Coin Detection Extension Report

1. Introduction

This report details the extension of a basic coin detection image processing pipeline to handle more challenging scenarios. The primary goal was to improve accuracy and robustness when dealing with complex backgrounds and interfering elements within the images. All code to be mentioned in this report can be found in the CS373_coin_detection_extension.py file and the Utils folder.

2. Modifications to the Pipeline

Overview

The modified pipeline is given below:

1. Convert to Greyscale
2. Contrast Stretching
3. Laplacian Edge Detection
4. Noise Reduction and Breaking down interfering thin objects (Median Filtering)
5. Adaptive Thresholding
6. Morphological Operations (Dilation and Erosion)
7. Connected Component Labelling
8. Finding Valid Coin Regions

2.1 Laplacian Filter for Edge Detection

Instead of utilizing a traditional edge detection method, a Laplacian filter was implemented. This decision was based on the Laplacian filter's ability to highlight rapid intensity changes, making it particularly effective for detecting the circular edges of coins. The specific Laplacian kernel utilized was the one provided in the assignment (refer to Fig. 15). While effective, this filter introduced noise into the image, necessitating further processing (See below). Relevant code can be found in Utils/edge_detection.py

```
def border_ignore_filter(pixel_array, kernel):
    image_height = len(pixel_array)
    image_width = len(pixel_array[0])
    result_array = [[0 for _ in range(image_width)] for _ in
range(image_height)]  # Initialize the array

    for row in range(1, image_height - 1):
```

```
        for col in range(1, image_width - 1):
            accumulator = 0
            for i in range(-1, 1 + 1):  # row
                for j in range(-1, 1 + 1):  # column
                    image_value = pixel_array[row + i][col + j]
                    kernel_value = kernel[i + 1][j + 1]  # +1 to shift
                    accumulator += image_value * kernel_value
            result_array[row][col] = accumulator

    return result_array

def laplacian_filter(px_stretched_grey):
    laplacian_kernel = [[1.0, 1.0, 1.0],
                        [1.0, -8.0, 1.0],
                        [1.0, 1.0, 1.0]]
    return border_ignore_filter(px_stretched_grey, laplacian_kernel)
```

2.2 Noise Reduction

A median filter was selected to mitigate the noise introduced by the Laplacian filter. This non-linear filter efficiently removes salt-and-pepper noise without compromising the sharpness of the detected edges, making it suitable for this application. In addition, it breaks down interfering handwriting in Hard Case 3 which can be then discarded in the bounding box validation stage (Stage 8) for having irregularly small sizes.
Below shows the image before applying the median filter on Hard Case 1:

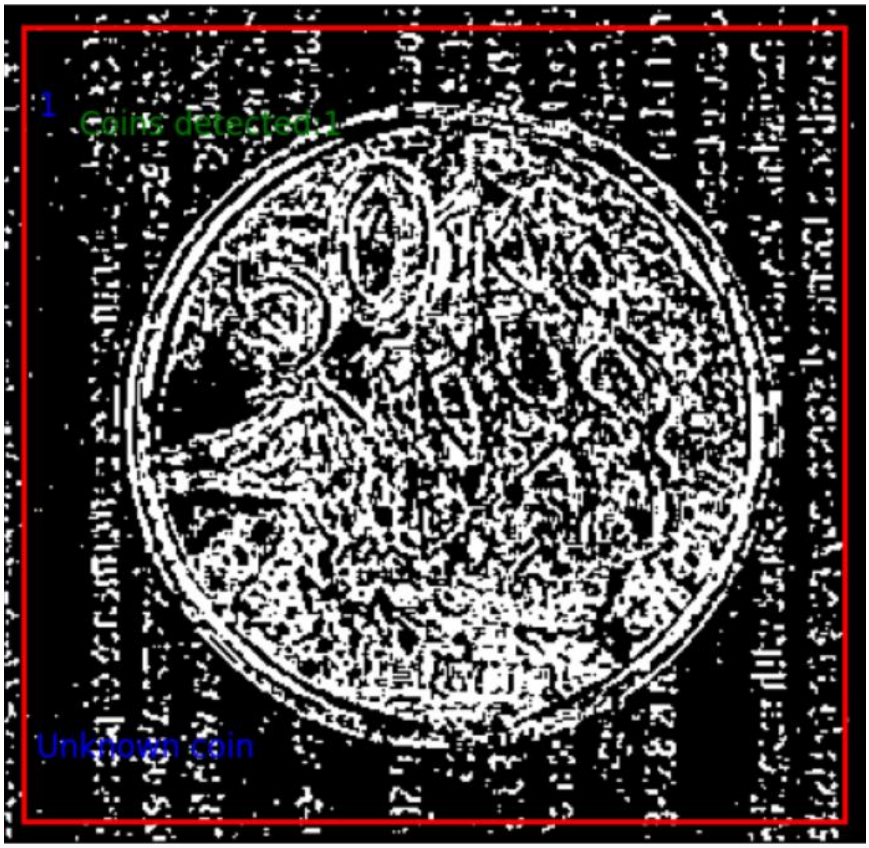
```
def median_filter(pixel_array):
    image_height = len(pixel_array)
    image_width = len(pixel_array[0])

    def get_extended_array():
        new_pixel_array = [row[:] for row in pixel_array]
        for r in range(len(new_pixel_array)):
            new_pixel_array[r].insert(0, 0)
            new_pixel_array[r].append(0)
        new_pixel_array.insert(0, [0 for _ in range(image_width + 2)])
        new_pixel_array.append([0 for _ in range(image_width + 2)])
        return new_pixel_array

    extended_array = get_extended_array()
    result_array = [[0 for _ in range(image_width)] for _ in
range(image_height)]  # Initialize the array

    for row in range(1, len(extended_array) - 1):
        for col in range(1, len(extended_array[row]) - 1):
            value_array = []
            for i in range(-1, 1 + 1):
                for j in range(-1, 1 + 1):
                    pixel_value = extended_array[row + i][col + j]
                    value_array.append(pixel_value)
            value_array.sort()
            length = len(value_array)
            if length % 2 == 1:
                result_array[row - 1][col - 1] = value_array[length //
2]
            else:
```

```
                result_array[row - 1][col - 1] = (value_array[length //
2 - 1] + value_array[length // 2]) / 2

    return result_array

def apply_median(pixel_array, times=3):
    result_array = pixel_array
    for i in range(times):
        result_array = median_filter(result_array)
    return result_array
```

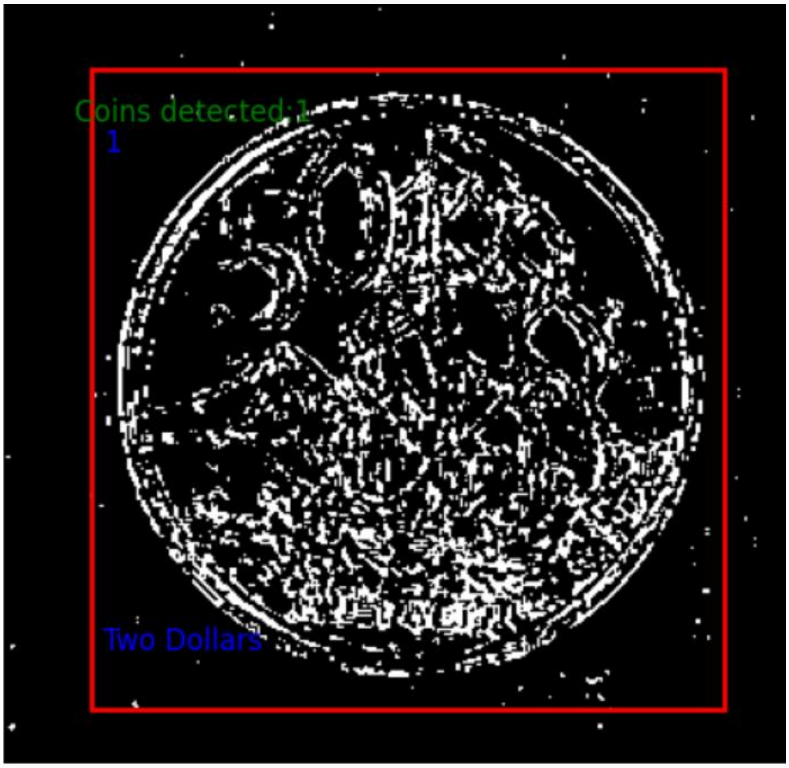
2.3 Handling Complex Backgrounds

Hard Case 1 presented a complex background that posed challenges for traditional global thresholding techniques (22, as required in the assignment). To overcome this, adaptive thresholding was implemented. This method calculates a threshold value for each pixel based on its local neighborhood, allowing for robust object identification even in the presence of varying illumination and intricate background details.

Below is the picture of Hard Case 1 using a global thresholding of 22, which the pipeline failed to detect:



Below is the picture of Hard Case 1 with adaptive thresholding, which eliminates the background interferences into several tiny dots that can be then eliminated in the erosion stage.



2.4 Addressing Interfering Elements

The presence of handwriting in Hard Case 3 introduced challenges for accurate coin detection as it cannot be eliminated with adaptive thresholding as in Hard Case 1. To address this, the blurring mean filter was removed from the pipeline to prevent the handwriting from being spread and becoming monolithic. Instead, a median filter was used to break down the writing into smaller components prior to morphological operations. Erosion was then applied, further shrinking and ultimately eliminating these unwanted elements, leading to a cleaner segmentation.

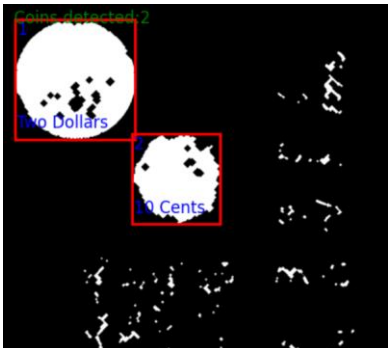


Figure 1 Hard Case 3 after using the median filter and erosion

2.4 Bounding box Validation

To address the challenges of interfering objects posed in the hard cases, validating the sizes and shapes of bounding boxes is necessary. Boxes of reasonable sizes and width/length ratio are retained. This solves the issue of having gigantic interfering objects as in Hard Case 2.

```
def find_valid_region_bounds(image):
    region_dict = find_region_bounds(image)
    valid_regions = {}
    value = 1

    for region_id, region_bounds in region_dict.items():
        min_x, max_x = region_bounds['min_x'], region_bounds['max_x']
        min_y, max_y = region_bounds['min_y'], region_bounds['max_y']

        width = max_x - min_x + 1
        height = max_y - min_y + 1

        # Check if the size of the box is within valid range
        if width >= 100 and width <= 400 and height >= 100 and height
<= 400:
            # Check if the ratio of width/height is close to 1
            aspect_ratio = width / height
            if 0.8 <= aspect_ratio <= 1.2:
                valid_regions[value] = {
                    'min_x': min_x, 'max_x': max_x,
                    'min_y': min_y, 'max_y': max_y
                }
                value += 1

    return valid_regions, value - 1
```

2.5 Coin Counting and Type Identification

Coin Counting:

Following segmentation, connected component analysis was employed to identify individual objects. The bounding box validation algorithm in the previous section outputs the valid regions and the total amount of them.

Type Identification:

Coin type identification relied on the principle that different coin denominations typically have distinct sizes. The area of each valid bounding box was calculated and compared to pre-defined size ranges for each coin type. In a controlled environment (e.g. a vending machine), it is possible to keep the distance between cameras and coins constant. The area of bounding boxes is calculated to identify the types of coins with a reasonable tolerance of discrepancies.

Figure 2 Hard Case 3 after applying the median filter and erosion

3. Results

The extended pipeline demonstrated marked improvements in performance on the challenging hard-level images compared to the basic approach.

- The combination of the Laplacian filter and median filter successfully extracted coin edges while minimizing noise interference.
- Adaptive thresholding proved highly effective in segmenting coins against the complex background of Hard Case 1.
- The removal of the blurring mean filter, coupled with the implementation of the median filter and erosion, significantly reduced false positives in Hard Case 3 caused by the presence of handwriting.
- Bounding box validation identifies coins properly.

Below are the results the pipeline produced for all hard cases. The pipeline also succeeded in detecting coins in all easy cases, whose results are not shown for brevity.

●原材料名:準チョコレート
全粉乳、ココアバター)(国
油脂、乳糖、砂糖、いちご果
ココアパウダー、全粉乳、カ
ー/乳化剤、着色料(赤ビー
イーストフード、(一部に小麦・
●内容量:3枚 ●賞味期限:
存方法:28℃以下の涼しい
保存してください。●製造者:
神戸市中央区御幸通7-1-15
株式会社(霞ヶ浦工場)
〒1751
(商品名中) 小麦、乳、大豆
ごまを含む製品と共通
ます。
た後、冷えて固まると表面が白くなること
離したもので、風味は変わりますが召し上
ほど。●万一品質に不都合がござい
ご連絡ください。お取り替えいたします。
2
One Dollar
1
Two Dollars
3
50 Cents

Coins detected:2
1
Two Dollars
2
10 Cents
w = x2 - x1
h = y2 - y1
x2 = w + x1
y2 = h + y1
2 3 4 5 6
x1 y1 w h score

4. Conclusion

The modifications incorporated into the coin detection pipeline demonstrably enhanced its robustness and accuracy in handling challenging image scenarios. The use of a Laplacian filter, adaptive thresholding, morphological operations, and size-based analysis significantly improved the system's performance. Future work could explore the integration of machine learning algorithms for coin type classification, potentially enabling the system to handle variations in camera perspectives and lighting conditions more robustly.