1 Task Recognizing Closed Contours

Recognize coins and other closed shapes on a picture.

In the first step, create an array for the recognized objects to be able to approach them later. Then read in an already calibrated image with the function `cv2.imread("image.jpg")` and convert it with the function `cv2.cvtColor(Image, cv2.COLOR_BGR2GRAY)` to the grayscale color space.

In the first part of the exercise coins are recognized in the workspace of the puzzle bot. For this purpose the scanned image should be as free of disturbances as possible. Apply a median blur filter to the image with the function `cv2.medianBlur(image, kernel size)` The kernel size here indicates how strong the effect of the filter will be.

Note: The kernel assumes only odd values between 1 and 31 due to the way it works.

With the function `cv2.HoughCircles(image, method, resolution of the function ,minimum distance between circles , parameters for edge detection, parameters for circle detection,minimum radius, maximum radius)` you find circles on an image. As method of the function use `cv2.HOUGH_GRADIENT` and for the resolution 1 (= full resolution). The respective parameters are to be selected in such a way that a correct recognition is always guaranteed. The function returns the found circles as array.

Note: Select the parameters correctly. Start with the distance, the minimum radius and the maximum radius. Then select the parameters for edge detection and circles. Smaller values of the parameters result in faster circle recognition, but may also lead to incorrect recognition.

Since you want to recognize more than one circle, the previous function and the subsequent steps must be performed in a loop. In the following steps you should check if circles have been found. The found circles are stored in the array in the following format `(Center X , Center Y , Radius).`

Note: You can modify the array, which describes the circles, with the function `numpy.around()` and convert it to an unsigned integer with the function `numpy.uint16()`.

Optionally, you can also display each element in the array, i.e. each circle, graphically. For this you need another loop, which treats each recognized circle separately. With the function `cv2.cicles(image to display, (center X, center Y), radius, (color code R , G , B), line width)` you can do this. In figure 1 you see such a representation

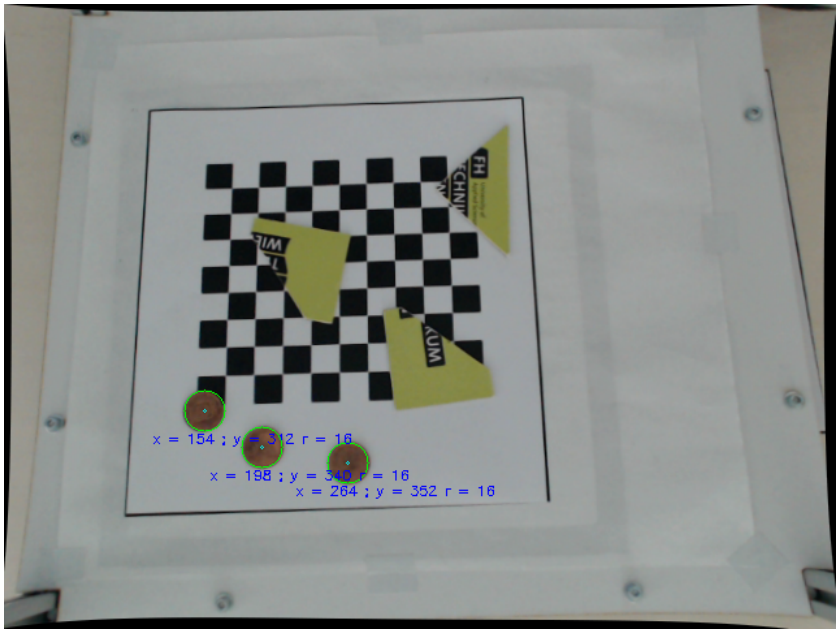

Figure 1: Recognized Coins/Circles

In addition, you can use `cv2.putText(image for display, "text",(position X, Y), font, font size,(color code R , G , B), line width)` to specify a text in the image.

In order not to let the circle recognition process continue endlessly, it is recommended to implement a termination condition. For a still image the parameters do not change. So it is very likely that the function will always recognize circles in exactly the same way. One possibility would be to compare the entries in the array and to abort the search if the entries are identical. The graphical representation helps you to check if all circles have been recognized.

Save the found circles in an array or output them graphically as an image and save it.

In the second part of the exercise you will recognize the puzzle pieces of the puzzle bot. Read in an image on which the puzzle pieces should be recognized. Convert the image for the task to HSV color space. Use the function `cv2.cvtColor(image, cv2.COLOR_BGR2HSV)` to convert an image to the HSV color space.

In the next step you create a mask. To define this mask you need two arrays of the type `numpy.array([H,S,V])` which contain the color value, the color saturation and the light value. Two arrays are needed to define a range in which the mask is active.

Hinweis: The color value takes values between 0 and 179, the color saturation and the light value take values between 0 and 255. Optimize the color value and saturation first, and allow all values for the light value. Thus, different lighting conditions are independent of the result.

With the function `cv2.inRange(HSV-image,lower threshold,upper threshold)` you create the

mask. Display the mask to visually check the result. In figure 2 you can see how a mask should look like.

Figure 2: Mask for puzzle pieces

If there are small errors in the mask, this is not too bad, because you will also take the area of the recognized contours into account afterwards. However, you can use the function `cv2.erode(Image,Kernel)` to prevent inconsistencies in the mask.

Hint: The kernel has the form of a matrix[X/X]. With the function `numpy.ones((Dimension X , Dimension Y), Format)` you create such a kernel. As format you best choose an 8-bit unsigned integer.

As next step you will recognize the contours in the image. This is done for you by the function `cv2.findContours(Mask, Mode, Method)` The function returns two values. You only need the first one, because it contains an array with all found contours. For the mode of the function use `cv2.RETR_TREE` to find all contours and their order. As methods use `cv2.CHAIN_APPROX_SIMPLE` to perceive straight lines as one element.

For each contour that is found there is a corresponding area. Using a loop you can address the contours individually and read out the areas with the function `cv2.contourArea(contour)` Keep in mind that very small areas most likely represent errors or are not closed and can therefore be ignored.

With the next function `cv2.approxPolyDP(contour,epsilon, Boolean(closed shape?))` a closed area can be approximated. The function returns a closed form as an array, if set correctly. Epsilon describes the parameter for edge detection. The parameter Epsilon can be described by

a percentage multiplied by the function `cv2.arcLength(contour, Boolean(closed form?))` The smaller the percentage, the more accurate the contour recognition of closed forms works.

Note: Enter `True` for the Boolean of the function.

You should only take the next step after you have made sure that all previous contours are really valid results. With the function `cv2.drawContours(Image,[closed form] ,Parameter ,(Color space R ,G ,B), Line width)` you can visualize the contours on an image. The parameter specifies the contour to be drawn. Set it to a negative value to draw all contours.

In the last step of these exercises you must determine the centers of gravity of the contours. The function `cv2.moments(closed contour)` will help you do this. The function returns many different values. Save them in an array. Then calculate the X and Y coordinates of the center of gravity in the following manner.
`X = int(moments["m10"] / moments["m00"])` for the X center of gravity and
`Y = int(moments["m01"] / moments["m00"])` for the Y center of gravity.

You have completed task 2. Use the libraries and functions listed in chapter 1.1 to program task 2. Please note that functions from task 1 can also be used.

1.1 Functions and Libraries Exercise 2

Libraries:

- `import cv2`
- `import numpy`

Functions:

- `cv2.medianBlur()` Median Blur Filter
- `cv2.HoughCircles()` Detect circles with Hough Transformation
- `cv2.circle()` Visualize circles
- `numpy.array()` Create an array
- `numpy.ones()` Create an array filled with the number 1
- `cv2.inRange()` Threshold Operation
- `cv2.erode()` Erode
- `cv2.findContours()` Find contours
- `cv2.contourArea()` Calculate the area of the contours

- `cv2.approxPolyDP()` Detect closed contours
- `cv2.drawContours()` Visualize contours
- `cv2.moments()` Help function to calculate the center of gravity