CSE 573 Project 1

- NIKHIL SRIHARI
50291966
nikhilsr

1. IMAGE FEATURES AND HOMOGRAPHY:

Code:

```
importnumpyas np
import cv2
import random

UBIT ='nikhilsr'
np.random.seed(sum([ord(c)for c in UBIT]))

mountain1ImageLocation ='./proj2_data/data/mountain1.jpg'
mountain2ImageLocation ='./proj2_data/data/mountain2.jpg'


defwriteImage(img,outputFileName):
    cv2.imwrite(outputFileName,img)
return1


defreadImage(imageLocation):
img= cv2.imread(imageLocation,1)
returnimg


defmain():
print("Task 1 :")
print(" Task 1.1 : ")
    img1 =readImage(mountain1ImageLocation)
    img2 =readImage(mountain2ImageLocation)
    img1_ = img1.copy()
    img2_ = img2.copy()
    img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    sift = cv2.xfeatures2d.SIFT_create()
    img1_keyPoints, img1_desc =sift.detectAndCompute(img1_gray,None)
    img2_keyPoints, img2_desc =sift.detectAndCompute(img2_gray,None)
    img1_withHighlightedKeyPoints = cv2.drawKeypoints(img1_gray, img1_keyPoints, img1, 
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    img2_withHighlightedKeyPoints = cv2.drawKeypoints(img2_gray, img2_keyPoints, img2, 
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
writeImage(img1_withHighlightedKeyPoints,"task1_sift1.jpg")
writeImage(img2_withHighlightedKeyPoints,"task1_sift2.jpg")
print("     Task 1.1 Completed. Keypoints have been detected.")
print(" Task 1.2 : ")
    K =2
    M2NRatio =0.75
    FLANN_INDEX_KDTREE =0
numOfChecks=100
index_params=dict(algorithm = FLANN_INDEX_KDTREE, trees =5)
search_params=dict(checks =numOfChecks)
flann= cv2.FlannBasedMatcher(index_params,search_params)
    matches =flann.knnMatch(img1_desc, img2_desc, k=K)
goodMatches=[]
for m, n in matches:
if(m.distance<(M2NRatio *n.distance)):
goodMatches.append(m)
knnMatchedImg= cv2.drawMatches(img1_withHighlightedKeyPoints, img1_keyPoints, 
img2_withHighlightedKeyPoints, img2_keyPoints,goodMatches,None, flags=2)
writeImage(knnMatchedImg,"task1_matches_knn.jpg")
print("     Task 1.2 Completed. Good Keypoint matches have been mapped.")
```

```
print(" Task 1.3 : ")
    H =None
    MIN_MATCH_COUNT=10
if(len(goodMatches)>= MIN_MATCH_COUNT):
src_pts=np.float32([ img1_keyPoints[m.queryIdx].ptfor mingoodMatches]).reshape(-1,1,2)
dst_pts=np.float32([ img2_keyPoints[m.trainIdx].ptfor mingoodMatches]).reshape(-1,1,2)
        H, mask = cv2.findHomography(src_pts,dst_pts, cv2.RANSAC,5.0)
print("     The Homography matrix H : ")
print("     "+str(H))
else:
print("     Not enough good matches were found with minimum match count being set at
"+str(MIN_MATCH_COUNT))
print("     Task 1.3 Completed.")
print(" Task 1.4 : ")
    MATCHES_TO_BE_MAPPED =10
    MIN_MATCH_COUNT =10
    goodMatches1 =random.sample(goodMatches, MATCHES_TO_BE_MAPPED)
if(len(goodMatches1)>= MIN_MATCH_COUNT):
        src_pts1 =np.float32([ img1_keyPoints[m.queryIdx].ptfor min goodMatches1
]).reshape(-1,1,2)
        dst_pts1 =np.float32([ img2_keyPoints[m.trainIdx].ptfor min goodMatches1
]).reshape(-1,1,2)
        H1, mask1 = cv2.findHomography(src_pts1, dst_pts1, cv2.RANSAC,5.0)
        matchesMask1 = mask1.ravel().tolist()
        height1, width1, channels1 = img1_withHighlightedKeyPoints.shape
        pts1 =np.float32([[0,0],[0,height1-1],[width1-1,height1-1],[width1-
1,0]]).reshape(-1,1,2)
        dst1 = cv2.perspectiveTransform(pts1,H1)
else:
print("     Not enough good matches were found with minimum match count being set at
"+str(MIN_MATCH_COUNT))
        matchesMask1 =None
draw_params=dict(matchColor=(0,255,255),singlePointColor=None,matchesMask=
matchesMask1, flags =2)
matchedImg= cv2.drawMatches(img1_withHighlightedKeyPoints, img1_keyPoints,
img2_withHighlightedKeyPoints, img2_keyPoints, goodMatches1,None,**draw_params)
writeImage(matchedImg,"task1_matches.jpg")
print("     Task 1.4 Completed.")
print(" Task 1.5 : ")
    img1_height, img1_width = img1_.shape[:2]
    img1_pixels
=np.float32([[0,0],[0,img1_height],[img1_width,img1_height],[img1_width,0]]).reshape(-
1,1,2)
    img2_height, img2_width = img2_.shape[:2]
    img2_pixels
=np.float32([[0,0],[0,img2_height],[img2_width,img2_height],[img2_width,0]]).reshape(-
1,1,2)
    img2_pixels = cv2.perspectiveTransform(img2_pixels, H)
warpedImg_pixels=np.concatenate((img1_pixels, img2_pixels), axis=0)
[xMin,yMin]= np.int32(warpedImg_pixels.min(axis=0).ravel()-0.5)
[xMax,yMax]= np.int32(warpedImg_pixels.max(axis=0).ravel()+0.5)
    t =[-xMin,-yMin]
Ht=np.array([[1,0,t[0]],[0,1,t[1]],[0,0,1]])
warpedImg= cv2.warpPerspective(img2_, Ht.dot(H),(xMax-xMin,yMax-yMin))
warpedImg[ t[1]:(img1_height+t[1]), t[0]:(img1_width+t[0])]= img1_
writeImage(warpedImg,"task1_pano.jpg")
print("     Task 1.5 Completed.")


main()
```

Homography Matrix:

```
    The Homography matrix H :
[[ 1.58799966e+00 -2.91541838e-01 -3.95539425e+02]
[ 4.48199617e-01  1.43139761e+00 -1.90370131e+02]
[ 1.20864262e-03 -5.94920214e-05  1.00000000e+00]]
```

Output Images:

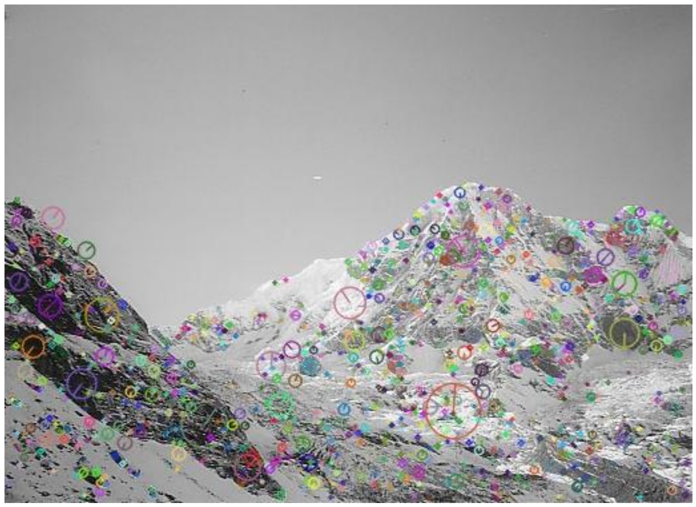

Fig 1.1. task1_sift2.jpg (Extracted SIFT features and draw the keypoints for mountain1.jpg)

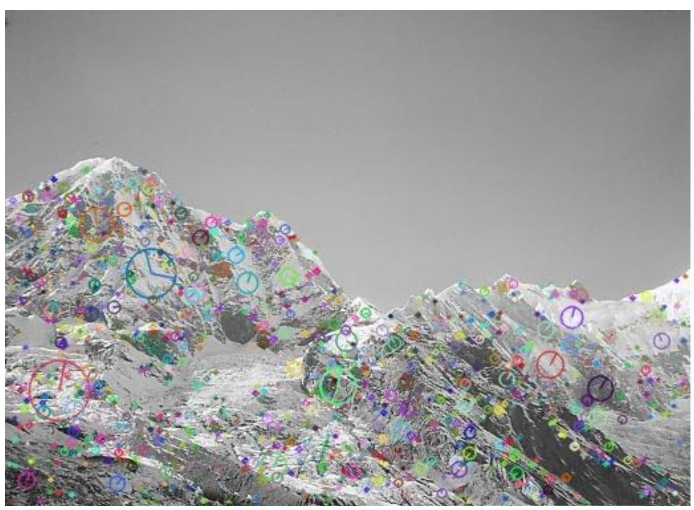

Fig 1.2. task1_sift2.jpg(Extracted SIFT features and draw the keypoints for mountain2.jpg)

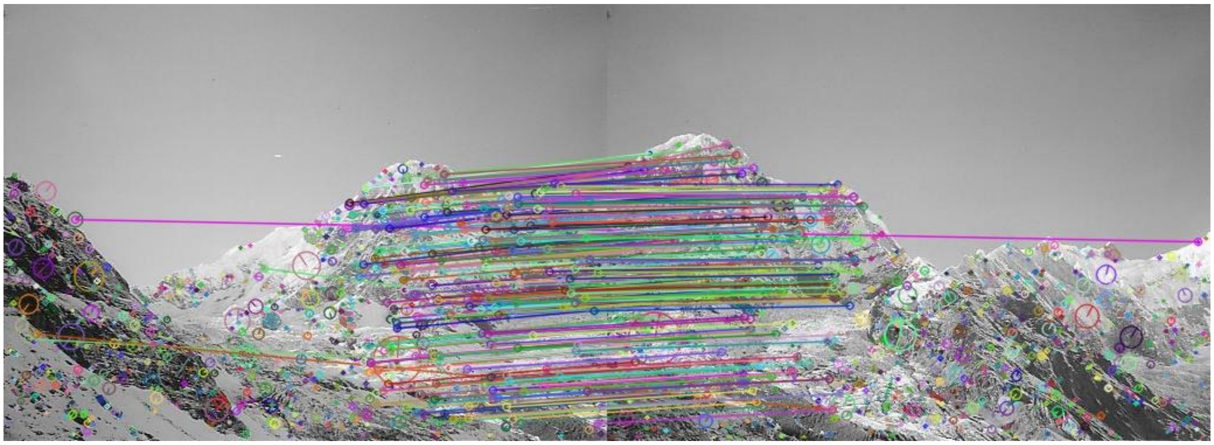

Fig 1.3. task1_matches_knn.jpg

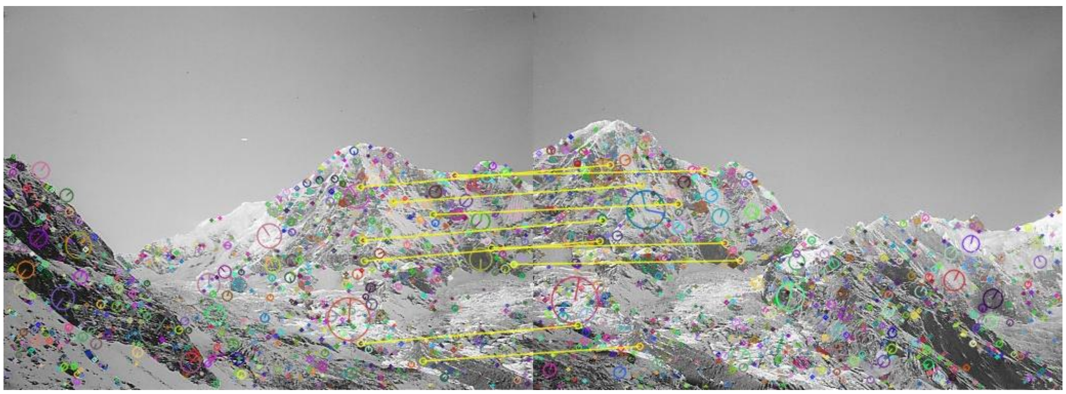

Fig 1.4. task1_matches.jpg

Fig 1.4. task1_pano.jpg

2. EPIPOLAR GEOMETRY:

Code:

```
import cv2
importnumpyas np
import random
from matplotlib importpyplotasplt
from matplotlib import cm
import math

UBIT ='nikhilsr'
np.random.seed(sum([ord(c)for c in UBIT]))

tsucubaLeft_ImageLocation='./proj2_data/data/tsucuba_left.png'
tsucubaRight_ImageLocation='./proj2_data/data/tsucuba_right.png'


defwriteImage(img,outputFileName):
    cv2.imwrite(outputFileName,img)
return1


defreadImage(imageLocation):
img= cv2.imread(imageLocation,1)
returnimg


defmain():
print("Task 2 :")
print(" Task 2.1 : ")
    img1 =readImage(tsucubaLeft_ImageLocation)
    img2 =readImage(tsucubaRight_ImageLocation)
    img1_ = img1.copy()
    img2_ = img2.copy()
    img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    sift = cv2.xfeatures2d.SIFT_create()
    img1_keyPoints, img1_desc =sift.detectAndCompute(img1_gray,None)
    img2_keyPoints, img2_desc =sift.detectAndCompute(img2_gray,None)
    img1_withHighlightedKeyPoints = cv2.drawKeypoints(img1_gray, img1_keyPoints, img1,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    img2_withHighlightedKeyPoints = cv2.drawKeypoints(img2_gray, img2_keyPoints, img2,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
writeImage(img1_withHighlightedKeyPoints,"task2_sift1.jpg")
writeImage(img2_withHighlightedKeyPoints,"task2_sift2.jpg")
    K =2
    M2NRatio =0.75
    FLANN_INDEX_KDTREE =1
numOfChecks=100
index_params=dict(algorithm = FLANN_INDEX_KDTREE, trees =5)
search_params=dict(checks =numOfChecks)
flann= cv2.FlannBasedMatcher(index_params,search_params)
    matches =flann.knnMatch(img1_desc, img2_desc, k=K)
goodMatches=[]
    img1_points =[]
    img2_points =[]
for m, n in matches:
if(m.distance<(M2NRatio *n.distance)):
goodMatches.append(m)
            img1_points.append(img1_keyPoints[m.queryIdx].pt)
            img2_points.append(img2_keyPoints[m.trainIdx].pt)
knnMatchedImg= cv2.drawMatches(img1_withHighlightedKeyPoints, img1_keyPoints,
img2_withHighlightedKeyPoints, img2_keyPoints,goodMatches,None, flags=2)
```

```
writeImage(knnMatchedImg,"task2_matches_knn.jpg")
print("     Task 2.1 Completed. Keypoints have been detected. KNN matched image has been created")
print(" Task 2.2 : ")
#img1_points = np.array(random.sample(list(np.int32(img1_points)), 10))
#img2_points = np.array(random.sample(list(np.int32(img2_points)), 10))
    img1_points =np.array(random.sample(img1_points,10),dtype=np.int32)
    img2_points =np.array(random.sample(img2_points,10),dtype=np.int32)
    F, mask = cv2.findFundamentalMat(img1_points, img2_points, cv2.FM_LMEDS)
print("     The Fundamental Matrix F : ")
print("     "+str(F))
print("     Task 2.2 Completed.")
print(" Task 2.3 : ")
    img1_inlierPoints = img1_points[mask.ravel()==1]
    img2_inlierPoints = img2_points[mask.ravel()==1]
el_LonR=(cv2.computeCorrespondEpilines(img1_inlierPoints.reshape(-1,1,2),1,
F)).reshape(-1,3)
el_RonL=(cv2.computeCorrespondEpilines(img2_inlierPoints.reshape(-1,1,2),2,
F)).reshape(-1,3)
r,c,v= img1.shape
for r, img1_inlierPoint, img2_inlierPoint inzip(el_RonL, img1_inlierPoints,
img2_inlierPoints):
color=(0,255,255)
        x0,y0 =map(int,[0,-r[2]/r[1]])
        x1,y1 =map(int,[c,-(r[2]+r[0]*c)/r[1]])
img_epi_left= cv2.line(img1,(x0,y0),(x1,y1), color,1)
img_epi_left= cv2.circle(img_epi_left,tuple(img1_inlierPoint),5,color,-1)
r,c,v= img2.shape
for r, img2_inlierPoint, img1_inlierPoint inzip(el_LonR, img2_inlierPoints,
img1_inlierPoints):
color=(255,255,0)
        x0,y0 =map(int,[0,-r[2]/r[1]])
        x1,y1 =map(int,[c,-(r[2]+r[0]*c)/r[1]])
img_epi_right= cv2.line(img2,(x0,y0),(x1,y1),color,1)
img_epi_right= cv2.circle(img_epi_right,tuple(img2_inlierPoint),5,color,-1)
writeImage(img_epi_left,"task2_epi_left.jpg")
writeImage(img_epi_right,"task2_epi_right.jpg")
print("     Task 2.3 Completed.")
print(" Task 2.4 : ")
    stereo = cv2.StereoSGBM_create(numDisparities=64,blockSize=25)
img_disparity=stereo.compute(img1_gray, img2_gray)
thresholdImg=(cv2.threshold(img_disparity,0.6,1.0, cv2.THRESH_BINARY))[1]
#writeImage(img_disparity, "task2_disparity.jpg")
plt.subplot(122)
plt.imsave('task2_disparity.jpg',img_disparity,cmap=cm.gray)
print("     Task 2.4 Completed.")


main()
```

Fundamental Matrix:

```
     The Fundamental Matrix F :
 [[ 6.17816316e-04  8.08786066e-03 -1.57507483e-01]
 [ 2.50424086e-04  3.63420634e-03 -1.54443817e-01]
 [-1.11612122e-01 -1.35327244e+00  1.00000000e+00]]
```

Output Images:

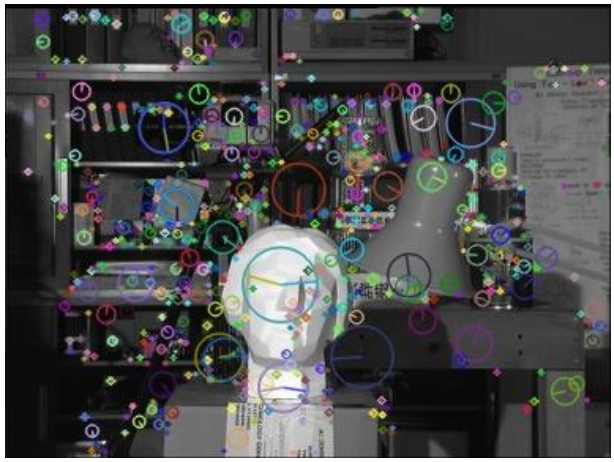

Fig 2.1. task2_sift1.jpg

Fig 2.2. task2_sift2.jpg

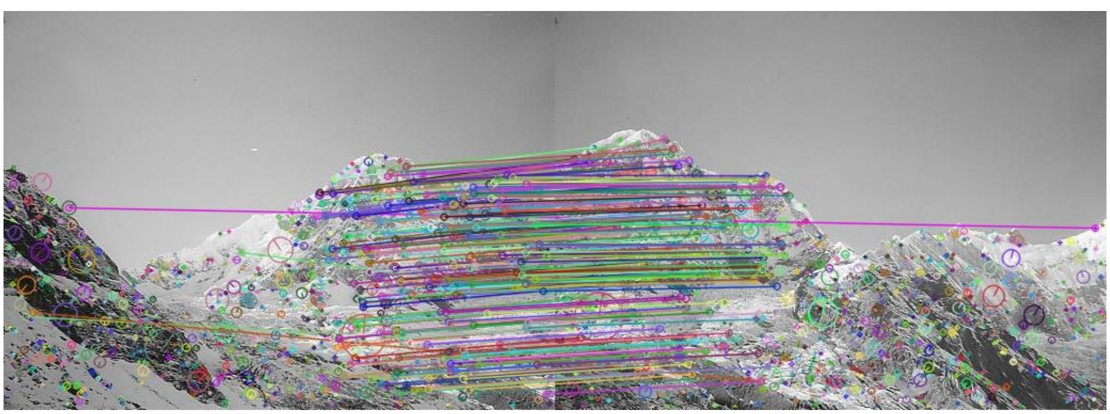

Fig 2.3. task2_matches_knn.jpg

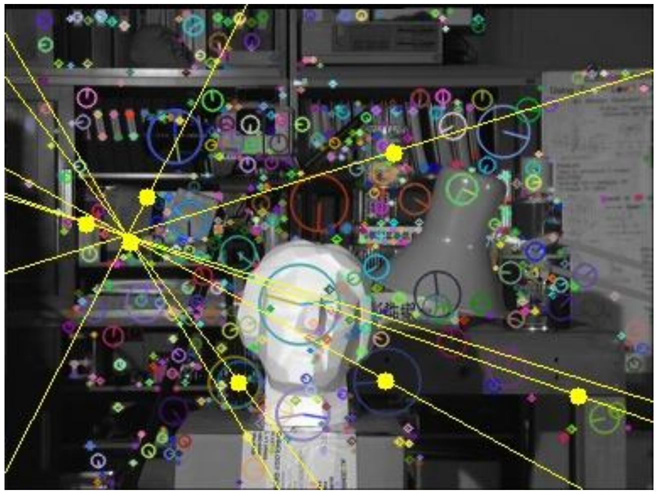

Fig 2.4. task2_epi_left.jpg

Fig 2.5. task2_epi_right.jpg

Fig 2.6. task2_disparity.jpg

3. K- MEANS CLUSTERING:

Code:

Implement the k-means clustering algorithm (kMeansClustering.py):

```
importnumpyas np
from math import sqrt
importmatplotlib.pyplotasmplt
importmatplotlib.markersasmmarkers


class KMeansClustering():

        def__init__(self,numOfClusters):
                self.K=numOfClusters
                self.dataPoints=None
                self.initialClusters=None
                self.numberOfDataPoints=None
                self.dataPointsDimension=None
                self.allClusterCenters=[]
                self.finalClusterCenters=None
                self.allDataPointsDistribution=[]
                self.finalDataPointsDistribution=None


        def __plotImage(self,imageType,iterNum):

        currentClusterCenters=(self.allClusterCenters[len(self.allClusterCenters)-
1]).copy()
                currentClusterCenters=np.around(currentClusterCenters, decimals=2)

        currentDataPointsDistribution=self.allDataPointsDistribution[len(self.allDataP
ointsDistribution)-1]
                dp_x_coordinates=self.dataPoints[:,0]
                dp_y_coordinates=self.dataPoints[:,1]
                cc_x_coordinates=currentClusterCenters[:,0]
                cc_y_coordinates=currentClusterCenters[:,1]
                clusterGroupings=np.array([1,2,3])
                fig,ax=mplt.subplots()
                if(imageType=="Classification"):
                        plotName='task3_iter'+str(iterNum)+'_a.jpg'
                        dataGroupings=[]
                        i=0
                        while(i<self.numberOfDataPoints):
                                if(currentDataPointsDistribution[i][0]==1):
                                        dataGroupings.append(1)
                                elif(currentDataPointsDistribution[i][1]==1):
                                        dataGroupings.append(2)
                                else:
                                        dataGroupings.append(3)
                                i=i+1
                        dataGroupings=np.array(dataGroupings)

        ax.scatter(dp_x_coordinates[dataGroupings==1],dp_y_coordinates[dataGroupings==
1], c='red',facecolors='full',
marker=mmarkers.MarkerStyle(marker='^',fillstyle='full'),edgecolors='black')

        ax.scatter(dp_x_coordinates[dataGroupings==2],dp_y_coordinates[dataGroupings==
2], c='green',facecolors='full',
marker=mmarkers.MarkerStyle(marker='^',fillstyle='full'),edgecolors='black')

        ax.scatter(dp_x_coordinates[dataGroupings==3],dp_y_coordinates[dataGroupings==
```

```
3], c='blue',facecolors='full',
marker=mmarkers.MarkerStyle(marker='^',fillstyle='full'),edgecolors='black')

        ax.scatter(cc_x_coordinates[clusterGroupings==1],cc_y_coordinates[clusterGroup
ings==1], c='red',facecolors='full',
marker=mmarkers.MarkerStyle(marker='o',fillstyle='full'),edgecolors='red')

        ax.scatter(cc_x_coordinates[clusterGroupings==2],cc_y_coordinates[clusterGroup
ings==2], c='green',facecolors='full',
marker=mmarkers.MarkerStyle(marker='o',fillstyle='full'),edgecolors='green')

        ax.scatter(cc_x_coordinates[clusterGroupings==3],cc_y_coordinates[clusterGroup
ings==3], c='blue',facecolors='full',
marker=mmarkers.MarkerStyle(marker='o',fillstyle='full'),edgecolors='blue')
                else:
                        plotName='task3_iter'+str(iterNum)+'_b.jpg'

        ax.scatter(dp_x_coordinates,dp_y_coordinates,facecolors='none',
marker=mmarkers.MarkerStyle(marker='^',fillstyle='none'),edgecolors='black')

        ax.scatter(cc_x_coordinates[clusterGroupings==1],cc_y_coordinates[clusterGroup
ings==1], c='red',facecolors='full',
marker=mmarkers.MarkerStyle(marker='o',fillstyle='none'),edgecolors='red')

        ax.scatter(cc_x_coordinates[clusterGroupings==2],cc_y_coordinates[clusterGroup
ings==2], c='green',facecolors='full',
marker=mmarkers.MarkerStyle(marker='o',fillstyle='none'),edgecolors='green')

        ax.scatter(cc_x_coordinates[clusterGroupings==3],cc_y_coordinates[clusterGroup
ings==3], c='blue',facecolors='full',
marker=mmarkers.MarkerStyle(marker='o',fillstyle='none'),edgecolors='blue')
                i=0
                while(i<self.numberOfDataPoints):

        ax.annotate("("+((str)(dp_x_coordinates[i]))+","+((str)(dp_y_coordinates[i]))+
")",(dp_x_coordinates[i],dp_y_coordinates[i]))
                        i=i+1
                i=0
                while(i<3):

        ax.annotate("("+((str)(cc_x_coordinates[i]))+","+((str)(cc_y_coordinates[i]))+
")",(cc_x_coordinates[i],cc_y_coordinates[i]))
                        i=i+1
                #mplt.show()
                fig.savefig(plotName, dpi=fig.dpi)


        def __calculateEuclideanDistance(self, x1, x2):
                if((str(type(x1))=="<class
'numpy.ndarray'>")and(str(type(x2))=="<class 'numpy.ndarray'>")):
                        x1 =np.array(x1)
                        x2 =np.array(x2)
                        return sqrt(np.sum(np.square(x1-x2)))
                else:
                        returnsqrt(x1**2- x2**2)


        deffit(self,dataPoints,initialClusters=None,plotGraphs=False):
                self.dataPoints=dataPoints

        self.initialClusters=initialClustersif(str(type(initialClusters))=="<class
'numpy.ndarray'>")elseself.dataPoints[0:self.K]
                self.numberOfDataPoints=(dataPoints).shape[0]
```

```
                self.dataPointsDimension=(dataPoints).shape[1]
                self.allClusterCenters.append(self.initialClusters)
                #Starting k means clustering logic
                iterNum=0

        while((iterNum==0)or((np.array_equal(currentClusterCenter,nextClusterCenter))=
=False)):
                        # For current cluster centers, find optimal point
distribution

        currentClusterCenter=self.allClusterCenters[len(self.allClusterCenters)-1]

        currentDataPointsDistribution=np.zeros((self.numberOfDataPoints,self.K))
                        i=0
                        while(i<self.numberOfDataPoints):
                                dataPoint=self.dataPoints[i]
                                temp=[]
                                j=0
                                while(j<self.K):

        temp.append(self.__calculateEuclideanDistance(dataPoint,currentClusterCenter[j
]))
                                        j=j+1

        currentDataPointsDistribution[i][temp.index(min(temp))]=1
                                i=i+1

        (self.allDataPointsDistribution).append(currentDataPointsDistribution)
                        if(plotGraphs==True):
                                self.__plotImage("Classification", iterNum+1)
                        # For current points distribution, find optimal cluster
centers

        nume=(np.dot((np.transpose(currentDataPointsDistribution)),dataPoints))

        partial_den=(np.transpose(currentDataPointsDistribution)).sum(axis=1)
                        den_create=[]
                        for a inrange(self.dataPointsDimension):
                                den_create.append(partial_den)
                        den =np.transpose(np.array(den_create))
                        nextClusterCenter=(nume/ den )
                        self.allClusterCenters.append(nextClusterCenter)
                        if(plotGraphs==True):
                                self.__plotImage("UpdateClusterCenter", iterNum+1)
                        # Increment iteration number
                        iterNum=iterNum+1

        if((self.numberOfDataPoints)>1000andlen(self.allClusterCenters)>=7):
                                self.allClusterCenters=self.allClusterCenters[4:]

        self.allDataPointsDistribution=self.allDataPointsDistribution[4:]

        self.allClusterCenters=self.allClusterCenters[:len(self.allClusterCenters)-1]
                self.finalClusterCenters=nextClusterCenter
                self.finalDataPointsDistribution=currentDataPointsDistribution
```

Task 3 Code (Task3.py) :

```
importnumpyas np
fromkMeansClusteringimportKMeansClustering
import cv2
```

```
baboonImageLocation='./proj2_data/data/baboon.jpg'


defreadImage(imageLocation):
img= cv2.imread(imageLocation,1)
returnimg


defwriteImage(img,outputFileName):
        cv2.imwrite(outputFileName,img)
        return1


defmain():
        print("Task 3 :")
        print("  Task 3.1, 3.2, 3.3 :")
        X
=np.array([[5.9,3.2],[4.6,2.9],[6.2,2.8],[4.7,3.2],[5.5,4.2],[5.0,3.0],[4.9,3.1],[6.7,
3.1],[5.1,3.8],[6.0,3.0]])
        initialClusters=np.array([[6.2,3.2],[6.6,3.7],[6.5,3.0]])
        kMeansCluster=KMeansClustering(3)
        kMeansCluster.fit(X,initialClusters,True)
        print("          All Cluster Centers :
"+(str)(kMeansCluster.allClusterCenters))
        print("          Final Cluster Centers :
"+(str)(kMeansCluster.finalClusterCenters))
        print("          All Data Points Distribution :
"+(str)(kMeansCluster.allDataPointsDistribution))
        print("          Final Data Points Distribution :
"+(str)(kMeansCluster.finalDataPointsDistribution))
        print()
        print("  Task 3.4 :")
        baboonImg=readImage(baboonImageLocation)
        #baboonImg = cv2.resize(baboonImg , (128, 128))
        baboonData=[]
        i=0
        while(i<len(baboonImg)):
                j=0
                while(j<len(baboonImg[0])):
                        baboonData.append(baboonImg[i][j])
                        j=j+1
                i=i+1
        baboonData=np.array(baboonData)
        K =[3,5,10,20]
        for kin K:
                baboonImg_copy=baboonImg.copy()
                kMeansCluster=KMeansClustering(k)
                kMeansCluster.fit(baboonData)
                print("          Final Cluster Centers for K = "+(str)(k)+" :
"+(str)(kMeansCluster.finalClusterCenters))
                i=0
                while(i<len(baboonImg)):
                        j=0
                        while(j<len(baboonImg[0])):
                                baboonImg_copy[i][j]=
kMeansCluster.finalClusterCenters[np.argmax(kMeansCluster.finalDataPointsDistribution[
(i*len(baboonImg))+j])]
                                j=j+1
                        i=i+1
                writeImage(baboonImg_copy,"task3_baboon_"+str(k)+".jpg")
        print("  Task 3.5 :")
```

```
main()
```

Code Output:

```
Task 3 :
    Task 3.1, 3.2, 3.3 :
        All Cluster Centers : [
       array([[6.2, 3.2],
               [6.6, 3.7],
               [6.5, 3. ]]),
       array([[5.17142857, 3.17142857],
              [5.5       , 4.2       ],
              [6.45      , 2.95      ]]),
       array([[4.8  , 3.05 ],
              [5.3  , 4.   ],
              [6.2  , 3.025]])]

        Final Cluster Centers :
        [[4.8   3.05 ]
         [5.3   4.   ]
         [6.2   3.025]]

        All Data Points Distribution : [
        array([[1., 0., 0.],   -> 1st Cluster for point [5.9, 3.2]
               [1., 0., 0.],   -> 1st Cluster for point [4.6, 2.9]
               [0., 0., 1.],   -> 3rd Cluster for point [6.2, 2.8]
               [1., 0., 0.],   -> 1st Cluster for point [4.7, 3.2]
               [0., 1., 0.],   -> 2nd Cluster for point [5.5, 4.2]
               [1., 0., 0.],   -> 1st Cluster for point [5.0, 3.0]
               [1., 0., 0.],   -> 1st Cluster for point [4.9, 3.1]
               [0., 0., 1.],   -> 3rd Cluster for point [6.7, 3.1]
               [1., 0., 0.],   -> 1st Cluster for point [5.1, 3.8]
               [1., 0., 0.]]),-> 1st Cluster for point [6.0, 3.0]
       array([ [0., 0., 1.],   -> 3rd Cluster for point [5.9, 3.2]
               [1., 0., 0.],   -> 1st Cluster for point [4.6, 2.9]
               [0., 0., 1.],   -> 3rd Cluster for point [6.2, 2.8]
               [1., 0., 0.],   -> 1st Cluster for point [4.7, 3.2]
               [0., 1., 0.],   -> 2nd Cluster for point [5.5, 4.2]
               [1., 0., 0.],   -> 1st Cluster for point [5.0, 3.0]
               [1., 0., 0.],   -> 1st Cluster for point [4.9, 3.1]
               [0., 0., 1.],   -> 3rd Cluster for point [6.7, 3.1]
               [0., 1., 0.],   -> 2nd Cluster for point [5.1, 3.8]
               [0., 0., 1.]]),-> 3rd Cluster for point [6.0, 3.0]
       array([ [0., 0., 1.],   -> 3rd Cluster for point [5.9, 3.2]
               [1., 0., 0.],   -> 1st Cluster for point [4.6, 2.9]
               [0., 0., 1.],   -> 3rd Cluster for point [6.2, 2.8]
               [1., 0., 0.],   -> 1st Cluster for point [4.7, 3.2]
               [0., 1., 0.],   -> 2nd Cluster for point [5.5, 4.2]
               [1., 0., 0.],   -> 1st Cluster for point [5.0, 3.0]
               [1., 0., 0.],   -> 1st Cluster for point [4.9, 3.1]
               [0., 0., 1.],   -> 3rd Cluster for point [6.7, 3.1]
               [0., 1., 0.],   -> 2nd Cluster for point [5.1, 3.8]
               [0., 0., 1.]])]-> 3rd Cluster for point [6.0, 3.0]

        Final Data Points Distribution :
        [[0. 0. 1.]  -> 3rd Cluster for point [5.9, 3.2]
         [1. 0. 0.]  -> 1st Cluster for point [4.6, 2.9]
         [0. 0. 1.]  -> 3rd Cluster for point [6.2, 2.8]
         [1. 0. 0.]  -> 1st Cluster for point [4.7, 3.2]
         [0. 1. 0.]  -> 2nd Cluster for point [5.5, 4.2]
         [1. 0. 0.]  -> 1st Cluster for point [5.0, 3.0]
         [1. 0. 0.]  -> 1st Cluster for point [4.9, 3.1]
         [0. 0. 1.]  -> 3rd Cluster for point [6.7, 3.1]
         [0. 1. 0.]  -> 2nd Cluster for point [5.1, 3.8]
         [0. 0. 1.]] -> 3rd Cluster for point [6.0, 3.0]

[Finished in 2.4s]
```

This is the Output produced by my code. My code actually runs to completion of the k means clustering algorithm. After the completion, I print out the cluster center and the classification matrix for all iteration as well as the final classification matrix and cluster center as can be seen above.(The part is yellow is hand written – It is to help the evaluator match the classification matrix to the cluster centers and the data). The updated cluster centers and classification vectors for 3.1,3.2,3.3 are displayed as part of the ‘All Cluster Centers’ and ‘All Data Points Distribution’ output above.

Output Images:

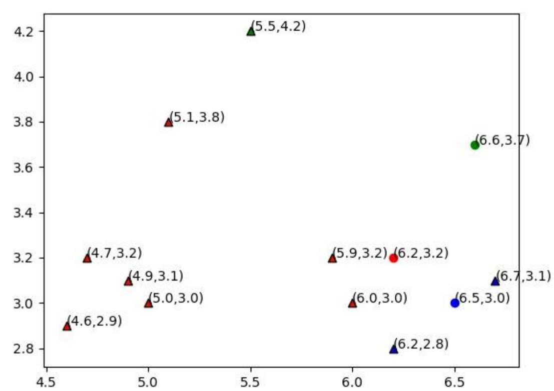

Fig 3.1. task3_iter1_a.jpg

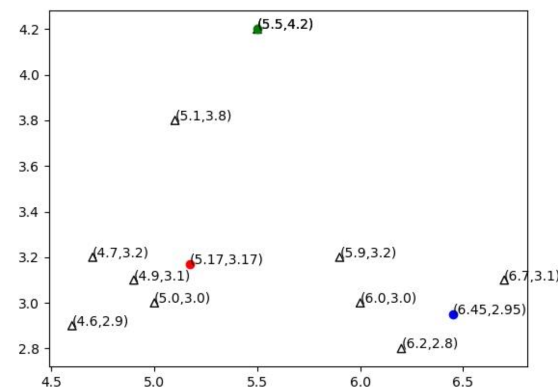

Fig 3.2. task3_iter1_b.jpg

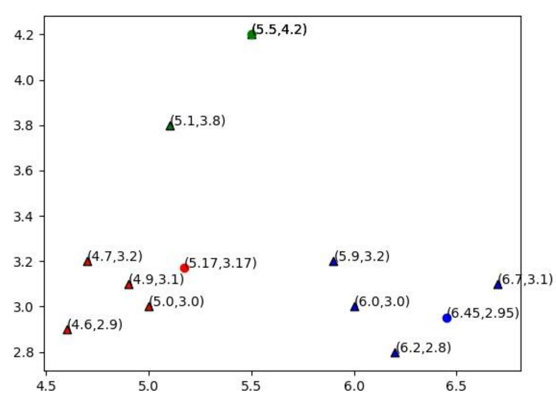

Fig 3.3. task3_iter2_a.jpg

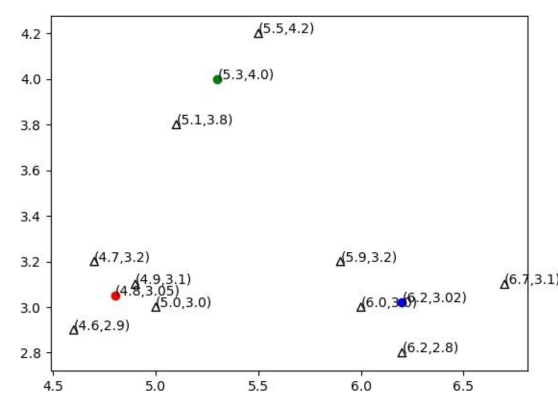

Fig 3.4. task3_iter2_b.jpg

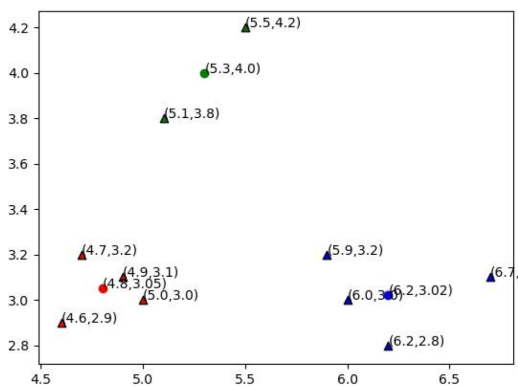

Fig 3.5. task3_iter3_a.jpg

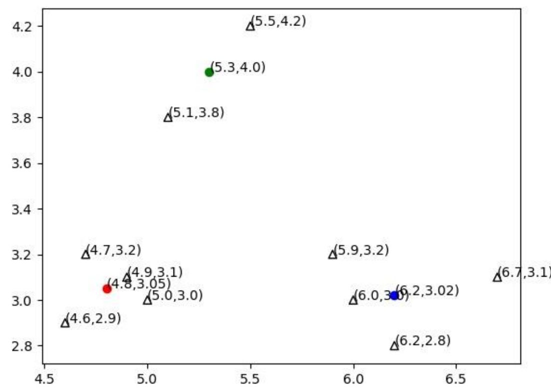

Fig 3.5. task3_iter3_b.jpg

Fig 3.6. task3_baboon_3.jpg

Fig 3.7. task3_baboon_5.jpg

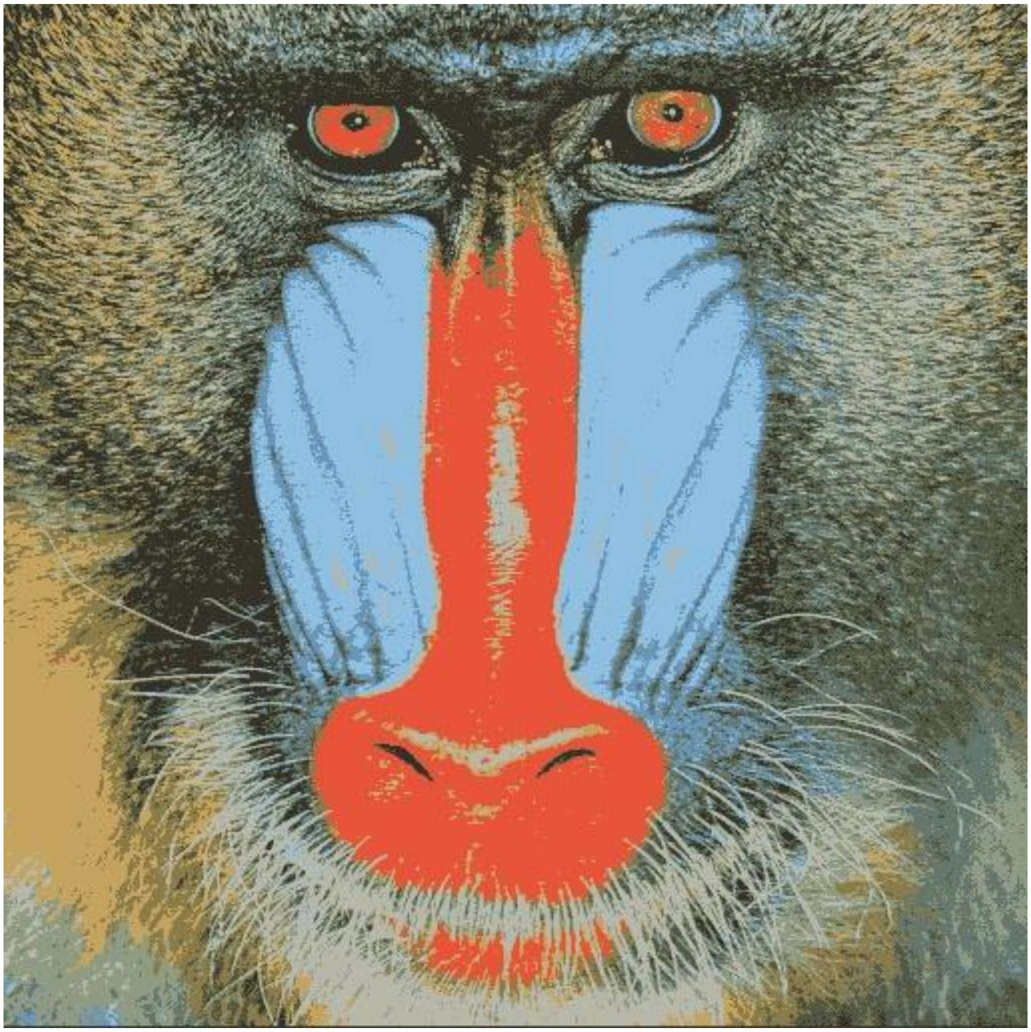

Fig 3.8. task3_baboon_10.jpg

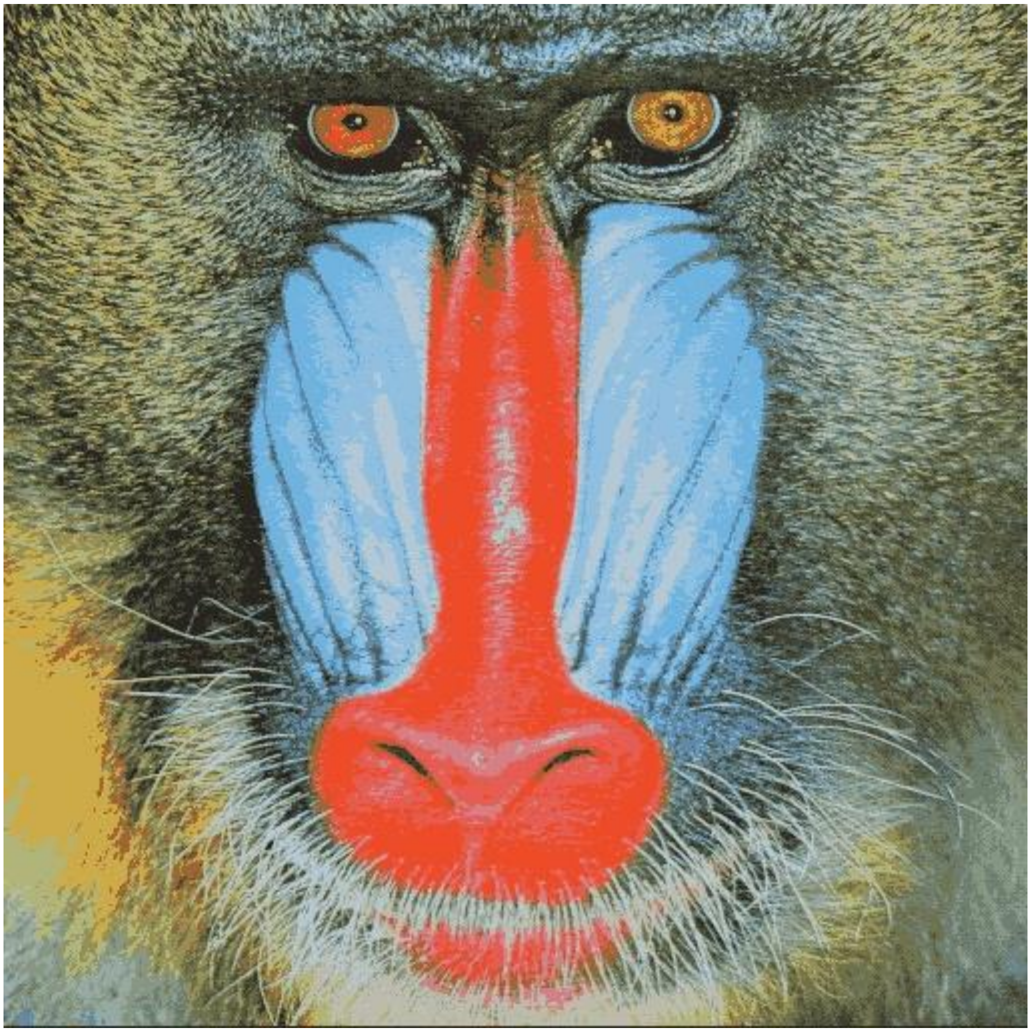

Fig 3.9. task3_baboon_20.jpg

3 BONUS. GMM:

Code:

Implement the GMM algorithm (gaussianMixtureModel.py):

```
import numpy as np
import math
import matplotlib.pyplot as mplt
import matplotlib.markers as mmarkers
from matplotlib.patches import Ellipse


class GaussianMixtureModel():

        def __init__(self, numOfClusters, taskNum):
                self.taskNum = taskNum
```

```
                self.C = numOfClusters
                self.Theta = None
                self.ProbabilityDistribution = None
                self.totNumOfIterations = None


        def __plot_cov_ellipse(self, cov, pos, nstd=2, ax=None, **kwargs):
                def eigsorted(cov):
                        vals, vecs = np.linalg.eigh(cov)
                        order = vals.argsort()[::-1]
                        return vals[order], vecs[:,order]
                if ax is None:
                        ax = mplt.gca()
                vals, vecs = eigsorted(cov)
                theta = np.degrees(np.arctan2(*vecs[:,0][::-1]))
                width, height = 2 * nstd * np.sqrt(vals)
                ellip = Ellipse(xy=pos, width=width, height=height,
angle=theta, **kwargs)
                ax.add_artist(ellip)
                return ellip


        def __drawGraph(self, X, currTheta, p, N, iterNum):
                points = []
                j=0
                while(j<self.C):
                        points.append([])
                        j=j+1
                i=0
                while(i<N):
                        (points[np.argmax(p[i])]).append(X[i])
                        i=i+1
                fig = mplt.figure(0)
                ax = fig.add_subplot(111, aspect='equal')
                self.__plot_cov_ellipse(cov=currTheta["BigSigma"][0],
pos=currTheta["Mu"][0], ax=ax, color='red', alpha=0.5)
                self.__plot_cov_ellipse(cov=currTheta["BigSigma"][1],
pos=currTheta["Mu"][1], ax=ax, color='green', alpha=0.5)
                self.__plot_cov_ellipse(cov=currTheta["BigSigma"][2],
pos=currTheta["Mu"][2], ax=ax, color='blue', alpha=0.5)

        ax.scatter((np.array(points[0]))[:,0],(np.array(points[0]))[:,1],
c='red', facecolors='full', marker=mmarkers.MarkerStyle(marker='o',
fillstyle='none'), edgecolors='red')

        ax.scatter((np.array(points[1]))[:,0],(np.array(points[1]))[:,1],
c='green', facecolors='full', marker=mmarkers.MarkerStyle(marker='o',
fillstyle='none'), edgecolors='green')

        ax.scatter((np.array(points[2]))[:,0],(np.array(points[2]))[:,1],
c='blue', facecolors='full', marker=mmarkers.MarkerStyle(marker='o',
fillstyle='none'), edgecolors='blue')
                ax.set_xlim(0, 110)
                ax.set_ylim(30, 110)
                fig.savefig("task3_gmm_iter"+(str)(iterNum+1)+".jpg",
dpi=fig.dpi)
```

```
    def __BigSigmaValCheck(self, BigSigma, D):
        temp = BigSigma[0][0]
        i=0
        while(i<D):
            j=0
            while(j<D):
                if ( (i!=0 and j!=0) ):
                    if (BigSigma[i][j]!=temp):
                        return BigSigma
                j=j+1
            i=i+1
        BigSigma[0][0] = 1.1 * BigSigma[0][0]
        BigSigma[D-1][D-1] = 1.1 * BigSigma[D-1][D-1]
        return BigSigma


    def __calculateGaussianProb(self, x, Mu, BigSigma, D):
        BigSigma_i = np.linalg.inv(BigSigma)
        diff = x-Mu
        diff_t = np.transpose(diff)
        num = math.exp( -0.5 * (np.dot(np.dot(diff_t, BigSigma_i),
diff)) )
        den = math.pow((2*3.14), (D/2)) *
math.sqrt(np.linalg.det(BigSigma))
        return num/den


    def __logLikelihood(self, X, N, C, D, pi, Mu, BigSigma):
        tot = 0
        i=0
        while(i<N):
            x = X[i]
            summ = 0
            j=0
            while(j<self.C):
                summ = summ + ( pi[j] *
self.__calculateGaussianProb(x, Mu[j], BigSigma[j], D) )
                j=j+1
            tot = tot + math.log(summ)
            i=i+1
        return tot


    def fit(self, X, initialParams):
        N = len(X)
        D = len(X[0])
        logLikelihoods = [-1000000000000000000]
        iterNum = 0
        while(1==1):
            #Step 1 : E
            if (iterNum==0):
                currTheta = initialParams
            else:
                currTheta = nextTheta
            currGuassProbMatrix = np.zeros((N,self.C))
            p = np.zeros((N,self.C))
```

```
                        i=0
                        while(i<N):
                                x = X[i]
                                j=0
                                while(j<self.C):
                                        currGuassProbMatrix[i][j] =
self.__calculateGaussianProb(x, (currTheta["Mu"][j]),
(currTheta["BigSigma"][j]),  D)
                                        j=j+1
                                i=i+1
                        i=0
                        while(i<N):
                                j=0
                                while(j<self.C):
                                        p[i][j] =
((currTheta["pi"][j])*(currGuassProbMatrix[i][j])) /
(np.sum(currGuassProbMatrix[i]))
                                        j=j+1
                                i=i+1
                        #Step 2 : M
                        m = np.sum(p, axis=0)
                        m_sum = np.sum(m)
                        pt_X = (np.dot(np.transpose(p), X))
                        nextTheta = currTheta.copy()
                        j=0
                        while(j<self.C):
                                nextTheta["pi"][j] = m[j] / m_sum
                                j=j+1
                        j=0
                        while(j<self.C):
                                nextTheta["Mu"][j] = (1/m[j]) * pt_X[j]
                                j=j+1
                        j=0
                        while(j<self.C):
                                summ = 0
                                i=0
                                while(i<N):
                                        diff = (X[i]-
nextTheta["Mu"][j]).reshape(1, D)

                                        diff_t = np.transpose(diff)
                                        summ = summ + (p[i][j]) * np.dot(
diff_t, diff )

                                        i=i+1
                                nextTheta["BigSigma"][j] =
self.__BigSigmaValCheck( ((1/m[j])*summ), D )
                                j=j+1
                        #Increment iterNum
                        logLikelihood = self.__logLikelihood(X, N, self.C, D,
nextTheta["pi"], nextTheta["Mu"], nextTheta["BigSigma"])
                        if (self.taskNum=="A"):
                                if (iterNum==0):
                                        print("        New Theta Values :
"+str(nextTheta))
                                        print("        Current Theta Values :
"+str(currTheta))
                                        print("        ProbabilityDistribution
Matrix : "+str(p))
```

```
                              print()
                    else:
                              if (iterNum>=0 and iterNum<=4):

        print("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX")
                                        print("         New Theta Values : 
"+str(nextTheta))
                                        print("         Current Theta Values : 
"+str(currTheta))
                                        print("         ProbabilityDistribution 
Matrix : "+str(p))
                                        print()
                                        self.__drawGraph(X, currTheta, p, N, 
iterNum)
                    if (logLikelihood<logLikelihoods[len(logLikelihoods)-
1]):
                              self.Theta =currTheta
                              self.ProbabilityDistribution = p
                              self.totNumOfIterations = iterNum
                              self.finalLogLikelihood = 
logLikelihoods[len(logLikelihoods)-1]
                              break
                    else:
                              if (iterNum>100):
                                        print("         Breaking coz there was 
no convergence in 100 iterations")
                                        break
                              else:
                                        logLikelihoods.append(logLikelihood)
                    iterNum = iterNum + 1
```

Task 3 Bonus Code (Task3_Bonus.py) :

```
import numpy as np
from gaussianMixtureModel import GaussianMixtureModel
import cv2


def main():
        print(" Task 3.5 A :")
        X = np.array([[5.9, 3.2], [4.6, 2.9], [6.2, 2.8], [4.7, 3.2], [5.5, 
4.2], [5.0, 3.0], [4.9, 3.1], [6.7, 3.1], [5.1, 3.8], [6.0, 3.0]])
        initialMus = np.array([[6.2, 3.2], [6.6, 3.7], [6.5, 3.0]])
        initialBigSigma = np.array([ [[0.5, 0],[0, 0.5]], [[0.5, 0],[0, 0.5]], 
[[0.5, 0],[0, 0.5]] ])
        initialPi = np.array([ (1/3), (1/3), (1/3) ])
        initialParams = { "Mu": initialMus, "BigSigma": initialBigSigma, "pi": 
initialPi }
        gaussianMixtureModel = GaussianMixtureModel(3, 'A')
        gaussianMixtureModel.fit(X, initialParams)
        print()
        print("         Total Num of Iterations taken to reach Convergence : 
"+str(gaussianMixtureModel.totNumOfIterations))
```

```
        print("        Final Theta Values : "+str(gaussianMixtureModel.Theta))
        print("        Final Probability Distribution Matrix :
"+str(gaussianMixtureModel.ProbabilityDistribution))
        print(" Task 3.5 B :")
        # Old Faithful dataset
        X = np.array( [[3.600, 79], [1.800, 54], [3.333, 74], [2.283, 62],
[4.533, 85], [2.883, 55], [4.700, 88], [3.600, 85], [1.950, 51], [4.350, 85],
[1.833, 54], [3.917, 84], [4.200, 78], [1.750, 47], [4.700, 83], [2.167, 52],
[1.750, 62], [4.800, 84], [1.600, 52], [4.250, 79], [1.800, 51], [1.750, 47],
[3.450, 78], [3.067, 69], [4.533, 74], [3.600, 83], [1.967, 55], [4.083, 76],
[3.850, 78], [4.433, 79], [4.300, 73], [4.467, 77], [3.367, 66], [4.033, 80],
[3.833, 74], [2.017, 52], [1.867, 48], [4.833, 80], [1.833, 59], [4.783, 90],
[4.350, 80], [1.883, 58], [4.567, 84], [1.750, 58], [4.533, 73], [3.317, 83],
[3.833, 64], [2.100, 53], [4.633, 82], [2.000, 59], [4.800, 75], [4.716, 90],
[1.833, 54], [4.833, 80], [1.733, 54], [4.883, 83], [3.717, 71], [1.667, 64],
[4.567, 77], [4.317, 81], [2.233, 59], [4.500, 84], [1.750, 48], [4.800, 82],
[1.817, 60], [4.400, 92], [4.167, 78], [4.700, 78], [2.067, 65], [4.700, 73],
[4.033, 82], [1.967, 56], [4.500, 79], [4.000, 71], [1.983, 62], [5.067, 76],
[2.017, 60], [4.567, 78], [3.883, 76], [3.600, 83], [4.133, 75], [4.333, 82],
[4.100, 70], [2.633, 65], [4.067, 73], [4.933, 88], [3.950, 76], [4.517, 80],
[2.167, 48], [4.000, 86], [2.200, 60], [4.333, 90], [1.867, 50], [4.817, 78],
[1.833, 63], [4.300, 72], [4.667, 84], [3.750, 75], [1.867, 51], [4.900, 82],
[2.483, 62], [4.367, 88], [2.100, 49], [4.500, 83], [4.050, 81], [1.867, 47],
[4.700, 84], [1.783, 52], [4.850, 86], [3.683, 81], [4.733, 75], [2.300, 59],
[4.900, 89], [4.417, 79], [1.700, 59], [4.633, 81], [2.317, 50], [4.600, 85],
[1.817, 59], [4.417, 87], [2.617, 53], [4.067, 69], [4.250, 77], [1.967, 56],
[4.600, 88], [3.767, 81], [1.917, 45], [4.500, 82], [2.267, 55], [4.650, 90],
[1.867, 45], [4.167, 83], [2.800, 56], [4.333, 89], [1.833, 46], [4.383, 82],
[1.883, 51],
[4.933, 86], [2.033, 53], [3.733, 79], [4.233, 81], [2.233, 60], [4.533, 82],
[4.817, 77], [4.333, 76], [1.983, 59], [4.633, 80], [2.017, 49], [5.100, 96],
[1.800, 53], [5.033, 77], [4.000, 77], [2.400, 65], [4.600, 81], [3.567, 71],
[4.000, 70], [4.500, 81], [4.083, 93], [1.800, 53], [3.967, 89], [2.200, 45],
[4.150, 86], [2.000, 58], [3.833, 78], [3.500, 66], [4.583, 76], [2.367, 63],
[5.000, 88], [1.933, 52], [4.617, 93], [1.917, 49], [2.083, 57], [4.583, 77],
[3.333, 68], [4.167, 81], [4.333, 81], [4.500, 73], [2.417, 50], [4.000, 85],
[4.167, 74], [1.883, 55], [4.583, 77], [4.250, 83], [3.767, 83], [2.033, 51],
[4.433, 78], [4.083, 84], [1.833, 46], [4.417, 83], [2.183, 55], [4.800, 81],
[1.833, 57], [4.800, 76], [4.100, 84], [3.966, 77], [4.233, 81], [3.500, 87],
[4.366, 77], [2.250, 51], [4.667, 78], [2.100, 60],
[4.350, 82], [4.133, 91], [1.867, 53], [4.600, 78], [1.783, 46], [4.367, 77],
[3.850, 84], [1.933, 49], [4.500, 83], [2.383, 71], [4.700, 80], [1.867, 49],
[3.833, 75], [3.417, 64], [4.233, 76], [2.400, 53], [4.800, 94], [2.000, 55],
[4.150, 76], [1.867, 50], [4.267, 82], [1.750, 54], [4.483, 75], [4.000, 78],
[4.117, 79], [4.083, 78], [4.267, 78], [3.917, 70], [4.550, 79], [4.083, 70],
[2.417, 54], [4.183, 86], [2.217, 50], [4.450, 90], [1.883, 54], [1.850, 54],
[4.283, 77], [3.950, 79], [2.333, 64], [4.150, 75], [2.350, 47], [4.933, 86],
[2.900, 63], [4.583, 85], [3.833, 82], [2.083, 57], [4.367, 82], [2.133, 67],
[4.350, 74], [2.200, 54], [4.450, 83], [3.567, 73], [4.500, 73], [4.150, 88],
[3.817, 80], [3.917, 71], [4.450, 83], [2.000, 56], [4.283, 79], [4.767, 78],
[4.533, 84], [1.850, 58], [4.250, 83], [1.983, 43], [2.250, 60], [4.750, 75],
[4.117, 81], [2.150, 46], [4.417, 90], [1.817, 46], [4.467, 74]] )
        initialMus = np.array([[4.0, 81], [2.0, 57], [4.0, 71]])
        initialBigSigma = np.array([ [[1.30, 13.98],[13.98, 184.82]], [[1.30,
13.98],[13.98, 184.82]], [[1.30, 13.98],[13.98, 184.82]] ])
        initialPi = np.array([ (1/3), (1/3), (1/3) ])
```

```
        initialParams = { "Mu": initialMus, "BigSigma": initialBigSigma, "pi":
initialPi }
        gaussianMixtureModel = GaussianMixtureModel(3, 'B')
        gaussianMixtureModel.fit(X, initialParams)
        print()
        print("         Total Num of Iterations taken to reach Convergence :
"+str(gaussianMixtureModel.totNumOfIterations))
        print("         Final Probability Distribution Matrix :
"+str(gaussianMixtureModel.ProbabilityDistribution))
        print("         Final Theta Values : "+str(gaussianMixtureModel.Theta))
        print("         Final Probability Distribution Matrix :
"+str(gaussianMixtureModel.ProbabilityDistribution))
        print()


main()
```

Output Images:

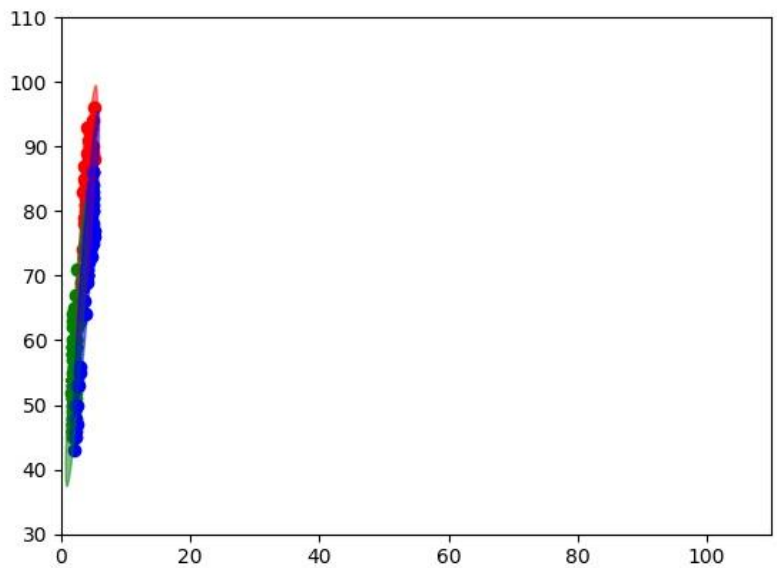

Fig task3_gmm_iter1.jpg

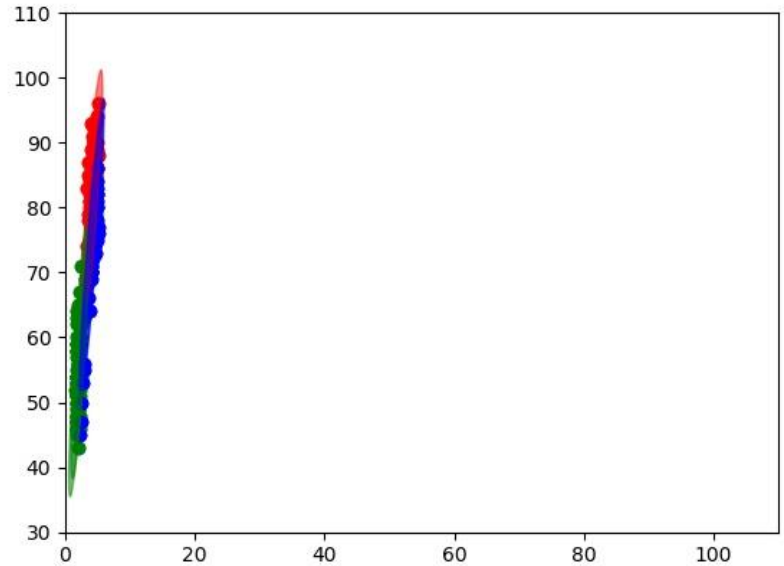

Fig task3_gmm_iter2.jpg

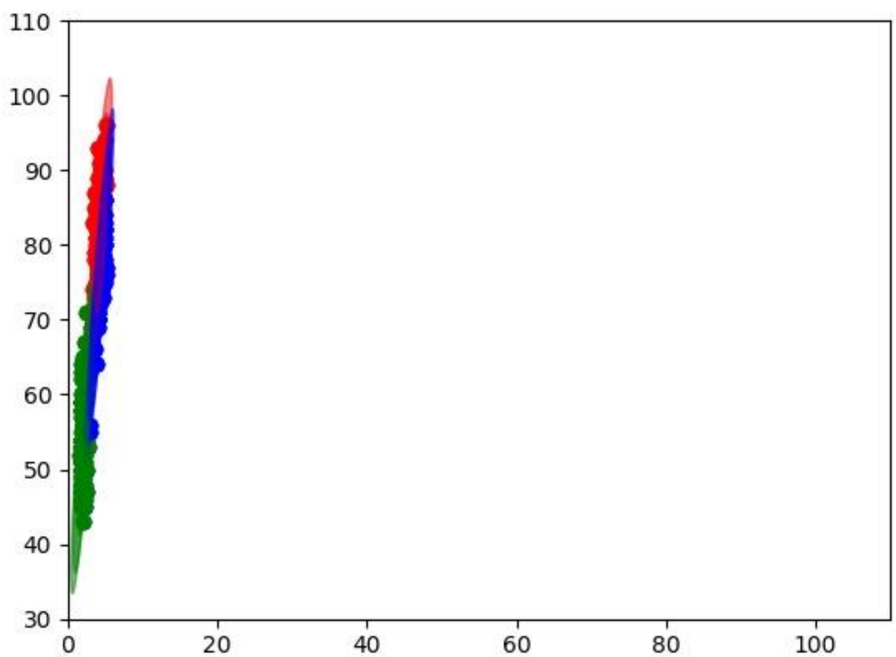

Fig task3_gmm_iter3.jpg

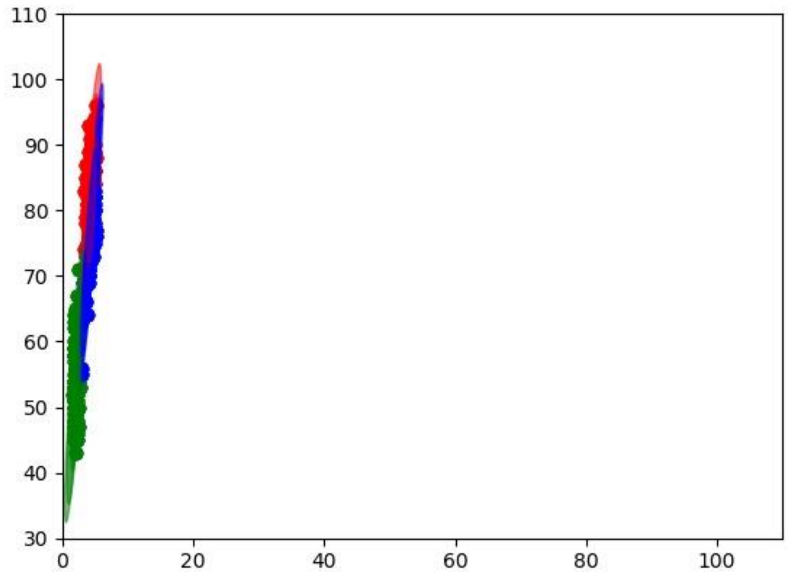

Fig task3_gmm_iter4.jpg

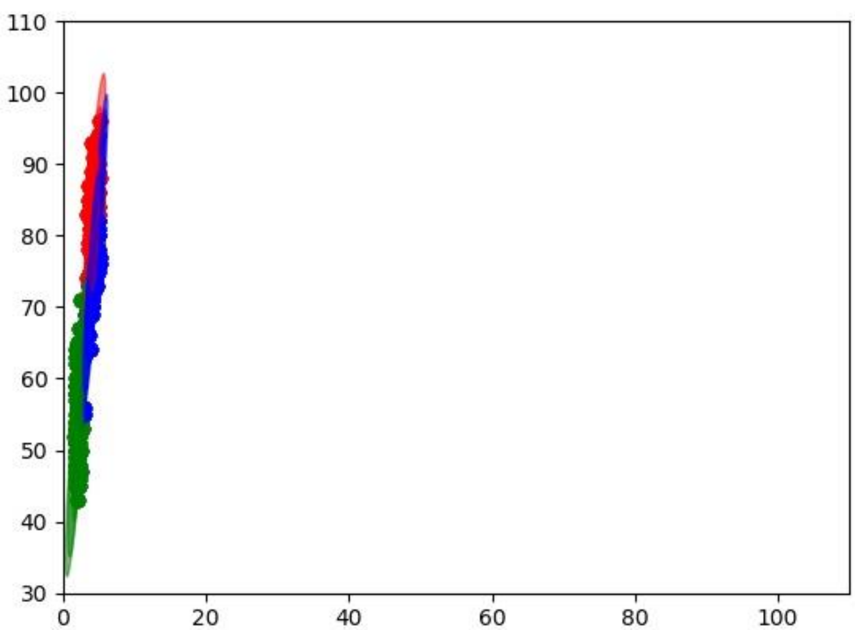

Fig task3_gmm_iter5.jpg

Code Output:

Task 3.5 A:

```
Task 3.5 A :
  New Theta Values : {'Mu': array([[5.3165079 , 3.21527292],
 [5.61129795, 3.38505311],
 [5.60443565, 3.14420061]]), 'BigSigma': array([[[ 0.40837607, -0.00690868],
  [-0.00690868,  0.15414263]],

 [[ 0.40543064, -0.08101016],
  [-0.08101016,  0.24507592]],

 [[ 0.49728723, -0.04157679],
  [-0.04157679,  0.11914189]]]), 'pi': array([0.50656611, 0.2067513 , 0.28668259])}
  Current Theta Values : {'Mu': array([[5.3165079 , 3.21527292],
 [5.61129795, 3.38505311],
 [5.60443565, 3.14420061]]), 'BigSigma': array([[[ 0.40837607, -0.00690868],
  [-0.00690868,  0.15414263]],

 [[ 0.40543064, -0.08101016],
  [-0.08101016,  0.24507592]],

 [[ 0.49728723, -0.04157679],
  [-0.04157679,  0.11914189]]]), 'pi': array([0.50656611, 0.2067513 , 0.28668259])}
  ProbabilityDistribution Matrix : [[0.14778737 0.07715177 0.10839419]
  [0.219909   0.03006055 0.08336377]
  [0.13466313 0.05990598 0.13876423]
  [0.21410142 0.04279622 0.0764357 ]
  [0.13790085 0.14210056 0.05333192]
  [0.19947439 0.04149969 0.09235925]
  [0.2043486  0.04337258 0.08561216]
  [0.10651287 0.09541787 0.13140259]
  [0.1793481  0.08995671 0.06402852]
  [0.14450797 0.06690908 0.12191629]]


  Total Num of Iterations taken to reach Convergence : 5
  Final Theta Values : {'Mu': array([[5.01492339, 3.05036551],
 [5.50832159, 3.71922232],
 [5.84045756, 3.03012592]]), 'BigSigma': array([[[ 0.25491429, -0.00729867],
  [-0.00729867,  0.01488842]],

 [[ 0.14895973, -0.12485142],
  [-0.12485142,  0.23540046]],

 [[ 0.53901973,  0.00402816],
  [ 0.00402816,  0.01606941]]]), 'pi': array([0.36048106, 0.10133229, 0.53818665])}
  Final Probability Distribution Matrix : [[7.32424683e-02 4.45635433e-02 2.28296847e-01]
  [2.52114273e-01 8.86086625e-06 1.56253211e-01]
  [5.60753242e-02 3.79272870e-02 2.78819254e-01]
  [2.91465096e-01 2.84440148e-04 1.00132806e-01]
  [5.10427888e-18 1.27317674e-01 3.65555982e-20]
  [2.41122454e-01 4.29839877e-04 1.69939324e-01]
  [2.61438558e-01 4.11860654e-04 1.41605808e-01]
  [1.17327038e-02 1.73330008e-03 4.85456269e-01]
  [7.99117908e-08 1.27317645e-01 3.11134338e-09]
  [8.01884728e-02 1.62696527e-02 3.31653267e-01]]
```

'New Theta Values', 'Current Theta Values', 'ProbabilityDistribution Matrix' are for the first iteration. The final values are for the final iteration. The theta values contain the new cluster centers as 'Mu'

Task 3.5 B:

Task 3.5 B :

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
ITER 1
New Theta Values : {'Mu': array([[ 3.96276561, 79.35728095],
[ 2.37099456, 59.47584954],
[ 3.93478598, 71.88809962]]), 'BigSigma': array([[[  0.67499521,   6.99959018],
 [  6.99959018, 100.578064  ]],

[[  0.70503148,   8.11314944],
 [  8.11314944, 121.71548821]],

[[  0.95454952,  10.66112771],
 [ 10.66112771, 139.10514248]]]), 'pi': array([0.35282341, 0.29215838, 0.35501822])}
Current Theta Values : {'Mu': array([[ 3.96276561, 79.35728095],
[ 2.37099456, 59.47584954],
[ 3.93478598, 71.88809962]]), 'BigSigma': array([[[  0.67499521,   6.99959018],
 [  6.99959018, 100.578064  ]],

[[  0.70503148,   8.11314944],
 [  8.11314944, 121.71548821]],

[[  0.95454952,  10.66112771],
 [ 10.66112771, 139.10514248]]]), 'pi': array([0.35282341, 0.29215838, 0.35501822])}

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
ITER 2
New Theta Values : {'Mu': array([[ 4.12175137, 81.10000707],
[ 2.22936693, 57.56366544],
[ 4.0547864 , 73.22831083]]), 'BigSigma': array([[[  0.39259018,   3.52115658],
 [  3.52115658,  59.38307281]],

[[  0.44836612,   5.3053511 ],
 [  5.3053511 ,  93.08788902]],

[[  0.78665328,   8.74656137],
 [  8.74656137, 115.34614251]]]), 'pi': array([0.36697391, 0.28290797, 0.35011812])}
Current Theta Values : {'Mu': array([[ 4.12175137, 81.10000707],
[ 2.22936693, 57.56366544],
[ 4.0547864 , 73.22831083]]), 'BigSigma': array([[[  0.39259018,   3.52115658],
 [  3.52115658,  59.38307281]],

[[  0.44836612,   5.3053511 ],
 [  5.3053511 ,  93.08788902]],

[[  0.78665328,   8.74656137],
 [  8.74656137, 115.34614251]]]), 'pi': array([0.36697391, 0.28290797, 0.35011812])}

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
ITER 3
New Theta Values : {'Mu': array([[ 4.23511269, 82.19064976],
[ 2.07695134, 55.47965479],
[ 4.18097714, 74.61624226]]), 'BigSigma': array([[[ 0.17621894,  1.03804286],
 [ 1.03804286, 31.73936007]],

[[ 0.15789017,  1.75813042],
 [ 1.75813042, 51.27041065]],

[[ 0.53863619,  5.86547915],
 [ 5.86547915, 80.99117408]]]), 'pi': array([0.40173653, 0.28745221, 0.31081126])}
Current Theta Values : {'Mu': array([[ 4.23511269, 82.19064976],
[ 2.07695134, 55.47965479],
[ 4.18097714, 74.61624226]]), 'BigSigma': array([[[ 0.17621894,  1.03804286],
 [ 1.03804286, 31.73936007]],

[[ 0.15789017,  1.75813042],
 [ 1.75813042, 51.27041065]],
```

```
[[ 0.53863619,  5.86547915],
 [ 5.86547915, 80.99117408]]]), 'pi': array([0.40173653, 0.28745221, 0.31081126])}

 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 ITER 4
 New Theta Values : {'Mu': array([[ 4.25950656, 82.33288169],
[ 2.02439588, 54.53257604],
[ 4.27264361, 75.77831487]]), 'BigSigma': array([[[ 0.14670209,  0.5796213 ],
 [ 0.5796213 , 26.36592017]],

[[ 0.06347558,  0.48745794],
 [ 0.48745794, 35.01691814]],

[[ 0.30284402,  3.04277627],
 [ 3.04277627, 47.27348264]]]), 'pi': array([0.45227837, 0.29567925, 0.25204238])}
 Current Theta Values : {'Mu': array([[ 4.25950656, 82.33288169],
[ 2.02439588, 54.53257604],
[ 4.27264361, 75.77831487]]), 'BigSigma': array([[[ 0.14670209,  0.5796213 ],
 [ 0.5796213 , 26.36592017]],

[[ 0.06347558,  0.48745794],
 [ 0.48745794, 35.01691814]],

[[ 0.30284402,  3.04277627],
 [ 3.04277627, 47.27348264]]]), 'pi': array([0.45227837, 0.29567925, 0.25204238])}

 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 ITER 5
 New Theta Values : {'Mu': array([[ 4.26589659, 82.56841618],
[ 2.01703985, 54.38982439],
[ 4.27760722, 76.14633309]]), 'BigSigma': array([[[ 0.14632566,  0.49772435],
 [ 0.49772435, 25.14813601]],

[[ 0.05460536,  0.37029901],
 [ 0.37029901, 33.79836547]],

[[ 0.25346635,  2.39149908],
 [ 2.39149908, 39.49935999]]]), 'pi': array([0.47984179, 0.30286244, 0.21729576])}
 Current Theta Values : {'Mu': array([[ 4.26589659, 82.56841618],
[ 2.01703985, 54.38982439],
[ 4.27760722, 76.14633309]]), 'BigSigma': array([[[ 0.14632566,  0.49772435],
 [ 0.49772435, 25.14813601]],

[[ 0.05460536,  0.37029901],
 [ 0.37029901, 33.79836547]],

[[ 0.25346635,  2.39149908],
 [ 2.39149908, 39.49935999]]]), 'pi': array([0.47984179, 0.30286244, 0.21729576])}
```

'New Theta Values', 'Current Theta Values', 'ProbabilityDistribution Matrix' for the first five iterations. The theta values contain the new cluster centers as 'Mu'.
The GMM code works properly. However, I am not too sure about the eclipse plots I have plotted – I am not too sure if I have used the plot_cov_ellipse function and the Eclipse class correctly. Hence, I have included the values as well here. Pls consider these values to check the validity of my GMM code, instead of the graphs, if the graphs aren't accurate.

As a bonus, on this bonus, I let the algorithm run till the max log likelihood stops increasing. The code ran till the 5th iteration, at which point it fell into a local maxima. I didn't have the time to alter my code to cross this local maxima – and it wasn't asked for in the question. In any case, here is the output from the final iteration:

```
 Total Num of Iterations taken to reach Convergence : 5
 Final Theta Values : {'Mu': array([[ 4.2729826 , 82.72681531],
[ 2.01800222, 54.38583395],
[ 4.27176979, 76.26936   ]]), 'BigSigma': array([[[ 0.14804665,  0.47072552],
 [ 0.47072552, 24.79188924]],

[[ 0.05520698,  0.36669436],
 [ 0.36669436, 33.70740478]],

[[ 0.24319134,  2.2293113 ],
 [ 2.2293113 , 37.75175862]]]), 'pi': array([0.49289727, 0.31156907, 0.19553366])}
```