CO543 – Image Processing

Lab 05

Image segmentation

Objectives

- What does the function kmeans() do and how? What does the function reshape do?

Image segmentation is an image processing task in which the image is segmented or partitioned into multiple regions such that the pixels in the same region share common characteristics. There are two forms of image segmentation:

1. Local segmentation – It is concerned with a specific area or region of the image.
2. Global segmentation – It is concerned with segmenting the entire image.

Amplitude segmentation

Amplitude segmentation is the simplest way to segment an image. It is useful (efficient) when amplitude (the intensity of the pixel) defines the scene regions precisely enough. To perform amplitude segmentation, We use a first order histogram.

Manual determination of threshold

In this, we will determine the segmentation threshold based on the histogram. If there are several regions with the more-or-less uniform color within each of them, then we expect histogram to be bimodal or multimodal.
The interpretation of the histogram depends on the histogram itself and/or knowledge of the image content (e.g. number of regions and similarity of their colors).
An example of such a histogram is given in Figure 01. In this histogram we see several groups, and we would choose a threshold value as 120 for segmentation. An optimal choice of the threshold corresponds to the value between two maxima.

First we will be inspecting the histogram.

```
import cv2 as cv import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('testpat1.tif',0)
#Magnitude spectrum of the image f
= np.fft.fft2(img)
```

```
fshift = np.fft.fftshift(f)    magnitude_spectrum =

20*np.log(np.abs(fshift)) plt.subplot(121), plt.imshow(img,

cmap = 'gray') plt.title('Input Image'), plt.xticks([]),

plt.yticks([]) plt.subplot(122),plt.imshow(magnitude_spectrum,

cmap = 'gray') plt.title('Magnitude Spectrum'),

plt.xticks([]), plt.yticks([]) plt.show()

#histogram of the image
plt.hist(img.ravel(),256,[0,256])
plt.title('Histogram') plt.show()
```

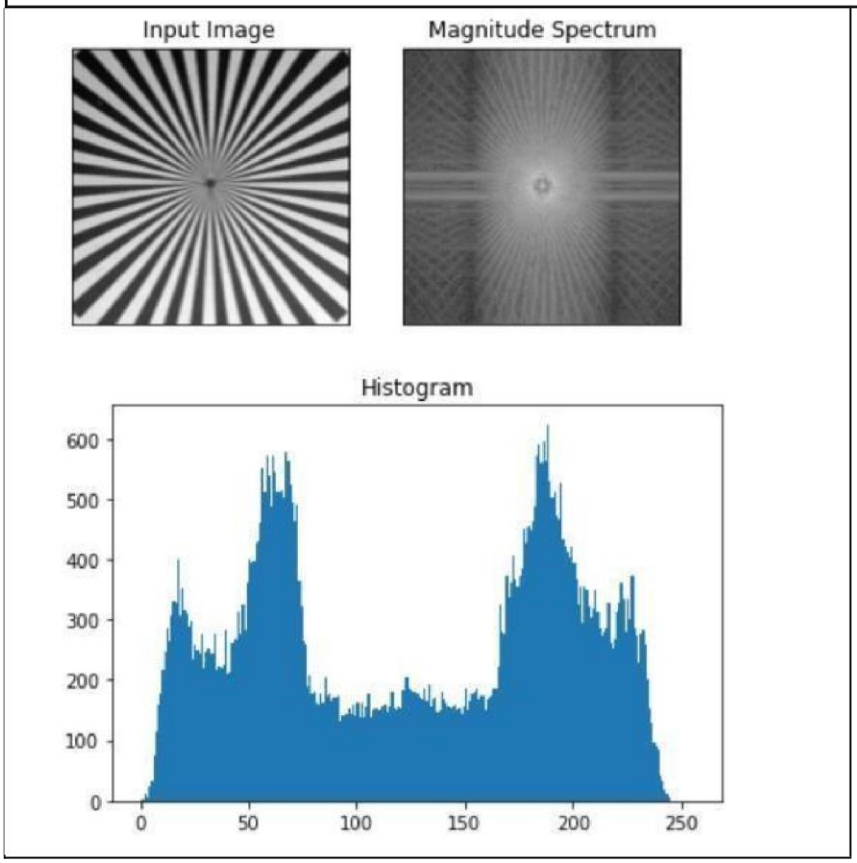

Figure 01: Histogram and the Magnitude transformation of the image

For segmentation we can use the function *cv.threshold()* which will segment the input image with thresholds that are given as a second input parameter.
There are few algorithms that are used for thresholding. You may read this.

```
# applying different thresholding techniques on the input image

# all pixels value above 120 will be set to 255 ret, thresh1 =

cv.threshold(img, 120, 255, cv.THRESH_BINARY) ret, thresh2 =

cv.threshold(img, 120, 255, cv.THRESH_BINARY_INV) ret, thresh3

= cv.threshold(img, 120, 255, cv.THRESH_TRUNC) ret, thresh4 =

cv.threshold(img, 120, 255, cv.THRESH_TOZERO) ret, thresh5 =

cv.threshold(img, 120, 255, cv.THRESH_TOZERO_INV)

titles = ['Original
Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']

images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]

for i in range(6):


plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
plt.title(titles[i])
plt.xticks([]),plt.yticks([]) plt.show()
```

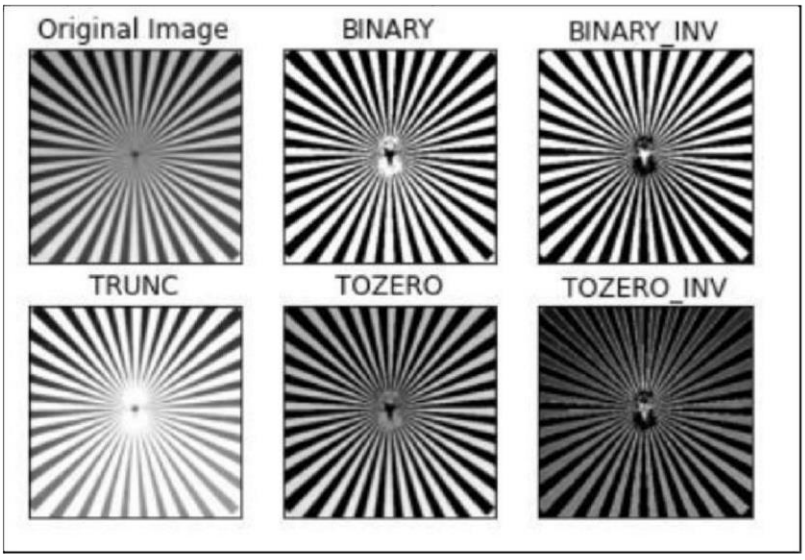

Figure 02: Output of the image thresholding algorithms

Adaptive Thresholding

In the previous section, we used a global value as threshold value. But it may not be good in all the conditions where the image has different lighting conditions in different areas. In that case, we go for adaptive thresholding.

Here, the algorithm calculates the threshold for a small region of the image. So we get different thresholds for different regions of the same image and it gives us better results for images with varying illumination.

Adaptive Method - It decides how thresholding value is calculated.

- cv2.ADAPTIVE_THRESH_MEAN_C : threshold value is the mean of neighbourhood area.
- cv2.ADAPTIVE_THRESH_GAUSSIAN_C : threshold value is the weighted sum of neighbourhood values where weights are a gaussian window.

```
img = cv2.imread('testpat1.tif',0) img =

cv2.medianBlur(img,5) ret,th1 =

cv2.threshold(img,120,255,cv2.THRESH_BINARY)



th2        =        cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY,11,2)                                    th3                =
cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINA

RY,11,2) titles = ['Original Image', 'Global Thresholding (v = 120)',

'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding'] images =

[img, th1, th2, th3] for i in range(4):



    plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
plt.title(titles[i])      plt.xticks([]),plt.yticks([])
plt.show()
```

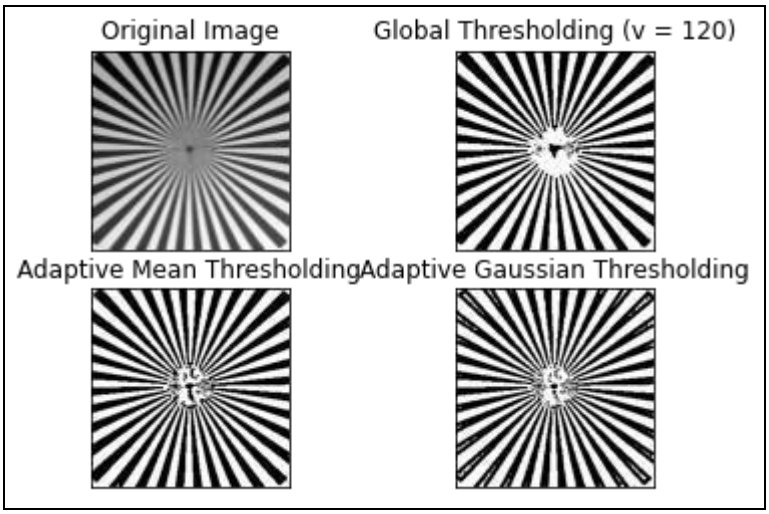

Figure 03: Output of the adaptive thresholding algorithms

Automatic determination of the threshold

In this part of the exercise, we will demonstrate the success of K − means algorithm in automatic determination of the threshold. We will use kmeans(). Output variable c of the function contains the calculated centers of the classes in which we group the image pixel intensities.

```
img = cv.imread('testpat1.tif',0) reshapedImage = np.float32(img.reshape(1,-
1)) numberOfClusters = 2 stopCriteria = (cv.TERM_CRITERIA_EPS +
cv.TERM_CRITERIA_MAX_ITER, 100, 0.1)


ret, labels, clusters = cv.kmeans(reshapedImage, numberOfClusters, None,
stopCriteria, 10, cv.KMEANS_RANDOM_CENTERS) clusters =
np.uint8(clusters) intermediateImage = clusters[labels.flatten()]
clusteredImage = intermediateImage.reshape((img.shape))
plt.imshow(clusteredImage, cmap='gray')
```

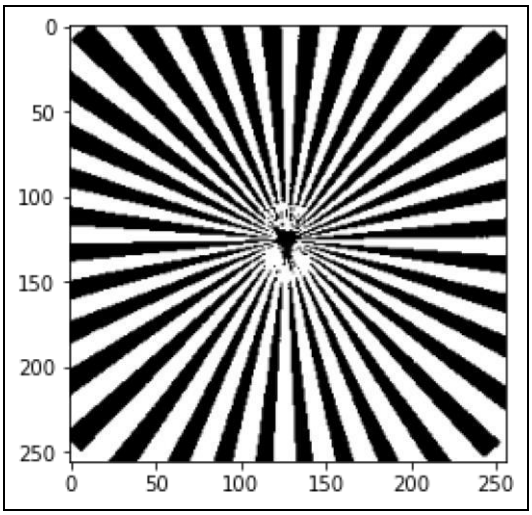

Figure 04: Output of the k-means thresholding algorithm

Extraction of edges (Canny Edge Detection)

For finding the edges in the image, we can use the function cv2.Canny(), which supports several methods for finding the edges. The function gives a binary image (same dimensions as the input image), which contains only the detected edges from the input image.

```
img = cv.imread('testpat1.tif',0) edges =

cv2.Canny(img,100,200) plt.imshow(edges,cmap = 'gray')

plt.title('Edge Image'), plt.xticks([]), plt.yticks([])

plt.show()
```

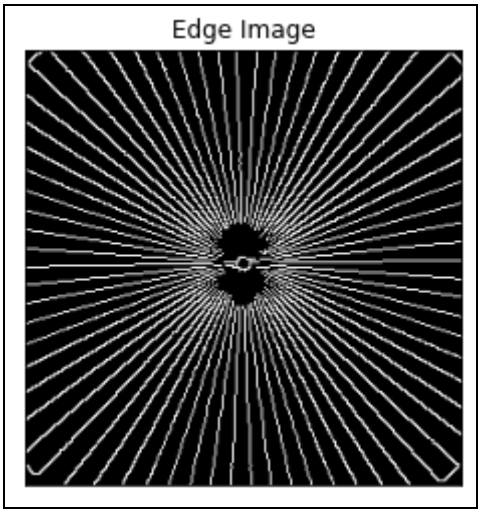

Figure 05: Output of the Canny Edge Detection
You may try sobel, prewitt, roberts, zerocross and log methods in given three images.

Texture segmentation

We will segment the textures based on selected features. First, for each pixel in the image, we will calculate the selected feature on the defined neighborhood of the pixel Image containing the feature values is the image we will segment. Gabor kernel filter used for this.

```
import numpy as np import cv2

import matplotlib.pyplot as

plt

img = cv2.imread('testpat1.tif',0)

ksize = 5

sigma = 5

theta = 1*np.pi/4

lamda = 1*np.pi/4

gamma=0.9

phi = 0.8    kernel = cv2.getGaborKernel((ksize, ksize), sigma, theta, lamda, gamma, phi,

ktype=cv2.CV_32F)  fimg = cv2.filter2D(img, cv2.CV_8UC3, kernel)

kernel_resized = cv2.resize(kernel, (400, 400))
plt.subplot(1,3,1), plt.imshow(img, cmap='gray')
plt.subplot(1,3,2), plt.imshow(fimg, cmap='gray')
plt.subplot(1,3,3), plt.imshow(kernel_resized, cmap='gray')
plt.show()
```

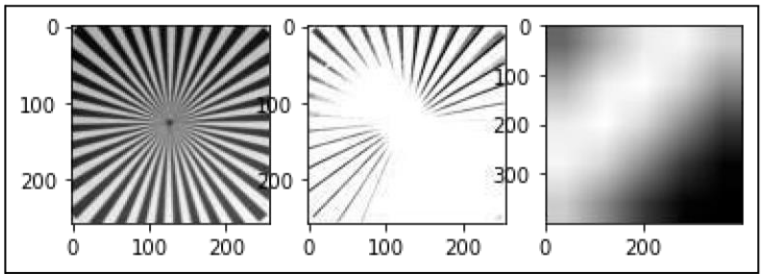

Image Inpainting

OpenCV implements two inpainting algorithms:

1. “An Image Inpainting Technique Based on the Fast Marching Method”, Alexandru Telea, 2004:

   This is based on Fast Marching Method (FMM). Looking at the region to be inpainted, the algorithm first starts with the boundary pixels and then goes to the pixels inside the boundary. It replaces each pixel to be inpainted with a weighted sum of the pixels in the background, with more weight given to nearer pixels and boundary pixels.

2. “Navier-Stokes, Fluid Dynamics, and Image and Video Inpainting”, Bertalmio, Marcelo, Andrea L. Bertozzi, and Guillermo Sapiro, 2001: This algorithm is inspired by partial differential equations. Starting from the edges (known regions) towards the unknown regions, it propagates isophote lines (lines that join same-intensity points). Finally, variance in an area is minimized to fill colors.

```
import numpy as np  import
cv2


# Open the image.
img = cv2.imread('cat_damaged.png')


# Load the mask.
mask = cv2.imread('cat_mask.png', 0)


# Inpaint.
dst = cv2.inpaint(img, mask, 3, cv2.INPAINT_NS)


# Write the output.
cv2.imwrite('cat_inpainted.png', dst)
```

By using segmentation Capture the Damaged area and restore the original image with cv2.inpaint Filter

Exercise

1. Read the image texture.tif. Display the image. How many textures are there in the image? Describe them. Find images here (link updated): https://drive.google.com/drive/folders/1fPzXha_xapiXJhEbiatJ2pWpOyYfJ53?usp=sharing
2. Select several features and calculate them on blocks of size of 12 × 12 using Gabor filter. Display the calculated features and estimate which ones can be used to segment given structure.
   For the selected images apply the K-means method and comment on the results.
3. Calculate the spectra energy (without the DC component) feature on the texture.tif image, on the blocks of size 12×12. Is this feature good for segmentation of the textures on this image?
   Segment the energy image using the K-means method and comment on the results.
4. By using segmentation and cv2.inpaint restore the "Efac.jpg" image. In your report explain the steps you used to achieve it.

Submission

You need to submit a jupyter notebook file containing the relevant programs and functions named according to the relevant question names as indicated in the lab sheet. Make sure to include the input images you used to run the codes as well.
You need to submit a PDF file (e20XXXresults.pdf) displaying all results from your code (your input and output images under each section after performing the required functions). You can submit a single ZIP file as e20XXXlab05_segmentation.zip including all:
e20XXXlab05_segmentation.zip
• Python source codes
• All Input and output images
• A PDF file with results
Note: XXX indicates your registration number in all cases.