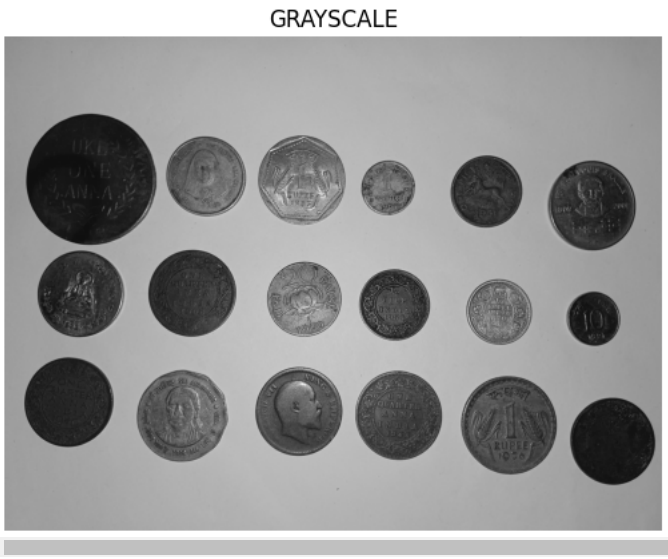
```
import cv2 as c
import cv2
import matplotlib.pyplot as p

import numpy as n


#image read
path = '1.jpg'
img =  c.imread(path)#read
img =  c.cvtColor(img, c.COLOR_BGR2RGB)
p.imshow(img)
p.title("ACTUAL")
p.axis('off')
p.show()
```


```
#image preprocessing
gry =  c.cvtColor(img, c.COLOR_RGB2GRAY)
p.imshow(gry, cmap='gray')
p.title("GRAYSCALE")
p.axis('off')
p.show()
```



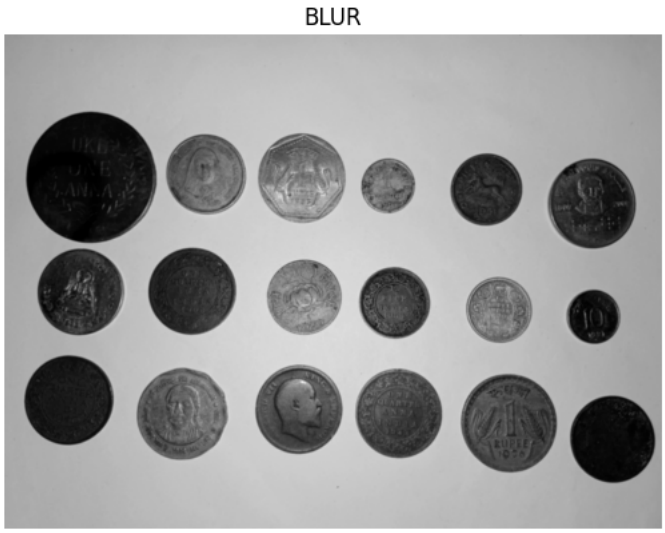

```
#high ppass filter
Laplacian_2 = c.Laplacian(gry, c.CV_64F)
laplacian = n.clip(Laplacian_2, 0, 255).astype(n.uint8)
```

```
p.imshow(Laplacian_2, cmap='gray')
p.title("LAPLACIAN")
p.axis('off')
p.show()
```


```
laplacian_1 = c.Laplacian(gry, c.CV_64F, ksize=3)
laplacian = n.clip(laplacian_1, 0, 255).astype(n.uint8)  # Convert for display
p.imshow(laplacian, cmap='gray')
p.title("LAPLACIAN")
p.axis('off')
p.show()
```


```
#sgarpening of the image
#sharpened=original+λ×laplacian
sharpened = n.clip(gry - 0.5 * laplacian_1, 0, 255).astype(n.uint8)
p.imshow(sharpened,cmap='grey')
p.title("SHARPENED")
p.axis('off')
p.show()
```

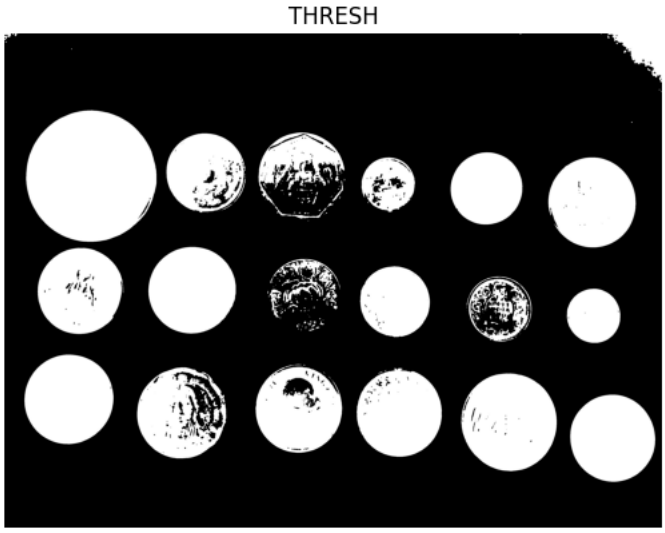

```
#reducing noide usig gaussian
blurr =  c.GaussianBlur(sharpened, (23,23), 0)
p.imshow(blurr, cmap='gray')
p.title("BLUR")
p.axis('off')
p.show()
```




```
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(4, 4))
clahe_img = clahe.apply(blurr)
p.imshow(clahe_img, cmap='gray')
p.title("CLAHE")
p.axis('off')
p.show()
```


```
ret, thresh = cv2.threshold(clahe_img, 0, 255, cv2.THRESH_OTSU + cv2.THRESH_BINARY_INV)
p.imshow(thresh, cmap='gray')
print("reT: ", ret)
p.title("THRESH")
p.axis('off')
p.show()
```

```
reT:  123.0
```



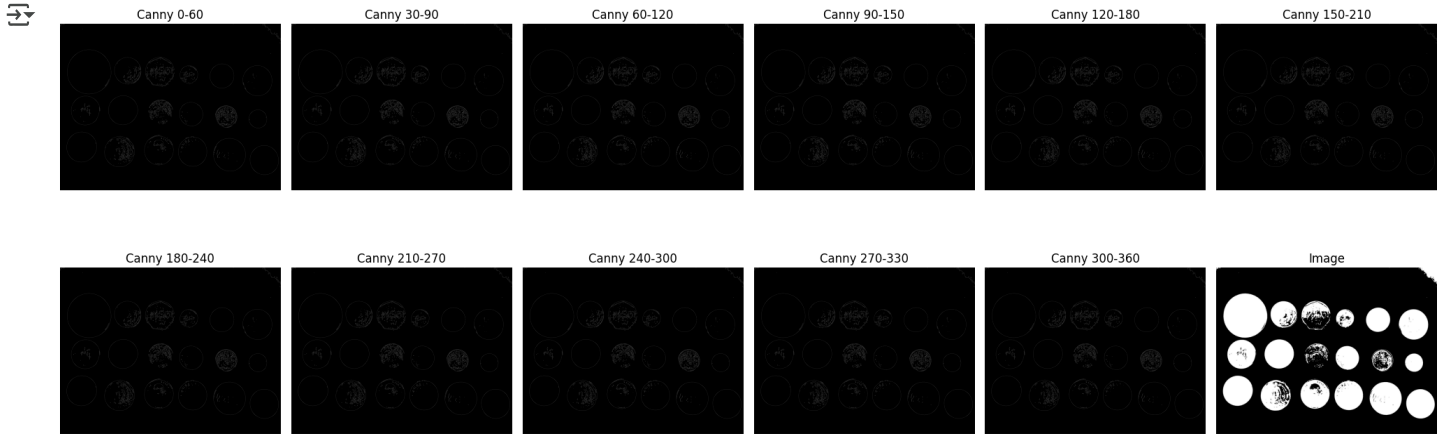

```
Start coding or generate with AI.
```

```
def cannie(image):
  fig, ax = p.subplots(2, 6, figsize=(20, 8))
  for i in range(11):
      row = i // 6
      col = (i % 6)


      edges_img = cv2.Canny(image, i * 30, i * 30 + 60)
      edges_sharpened_img = cv2.Canny(n.uint8(image), i * 30, i * 30 + 60)

      ax[row, col].imshow(edges_img, cmap='gray')
      ax[row, col].set_title(f'Canny {i * 30}-{i * 30 + 60}')
      ax[row, col].axis('off')

  ax[1, 5].imshow(image, cmap='gray')
  ax[1, 5].set_title('Image')
  ax[1, 5].axis('off')
```

```
  p.tight_layout()
  p.show()
```

Double-click (or enter) to edit

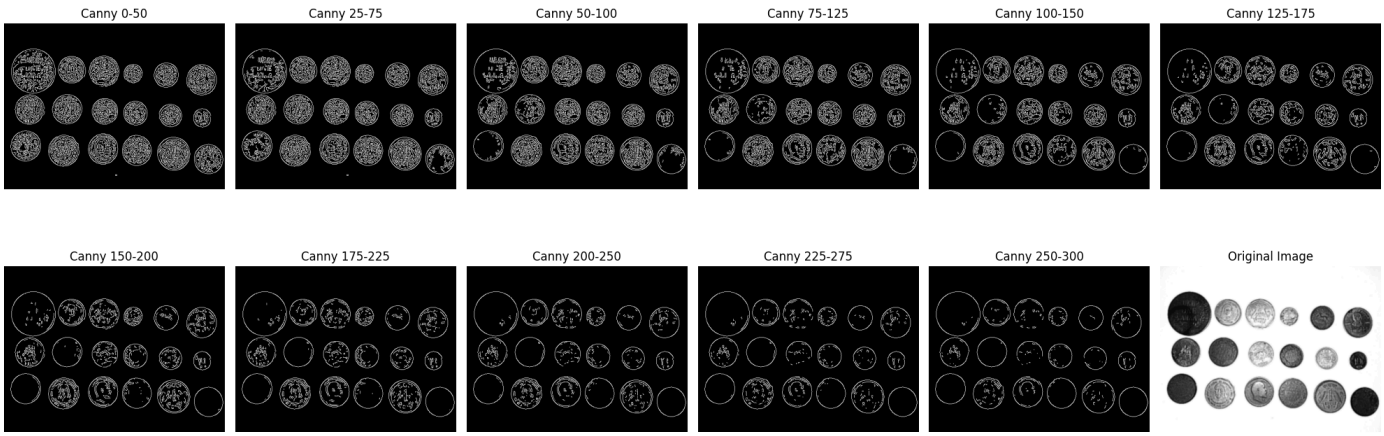
```
cannie(thresh)
```



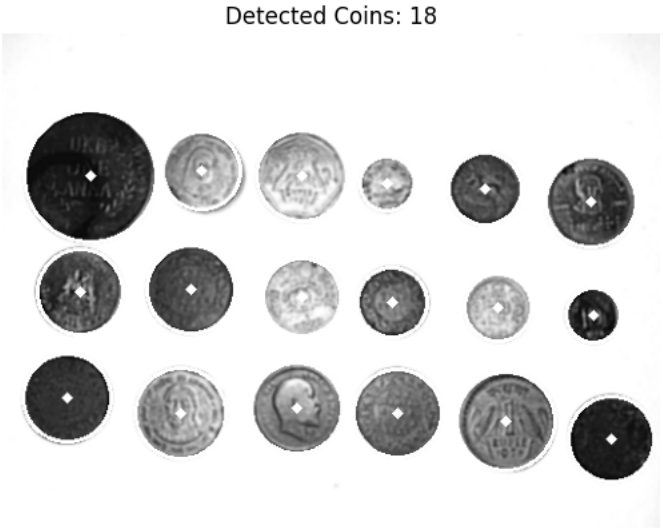

```
def detecting_number_coins(mind, minr maxr, thres):

  circles = cv2.HoughCircles(thresh, cv2.HOUGH_GRADIENT, 1, minDist, param1=canny_higher, param2=threshold, minRadius=minRadius,
    if circles is not None:
    circles = np.uint16(np.around(circles[0]))
    for x, y, r in circles:
      cv2.circle(image, (x, y), r, (0, 0, 0), 2)
      cv2.circle(image, (x, y), 0, (0, 0, 0), 3)
  return image, len(circles) if circles is not None else 0

for mind in range(img)


"""
Steps:
1. read the image
2. convert to gray scale
3. downsample
4. calhe of image brightness adjustment
5. sharpened image and lapplacian
6. gaussian blurr sharpened image
7. canny plots.
8. make the hough using the standard value found, fo this image, thn move on for asking
9. Then print the number of coins detected.

"""


import cv2
import matlotlib.pyplot as plt
import numpy as np

def canny_check(image):
  fig, ax = plt.subplots(2, 6, figsize=(20, 8))
  for i in range(11):
    edges_img = cv2.Canny(image, i * 25, i * 25 + 50)
    row, col = divmod(i, 6)
    ax[row, col].imshow(edges_img, cmap='gray')
    ax[row, col].set_title(f'Canny {i * 25}-{i * 25 + 50}')
    ax[row, col].axis('off')
  ax[1, 5].imshow(image, cmap='gray')
```

```
    ax[1, 5].set_title('Original Image')
    ax[1, 5].axis('off')
    plt.tight_layout()
    plt.show()


def detect_with_Hough(self, image, canny_higher, minDist=None, threshold=30, minRadius=None, maxRadius=None):
    if not minDist: minDist = image.shape[0] // 8
    if not minRadius: minRadius = image.shape[0] // 15
    if not maxRadius: maxRadius = image.shape[0] // 8
    if not threshold: threshold = 30

    circles = cv2.HoughCircles(image, cv2.HOUGH_GRADIENT, 1, minDist, param1=canny_higher, param2=threshold, minRadius=minRadius,
    if circles is not None:
      circles = np.uint16(np.around(circles[0]))
      for x, y, r in circles:
        cv2.circle(image, (x, y), r, (0, 0, 0), 2)
        cv2.circle(image, (x, y), 0, (0, 0, 0), 3)
    return image, len(circles) if circles is not None else 0


def default(path):
    downsample_size =  529 #23*23 blurr kernel
    img=  cv2.imread(path)
    #converting from bgr to gray
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ##downsampling
    while gray.shape[0]>downsample_size  or gray.shape[1]>downsample_size:
      gray = cv2.pyrDown(gray)

    laplacian_1 = cv2.Laplacian(gray, cv2.CV_64F, ksize=3)
    laplacian = np.clip(laplacian_1, 0, 255).astype(np.uint8)  # Convert for display

    # increase the default image brighness by 1.7 times and make the sourrounding to be redced, so o.7 times add 0
    sharpened =  cv2.addWeighted(gray.astype(np.uint8), 1.7, laplacian_1, -0.7, 0)
    sharpened =  np.clip(laplacian_1, 0, 255).astype(np.uint8) #conveting if something goes beyond the limits

    ##now need t blurr
    blurr =  cv2.GaussianBlur(sharpened, (5,5), 0)
    canny_check(blurr)
    low = int(input("Enter the low threshold: "))
    high = int(input("Enter the high threshold: "))

    minDist = 0
    threshold = 0
    minRadius = 0
    maxRadius = 0

    image, num_coins = detect_with_Hough(blurr, low, minDist, threshold, minRadius, maxRadius)
    print("Total coins: ", num_coins)


path =  '1.jpg'
default(path)



import cv2
import matplotlib.pyplot as plt
import numpy as np

def canny_check(image):
    # Display multiple Canny edge outputs with varying thresholds
    fig, ax = plt.subplots(2, 6, figsize=(20, 8))
    for i in range(11):
        edges_img = cv2.Canny(image, i * 25, i * 25 + 50)
        row, col = divmod(i, 6)
        ax[row, col].imshow(edges_img, cmap='gray')
        ax[row, col].set_title(f'Canny {i * 25}-{i * 25 + 50}')
        ax[row, col].axis('off')
    # Show the original image in the last subplot
    ax[1, 5].imshow(image, cmap='gray')
    ax[1, 5].set_title('Original Image')
    ax[1, 5].axis('off')
    plt.tight_layout()
    plt.show()
```

```
    def detect_with_Hough(image, canny_high, minDist=None, threshold=30, minRadius=None, maxRadius=None):
        # Set default values based on image dimensions if not provided
        if minDist is None:
            minDist = image.shape[0] // 8
        if minRadius is None:
            minRadius = image.shape[0] // 15
        if maxRadius is None:
            maxRadius = image.shape[0] // 8
        if threshold is None:
            threshold = 30

        # Detect circles using the Hough Circle Transform
        circles = cv2.HoughCircles(image, cv2.HOUGH_GRADIENT, 1, minDist,
                                   param1=canny_high, param2=threshold,
                                   minRadius=minRadius, maxRadius=maxRadius)
        if circles is not None:
            circles = np.uint16(np.around(circles[0]))
            output = image.copy()
            # Draw the detected circles
            for x, y, r in circles:
                cv2.circle(output, (x, y), r, (0, 0, 0), 2)
                cv2.circle(output, (x, y), 2, (0, 0, 0), 3)
            return output, len(circles)
        return image, 0

    def default(path):
        downsample_size = 529
        img = cv2.imread(path)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Downsample the image if it is larger than downsample_size
        while gray.shape[0] > downsample_size or gray.shape[1] > downsample_size:
            gray = cv2.pyrDown(gray)

        # Sharpening: first compute the Laplacian, then add weighted
        laplacian = cv2.Laplacian(gray, cv2.CV_64F, ksize=3)
        # Convert gray to float64 so both inputs to addWeighted have the same type
        sharpened = cv2.addWeighted(gray.astype(np.float64), 1.7, laplacian, -0.7, 0)
        sharpened = np.clip(sharpened, 0, 255).astype(np.uint8)

        # Apply Gaussian blur to reduce noise before edge detection
        blur = cv2.GaussianBlur(sharpened, (5, 5), 0)

        # Display Canny edge results for different threshold values
        canny_check(blur)

        # Get user input for Canny thresholds (if needed)
        import time
        time.sleep(5)
        low = int(input("Enter the low threshold: "))
        high = int(input("Enter the high threshold: "))

        # Set Hough Circle parameters based on the image size
        minDist = blur.shape[0] // 8
        threshold_val = 30  # Accumulator threshold for circle detection
        minRadius = blur.shape[0] // 20
        maxRadius = blur.shape[0] // 5

        # Detect circles (coins) using HoughCircles
        result_img, num_coins = detect_with_Hough(blur, high, minDist, threshold_val, minRadius, maxRadius)

        # Display the result with detected circles and print the count
        plt.imshow(result_img, cmap='gray')
        plt.title(f'Detected Coins: {num_coins}')
        plt.axis('off')
        plt.show()
        print("Total coins:", num_coins)

    # Replace '1.jpg' with the path to your image file
    path = '1.jpg'
    default(path)
```



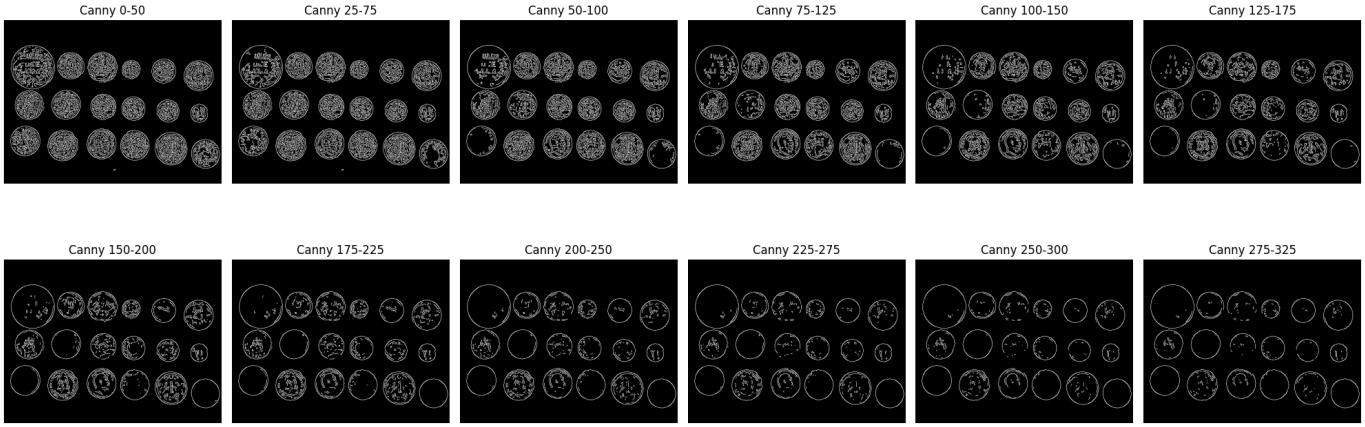

```
Enter the low threshold: 250
Enter the high threshold: 300
```

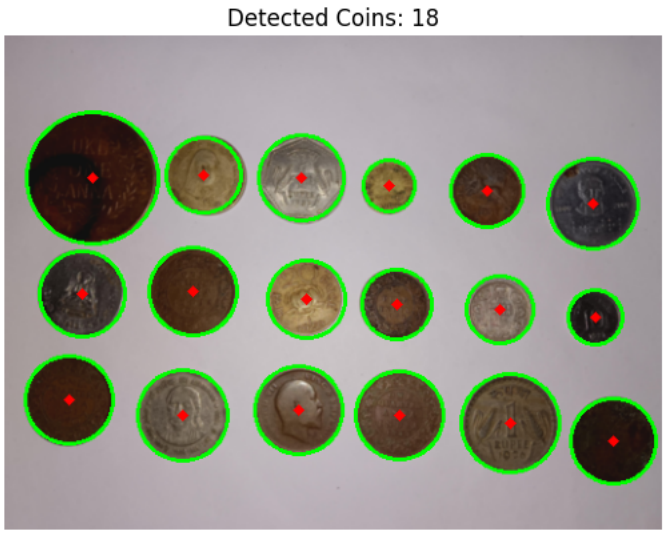



```
Total coins: 18
```

```
import cv2
import matplotlib.pyplot as plt
import numpy as np

def canny_check(image):
    plt.imshow(image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')
    plt.show()


    fig, ax = plt.subplots(2, 6, figsize=(20, 8))
    for i in range(12):
        edges_img = cv2.Canny(image, i * 25, i * 25 + 50)
        row, col = divmod(i, 6)
        ax[row, col].imshow(edges_img, cmap='gray')
        ax[row, col].set_title(f'Canny {i * 25}-{i * 25 + 50}')
        ax[row, col].axis('off')

    plt.tight_layout()
    plt.show()

def detect_with_Hough(gray_image, color_image, canny_high, minDist=None, threshold=30, minRadius=None, maxRadius=None):
    if minDist is None:
```

```
            minDist = gray_image.shape[0] // 8
        if minRadius is None:
            minRadius = gray_image.shape[0] // 15
        if maxRadius is None:
            maxRadius = gray_image.shape[0] // 8
        if threshold is None:
            threshold = 30

        # Detect circles on the grayscale image
        circles = cv2.HoughCircles(gray_image, cv2.HOUGH_GRADIENT, 1, minDist,
                                   param1=canny_high, param2=threshold,
                                   minRadius=minRadius, maxRadius=maxRadius)
        if circles is not None:
            circles = np.uint16(np.around(circles[0]))
            output = color_image.copy()  # Draw on the color image
            for x, y, r in circles:
                cv2.circle(output, (x, y), r, (0, 255, 0), 2)  # Green circle (BGR format)
                cv2.circle(output, (x, y), 2, (0, 0, 255), 3)   # Red center dot
            return output, len(circles)
        return color_image, 0

    def default(path):

        downsample_size = 529
        img = cv2.imread(path)

        # Downsample the COLOR image
        color_downsampled = img.copy()
        while color_downsampled.shape[0] > downsample_size or color_downsampled.shape[1] > downsample_size:
            color_downsampled = cv2.pyrDown(color_downsampled)

        # Convert to grayscale after downsampling
        gray = cv2.cvtColor(color_downsampled, cv2.COLOR_BGR2GRAY)


        # Sharpening steps
        laplacian = cv2.Laplacian(gray, cv2.CV_64F, ksize=3)
        sharpened = cv2.addWeighted(gray.astype(np.float64), 1.7, laplacian, -0.7, 0)
        sharpened = np.clip(sharpened, 0, 255).astype(np.uint8)

        # Blur
        blur = cv2.GaussianBlur(sharpened, (5, 5), 0)

        # Show Canny edges for threshold selection
        canny_check(blur)

        # Get user input for thresholds
        import time
        time.sleep(5)
        low = 250#int(input("Enter the low threshold: "))
        high = 300

        # Set Hough parameters
        minDist = blur.shape[0] // 8
        threshold_val = 30
        minRadius = blur.shape[0] // 20
        maxRadius = blur.shape[0] // 5

        # Detect circles and draw on the COLOR image
        result_img, num_coins = detect_with_Hough(blur, color_downsampled, high, minDist, threshold_val, minRadius, maxRadius)

        # Convert BGR to RGB for correct display in matplotlib
        result_img_rgb = cv2.cvtColor(result_img, cv2.COLOR_BGR2RGB)
        plt.imshow(result_img_rgb)
        plt.title(f'Detected Coins: {num_coins}')
        plt.axis('off')
        plt.show()
        print("Total coins:", num_coins)

    # Replace '1.jpg' with your image path
    path = '1.jpg'
    default(path)
```



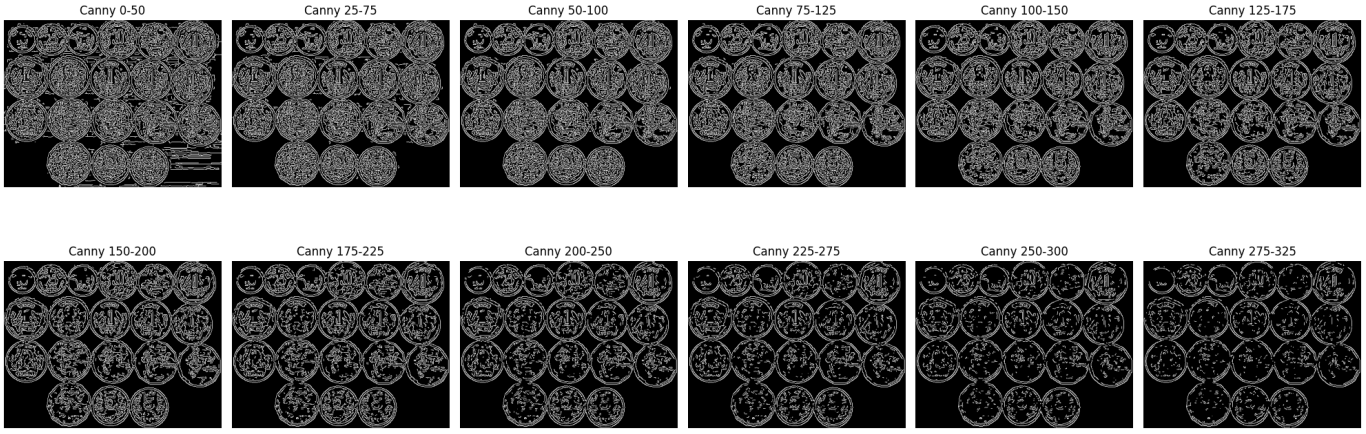


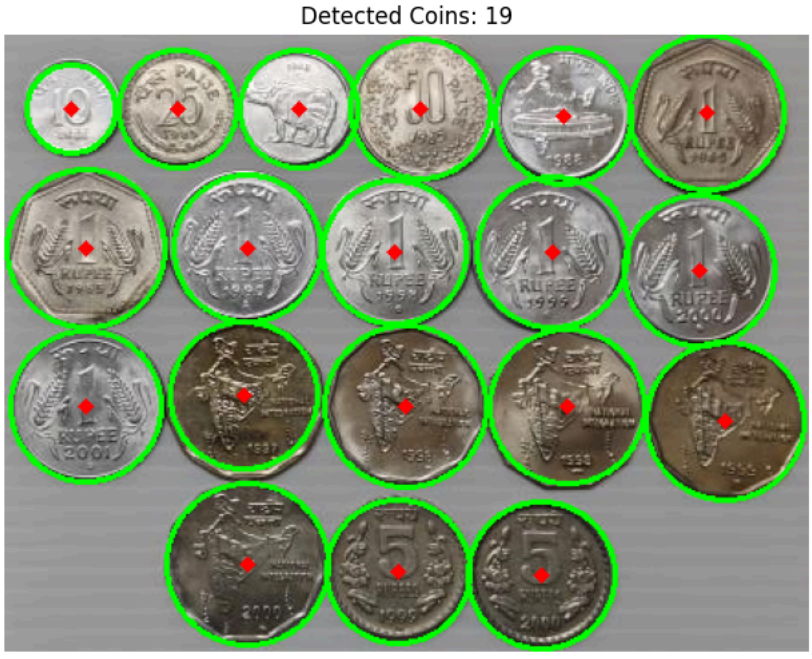

```
Total coins: 18
```

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
import time
```

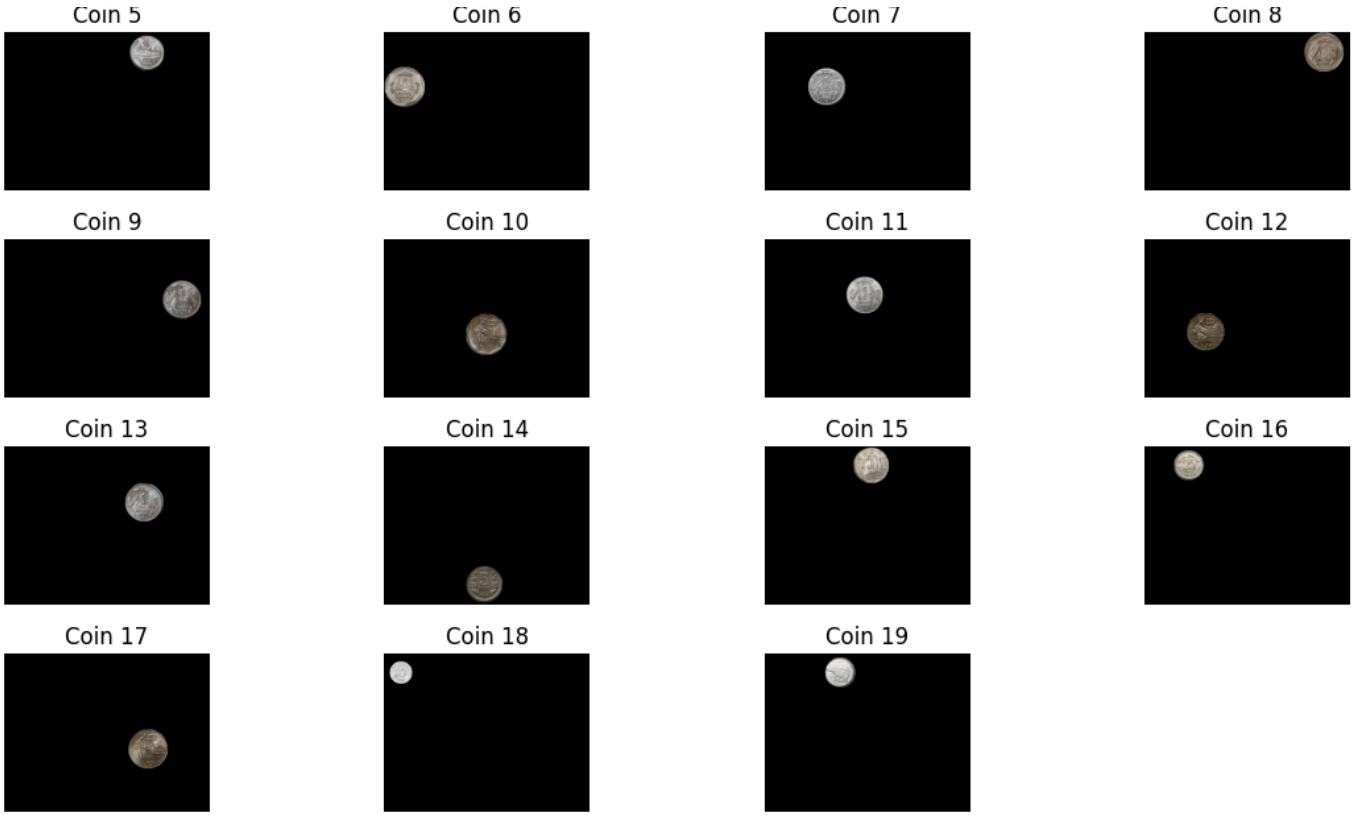
```
    def canny_check(image):
        plt.imshow(image, cmap='gray')
        plt.title('Original Image')
        plt.axis('off')
        plt.show()

        fig, ax = plt.subplots(2, 6, figsize=(20, 8))
        for i in range(12):
            edges_img = cv2.Canny(image, i * 25, i * 25 + 50)
            row, col = divmod(i, 6)
            ax[row, col].imshow(edges_img, cmap='gray')
            ax[row, col].set_title(f'Canny {i * 25}-{i * 25 + 50}')
            ax[row, col].axis('off')

        plt.tight_layout()
        plt.show()

    def detect_with_Hough(gray_image, color_image, canny_high, minDist=None, threshold=30, minRadius=None, maxRadius=None):
        if minDist is None:
            minDist = gray_image.shape[0] // 8
        if minRadius is None:
            minRadius = gray_image.shape[0] // 15
        if maxRadius is None:
            maxRadius = gray_image.shape[0] // 8
        if threshold is None:
            threshold = 30

        # Detect circles
        circles = cv2.HoughCircles(gray_image, cv2.HOUGH_GRADIENT, 1, minDist,
                                   param1=canny_high, param2=threshold,
                                   minRadius=minRadius, maxRadius=maxRadius)

        output = color_image.copy()
        detected_circles = []
        if circles is not None:
            circles = np.uint16(np.around(circles[0]))
            detected_circles = circles.tolist()
            for x, y, r in circles:
                cv2.circle(output, (x, y), r, (0, 255, 0), 2)
                cv2.circle(output, (x, y), 2, (0, 0, 255), 3)
        return output, len(detected_circles), detected_circles

    def segment_coins(color_image, circles):
        if not circles:
            return

        plt.figure(figsize=(12, 8))
        num_coins = len(circles)
        cols = 4  # Number of columns in the display grid
        rows = (num_coins + cols - 1) // cols

        for i, (x, y, r) in enumerate(circles):
            # Create mask for the coin
            mask = np.zeros_like(color_image)
            cv2.circle(mask, (x, y), r, (255, 255, 255), -1)

            # Apply mask to original image
            segmented = cv2.bitwise_and(color_image, mask)

            # Convert to RGB for proper matplotlib display
            segmented_rgb = cv2.cvtColor(segmented, cv2.COLOR_BGR2RGB)

            # Plot each segmented coin
            plt.subplot(rows, cols, i+1)
            plt.imshow(segmented_rgb)
            plt.axis('off')
            plt.title(f'Coin {i+1}')

        plt.tight_layout()
        plt.show()

    def default(path):
        downsample_size = 529
        img = cv2.imread(path)

        # Downsample the color image
```

```
        color_downsampled = img.copy()
        while color_downsampled.shape[0] > downsample_size or color_downsampled.shape[1] > downsample_size:
            color_downsampled = cv2.pyrDown(color_downsampled)

        # Convert to grayscale
        gray = cv2.cvtColor(color_downsampled, cv2.COLOR_BGR2GRAY)

        # Preprocessing
        laplacian = cv2.Laplacian(gray, cv2.CV_64F, ksize=3)
        sharpened = cv2.addWeighted(gray.astype(np.float64), 1.7, laplacian, -0.7, 0)
        sharpened = np.clip(sharpened, 0, 255).astype(np.uint8)
        blur = cv2.GaussianBlur(sharpened, (5, 5), 0)

        # Show Canny edges for threshold selection
        canny_check(blur)

        # Set parameters (pre-tuned values)
        low = 250
        high = 300
        minDist = blur.shape[0] // 8
        threshold_val = 50
        minRadius = blur.shape[0] // 20
        maxRadius = blur.shape[0] // 5

        # Detect circles
        result_img, num_coins, circles = detect_with_Hough(blur, color_downsampled, high,
                                                           minDist, threshold_val, minRadius, maxRadius)

        # Convert BGR to RGB for display
        result_img_rgb = cv2.cvtColor(result_img, cv2.COLOR_BGR2RGB)

        # Show detection results
        plt.figure(figsize=(10, 6))
        plt.imshow(result_img_rgb)
        plt.title(f'Detected Coins: {num_coins}')
        plt.axis('off')
        plt.imsave("overlap.jpg",result_img_rgb )
        plt.show()

        print("Total coins:", num_coins)

        # Show segmented coins
        if num_coins > 0:
            print("\nSegmented Coins:")
            segment_coins(color_downsampled, circles)

    # Replace '1.jpg' with your image path
    path = '3.jpg'
    default(path)
```






```
Total coins: 19

Segmented Coins:
```


Coin 5
Coin 6
Coin 7
Coin 8
Coin 9
Coin 10
Coin 11
Coin 12
Coin 13
Coin 14
Coin 15
Coin 16
Coin 17
Coin 18
Coin 19

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def process_coins_watershed(image_path):
    # Read and validate image
    img = cv2.imread(image_path)
    if img is None:
        raise ValueError("Could not read image at provided path")

    # Downsample if either dimension exceeds 529px(23*23 )
    while img.shape[0] > 529 or img.shape[1] > 529:
        img = cv2.pyrDown(img)

    # Convert to grayscale and blur
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (3, 3), 0)

    # Thresholding with Otsu's method (invert if coins are lighter than background)
    _, thresh = cv2.threshold(blurred, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

    # Morphological closing to fill small holes
    kernel_size = int(input("Enter closing kernel size (odd number, default 3): ") or 3)
    morph_iterations = int(input("Enter closing iterations (default 1): ") or 1)
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel, iterations=morph_iterations)

    # Sure background by dilation
    sure_bg = cv2.dilate(closed, kernel, iterations=3)

    # Sure foreground using distance transform and thresholding
    dist_transform = cv2.distanceTransform(closed, cv2.DIST_L2, 5)
    _, sure_fg = cv2.threshold(dist_transform, 0.5 * dist_transform.max(), 255, 0)
    sure_fg = np.uint8(sure_fg)

    # Unknown region is the difference between background and foreground
    unknown = cv2.subtract(sure_bg, sure_fg)

    # Marker labelling
    _, markers = cv2.connectedComponents(sure_fg)
    markers = markers + 1  # Ensure background is not 0 but 1
    markers[unknown == 255] = 0  # Mark unknown regions with zero

    # Apply watershed
    markers = cv2.watershed(img, markers)

    # Mark boundaries in yellow
    segmented = img.copy()
    segmented[markers == -1] = [0, 255, 255]

    # Count coins by counting unique labels (ignoring background and boundaries)
    unique_markers = np.unique(markers)
    # Typically background is labeled as 1, and -1 is boundary
    coin_count = len(unique_markers[(unique_markers > 1)])
    print(f"Detected coins: {coin_count}")

    cv2.imwrite('watershed_result.jpg', segmented)
    return segmented

# Usage
result = process_coins_watershed("2.jpg")
plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
plt.title('Segmented Image')
plt.axis('off')
plt.show()
```

```
Enter closing kernel size (odd number, default 3): 5
Enter closing iterations (default 1): 1
Detected coins: 4
```

Segmented Image

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
import time

def canny_check(image):
    plt.imshow(image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')
    plt.show()

    fig, ax = plt.subplots(2, 6, figsize=(20, 8))
    for i in range(12):
        edges_img = cv2.Canny(image, i * 25, i * 25 + 50)
        row, col = divmod(i, 6)
        ax[row, col].imshow(edges_img, cmap='gray')
        ax[row, col].set_title(f'Canny {i * 25}-{i * 25 + 50}')
        ax[row, col].axis('off')

    plt.tight_layout()
    plt.show()

def detect_with_Hough(gray_image, color_image, canny_high, minDist=None, threshold=30, minRadius=None, maxRadius=None):


    # Detect circles
    circles = cv2.HoughCircles(gray_image, cv2.HOUGH_GRADIENT, 1, minDist,
                               param1=canny_high, param2=threshold,
                               minRadius=minRadius, maxRadius=maxRadius)

    output = color_image.copy()
    detected_circles = []
    if circles is not None:
        circles = np.uint16(np.around(circles[0]))
        detected_circles = circles.tolist()
        for x, y, r in circles:
            cv2.circle(output, (x, y), r, (0, 255, 0), 2)
            cv2.circle(output, (x, y), 2, (0, 0, 255), 3)
    return output, len(detected_circles), detected_circles

def segment_coins(color_image, circles):
    if not circles:
        return

    plt.figure(figsize=(12, 8))
    num_coins = len(circles)
    cols = 4  # Number of columns in the display grid
    rows = (num_coins + cols - 1) // cols

    for i, (x, y, r) in enumerate(circles):
```

```
            # Create mask for the coin
            mask = np.zeros_like(color_image)
            cv2.circle(mask, (x, y), r, (255, 255, 255), -1)

            # Apply mask to original image
            segmented = cv2.bitwise_and(color_image, mask)

            # Convert to RGB for proper matplotlib display
            segmented_rgb = cv2.cvtColor(segmented, cv2.COLOR_BGR2RGB)

            # Plot each segmented coin
            plt.subplot(rows, cols, i+1)
            plt.imshow(segmented_rgb)
            plt.axis('off')
            plt.title(f'Coin {i+1}')

        plt.tight_layout()
        plt.show()

    def default(path, option):

        if option == 1:
          downsample_size = 529
          img = cv2.imread(path)

        # Downsample the color image
        color_downsampled = img.copy()
        while color_downsampled.shape[0] > downsample_size or color_downsampled.shape[1] > downsample_size:
            color_downsampled = cv2.pyrDown(color_downsampled)

        # Convert to grayscale
        gray = cv2.cvtColor(color_downsampled, cv2.COLOR_BGR2GRAY)

        # Preprocessing
        laplacian = cv2.Laplacian(gray, cv2.CV_64F, ksize=3)
        sharpened = cv2.addWeighted(gray.astype(np.float64), 1.7, laplacian, -0.7, 0)
        sharpened = np.clip(sharpened, 0, 255).astype(np.uint8)
        blur = cv2.GaussianBlur(sharpened, (5, 5), 0)

        # Show Canny edges for threshold selection
        canny_check(blur)

        # Set parameters (pr

        low = 250
        high = 300
        minDist = blur.shape[0] // 8
        threshold_val = 30
        minRadius = blur.shape[0] // 20
        maxRadius = blur.shape[0] // 5

        # Detect circles
        result_img, num_coins, circles = detect_with_Hough(blur, color_downsampled, high,
                                                           minDist, threshold_val, minRadius, maxRadius)

        # Convert BGR to RGB for display
        result_img_rgb = cv2.cvtColor(result_img, cv2.COLOR_BGR2RGB)

        # Show detection results
        plt.figure(figsize=(10, 6))
        plt.imshow(result_img_rgb)
        plt.title(f'Detected Coins: {num_coins}')
        plt.axis('off')
        plt.show()

        print("Total coins:", num_coins)

        # Show segmented coins
        if num_coins > 0:
            print("\nSegmented Coins:")
            segment_coins(color_downsampled, circles)

    # Replace '1.jpg' with your image path
    path = '1.jpg'
    option =  int(input("Enter 1: Deafault: Based on the image given as input , hyper paramters are fixed with the current input ima

    default(path, option)
```

```
"""
1. image read
2. downsample
3. convert the color to gray
4. sharpen the image
5. gaussian blurr
6. apply threshold
7. apply morphology
8. if there is overlapping of the image we will be making the erode
9. distance transform ,, srefg
10. watershed
11. total oins deteced
"""


import cv2
import numpy as np
import matplotlib.pyplot as plt


class WatershedProcessor:
    def __init__(self, downsample_size=512, gaussian_kernel=(3, 3), morph_kernel_size=(5, 5),
                 dist_thresh_factor=0.8, morphology_iterations=1, erode_iterations=3):
        self.downsample_size = downsample_size
        self.gaussian_kernel = gaussian_kernel
        self.morph_kernel_size = morph_kernel_size
        self.dist_thresh_factor = dist_thresh_factor
        self.morphology_iterations = morphology_iterations
        self.erode_iterations = erode_iterations

    def read_image(self, path):
        return cv2.imread(path)

    def downsample_images(self, image):
        while image.shape[0] > self.downsample_size or image.shape[1] > self.downsample_size:
            image = cv2.pyrDown(image)
        return image

    def apply_high_pass_filter(self, image):
        laplacian = cv2.Laplacian(image, cv2.CV_64F)
        sharpened = cv2.addWeighted(image, 1.5, cv2.convertScaleAbs(laplacian), -0.5, 0)
        return laplacian, sharpened

    def apply_threshold(self, image):
        ret, thresh = cv2.threshold(image, 0, 255, cv2.THRESH_OTSU + cv2.THRESH_BINARY_INV)
        print("Optimal Threshold:", ret)
        return thresh

    def apply_morphology(self, image):
        kernel = np.ones(self.morph_kernel_size, np.uint8)
        return cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel, iterations=self.morphology_iterations)

    def compute_distance_transform(self, image):
        dist_transform = cv2.distanceTransform(image, cv2.DIST_L2, 3)
        ret, sure_fg = cv2.threshold(dist_transform, self.dist_thresh_factor * dist_transform.max(), 255, cv2.THRESH_BINARY)
        return dist_transform, np.uint8(sure_fg)

    def erode(self, filled_image):
        kernel = np.ones(self.morph_kernel_size, np.uint8)
        sure_bg = cv2.erode(filled_image, kernel, iterations=self.erode_iterations)
        return sure_bg

    def apply_watershed(self, img, sure_fg, filled_image):
        difference_image = cv2.subtract(filled_image, sure_fg)
        total_coins, markers = cv2.connectedComponents(sure_fg)
        markers += 1
        markers[difference_image == 255] = 0

        cv2.watershed(img, markers)

        # Create output image
        segmented_img = img.copy()
        segmented_img[markers == -1] = [0, 0, 255]  # Mark boundaries in red

        return (total_coins - 1), segmented_img
```

```
        def process_watershed(self, path):

            img = self.read_image(path)  # Fixed function call
            if img is None:
                print(f"Error: Could not read image from {path}")
                return None

            cv2.imwrite('output/watershed_original.jpg', img)

            img = self.downsample_images(img)  # Fixed function call
            cv2.imwrite('output/watershed_downsampled.jpg', img)

            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            cv2.imwrite('output/watershed_gray.jpg', gray)

            laplacian, sharpened = self.apply_high_pass_filter(gray)  # Fixed function call
            cv2.imwrite('output/watershed_laplacian.jpg', laplacian)
            cv2.imwrite('output/watershed_sharpened.jpg', sharpened)

            blurred = cv2.GaussianBlur(sharpened, self.gaussian_kernel, 0)
            cv2.imwrite('output/watershed_blurred.jpg', blurred)

            thresh = self.apply_threshold(blurred)  # Fixed function call
            cv2.imwrite('output/watershed_thresh.jpg', thresh)

            filled_image = self.apply_morphology(thresh)  # Fixed function call
            cv2.imwrite('output/watershed_filled_image.jpg', filled_image)

            overlap = input("Press 1 if there is significant overlap in image: ")
            if overlap == '1':
                filled_image = self.erode(filled_image)  # Fixed function call
                cv2.imwrite('output/watershed_eroded.jpg', filled_image)

            distance_transform, sure_fg = self.compute_distance_transform(filled_image)  # Fixed function call
            cv2.imwrite('output/watershed_distance_transform.jpg', distance_transform)
            cv2.imwrite('output/watershed_sure_fg.jpg', sure_fg)

            total_coins, watershed_result = self.apply_watershed(img, sure_fg, filled_image)  # Fixed function call
            plt.imshow(watershed_result)
            plt.axis("off")


            print("Total coins detected:", total_coins)
            return watershed_result


    if __name__ == "__main__":

        path2 = "3.jpg"

        downsample_size = int(input("Enter downsample size for images (default 512): ") or 512)
        gaussian_kernel = tuple(map(int, input("Enter Gaussian kernel size (default 3 3): ").split() or [3, 3]))
        morph_kernel_size = tuple(map(int, input("Enter Morphological kernel size (default 5 5): ").split() or [5, 5]))
        dist_thresh_factor = float(input("Enter Distance threshold factor for segmentation (default 0.8): ") or 0.8)
        morphology_iterations = int(input("Enter number of iterations for morphological operations (default 1): ") or 1)
        erode_iterations = int(input("Enter number of iterations for erosion (default 3): ") or 3)

        watershed_params = {
            "downsample_size": downsample_size,
            "gaussian_kernel": gaussian_kernel,
            "morph_kernel_size": morph_kernel_size,
            "dist_thresh_factor": dist_thresh_factor,
            "morphology_iterations": morphology_iterations,
            "erode_iterations": erode_iterations
        }

        wt = WatershedProcessor(**watershed_params)
        wt.process_watershed(path2)
```

```
Enter downsample size for images (default 512):
Enter Gaussian kernel size (default 3 3):
Enter Morphological kernel size (default 5 5):
Enter Distance threshold factor for segmentation (default 0.8):
Enter number of iterations for morphological operations (default 1): 4
Enter number of iterations for erosion (default 3):
Optimal Threshold: 166.0
Press 1 if there is significant overlap in image: 1
Total coins detected: 1
```

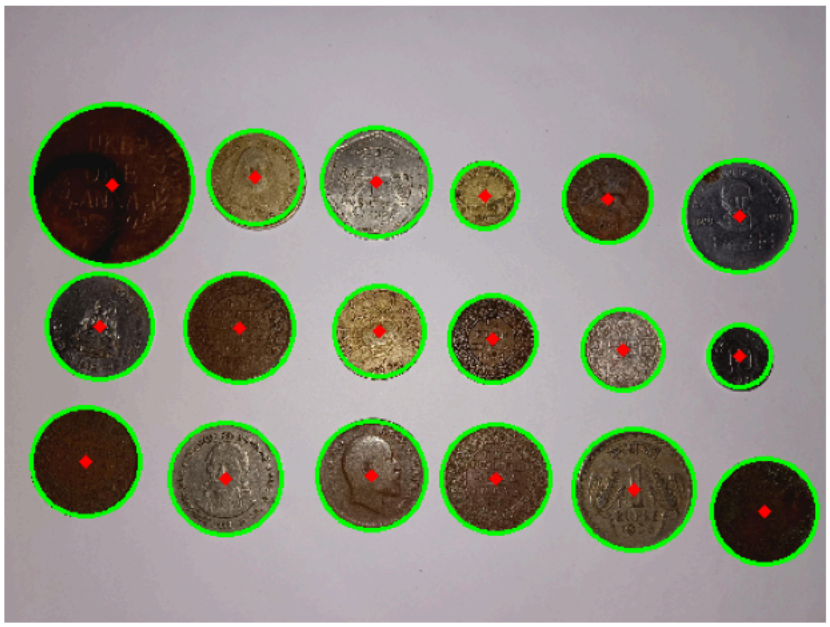
```
bimport cv2
import numpy as np
import matplotlib.pyplot as plt

def watershed(path):
    # Read and downsample image
    img = cv2.imread(path)
    if img is None:
        raise ValueError("Image not found or unable to load.")

    # Downsample while maintaining aspect ratio
    scale_factor = 529 / max(img.shape[:2])
    color_downsampled = cv2.resize(img, None, fx=scale_factor, fy=scale_factor)

    # Convert to grayscale
    gray = cv2.cvtColor(color_downsampled, cv2.COLOR_BGR2GRAY)

    # Improved preprocessing
    blurred = cv2.GaussianBlur(gray, (11, 11), 0)
    _, thresh = cv2.threshold(blurred, 0, 255, cv2.THRESH_OTSU + cv2.THRESH_BINARY_INV)

    # Morphological operations
    kernel = np.ones((3, 3), np.uint8)
    opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=2)

    # Proper sure background/foreground calculation
    sure_bg = cv2.dilate(opening, kernel, iterations=10)  # Fixed from erode to dilate
    dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
    _, sure_fg = cv2.threshold(dist_transform, 0.4 * dist_transform.max(), 255, 0)  # Lower threshold

    # Marker refinement
    sure_fg = np.uint8(sure_fg)
    unknown = cv2.subtract(sure_bg, sure_fg)

    # Connected components with background handling
    ret, markers = cv2.connectedComponents(sure_fg)
    markers += 1  # Background becomes 1
    markers[unknown == 255] = 0  # Unknown regions

    # Watershed algorithm
    markers = cv2.watershed(color_downsampled, markers)

    # Count unique regions (excluding background and boundaries)
    unique_markers = np.unique(markers)
    num_coins = len(unique_markers) - 2  # Subtract background (1) and boundaries (-1)
```

```
        # Visualization on downsampled image
        color_downsampled[markers == -1] = [0, 255, 0]  # Green boundaries

        # Display results
        print(f"Total coins detected: {num_coins}")
        plt.figure(figsize=(10, 6))
        plt.imshow(cv2.cvtColor(color_downsampled, cv2.COLOR_BGR2RGB))
        plt.title('Watershed Segmentation')
        plt.axis('off')
        plt.show()

    path = '1.jpg'
    watershed(path)
```

```
Total coins detected: 13
```


```
    import cv2
    import numpy as np
    import matplotlib.pyplot as plt

    def hybrid_coin_detection(image_path):
        # Load and preprocess image
        img = cv2.imread(image_path)
        if img is None:
            raise ValueError("Image not found")

        # Downsample while maintaining aspect ratio
        scale_factor = 529 / max(img.shape[:2])
        resized = cv2.resize(img, None, fx=scale_factor, fy=scale_factor)

        # Preprocessing pipeline
        gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
        blurred = cv2.GaussianBlur(gray, (7, 7), 3)

        # Hybrid detection approach
        # Step 1: Initial detection with Hough Circles
        circles = cv2.HoughCircles(blurred, cv2.HOUGH_GRADIENT, dp=1.2,
                                   minDist=30, param1=150, param2=30,
                                   minRadius=15, maxRadius=120)

        if circles is None:
            # Step 2: Fallback to Watershed
            _, thresh = cv2.threshold(blurred, 0, 255,
                                      cv2.THRESH_OTSU + cv2.THRESH_BINARY_INV)

            kernel = np.ones((3,3), np.uint8)
            opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=2)

            # Watershed markers from distance transform
            sure_bg = cv2.dilate(opening, kernel, iterations=3)
```

```
            dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
            _, sure_fg = cv2.threshold(dist_transform, 0.4*dist_transform.max(), 255, 0)

            sure_fg = np.uint8(sure_fg)
            unknown = cv2.subtract(sure_bg, sure_fg)

            _, markers = cv2.connectedComponents(sure_fg)
            markers += 1
            markers[unknown == 255] = 0

            markers = cv2.watershed(resized, markers)
            num_coins = len(np.unique(markers)) - 2
            result = resized.copy()
            result[markers == -1] = [0,255,0]

        else:
            # Use Hough results directly
            circles = np.uint16(np.around(circles[0]))
            num_coins = len(circles)
            result = resized.copy()
            for (x, y, r) in circles:
                cv2.circle(result, (x, y), r, (0,255,0), 2)
                cv2.circle(result, (x, y), 2, (0,0,255), 3)

        # Visualization
        plt.figure(figsize=(12,6))
        plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
        plt.title(f'Detected Coins: {num_coins}')
        plt.axis('off')
        plt.show()

        print(f"Total coins detected: {num_coins}")
        return num_coins, result
    # Usage
    path = '1.jpg'
    hybrid_coin_detection(path)
```




```
Total coins detected: 18
(18,
 array([[[144, 138, 149],
         [143, 137, 148],
         [145, 139, 150],
         ...,
         [147, 140, 147],
         [145, 138, 145],
         [144, 137, 144]],

        [[145, 139, 150],
         [144, 138, 149],
         [145, 139, 150],
         ...,
         [145, 138, 145],
         [144, 137, 144],
         [145, 138, 145]],

        [[141, 136, 147],
         [144, 138, 149],
         [144, 138, 149],
         ...,
         [146, 139, 146],
         [146, 139, 146],
         [147, 140, 147]],

        ...,
```

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def watershed_for_overlapping_coins(image_path):
    # Load image
    img = cv2.imread(image_path)
    if img is None:
        raise ValueError("Image not found")

    # Preprocessing
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    blurred = cv2.medianBlur(gray, 11)

    # Thresholding
    _, thresh = cv2.threshold(blurred, 0, 255, cv2.THRESH_OTSU + cv2.THRESH_BINARY_INV)

    # Morphological cleaning
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
    cleaned = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel, iterations=3)
    cleaned = cv2.morphologyEx(cleaned, cv2.MORPH_OPEN, kernel, iterations=2)

    # Distance transform
```

```
    dist_transform = cv2.distanceTransform(cleaned, cv2.DIST_L2, 5)
    cv2.normalize(dist_transform, dist_transform, 0, 1.0, cv2.NORM_MINMAX)

    # Regional maxima detection
    max_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7,7))
    dilated = cv2.dilate(dist_transform, max_kernel)
    regional_max = np.uint8(dist_transform >= dilated)

    # Marker identification
    _, markers = cv2.connectedComponents(regional_max)

    # Watershed preparation
    markers = markers + 1  # Background becomes 1
    markers[cleaned == 0] = 0  # Set background areas

    # Apply Watershed
    markers = cv2.watershed(img, markers)

    # Count coins (exclude background and boundaries)
    unique_markers = np.unique(markers)
    num_coins = len(unique_markers) - 2  # Subtract background (1) and boundaries (-1)

    # Visualization
    result = img.copy()
    result[markers == -1] = [0, 255, 0]  # Green boundaries

    plt.figure(figsize=(12,6))
    plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
    plt.title(f'Overlapping Coins Detected: {num_coins}')
    plt.axis('off')
    plt.show()

    return num_coins, result

# Usage
num_coins, result = watershed_for_overlapping_coins('1.jpg')
print(f"Detected coins: {num_coins}")
```


```
Detected coins: 863
```

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def watershed_for_overlapping_coins(image_path):
    # Load image
    img = cv2.imread(image_path)
    if img is None:
        raise ValueError("Image not found")
```