CSC3014 Computer Vision

Lab 5: Representation and Description

A. Region Labelling

Given a binary image, it is not hard for us to identify the number of regions and the location of them. However, it is a completely different story is we want the computer to do that for us. In this case, we need to first label the regions before the computer can process the regions accordingly. This can be done by using the cv2.connectedComponents() function. In fact, we had used this to generate the markers needed for watershed segmentation in the previous exercise. As shown below is the number of outputs you would obtain, as well as the number of input parameters required when using the function.

```
noRegion, labelledBinaryImage = cv2.connectedComponents(binaryImage, connectivity=n)
```

The function returns two arrays to you, first one is the number of regions, and second one is the labelled binary image. It is important to take note that the background is also considered as a region. Hence, if you only have, for example, 5 foreground regions, the function will indicate that there are 6 regions in total. Regarding the input parameters, you only have to provide the binary image that you would like it to process. The connectivity parameter is optional. If you did not specify the connectivity, then it will take the default that is 8-connectivity (each pixel has 8 neighbours). If you would like to use 4-connecivity (but very rare we would want to do that), then just put connectivity=4 as the second input parameter.

Exercise

Create the binary image shown below and label the regions by using cv2.connectedComponents(). Try to use 4-connectivity and 8-connectivity to observe the different between the two. Please keep this binary image because you will need to use it again in the subsequent exercise2.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	0	0	0	1	1	0	0	0
3	0	0	1	1	1	0	0	0	1	1	1	1	0
4	0	0	1	1	1	0	0	0	1	1	1	1	0
5	0	1	0	0	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	1	1	1	1	1	1	1	0
9	0	0	0	0	0	1	1	1	1	1	1	1	0
10	0	0	0	0	0	1	1	1	1	1	1	1	0
11	0	0	0	0	0	1	1	1	1	1	1	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0

B. Contours

Previously, we had learned how to obtain the boundary of a region by using morphological operations. But in addition to those, we can also use the cv2.findCountours() function to obtain the boundary of every single region in a binary image. From this point onwards, we will use the term contour instead of boundary, so that it matches with the term used by the function. As shown below is the number of outputs you would obtain, as well as the number of input parameters required when using the function.

```
contours, hierarchy = cv2.findContours(binaryImage, retrievalMode, approximationMethod)
```

This function also returns two arrays to you, the first one contains the set of points (coordinates) you need to form the contour of every single region. The second one is used to explain the relationship of different regions. This is useful if you have, for example, a region that is nested in another region. Then you can use the values inside hierarchy to trace that. But at this point, we are not interested with the relationship among the regions. Hence, you can replace hierarchy with _ when you use the function.

There are three input parameters that you need to provide when using the function. The first one is self-explanatory, that is the binary image that you would like to process. You have to take note on the second and third parameters when using the function because they control the number of contours, and the number of points to be saved for each contour respectively. The modes and methods that you can choose for each of them are summarized and described in the following tables.

Retrieval Mode	Description
cv2.RETR_TREE	Retrieve all contours (external and all nested contours).
cv2.RETR_EXTERNAL	Retrieve only the most outer contours.

Approximation Method	Description
cv2.CHAIN_APPROX_NONE	Store all the contour points (coordinates).
cv2.CHAIN_APPROX_SIMPLE	Compress contour points and leaves only their end points.

Exercise

Apply the cv2.findCountour() function on the binary image you created in the previous exercise. Try to understand how the contour points are stored and the difference(s) between the two approximation methods. It is very important to understand how the contour points are stored so that you can further process the contour points when necessary.

If you would like to display the contours on the image, you can use the cv2.drawContours() function. This step is only required for visualization purpose. As shown below is the number of outputs you would obtain, as well as the number of input parameters required when using the function.

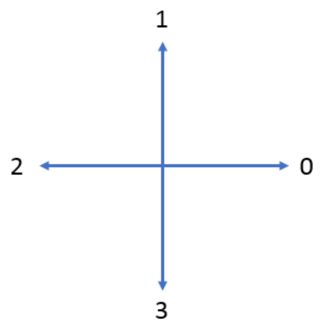
```
imageWithContours = cv2.drawContours(image, contours, noContours, colour, lineThickness)
```

The function only returns one array to you, which is the image with contours drawn using the specified colour and thickness. You need to provide five input parameters when using the function. First one is the image where you want to draw the contours. Second one is the contour points that you obtained when using the cv2.findContours() function. The third one is the number of contours you would like to draw. For example, if there are 8 sets of contour points, you can choose to only draw, for example, the first 5 of them. If you would like to draw all the contours, then just put a -1 for this parameter. Followed by this is the colour of the contours. You have to specific the colour in RGB format and enclosed the three values in round brackets separated by comma. For example, if you would like to use red colour then you have to put (0, 0, 255) as the fourth parameter (if the colour image is arranged in BGR order). The last one is the line thickness, which can be any integer value larger or equivalent to 1. As shown below is a sample code to extract and draw the contours on a colour image – "scissors.png" by using cv2.findCountours() and cv2.drawContours().

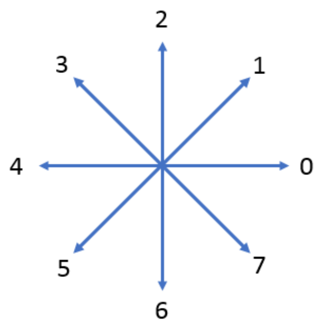
```
#Read colour image "scissors.png" as save it as scissors (arranged in BGR order).
scissors = cv2.imread("scissors.png")

#Convert to grayscale before binarization (use Otsu's method to determine the threshold).
scissorsGray = cv2.cvtColor(scissors,cv2.COLOR_BGR2GRAY)
_, scissorsBinary = cv2.threshold(scissorsGray, 0, 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)

#Find all the contours (external and nested contours).
contours,_ = cv2.findContours(scissorsBinary, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)

#Draw the contours on colour version of scissors with blue colour line with thickness of 5.
scissorsContours = cv2.drawContours(scissors.copy(), contours,- 1, (255, 0, 0), 5)

#Create a figure window to plot the colour version of scissors with all contours.
#Have to first convert from BGR to RGB order if using imshow from pyplot.
pt.figure()
pt.imshow(cv2.cvtColor(scissorsContours,cv2.COLOR_BGR2RGB))
```

C. Chain Code

Although we can use cv2.findContours() to obtain the contours for every single region in a binary image, there is no existing function for us to get the shape number required to describe the contours. Chain code treats the contour as a sequence of connected pixels. Based on the number of neighbours that a pixel can have (4-connectivity or 8-connectivity), there are two different sets of code that can be used to code the directions when moving from one pixel to another. As shown in figure below is two different schemes (4-connectivity or 8-connectivity) that can be used to code the directions. Chain code obtained using either one of the two schemes is also known as the Freeman chain code.



(a) 4-Connectivity



(b) 8-Connectivity

Exercise

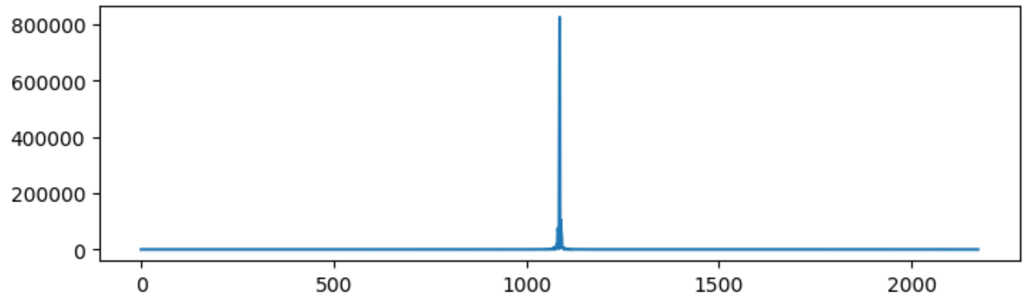
Based on one of the contours that you have extracted from the second exercise (contours extracted from the small binary image), write your own function to produce a shape number for it. In this case, assume a pixel can have up to 8 neighbours. To do this, you should first obtain the normal chain code before trying to rotate the chain code to (i) find the chain code with smallest magnitude (to make the chain code invariant to starting point), and (ii) to find the different of chain code (to make the chain code invariant to rotations).

D. Fourier Descriptors

Instead of saving the row and column coordinates that used to represent the contours, we can also choose to save their equivalent Fourier representations, known as the Fourier descriptors. In this case, we need to first convert first contour point to a pair of complex number, before we can use Fourier transform to transform them into a set of Fourier descriptors. Most of the time, we can get a very good approximation of the contour by using only a small amount of Fourier descriptors. In the sample code shown next page, we will try to observe the contour reconstructed by using only 2% of the Fourier descriptors.

In this case, we will use back the same colour image – "scissors.png". Assuming that you have obtained the binary version of "scissors", we will only be extracting the most outer contour this round. Then, we need to convert each contour point to a pair of complex number, and put them into a list. When the list is ready, Fast Fourier Transform (FFT) is applied to transform the set of complex numbers into frequency domain. Please take note that we should be using np.fft.fft() instead of np.fft.fft2(), since we are dealing with a vector instead of an image. After conversion, you still get a set of complex numbers, but the numbers will not be the same. Like what you have done to an image, we also need to perform a shift here to shift all the low frequency components to the center.

If you try to look at the magnitudes of all the Fourier Descriptors as shown below, you can see that the magnitudes of higher frequency components are very small and are almost negligible when compared to the low frequency components standing at the center. This gives us the option to only retain the low frequency components and remove the high frequency components (changing them to zeros). But when removing the high frequency components, we have to remove them balancedly from both ends (left and right hand sides) of a vector. For example, if you decided to remove 100 Fourier descriptors, then you have to remove 50 descriptors from each end.



After we have removed the high frequency components, all we need to do is to reverse the process above to reconstruct the contour points and subsequently the contour.

```
#Find only the most outer contours.
contours,_ = cv2.findContours(scissorsBinary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

#Convert each contour point into a pair of complex number and put all of them into a list.
complexNumberList = []
contourPoints = contours[0]
maxNumPoints = len(contourPoints)
for point in range(0,maxNumPoints):
    complexNumberList.append(contourPoints[point,0,0] + contourPoints[point,0,1]*1j)

#Perform DFT using FFT.
FourierDescriptors = np.fft.fft(complexNumberList)

#Shift all the low frequency components to the center, high frequency components to the sides.
shiftedFourierDescriptors = np.fft.fftshift(FourierDescriptors)

#Calculate and visualize the magnitude of each complex number.
magShiftedFourierDescriptors = np.abs(shiftedFourierDescriptors)
intMagShiftedFourierDescriptors = magShiftedFourierDescriptors.astype(np.int)
pt.figure()
pt.plot(list(range(0,maxNumPoints)),intMagShiftedFourierDescriptors)

#Determine the number of descriptors that should be removed (change to zero).
numRemovePercentage = 98
numRemoveDescriptor = int((maxNumPoints)*(numRemovePercentage/100))

#Remove the descriptors (change to zero) from both sides (left and right).
shiftedFourierDescriptors[0:int(numRemoveDescriptor/2)] = 0
shiftedFourierDescriptors[int(-numRemoveDescriptor/2)-1:-1] = 0

#Perform inverse shift and inverse DFT (using inverse FFT) to reconstruct the contour points.
invShiftedFourierDescriptors = np.fft.ifftshift(shiftedFourierDescriptors)
invFourierDescriptors = np.fft.ifft(invShiftedFourierDescriptors)

#Separate the row coordinates and column coordinates and round them to the nearest integers.
invRowCoordinates = invFourierDescriptors.real.astype(np.int)
invColCoordinates = invFourierDescriptors.imag.astype(np.int)

#Create copies of the original data structures used to hold the contour points.
#Copy the reconstructed row coordinates and column coordinates back to the original data structures.
invContourPoints = contourPoints.copy()
invContourPoints[:,0,0] = invRowCoordinates
invContourPoints[:,0,1] = invColCoordinates
invContours = contours.copy()
invContours[0] = invContourPoints

#Create a black image for drawing the contours (since we just want to see the contours).
[row,col,depth] = scissors.shape
blankImg = np.zeros((row,col,depth),dtype=np.uint8)
blankImgContours = cv2.drawContours(blankImg,invContours,-1,(255,0,0),5)
pt.figure()
pt.imshow(cv2.cvtColor(blankImgContours,cv2.COLOR_BGR2RGB))
```

Exercise

Try to adjust the percentage of Fourier descriptors (numRemovePercentage) that will be removed and observe the contour reconstructed based on the remaining descriptors.

E. Harris Corner Detection

There are many detectors that can be used to detect corners from an image, but among them, the one proposed by Chris Harris and Mike Stephens [1] is one of the most used detectors. This can be performed by using the cv2.cornerHarris() function. As shown below is a sample code of using this function to find the corners in grayscale version of "scissors".

```
#Find the corners in grayscale version of "scissors".
dst = cv2.cornerHarris(scissorsGray, 3, 3, 0.05)

#Create a copy of the colour version of "scissors"
scissorsCorners = scissors.copy()

#Set the threshold to be 0.01*dst.max(), so point with value more than this is considered as a corner point.
#Change the point to blue colour in the image.
scissorsCorners[dst>0.01*dst.max()] = [255, 0, 0]

#Plot the image to show the corner points.
pt.figure()
pt.imshow(cv2.cvtColor(scissorsCorners, cv2.COLOR_BGR2RGB))
```

The output from the cv2.cornerHarris() function is a set of locations where those corner points are located. It returns to you the score (or "probability") of a point being a corner point. Hence, you still need to set a threshold to determine the points that can be considered as the corner points. There are four input parameters to the function.

The first one is self-explanatory, which is the image where you would like to process. The second and third parameters are used to specify the neighbouring pixels that will be used in calculating the derivatives, and the size of the Sobel kernel. If you still remember what we have covered in edge detection, we can use Sobel kernel to calculate the derivatives of a region. This is required because the differences in intensity is measured based on the derivatives. The last parameter is known as the free parameter and it typically falls within the range of 0.04 to 0.06. Higher value gives you less false corners but you also tend to miss more real corners.

Exercise

Try to apply cv2.cornerHarris to a colour image of your own choice.

References

[1] C. Harris and M. Stephens, "A Combined Corner and Edge Detector", in Proceedings of 4th Alvey Vision Conference, 1988, pp.147-151.