y=find_cluster_elements(class_number, class_numbers, all_data)
    seggregated_data_x.append(x)
    seggregated_data_y.append(y)

np.random.shuffle(seggregated_data)
fig1 = plt.figure(figsize = (7,7))
splot = plt.subplot(1, 1, 1)

#print data from every gaussian using the seggregated data
plt.scatter(seggregated_data_x[0], seggregated_data_y[0], s=50,
c=colors[0])
plt.scatter(seggregated_data_x[1], seggregated_data_y[1], s=50,
c=colors[1])
plt.scatter(seggregated_data_x[2], seggregated_data_y[2], s=50,
c=colors[2])
plt.scatter(seggregated_data_x[3], seggregated_data_y[3], s=50,
c=colors[3])


#print the final surrondings of the data with respect to their
cluster centroids
for i in range(0, k):
    cur_mean = means[i]
    plt.scatter(cur_mean[0], cur_mean[1], s=120, c='r')
    plt.title('Final Gaussian Mixture Model')

    # calculate the eigenvalues and eigenvectors using covariance
matrix per gaussian
    v, w = np.linalg.eigh(covariances[i])
    #eigen vectors are calculated because they illustrate the
principle line of force where the data is pointing towards
    #whereas the eigen values are the coefficients attached to the
eigen vectors
    v = 2. * np.sqrt(2.) * np.sqrt(v)
```

```
    u = w[0] / np.linalg.norm(w[0])

    angle = np.arctan(u[1] / u[0])
    angle = 180. * angle / np.pi
    ell = mpl.patches.Ellipse(cur_mean, v[0], v[1], 180. + angle,
color=colors[i])
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.5)
    splot.add_artist(ell)
```

OUTPUT:



2. Deformable Contour Segmentation

Implement deformable contour segmentation based on the following algorithm. A couple of hints:

• While working on your implementation, you can test with the included 'ellipse.png' which represents an ideal image with strong edges and no noise.

• You can immediately update the snake point Pi you are working on, there is no need to keep a copy and update all at once.

• Don't forget to Gaussian blur before computing the gradients for generating the gradient magnitude Eimage. The amount of blurring may have an impact on your results.

• Think about how you want to handle the relationship between P1 and PN. In other words, what do Pi-1 and Pi+1 mean in these cases?

Part: A

Using the attached image 'implied_contour.png', find parameter settings which succeed to segment the implied contour, and also find parameter settings which fail.

- Include an analysis of the parameter settings. Why did they lead to failure or success?
- Include images of the final segmentations shown with the original image, like the figure on the right.

SOLUTION:

The parameters required to run the algorithm are nothing but the parameters on which the output of the process depends on mostly. They are Standard deviation (of gaussian kernel), Alpha, beta and gamma constants - inorder to calculate the Energy function in every epoch inorder to adjust the snake. Energy consists of two components. i.e. Internal energy and External energy. Here the Internal energy encourages the prior shape preferences. They are : smoothness, elasticity, particular known shape. Whereas, the External energy encourages the snake to fit on places where image structures exist, e.g., edges.

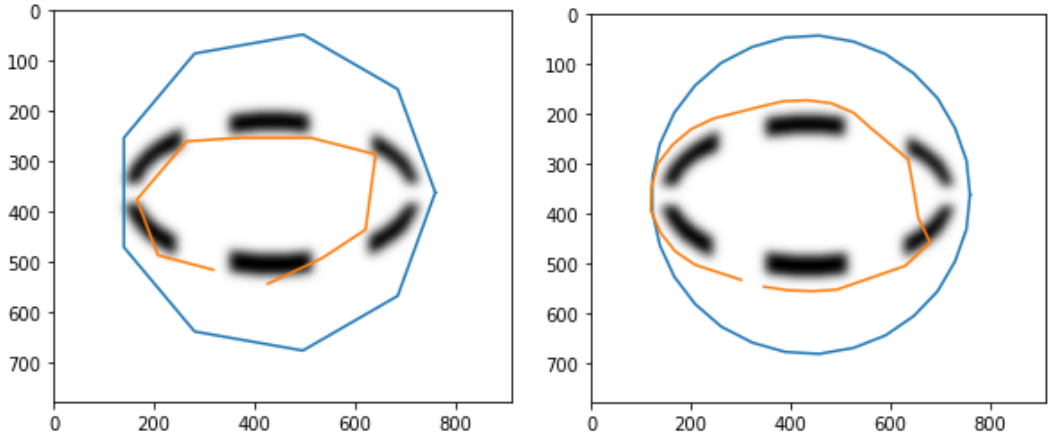


Figure 2.1 Output with n snake points= 10, 30 respectively. alpha=0.1, beta= 0.05, gamma= 0.5

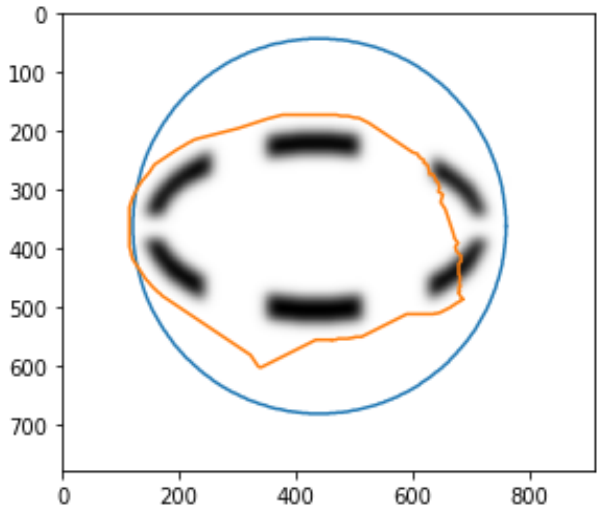


Figure 2.2 Output with n snake points= 500. alpha=0.1, beta= 0.05, gamma= 0.5

The figures 2.1 and 2.2 shows the output of the process with alpha=0.1, beta= 0.05, gamma= 0.5. Here, the alpha, and beta components are considered to be lesser in value because the lower the curvature, smoothness this helps in dealing with the missing data. The image has the structure of an almost complete oval. Hence, the locations between the points on the oval are tried to fill by the snake.

The snake however fails in some regions to cover the areas because this is the greedy approach and hence, a dynamic programming approach of this process can boost up to get a proper output. Moreover, the gaussian standard deviation value too affects the output of the snake deformation too as the external energy directly depends on the magnitude of the intensities which are indirectly varied when the gaussian with different standard deviation values are convolved over the image.

Part B:

Using the attached image ‘rectangle.png’, present your best results using: o An initialization with 10 snake points o An initialization with 30 snake points o An initialization with 500 snake points o Show all results and discuss what you observe.

SOLUTION:

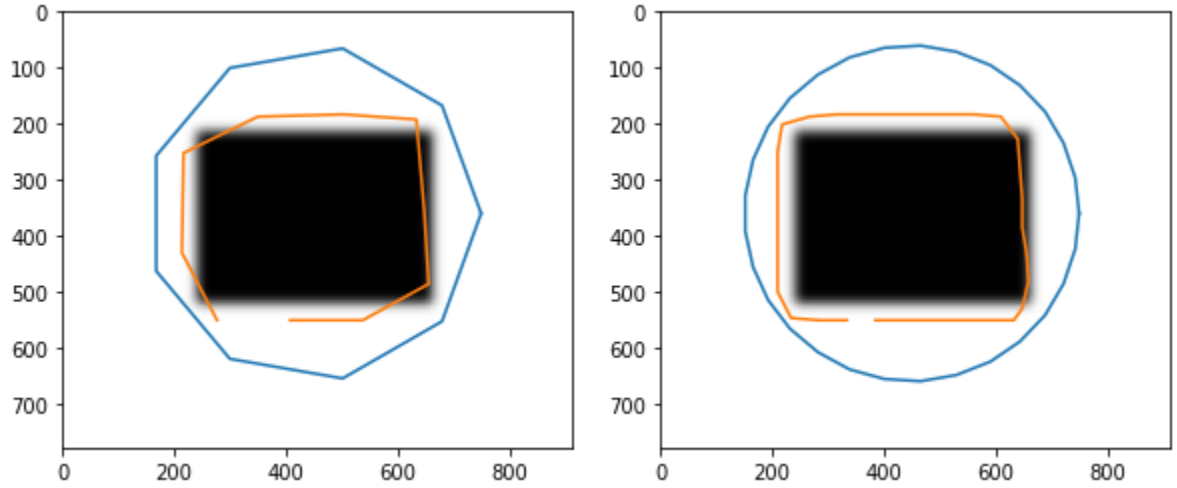


Figure 2.3 Output with n snake points= 10, 30 respectively. alpha=0.08, beta= 0.001, gamma= 0.5

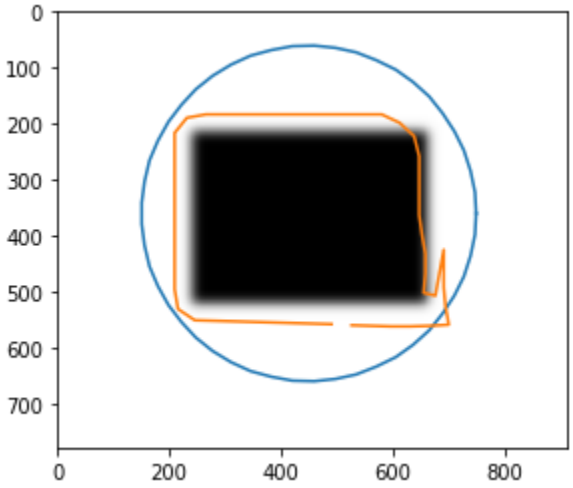


Figure 2.4 Output with n snake points= 500. alpha=0.08, beta= 0.001, gamma= 0.5.

We know that the Energy consists of two components. i.e. Internal energy and External energy. Here the Internal energy encourages the prior shape preferences. They are : smoothness, elasticity, particular known shape. Here, the alpha desires to keep the contour closer to the image. Whereas the beta presumes that the snake has to be circular in manner. That is why the beta value is taken to be lesser than the rest and alpha is taken to be bigger than beta. Whereas gamma deals with external energy contributes the most to the total energy.

Part C:

Using the attached image 'astronaut.png', show your best result in segmenting the head and hair (as in the figure on the right). Discuss your process for finding the parameters which gave your best result

SOLUTION:

In this case, the alpha beta gamma values are as follows: (0.15, 0.015, 0.001) respectively. The alpha values are taken to be the maximum because the internal energy that deals with the Continuity of the snake contour depends a lot when we deal with the non standard figure.

However, the elasticity of the snake (beta) is kept low but larger than the gamma value. The external energy contributes very less because the background of the astronaut is grey in color whereas the hair(head) when grayscale tends to be almost the same in shade. Thus the magnitude at that patch contributes a bit less in this case.

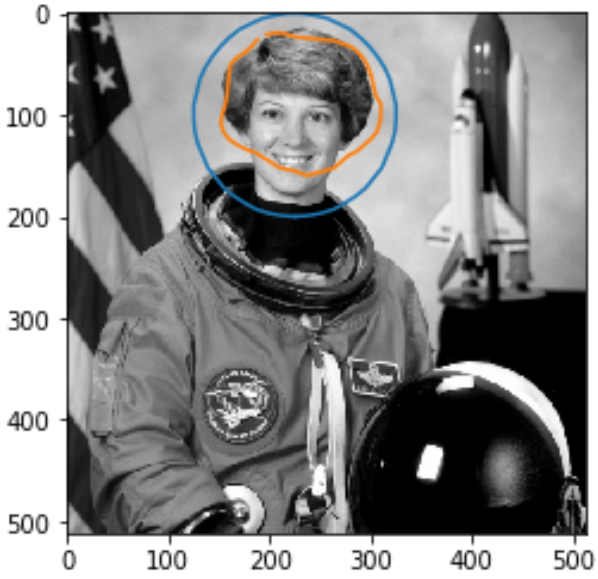


Figure 2.5 Output with n snake points= 50. alpha=0.15, beta= 0.0015, gamma= 0.001.

PYTHON CODE:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import math

#read image using path
def read_img(path):
    #status:complete
```

```
    img=cv2.imread(path)
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_gray = ((img_gray - np.min(img_gray)) * (1/(np.max(img_gray)
- np.min(img_gray)) * 255)).astype('uint8')
    return img_gray

def CommonUtil(image, init_x_size, init_y_size, pad_size, Mask):
    #status: complete
    #task: utility function to multiply kernel to image patch
    #output: adjusted image

    output_image= np.zeros((init_x_size, init_y_size))
    out_row=0
    out_col=0

    for x_pivot in range(pad_size, init_x_size+pad_size):
        out_col=0
        for y_pivot in range(pad_size, init_y_size+pad_size):
            patch=image[x_pivot-pad_size: x_pivot+pad_size+1,
y_pivot-pad_size: y_pivot+pad_size+1]
            output_image[out_row][out_col] = np.sum(np.multiply(Mask,
patch))
            out_col+=1
        out_row+=1
    return output_image


def Convolve(image, Mask):
    #status: complete
    #task: perform Convolution
    #parameters: image, Mask=kernel
    #returns: convolved image
    (init_x_size, init_y_size)=image.shape
    Mask=np.flipud(np.fliplr(Mask))
    pad_size=math.floor((Mask.shape[0])/2)
    image=np.pad(image, pad_size, mode='constant')
    output_image= CommonUtil(image, init_x_size, init_y_size,
pad_size, Mask)
    return output_image
```

```
def findMagnitude(xGrad, yGrad):
    #status: complete
    #params: xGrad, yGrad
    #returns: magnitude using the x and y gradients

    mags=np.zeros(xGrad.shape, np.float32)
#     print(mags.shape)
    x_len=xGrad.shape[0]
    y_len=xGrad.shape[1]
    for i in range(x_len):
        for j in range(y_len):
            mags[i][j]=np.sqrt(xGrad[i][j]**2 + yGrad[i][j]**2)%255
    return mags


def create_snake(center_x, center_y, radius, num_pts):
    #status: complete
    samples = np.linspace(0, 2*math.pi, num_pts)
    snake = np.zeros((num_pts,2))
    snake[:,0] = np.round(radius * np.cos(samples) + center_x)
    snake[:,1] = np.round(radius * np.sin(samples) + center_y)
    return snake[:snake.shape[0],0], snake[:snake.shape[0],1]


def find_avg_distance(distances):
    #status: complete
    summate=0
    for i in range(len(distances[0])-1):
        x1=distances[0][i]
        x2=distances[0][i+1]
        y1=distances[1][i]
        y2=distances[1][i+1]
        temp=math.sqrt((x2-x1)**2 + (y2-y1)**2)
        summate+=temp
    avg=summate/len(distances[0])
    return avg
```

```
def find_points_patch(point):
    #status: complete
    temp_x, temp_y=(-1, -1)
    patch_points_x=[]
    patch_points_y=[]
    for i in range(3):
        temp_y=-1
        for j in range(3):
            patch_points_x.append(point[0]+temp_x)
            patch_points_y.append(point[1]+temp_y)
            temp_y+=1
        temp_x+=1
    return np.array(patch_points_x), np.array(patch_points_y)


def find_image_patch(point, image):
    #status: complete
    patch=np.zeros((3, 3))
    temp_x, temp_y=(-1, -1)
    for i in range(patch.shape[0]):
        temp_y=-1
        for j in range(patch.shape[1]):
            if(((point[0]+temp_x)>= image.shape[0]) or
((point[1]+temp_y)>=image.shape[1])):
                continue

patch[i][j]=image[int(point[0]+temp_x)][int(point[1]+temp_y)]
            temp_y+=1
        temp_x+=1
    return patch


def find_energy(avg_cor, patch, point_x, point_y, points_x, points_y,
alpha, beta, gamma, current, prev, next_pt):
    #status: complete
    #find external energy
    E_external= -sum(sum(patch))
    x1=points_x[prev]
```

```
    x2=point_x
    y1=points_y[prev]
    y2=point_y
    temp_1 = math.sqrt((x2-x1)**2 + (y2-y1)**2)
    int_temp1 = (avg_cor-temp_1)**2
    #find internal energy
    a=(points_x[prev]-point_x)
    b=(points_x[next_pt]-point_x)
    int_temp2_x=(a+b)
    a=(points_y[prev]-point_y)
    b=(points_y[next_pt]-point_y)
    int_temp2_y=(a+b)
    int_temp2= int_temp2_x**2 +int_temp2_y**2
    Energy= alpha*int_temp1 + beta*int_temp2 + gamma*E_external
    return Energy, current


def snake_iteration(image, x, y, alpha, beta, gamma, iterations):
    #status: complete
    x_prewitt_filter=np.array([[-1, -1, -1]])
    y_prewitt_filter=(np.array([[1, 1, 1]]).T)
    conv_im_x=Convolve(image, x_prewitt_filter)
    conv_im_y=Convolve(image, y_prewitt_filter)
    mag_image=findMagnitude(conv_im_x, conv_im_y)

    for epoch in range(iterations):
        point_count, n_points=(len(x)-1, len(x))
        for i in range(n_points):
            avg_cor=find_avg_distance([x, y])
            prev=point_count%n_points
            current=(point_count+1)%n_points
            next_pt=(point_count+2)%n_points
            point_count+=1
            patch_point_x, patch_point_y =
find_points_patch(np.array([x[i], y[i]]))
            Energy_store=[]
            for point in range(patch_point_x.shape[0]):
                image_patch =
find_image_patch(np.array([patch_point_x[point],
```

```
patch_point_y[point]]), mag_image)
                Energy, current=find_energy(avg_cor, image_patch,
int(patch_point_x[point]), int(patch_point_y[point]), x, y, alpha,
beta, gamma, current, prev, next_pt)
                Energy_store.append(Energy)
            min_val, min_idx = min((val, idx) for (idx, val) in
enumerate(Energy_store))
            x[current]=patch_point_x[min_idx]
            y[current]=patch_point_y[min_idx]
    return x, y


def run_snake_process(image, iterations, points_x, points_y, alpha,
beta, gamma):
   #status: complete
    return snake_iteration(image, points_x, points_y, alpha, beta,
gamma, iterations)


def main():
    image=read_img("astronaut.png")
    out_image = cv2.GaussianBlur(image,(101,101),10)
#     snake_contour_x, snake_contour_y= create_snake(441, 367, 320,
10) #ellipse
#     snake_contour_x, snake_contour_y= create_snake(441, 362, 320,
15) #impliedContour
#     snake_contour_x, snake_contour_y= create_snake(450, 360, 300,
50) #rectangle
#     snake_contour_x, snake_contour_y= constructRectangle(122, 614,
167, 744)
    snake_contour_x, snake_contour_y= create_snake(100, 225, 100, 50)
#astronaut
#     epoches=50
    epoches=250
    xx, yy=(snake_contour_x.copy(), snake_contour_y.copy())
#     plt.plot(yy, xx)

#     alpha, beta, gamma=(0.002, 0.01, 0.0001)#ellipse
#     alpha, beta, gamma=(0.5, 0.05, 0.9)#implied
```

```
#     alpha, beta, gamma=(0.02, 0.001, 0.5) #rectangle
#     alpha, beta, gamma=(0.8, 0.001, 0.5) #rectangle
    alpha, beta, gamma=(0.15, 0.0015, 0.001) #astronaut
    snake_contour_x_, snake_contour_y_=run_snake_process(out_image,
epoches, snake_contour_x, snake_contour_y, alpha, beta, gamma)
    plt.imshow(image, cmap="gray")
    plt.plot(yy, xx)
    plt.plot(snake_contour_y_, snake_contour_x_)


if __name__ == "__main__":
    main()
```

3.Parallel Stereo Cameras

Consider a parallel camera setup with identical cameras separated by 300 mm. Each camera has a focal length measured in pixels of 4300 pixels. This roughly corresponds to a camera with a width of 4032 pixels and a field of view of ~50 degrees, corresponding to a focal length of ~50 mm. You will use focal length in pixels for this exercise.

• Plot disparity (x-axis) vs depth Z (y-axis) with disparity in the range [0, 200] pixels and Z in the range [0.001, 500000] mm. Use a plotting program rather than a hand drawn sketch.

• Generate two additional plots, with disparity in the range [0, 50] and disparity in the range [0, 20], but keeping Z fixed in the range [0.001, 500000] mm (also limit the axes to these values).

• In all 3 plots, label the x and y axis and include units.

• Describe the shape of the plot. With the shape of the plot and the disparity equation in mind, describe the relationship between disparity and depth Z

SOLUTION:

The output of disparity vs depth plot is shown in the figures 3.1, 3.2, 3.3. The relationship between disparity and depth Z is given by the formula:

$$Z = \frac{focal\ length * T}{disparity}$$

Where T is the distance between the identical cameras. In the question, it is given that the focal length is 4300 pixels and T is 300 mm. The value of Z is varied from [0.001 to 500000] and the value of disparity is varied differently in different sections from 0 to 200, 0 to 20, and 0 to 50 respectively. Hence, the units of Z is ($\frac{pixels * mm}{pixels}$)= pixels. Whereas the unit of disparity is pixels. The output suggests that the disparity and depth are inversely proportional. Furthermore, if the disparity tends more towards 0, higher the rate of increase in depth persists. This can be seen in the graph where the range of disparity is [0, 200]. But, in the case where the range of disparity is taken in the range of [0, 20] or [0, 50], the rate of change in depth in y axis is comparitively less. However, the disparity is inversely proportional to depth.

PYTHON CODE:

```
def disparityDepth(focal_length, T, depth_range, disparity_range):
    #status: complete
    disparity = np.linspace(0, disparity_range, disparity_range)
    y = ((focal_length)*T)/disparity
```

```
    disparity_plot=[]
    depth_plot=[]
    for i in range(len(y)):
        if(y[i]<depth_range):
            disparity_plot.append(disparity[i])
            depth_plot.append(y[i])
    plt.plot(disparity_plot, depth_plot, '-', linewidth=4)
    plt.xlabel("disparity")
    plt.ylabel("depth")
    plt.title("disparity vs depth")
```

OUTPUT:

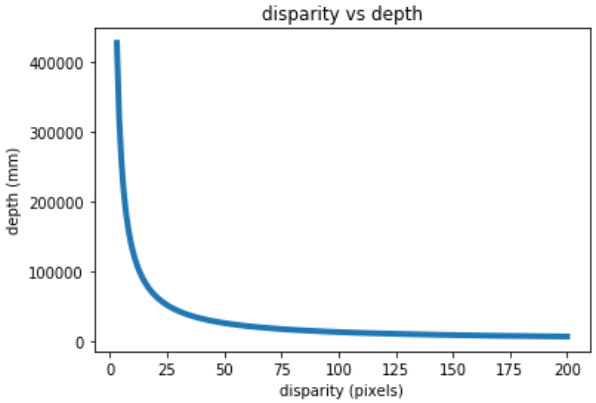


Figure: 3.1: Depth(upto ~ 500000 mm) vs disparity ([0, 200])

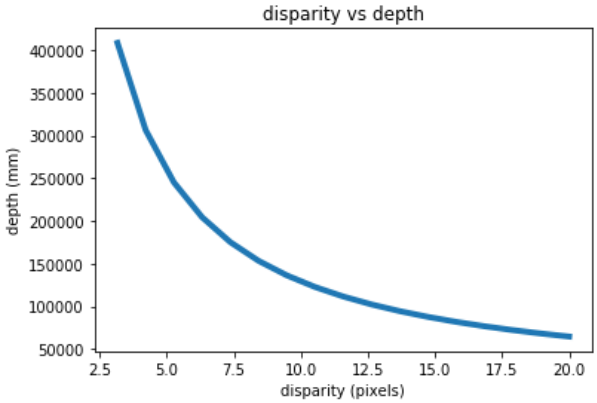


Figure 3.2. Depth (upto ~ 500000 mm) vs disparity ([0, 20])

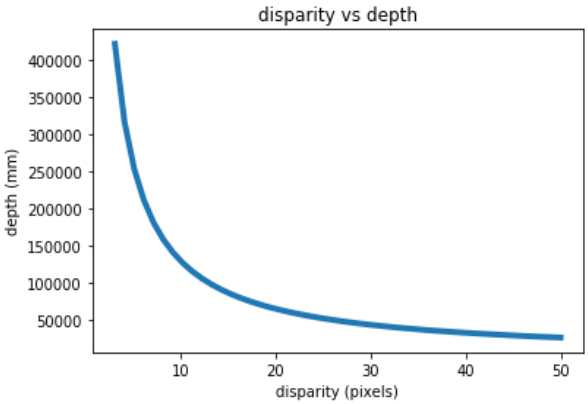


Figure 3.3. Depth (upto ~ 500000 mm) vs disparity ([0, 50])