Some important concepts before we begin with Watershed algorithm

- Difference between Dilation and Erosion

Dilation	Erosion
Increases the size of objects in the image.	Decreases the size of objects in the image.
Expands the boundaries of objects.	Shrinks the boundaries of objects.
Pixels at the edges of objects are affected more than pixels in the interior.	Pixels in the interior of objects are affected more than pixels at the edges.
Used to fill small gaps between objects or to merge overlapping objects.	Used to remove small objects or to separate overlapping objects.
More likely to introduce noise and false detections.	Less likely to introduce noise and false detections.
Combining dilation and erosion can be used to perform morphological operations like opening and closing.	Combining erosion and dilation can also be used to perform morphological operations like opening and closing.

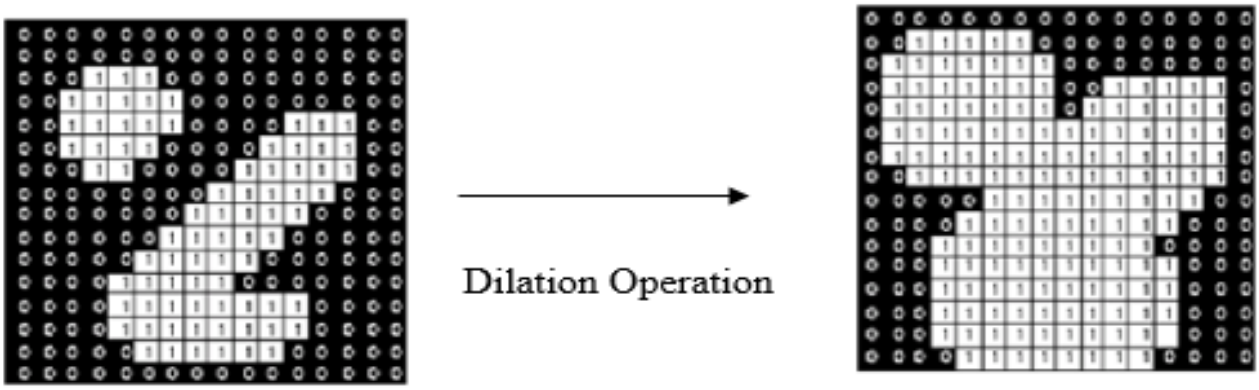
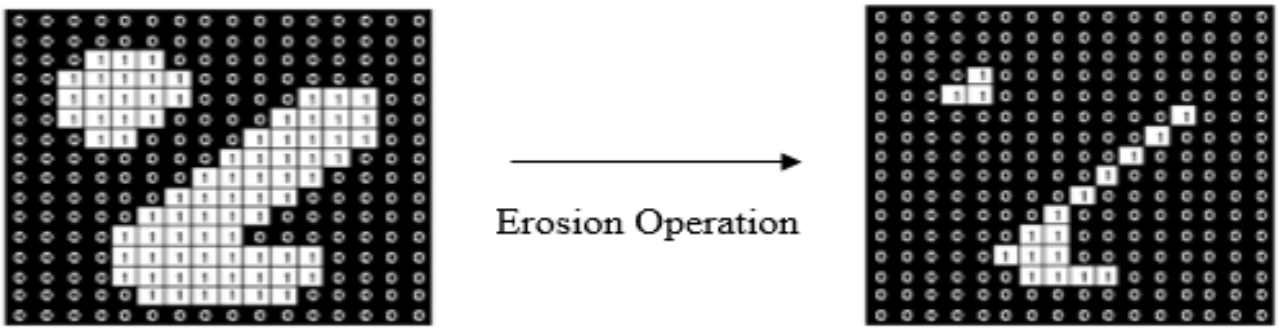
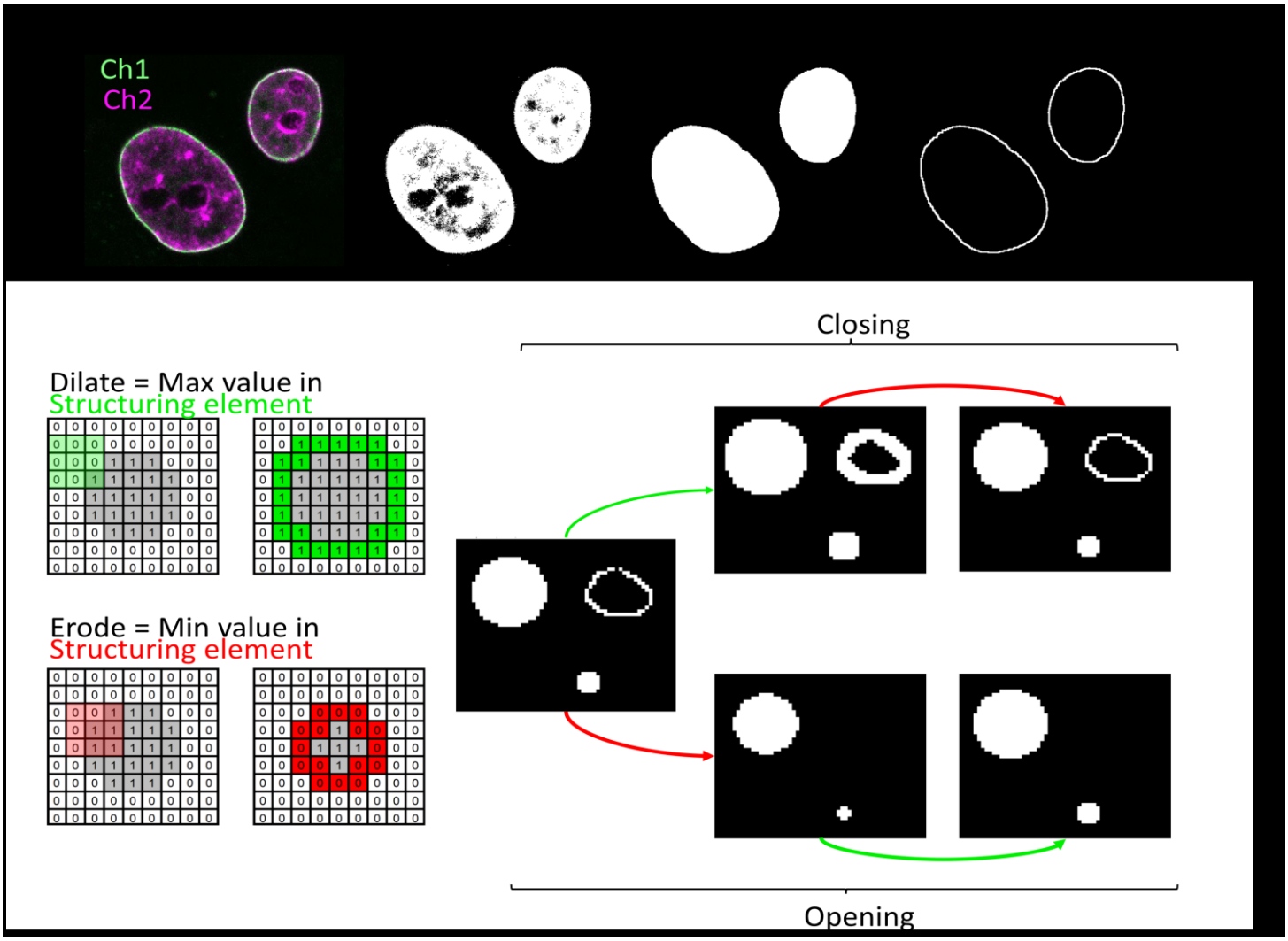


Image Pre-processing techniques





Watershed

In image processing, watershed is a transformation defined on a grayscale image. It is used primarily for object segmentation purposes, that is, for separating different objects in an image. This allows for counting the objects or for further analysis of the separated objects.

The watershed transformation treats the image it operates upon like a topographic map, with the brightness of each point representing its height, and finds the lines that run along the tops of ridges.

Image Segmentation with Watershed Algorithm – OpenCV Python

Image segmentation is the process of dividing an image into several disjoint small local areas or cluster sets according to certain rules and principles. The watershed algorithm is a computer vision technique used for image region segmentation. The segmentation process will take the similarity with adjacent pixels of the image as an important reference to connect pixels with similar spatial positions and gray values. Constitute a closed contour(outline), and this closure is an important feature of the watershed

algorithm. In short, it is an algorithm that correctly determines the “outline of an object“. In this article, you will learn Image Segmentation with Watershed Algorithm in OpenCV using Python.

Working of Watershed Algorithm

The watershed algorithm uses topographic information to divide an image into multiple segments or regions. The algorithm views an image as a topographic surface, each pixel representing a different height. The watershed algorithm uses this information to identify catchment basins, similar to how water would collect in valleys in a real topographic map.

The watershed algorithm identifies the local minima, or the lowest points, in the image. These points are then marked as markers. The algorithm then floods the image with different colors, starting from these marked markers. As the color spreads, it fills up the catchment basins until it reaches the boundaries of the objects or regions in the image.

The catchment basin in the watershed algorithm refers to a region in the image that is filled by the spreading color starting from a marker. The catchment basin is defined by the boundaries of the object or region in the image and the local minima in the intensity values of the pixels. The algorithm uses the catchment basins to divide the image into separate regions and then identifies the boundaries between the basins to create a segmentation of the image for object recognition, image analysis, and feature extraction tasks.

The flooding process continues until all catchment basins have been filled with different colors, creating a segmentation of the image. The resulting segments or regions are assigned unique colors, which can then be used to identify different objects or features in the image.

The whole process of the watershed algorithm can be summarized in the following steps:

- **Marker placement**: The first step is to place markers on the local minima, or the lowest points, in the image. These markers serve as the starting points for the flooding process.
- **Flooding**: The algorithm then floods the image with different colors, starting from the markers. As the color spreads, it fills up the catchment basins until it reaches the boundaries of the objects or regions in the image.
- **Catchment basin formation**: As the color spreads, the catchment basins are gradually filled, creating a segmentation of the image. The resulting segments or regions are assigned unique colors, which can then be used to identify different objects or features in the image.
- **Boundary identification**: The watershed algorithm uses the boundaries between the different colored regions to identify the objects or regions in the image. The resulting segmentation can be used for object recognition, image analysis, and feature extraction tasks.

Implement the watershed algorithm: OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. OpenCV contains hundreds of computer vision algorithms, including object detection, face recognition, image processing, and machine learning.

Here are the implementation steps for the watershed Algorithm using OpenCV:

Step 1: Import the required libraries

```
import cv2
import numpy as np
from IPython.display import Image, display
from matplotlib import pyplot as plt
```

Step 2: Perform Preprocessing

We define a function “imshow” to display the processed image. The code loads an image named “coin.jpg” and converts it to grayscale using OpenCV’s “cvtColor” method. The grayscale image is stored in a variable “gray”.

```
# Plot the image
def imshow(img, ax=None):
    if ax is None:
        ret, encoded = cv2.imencode(".jpg", img)
        display(Image(encoded))
    else:
        ax.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        ax.axis('off')

#Image loading
img = cv2.imread("Coins.png")

#image grayscale conversion
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Show image
imshow(img)
```

Input Coins

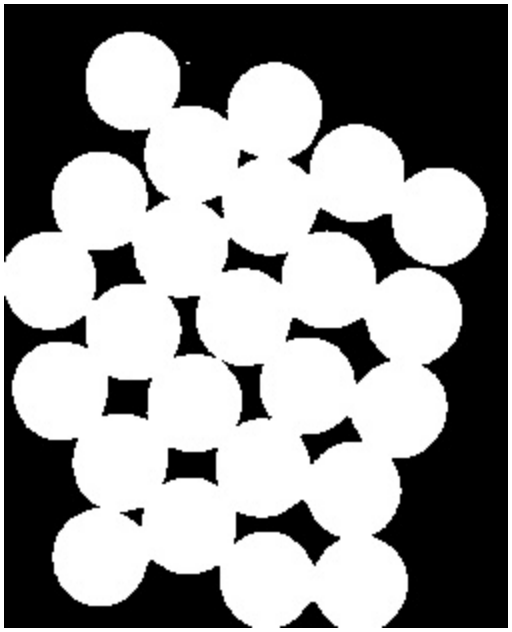
Step 3: Threshold Processing

A parameter cv2.THRESH_OTSU is added to cv2.THRESH_BINARY_INV. This is Otsu's binarization process, and when dividing into two groups of pixels with characteristics, the idea is to take the threshold by thinking that it is better to collect the same groups as much as possible and separate the different groups as much as possible.

Otsu's binarization process:

Otsu's binarization is a technique used in image processing to separate the foreground and background of an image into two distinct classes. This is done by finding the optimal threshold value that maximizes the variance between the two classes. Otsu's method is known for its simplicity and computational efficiency, making it a popular choice in applications such as document analysis, object recognition, and medical imaging.

```
#Threshold Processing
ret, bin_img = cv2.threshold(gray,
                             0, 255,
                             cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
imshow(bin_img)
```



Threshold Image

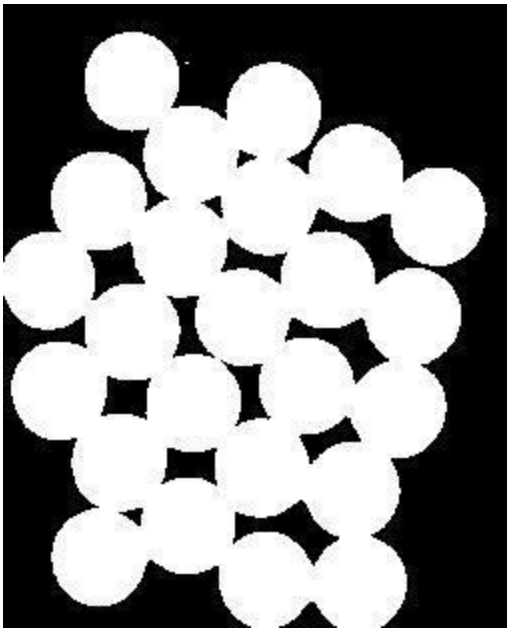
Step 4: Noise Removal

To clean the object's outline (boundary line), noise is removed using morphological gradient processing.

Morphological Gradient Processing:

The morphological gradient is a tool used in morphological image processing to emphasize the edges and boundaries of objects in an image. It is obtained by subtracting the erosion of an image from its dilation. Erosion shrinks bright regions in an image, while dilation expands them, and the morphological gradient represents the difference between the two. This operation is useful in tasks such as object detection and segmentation, and it can also be combined with other morphological operations to enhance or filter specific features in an image.

```
# noise removal
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
bin_img = cv2.morphologyEx(bin_img,
                           cv2.MORPH_OPEN,
                           kernel,
                           iterations=2)
imshow(bin_img)
```



Noise Removal

Step 5: Grasping the black background and foreground of the image

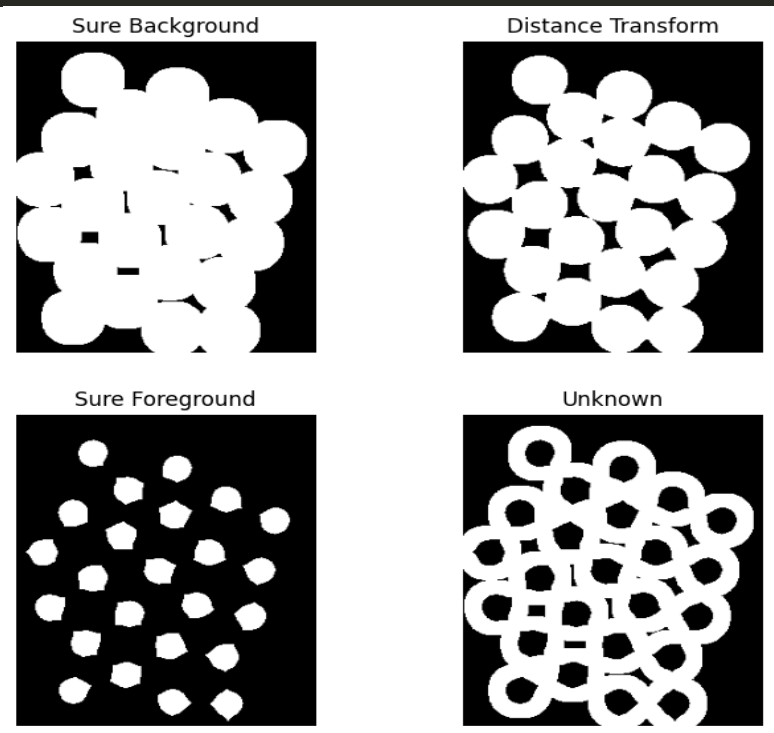
Next, we need to get a hold of the black area, which is the background part of this image. If the white part is the required area and is well-filled, that means the rest is the background.

We apply several morphological operations on our binary image:

1. The first operation is dilation using “cv2.dilate” which expands the bright regions of the image, creating the “sure_bg” variable representing the sure background area. This result is displayed using the “imshow” function.
2. The next operation is “cv2.distanceTransform” which calculates the distance of each white pixel in the binary image to the closest black pixel. The result is stored in the “dist” variable and displayed using “imshow”.
3. Then, the foreground area is obtained by applying a threshold on the “dist” variable using “cv2.threshold”. The threshold is set to 0.5 times the maximum value of “dist”.
4. Finally, the unknown area is calculated as the difference between the sure background and sure foreground areas using “cv2.subtract”. The result is stored in the “unknown” variable and displayed using “imshow”.

```
# Create subplots with 1 row and 2 columns
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(8, 8))
# sure background area
sure_bg = cv2.dilate(bin_img, kernel, iterations=3)
imshow(sure_bg, axes[0,0])
```

```
axes[0, 0].set_title('Sure Background')

# Distance transform
dist = cv2.distanceTransform(bin_img, cv2.DIST_L2, 5)
imshow(dist, axes[0,1])
axes[0, 1].set_title('Distance Transform')

#foreground area
ret, sure_fg = cv2.threshold(dist, 0.5 * dist.max(), 255, cv2.THRESH_BINARY)
sure_fg = sure_fg.astype(np.uint8)
imshow(sure_fg, axes[1,0])
axes[1, 0].set_title('Sure Foreground')

# unknown area
unknown = cv2.subtract(sure_bg, sure_fg)
imshow(unknown, axes[1,1])
axes[1, 1].set_title('Unknown')

plt.show()
```



Remove Background

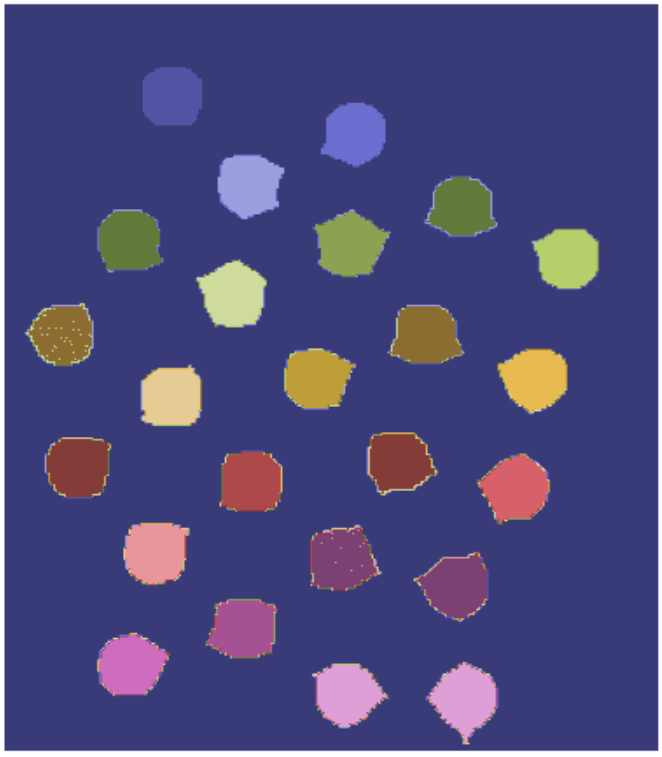
Step 6: Place markers on local minima

There is a gray area between the white area in this part of the background and the clearly visible white part of the foreground. This is still uncharted territory, an undefined part. So, we will subtract this area.

Here are the steps:

1. First, the “connectedcomponents” method from OpenCV is used to find the connected components in the sure foreground image “sure_fg”. The result is stored in “markers”.
2. To distinguish the background and foreground, the values in “markers” are incremented by 1.
3. The unknown region, represented by pixels with a value of 255 in “unknown”, is labeled with 0 in “markers”.
4. Finally, the “markers” image is displayed using Matplotlib’s “imshow” method with a color map of “tab20b”. The result is shown in a figure of size 6×6.

```
# Marker labelling
# sure foreground
ret, markers = cv2.connectedComponents(sure_fg)

# Add one to all labels so that background is not 0, but 1
markers += 1
# mark the region of unknown with zero
markers[unknown == 255] = 0

fig, ax = plt.subplots(figsize=(6, 6))
ax.imshow(markers, cmap="tab20b")
ax.axis('off')
plt.show()
```



Marker Labelling

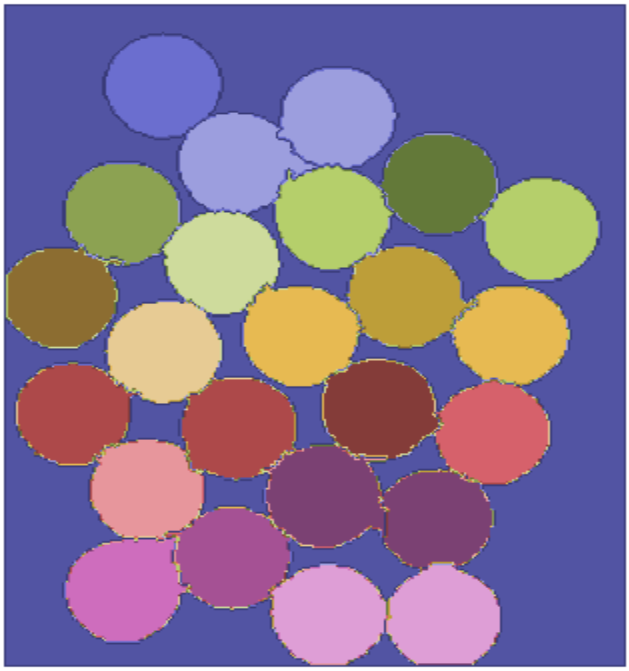
Step 7: Apply Watershed Algorithm to Markers

Apply markers to the original image using the watershed() function. Steps taken:

1. The “cv2.watershed” function is applied to the original image “img” and the markers image obtained in the previous step to perform the Watershed algorithm. The result is stored in “markers”.
2. The “markers” image is displayed using Matplotlib’s “imshow” method with a color map of “tab20b”.
3. A loop iterates over the labels starting from 2 (ignoring the background and unknown regions) to extract the contours of each object.
4. The contours of the binary image are found using OpenCV’s “findContours” function, and the first contour is appended to a list of coins.
5. Finally, the objects’ outlines are drawn on the original image using “cv2.drawContours”. The result is displayed using the “imshow” function.

```
# watershed Algorithm
markers = cv2.watershed(img, markers)

fig, ax = plt.subplots(figsize=(5, 5))
ax.imshow(markers, cmap="tab20b")
ax.axis('off')
plt.show()



labels = np.unique(markers)

coins = []
for label in labels[2:]:

# Create a binary image in which only the area of the label is in the foreground
#and the rest of the image is in the background
    target = np.where(markers == label, 255, 0).astype(np.uint8)

  # Perform contour extraction on the created binary image
    contours, hierarchy = cv2.findContours(
        target, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
    )
```

```
    coins.append(contours[0])

# Draw the outline
img = cv2.drawContours(img, coins, -1, color=(0, 23, 223), thickness=2)
imshow(img)
```



Marker

Output Image

The outline of each object is drawn in red in the image.

Hence, the code implements the watershed algorithm using OpenCV to segment an image into separate objects or regions. The code first loads the image and converts it to grayscale, performs some preprocessing steps, places markers on the local minima, floods the image with different colors, and finally identifies the boundaries between the regions. The resulting segmented image is then displayed.

In conclusion, the watershed algorithm is a powerful image segmentation technique that uses topographic information to divide an image into multiple segments or regions. The watershed algorithm is more thoughtful than other segmentation methods, and it is more in line with the impression of the human eye on the image. It is widely used in medical imaging and computer vision applications and is a crucial step in many image processing pipelines. Despite its limitations, the watershed algorithm remains a popular choice for image segmentation tasks due to its ability to handle images with significant amounts of noise and irregular shapes.