Laboratory 3. Basic image segmentation techniques

Segmentation subdivides an image into constituent regions or objects. The level of detail to which the subdivision is carried depends on the problem being solved. Segmentation of nontrivial images is one of the most difficult tasks in image processing. Segmentation accuracy determines the eventual success or failure of more complex computer vision algorithms used in various analysis procedures.

3.1 Image thresholding

Image thresholding is necessary in many Computer Vision applications to split the image into components or segments – otherwise called segmentation. For example, counting the cars in a parking area requires segmenting the car images from the pavement background. One possibility is to analyze the colors and separate the gray pavement composing the background.

In simple thresholding, for every pixel is applied the same threshold value. If the pixel value is smaller than the threshold, it is set to 0, otherwise it is set to a maximum value. The resulting image is binary. The function `cv2.threshold` is used to apply the thresholding and it has the following syntax:

```
retval, dst = cv2.threshold(src, thresh, maxval, type[, dst])
```

with parameters:

`src` is the input array ot image (multiple-channel, 8-bit or 32-bit floating point).

`thresh` is the threshold value.

`maxval` is the maximum value to use with the `THRESH_BINARY` and `THRESH_BINARY_INV` thresholding types.

`type` is the thresholding type (`cv2.THRESH_BINARY`, `cv2.THRESH_BINARY_INV`, `cv2.THRESH_TRUNC`, `cv2.THRESH_TOZERO`, `cv2.THRESH_TOZERO_INV`). More information on the thresholding types can be found here.

`dst` is the output array or image of the same size and type and the same number of channels as `src`.

`retval` is the threshold value in case of using thresholding types such as Otsu or Triangle.

- **Ex. 3.1** Read the images '*grayShades.jpg*' and '*grayFlowers.jpg*' as grayscale images. Experiment with multiple types of thresholding by changing the parameter `type` and keeping the same threshold value for each image separately. Display and compare the output images.
- **Ex. 3.2** Read the image '*adeverinta.jpg*' as grayscale and repeat the previous exercise. Comment on the output.

3.1.1 Adaptive thresholding

Simple thresholding uses a global threshold value (the same value for all the pixels in the image). Adaptive thresholding is the method where the threshold value is computed for smaller regions and therefore, there will be different threshold values for different regions.

In OpenCV, adaptive thresholding is implemented by the method `cv2.adaptiveThreshold`. Following are the syntax of this function and its parameters.

```
dst = cv2.adaptiveThreshold(src, maxValue, adaptiveMethod, thresholdType,
   blockSize, C)
```

`src, maxValue, dst, thresholdType` – represent the same parameters as in a simple threshold operation.

`adaptiveMethod` – method to compute the threshold value. The 2 possible options are: `cv2.ADAPTIVE_THRESH_MEAN_C` (the mean of the neighborhood `blockSize` x `blockSize` minus constant `C`) or `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` (the threshold is a gaussian-weighted sum of the neighborhood values, minus constant `C`).

`blockSize` – indicates the size of the neighborhood area

`C` – constant subtracted from the mean.

- **Ex. 3.3** Read the image '*adeverinta.jpg*' as grayscale and apply adaptive thresholding (both options). Compare the results with the ones from Ex. 3.2 and justify the outputs. The image has different lighting conditions in different areas and a smoothing filter applied before using adaptive thresholding will reduce the noise.

3.1.2 Otsu's thresholding

Otsu algorithm computes the optimal threshold value for *bimodal* images. A *bimodal* image contains mainly 2 distinct colors, so the histogram presents only 2 peaks. An example is presented in Figure 1. A good threshold would be in between those 2 peaks, at the minimum bin value. The Otsu method determines the optimal global threshold value from the image histogram.

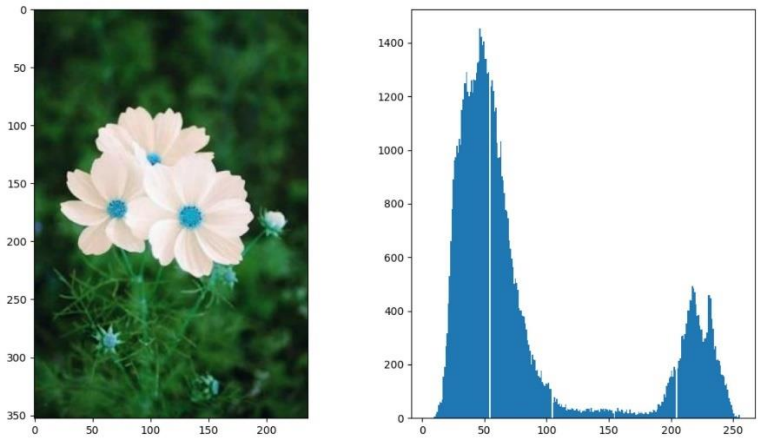

Figure 1. Original image (left) and its bimodal histogram of the intensity channel (right)

The `cv2.threshold()` function is used to apply Otsu's algorithm, where `cv2.THRESH_OTSU` is passed as an extra flag. The algorithm iterates over all possible thresholds. For a given threshold, it divides the image intensity values into 2 classes and calculates the intra-class variances for the two classes. The optimal threshold value is the one that minimizes the total intra-class variance (weighted sum of intra-class variances for the 2 classes). This optimal threshold is returned as the first output. An arbitrary example of function call is:

```
dst,threshOt = cv2.threshold(src,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
```

- **Ex. 3.4** Read the images '*rose.jpg*' and '*yellowFl.jpg*' as grayscale and apply Otsu thresholding. Compare the results with the previous methods. Experiment also with gaussian smoothing before the Otsu binarization.

3.2 Morphological Operations

In the process of segmenting images, after applying thresholding, white regions corresponding to distinct objects usually present imperfections that must be removed (or vice versa, black regions have unwanted white dots and spots). Those imperfections can be removed with morphological operations.

3.2.1 Dilation

Dilation signifies 'expansion' or becoming larger. White regions are enlarged using a structuring element moved along the boundary of the white area. An example is presented in Figure 2, where a white square is dilated using as structuring element a white circle.

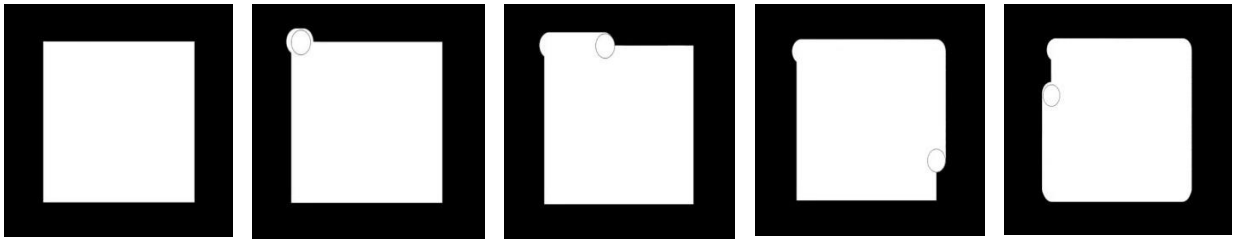

Figure 2. Dilation of a white square (left → right) with circle as structuring element

Structuring element can have various shapes and the results after dilation will be different. The OpenCV function that creates a structuring element is `cv2.getStructuringElement`. The function that implements dilation in OpenCV is:

```
dst = cv2.dilate(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]] )
```

with the following arguments:

`src` input image; the number of channels can be arbitrary, but the depth should be one of CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.

`dst` output image of the same size and type as `src`.

`kernel` structuring element used for dilation; if `kernel` is not specified, a 3 x 3 rectangular structuring element is used.

`anchor` position of the anchor within the element; default value (-1, -1) means that the anchor is at the element center.

`iterations` number of times dilation is applied; by default is 1.

`borderType` pixel extrapolation method.

`borderValue` border value in case of a constant border

- **Ex. 3.5** Read the image '*euro.jpg*' and transform it into a binary image using a simple threshold operation with 212 as threshold value. The coins should be mostly white (with some imperfections – see Figure 3 – the mask on the left side) on a black background. Try to remove the imperfections by dilation,

using 2 structuring elements: `cv2.MORPH_ELLIPSE` of size 7x7 and then a smaller kernel of 3x3. Start with 1 iteration for both kernels, then increase to 2 iterations. An example of the desired output mask is presented in Figure 3, on the right side. The desired output should label with white filled circles both coins and the white circles must not be joined together. Comment on the results.

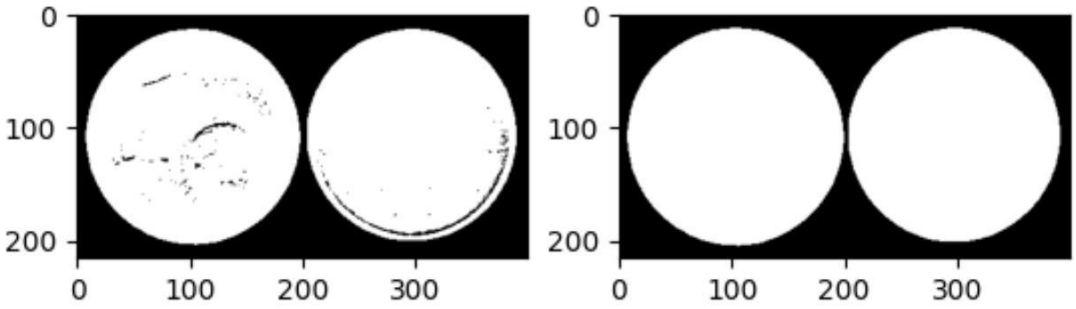

Figure 3. Left: original mask with imperfections. Right: correct mask obtained with morphological imperfections.

3.2.2 Erosion

The opposite of image dilation is erosion. Erosion means to gradually wear away or destroy (soil or rock). The effect of erosion is to shrink a shape. In dilation, mass is added to white regions, while in erosion mass is removed from the boundary of the white region, as if an eraser is moved along the boundary of the object. An example of erosion is presented in Figure 3.

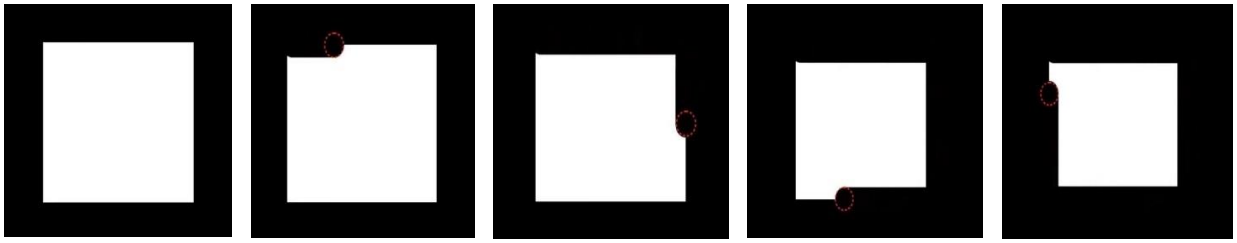

Figure 4. White square eroded using a structural element shaped as a circle.

The OpenCV function that implements erosion is `cv2.erode` and it has the same set of parameters as `cv2.dilate`.

```
dst = cv2.erode(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]] )
```

3.2.3 Opening and Closing

In case a binary mask contains small white spots that should be removed, the operations needed are erosion followed by dilation. Combined in this order, this operation is called ***morphological opening*** and is illustrated in Figure 4.

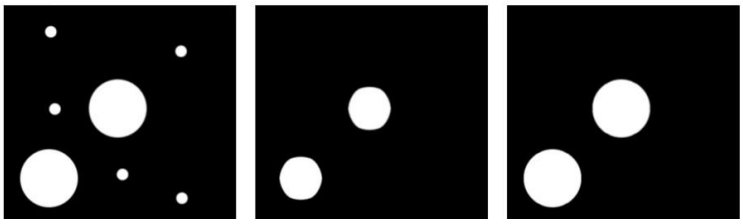

Figure 5. Left: original binary image. Middle: image after erosion. Right: image after dilation.

Small black holes inside white regions can be filled by performing dilation followed by erosion. This operation is called ***morphological closing*** and it is displayed in Figure 5.

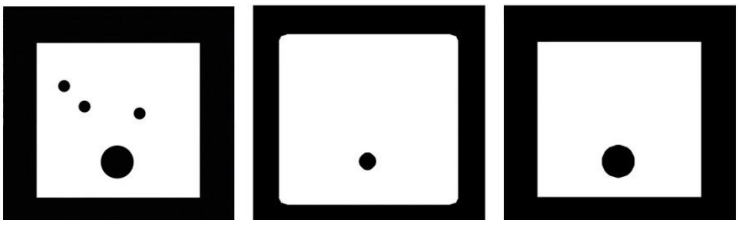

Figure 6. Left: original binary mask with 3 small holes. Middle: image after dilation. Right: image after erosion.

In OpenCV, the opening and closing operations are implemented using the function `cv2.morphologyEx`, with the following syntax:

```
imOpened = cv2.morphologyEx(src, cv2.MORPH_OPEN, kernel[, dst[, anchor[,
iterations[, borderType[, borderValue]]]]] )
imClosed = cv2.morphologyEx(src, cv2.MORPH_CLOSE, kernel[, dst[, anchor[,
iterations[, borderType[, borderValue]]]]] )
```

The function's parameters are:

`src` Source image. The number of channels can be arbitrary. The depth could be: CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.

`dst` Destination image of the same size and type as the source image.

`op` Type of a morphological operation (`cv2.MORPH_OPEN` / `cv2.MORPH_CLOSE`). Other types of morphological operations are: `cv2.MORPH_ERODE`, `cv2.MORPH_DILATE`, `cv2.MORPH_GRADIENT`, `cv2.MORPH_TOPHAT`, `cv2.MORPH_BLACKHAT`, `cv2.MORPH_HITMISS`. More information can be found here .

`kernel` Structuring element. It can be created using `cv2.getStructuringElement`.

`anchor` Anchor position with the kernel. Negative values mean that the anchor is at the kernel center.

`iterations` Number of times erosion and dilation are applied. Default value is 1. For 2 iterations, it will apply successively erosion → erosion → dilation → dilation.

`borderType` Pixel extrapolation method. More details can be found here.

`borderValue` Border value in case of a constant border.

- **Ex. 3.6** Read the image '*Coins.png*' and convert it into a grayscale image. Split it into the 3 color channels. Then decide which of the previous 4 matrices (grayscale, blue, green, red) can be used to obtain a binary mask for the coins. Use thresholding and morphological operations (dilatation, erosion, opening, closing) to obtain that binary mask, as illustrated in Figure 6.

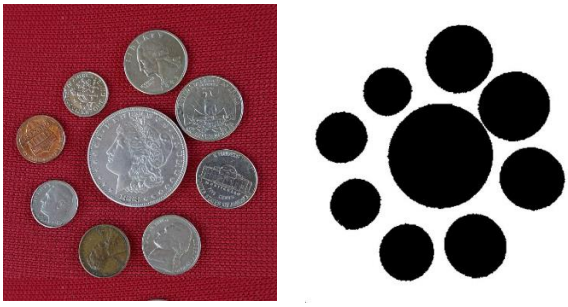

Figure 7. Coins image and binary mask.

3.3 Edge detection

Edges provide the topology and structure information of objects in an image. In order to accurately segment an object or multiple objects from the background, we can use the objects boundaries or edges. Detection of edges can be realized with Sobel filters and also with a second order derivative of an image, as a Laplacian operator. These methods have been introduced in Laboratory 2. Next sections present other more advanced methods available in OpenCV for edge detection.

3.3.1 Canny edge detector

The Canny edge detector is an algorithm composed of multiple stages, meant to extract a wide range of edges from images. It was developed by John F. Canny in 1986. The main steps necessary to obtain the edges are:

1. Noise reduction – edge detection is sensitive to image noise, so smoothing with a Gaussian blur kernel removes that noise.

2. Intensity gradients calculation – Sobel filters are needed for horizontal (G_x) and vertical gradients (G_y). Then, for each pixel, the gradient magnitude and direction is computed.

3. Non-maximum suppression – The pixels are checked if they are local maximum gradients in the edge direction. Only local maxima are considered, so the output image at this point will contain thin edges.

4. Hysteresis thresholding – at this stage it is decided which pixels belong to real edges, by comparing the intensity gradient with 2 threshold values: *maxThrsh* and *minThrsh*. Any edges with intensity gradient more than *maxThrsh* are sure to be edges and those below *minThrsh* are sure to be non-edges, so discarded. Those who lie between these 2 thresholds are classified edges or non-edges based on their connectivity to edges.

The OpenCV function that implements this algorithm is `cv2.canny`, with the following possible syntaxes:

```
dst = cv2.Canny(src, threshold1, threshold2[, dst[, apertureSize[, L2gradient]]])
dst = cv2.Canny(dx, dy, threshold1, threshold2[, dst[, L2gradient]])
```

The function parameters are:

`src` 8-bit input image.

`dst` output edge map; single channel 8-bit image, which has the same size as `src` .

`threshold1` first threshold for the hysteresis procedure.

`threshold2` second threshold for the hysteresis procedure.

`apertureSize` aperture size for the Sobel operator.

`L2gradient` a flag, indicating whether L_2 norm should be used instead of the default norm L_1, while computing the gradient magnitude

`dx` 16-bit x derivative of input `src` image (CV_16SC1 or CV_16SC3).

`dy` 16-bit y derivative of input `src` image (same type as `dx`).

- **Ex. 3.7** Write an application to find the edges using Canny detection and test it on the '*rose.jpg*'. The threshold values will be varied using two trackbars. What is the effect of threshold values? Experiment with multiple images. ***Indication***: use the function `cv2.createTrackbar()` and create a callback function which is called every time the trackbars are used. A similar example in C++ can be found here.

```
dst = cv2.Canny(imGray, 50, 150)
```

3.3.2 Contour analysis in OpenCV

There are several methods for edge detection. One convenient function in OpenCV for retrieving objects boundaries is `cv2.findContours`, with the syntax:

```
contours, hierarchy = cv2.findContours(src, mode, method[, contours[, hierarchy[, offset]]])
```

and parameters :

`src` - input image (8-bit single-channel). Non-zero pixels are treated as 1's. Zero pixels remain 0's, so the image is treated as binary. You can use `compare`, `inRange`, `threshold` , `adaptiveThreshold`, `Canny`, and others to create a binary image out of a grayscale or color one.

`contours` - Detected contours. Each contour is stored as a vector of points.

`hierarchy` - Optional output vector containing information about the image topology.

`mode` - Contour retrieval mode (`cv2.RETR_EXTERNAL`, `cv2.RETR_LIST`, `cv2.RETR_CCOMP`, `cv2.RETR_TREE`,…). More information about `mode` parameter can be found here.

`method` - Contour approximation method. (`cv2.CHAIN_APPROX_NONE`, `cv2.CHAIN_APPROX_SIMPLE`, `cv2.CHAIN_APPROX_TC89_L1` etc.)

`offset` - Optional offset by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context.

Contours properties can be extracted or underlined using the functions:

- `cv2.drawContours`
- `cv2.moments` (find out center of Mass, etc.)
- `cv2.contourArea`
- `cv2.arcLength`
- `cv2.minAreaRect`
- `cv2.minEnclosingCircle`
- `cv2.fitEllipse`

More details on the contours properties can be found here.

- **Ex. 3.8** Starting from the binary mask obtained in Ex.3.6, use contour detection to count the number of coins present in the image '*Coins.png*'. Use `cv2.RETR_LIST` as parameter *mode*, and `cv2.CHAIN_APPROX_SIMPLE` as *method*. Then use the function `cv2.drawContours` to draw all the contours detected previously.

3.4 Connected component analysis

Connected components analysis (or labeling) scans an image and groups its pixels into components based on pixel connectivity, i.e. all pixels in a connected component share similar pixel intensity values and are in some way connected with each other. Once all groups have been determined, each pixel is labeled with a gray level or a color (color labeling) according to the component it was assigned to. Extracting and labeling of various disjoint and connected components in an image is central to many automated image analysis applications.

Connected component labeling works by scanning an image, pixel-by-pixel (from top to bottom and left to right) in order to identify connected pixel regions – regions of adjacent pixels which share the same set of intensity values *V*. For a binary image *V={1}*. In a grayscale image, *V* takes values only in a certain range of values, for example: *V={51, 52, 53, ..., 77, 78, 79, 80}*.

Connected component labeling works on ***binary*** (or grayscale) images and different measures of connectivity are possible (generally *8-connectivity* considering 8 neighbors for one pixel, but *4-connectivity* is also possible). Considering a binary image, the connected components labeling operator scans the image by moving along a row until it comes to a point *p* (where *p* denotes the pixel to be labeled at any stage in the scanning process) for which *V={1}*. When this is true, it examines the four neighbors of *p* which have already been encountered in the scan (i.e. the neighbors (i) to the left of p, (ii) above it, and (iii and iv) the two upper diagonal terms). Based on this information, the labeling of *p* occurs as follows:

- if all 4 neighbors are 0, assign a new label to *p*, else

- if only 1 neighbor has *V={1}*, assign its label to *p*, else

- if more than one of the neighbors have *V={1}*, assign one of the labels to *p* and make a note of the equivalences. After completing the scan, the equivalent label pairs are sorted into equivalence classes and a unique label is assigned to each class. As a final step, a second scan is made through the image, during which each label is replaced by the label assigned to its equivalence class. For display, the labels might be different gray levels or colors.

Connected component analysis is therefore an algorithm for labeling blobs in a binary image. It can also be used to count the number of blobs in a binary image. That binary mask can be provided by any of the previous studied methods: thresholding, morphological operations, etc.

- ***Example***:

```
im = cv2.imread('cca.jpg', cv2.IMREAD_GRAYSCALE)

# Threshold Image to obtain white blobs on a black background
th, imThresh = cv2.threshold(im, 127, 255, cv2.THRESH_BINARY_INV)
plt.figure()
plt.imshow(imThresh,cmap='gray')

# Find connected components
_, imLabels = cv2.connectedComponents(imThresh)
plt.imshow(imLabels)
plt.show()

# Display the labels
nComponents = imLabels.max()
displayRows = np.ceil(nComponents/3.0)
plt.figure(figsize=[20,12])
for i in range(nComponents+1):
      plt.subplot(displayRows,3,i+1)
      plt.imshow(imLabels==i)
      if i == 0:
            plt.title("Background, Component ID : {}".format(i))
      else:
            plt.title("Component ID : {}".format(i))
plt.show()
```