CS4302 Practical 3 Image Analysis

170011474

November 2020

Contents

1 Input Files **2**

2 Q1 **3**

3 Q2 **5**

4 Q3 **7**

5 Q4 **10**

6 Q5 **11**

6.1 Steps 11

6.2 Processes 12

6.2.1 money_easy 12

6.2.2 money_medium 15

6.3 money_hard 17

6.4 money_very_hard 20

6.5 money_extreme 22

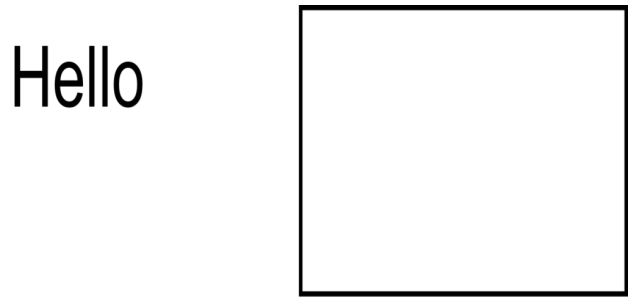
7 Q6 **26**

1 Input Files

The following 3 self-made images are used for this practical from Q1 to Q4:

(a) fig1

(b) fig2

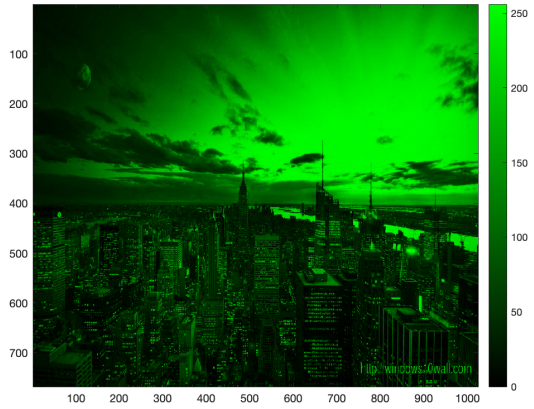


(c) fig3

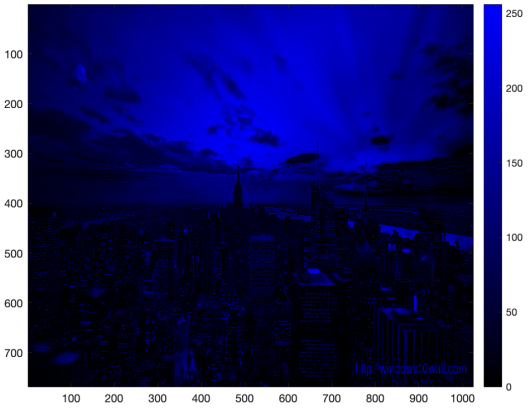
2 Q1

For each RGB image, it contains 3 color components: R (Red), G (Green), and B (Blue). Those figures are showing its components as monochrome images respectively. The reason why RGB color space can cover those images' color ranges is: RGB adopts an additive color scheme which predicts the appearance of colors made by coincident component lights, i.e. the perceived color can be predicted by summing the numeric representations of the component colors. Therefore, RGB uses red, green and blue as components to form many other colors, and those RGB images can be divided into 3 components to be shown individually.

(a) fig1 Red

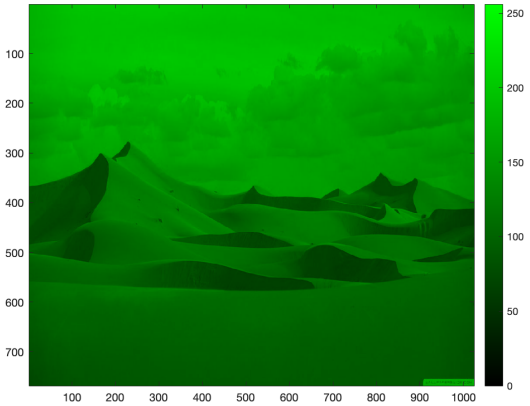


(b) fig1 Green

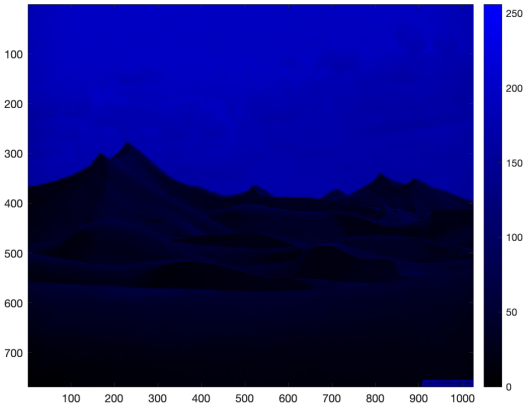


(c) fig1 Blue

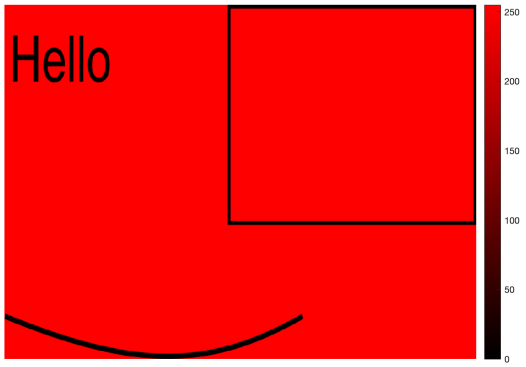
(a) fig2 Red



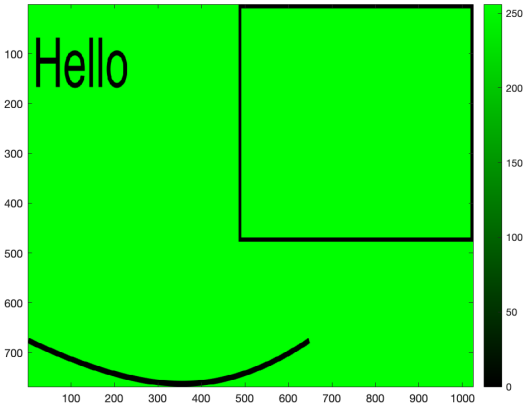
(b) fig2 Green



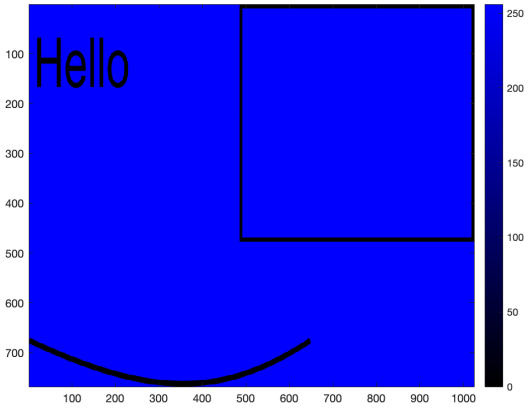
(c) fig2 Blue



(d) fig3 Red



(e) fig3 Green



(f) fig3 Blue

3 Q2

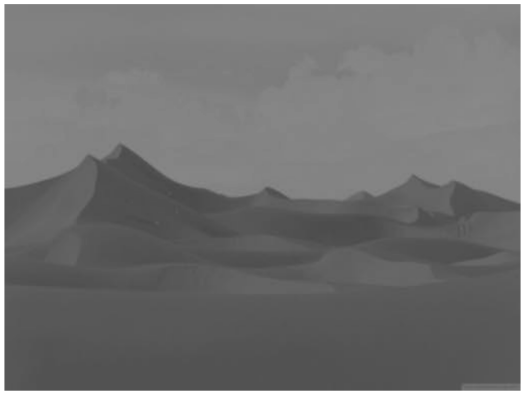
YCbCr is another color space and its 3 components are: Y for luminance, Cb for blue difference, and Cr for red difference. This scheme is used because sometimes we are more sensitive to spatial changes in the luminance and hence we can reduce the resolution for the chromaticity values. The reasons why Cb and Cr components do not have colors is: their values are simply pointing to the shifted difference. In fig3, it is slightly different from fig1 and fig2, where its Cb and Cr are almost the same. It is because it comes from a vector image and there is no original sufficient blue shift or red shift to be shown.

(a) fig1 Y

(b) fig1 Cb

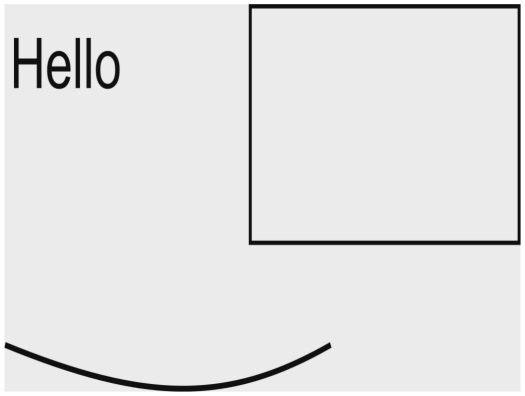
(c) fig1 Cr

(a) fig2 Y



(b) fig2 Cb

(c) fig2 Cr



(d) fig3 Y

(e) fig3 Cb

(f) fig3 Cr

4 Q3

To reduce sampling in both dimensions (horizontally and vertically), I wrote code to repeat 1 pixel's value for every 8 pixels. It is similar to YCbCr 4:2:0 down-sampling but it uses the factor of 8:

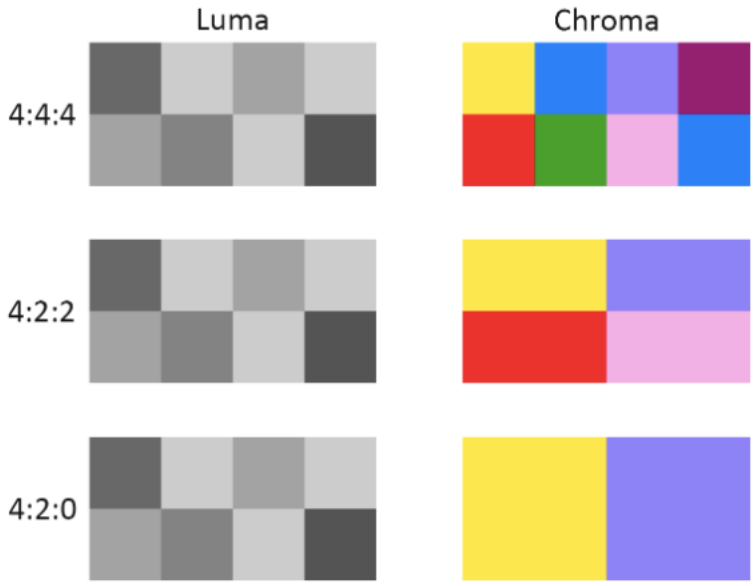
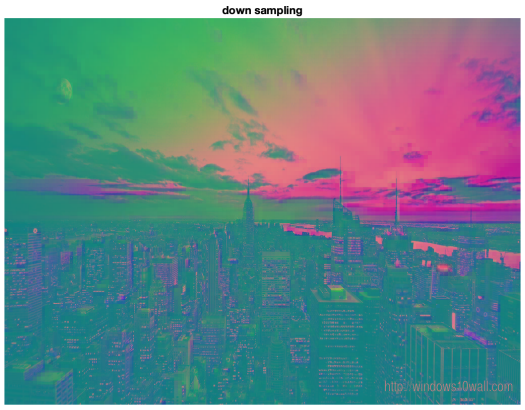


Figure 1: YCbCr

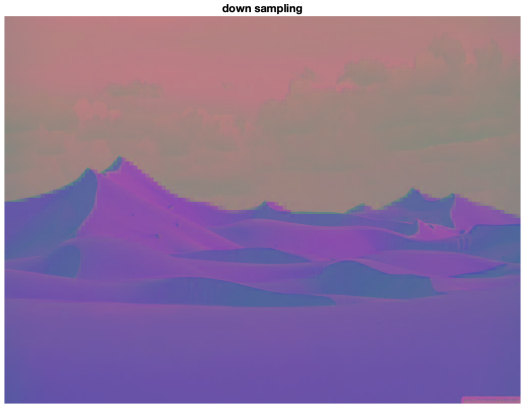
After down-sampling Cb and Cr, we can see that the objects in figures tend to have more jagged edges with colors. In another words, the resolution for colors has been dramatically reduced. The reason for that is: in the original files, for each pixel it might contain its own color value. However, down-sampling would brutally use one pixel's value to represent $8 * 8 = 64$ pixels. In fig3 (based on svg diagram), there is little change compared to others, since vectors inside are only in 1 color.

(a) fig1 Original

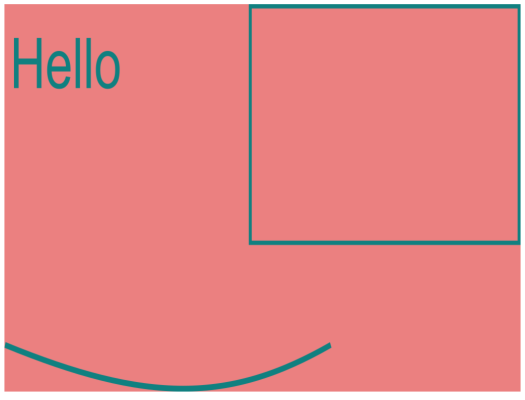


(b) fig1 Down-sampling

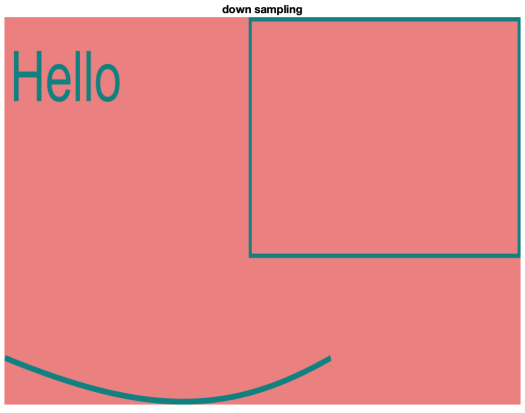
(c) fig2 Original



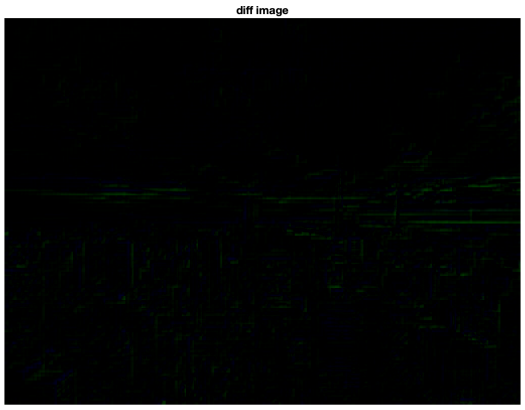
(d) fig2 Down-sampling



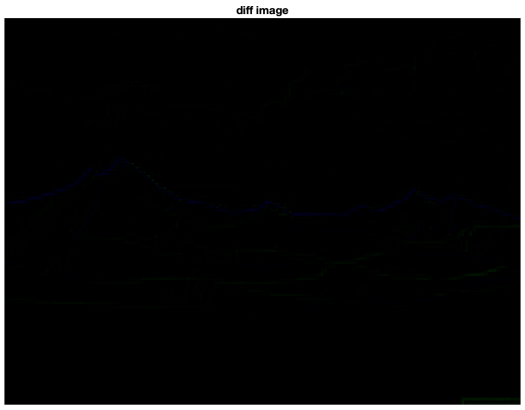
(e) fig3 Original



(f) fig3 Down-sampling



(a) fig1 diff



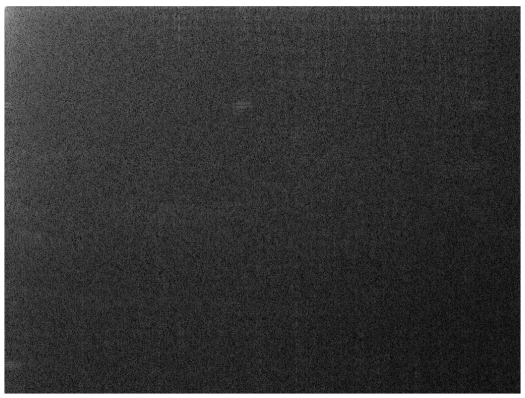
(b) fig2 diff

(c) fig3 diff

As we can see, in fig1 where there are more variations, the difference image shows a little green inside, while other pictures do not. This is because more variations would bring more obvious change after doing down-sampling, since information loss can be more.

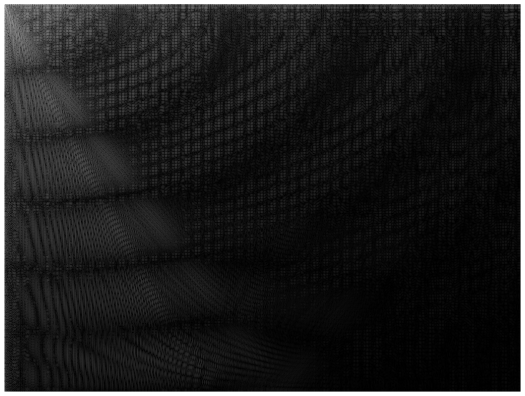
5 Q4

DCT2 (2D Discrete Cosine Transform) converts signals from spatial domain to frequency domain in 2-dimensional. It keeps more important parts of an image (such as lower frequency parts) and discards higher frequency parts (with many variations). After `dct2`, *lower frequency information would be kept in the upper left corner*. Higher frequency information is in other places.



(a) fig1 Frequency

(b) fig2 Frequency



(c) fig3 Frequency

As we can see from figures. Fig 1 has more variations compared to Fig 2, so in its corresponding frequency diagram, it contains more power in higher frequency places, and it seems more "noisy". In fig 3, the svg figure only contains some shapes, texts and a curve, it would lead to some "surface texture" like wave, which is more "regular" than fig 1 and fig2. Similarly, if a picture only contains a line, then it would show as some "dash" lines.

6 Q5

To get coins' sizes, use

```
d = drawline;
pos = d.Position;
diffPos = diff(pos);
diameter = hypot(diffPos(1),diffPos(2));
```

to draw a line on an image. In this way, the returned value would tell the distance of two points and hence coins' diameters would be approximated. The radius range is [130, 270] in this case, which would be used as a parameter in `imfindcircles`.

6.1 Steps

1. Read an image and then turn it into a gray scale image.

2. Edge detection: use `edge` function with the most powerful algorithm canny to detect edges of coins. Therefore, edges of circular objects would be more obvious.

3. Dilating: make the coin edges complete without holes. It uses a `strel` with a disk object to complete edges.

4. Use `imclearborder` to remove objects touching the image border and fill the edge. In this way, the object seems more "circular" and it would be more likely to be recognized as a circle.

5. Use `imfindcircles()` with a pre-set sensitivity to find circular objects. Since some coins, such as 20p, is not exactly a circle in a picture, we need to enhance the parameter `sensitivity` to recognize it.

For `money_easy`, the sensitivity is 0.94. For `money_medium`, the sensitivity is 0.945. For `money_hard`, the sensitivity is 0.951. For `money_very_hard` and `money_extreme`, the sensitivity is 0.955. Those parameters are provided by experiments.

6.2 Processes

6.2.1 money_easy

Figure 2: money_easy grayscale

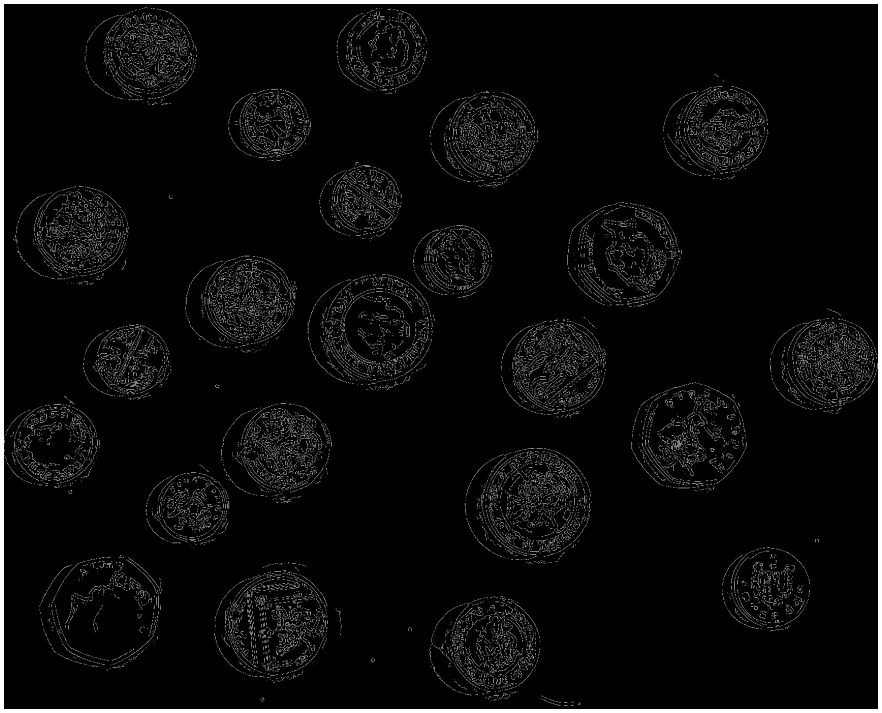


Figure 3: money_easy edge detection

Figure 4: money_easy after dilating

Figure 5: money_easy after clearing border

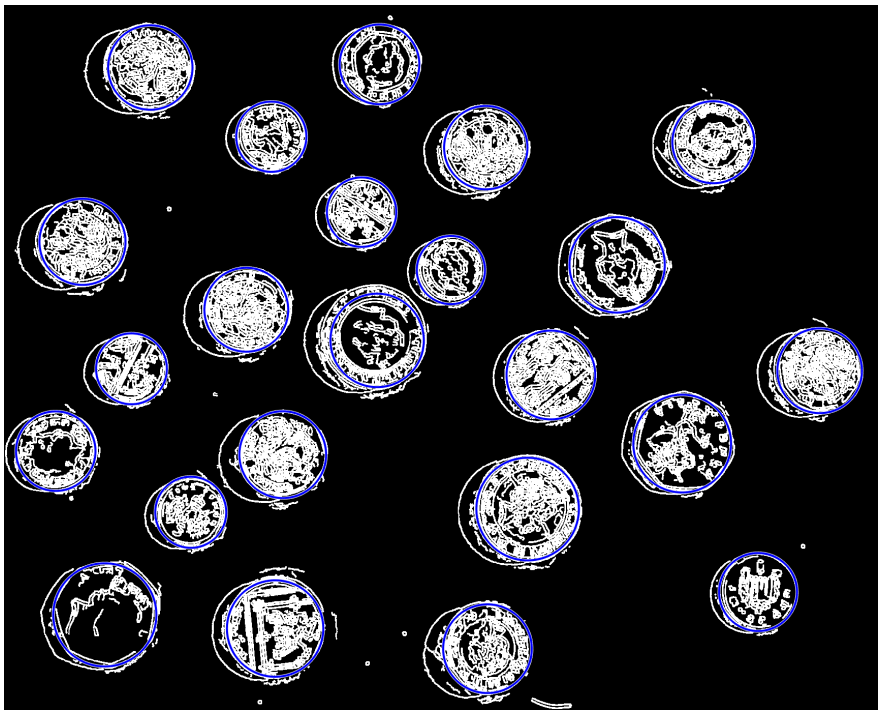


Figure 6: money_easy results

6.2.2 money_medium

Figure 7: money_medium grayscale

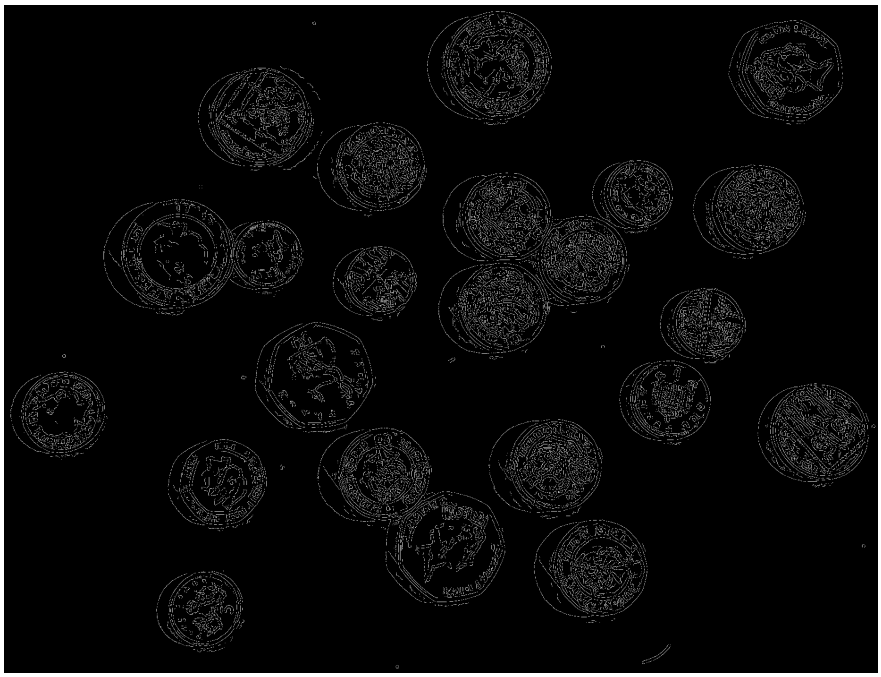


Figure 8: money_medium edge detection

Figure 9: money_medium after dilating

Figure 10: money_medium after clearing border

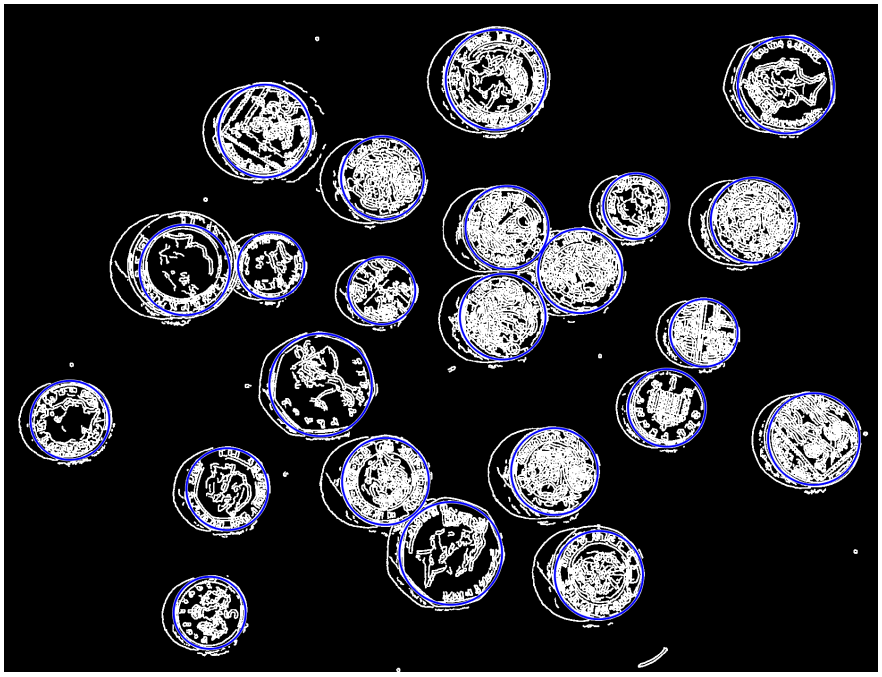


Figure 11: money_medium results

6.3 money_hard

Figure 12: money_hard gray scale

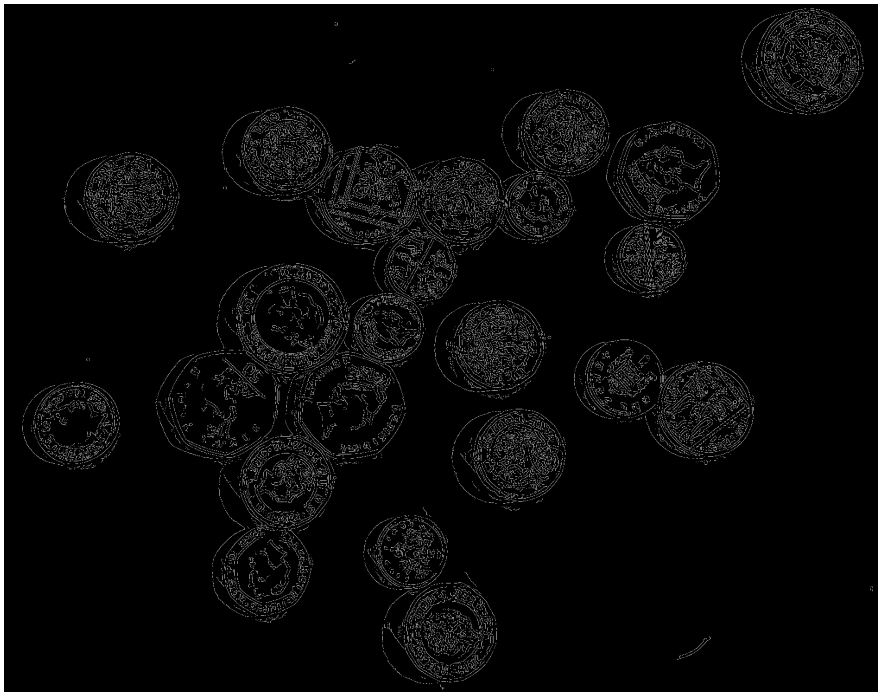


Figure 13: money_hard edge detection

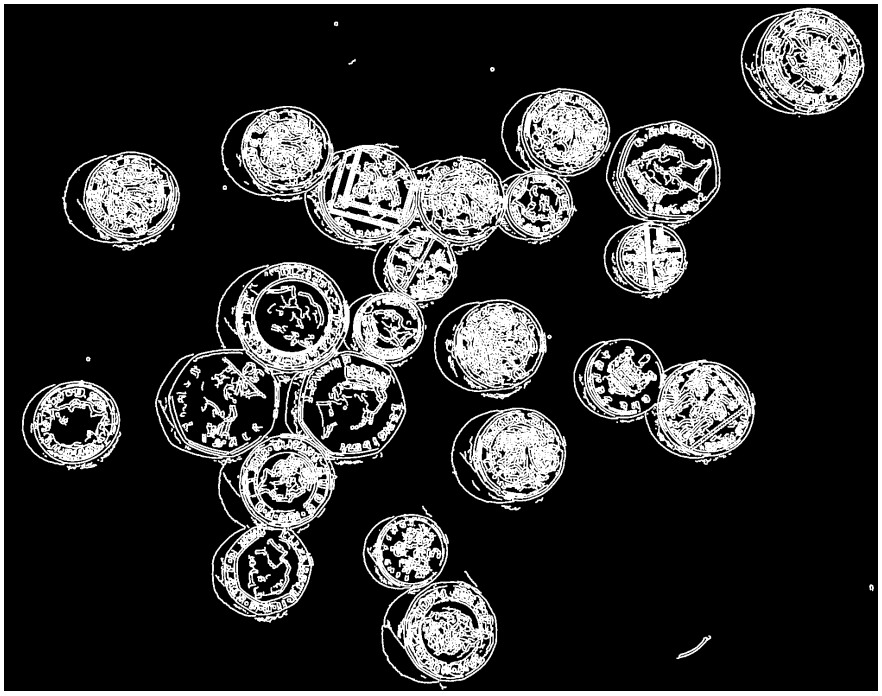


Figure 14: money_hard after dilating

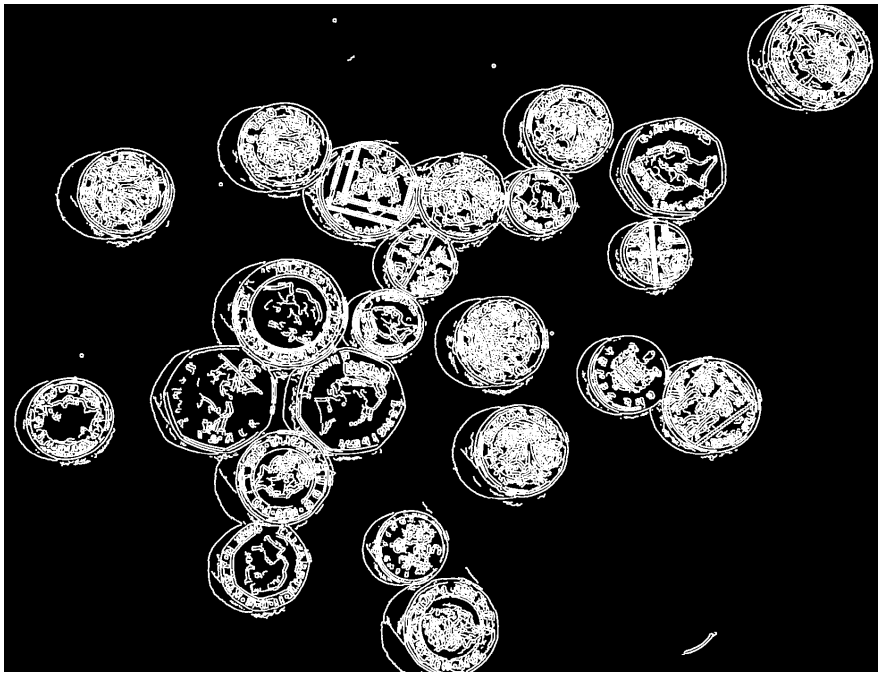


Figure 15: money_hard after clearing border

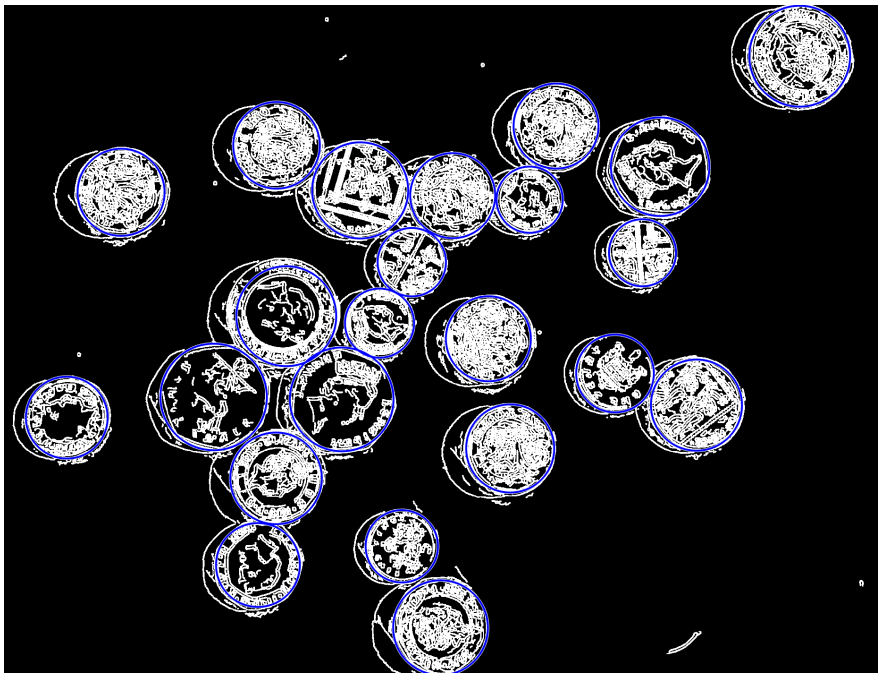


Figure 16: money_hard results

6.4 money_very_hard

Figure 17: money_very_hard gray scale

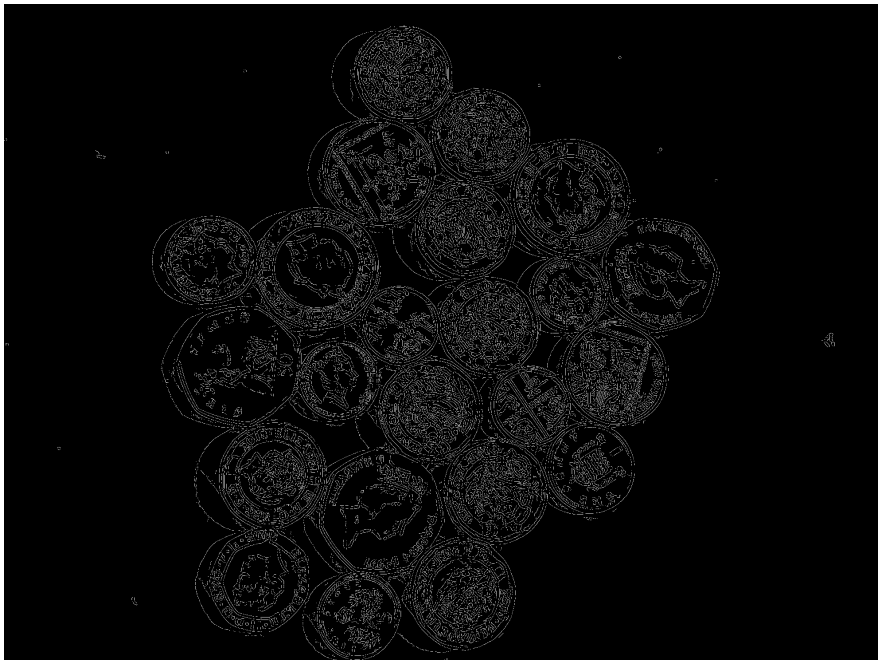


Figure 18: money_very_hard edge detection

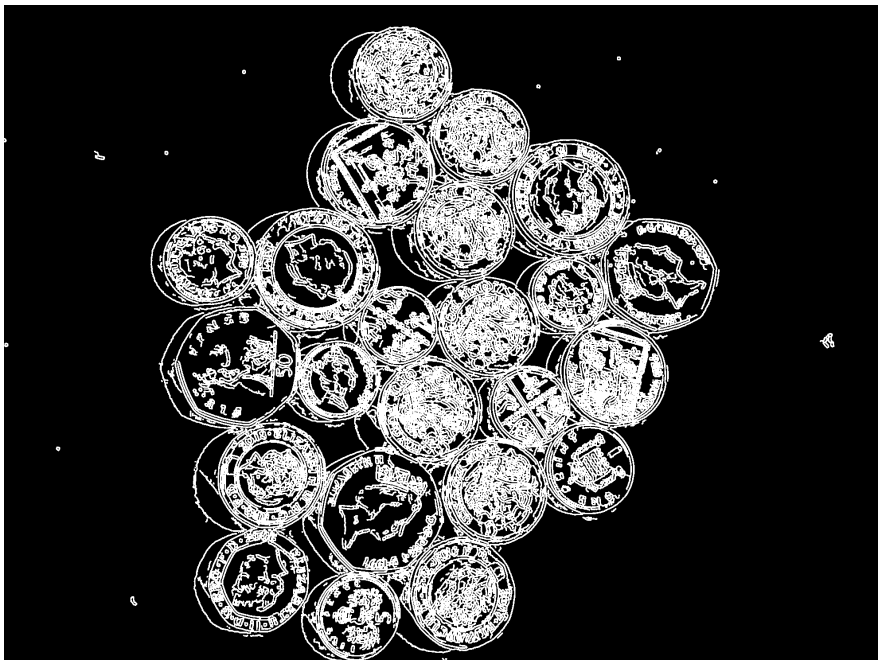


Figure 19: money_very_hard after dilating

Figure 20: money_very_hard after clearing border

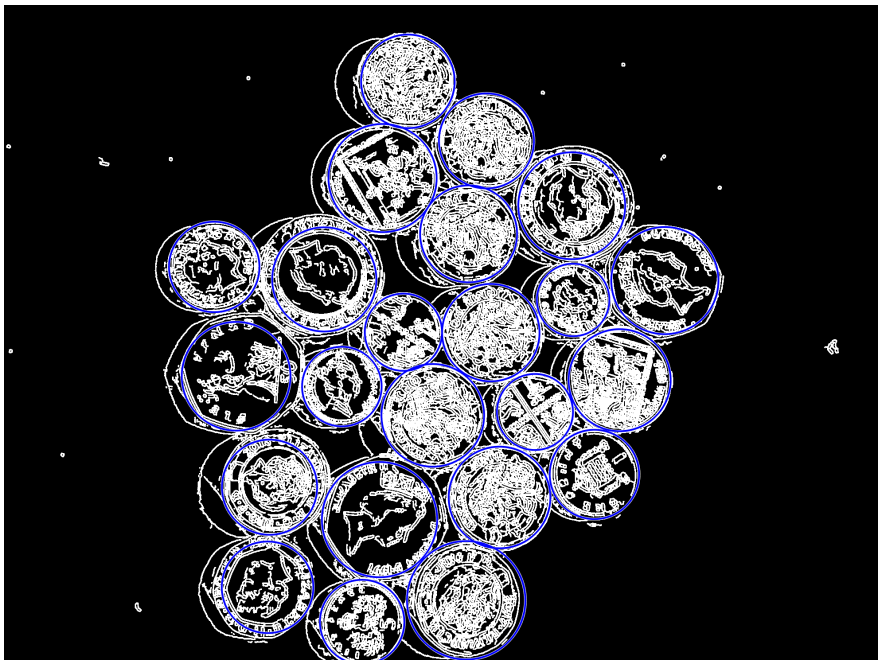


Figure 21: money_hard results

6.5 money_extreme

Figure 22: money_extreme gray scale

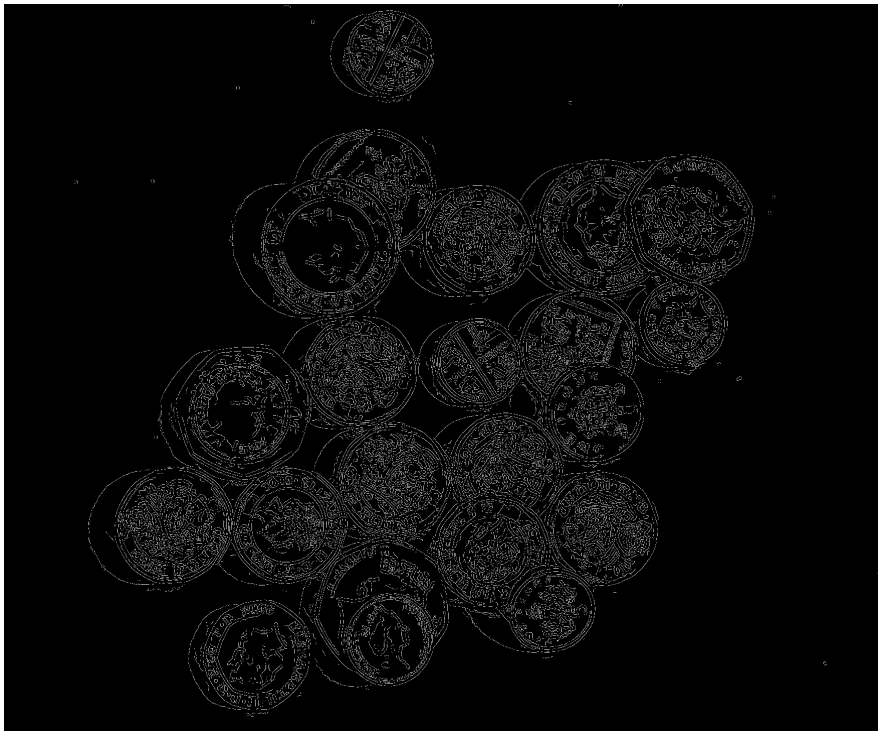


Figure 23: money_extreme edge detection

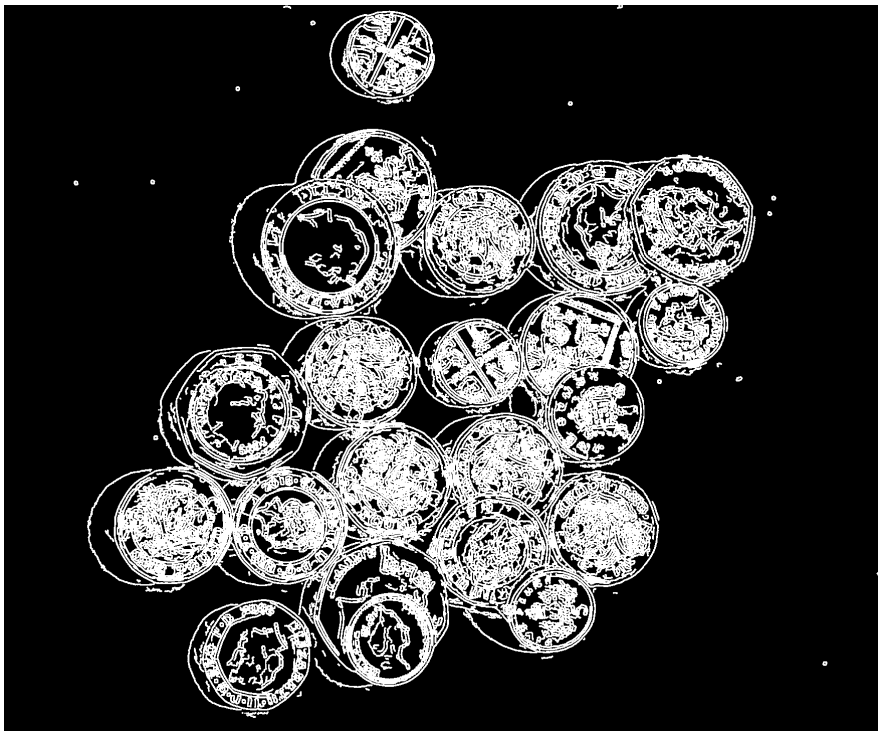


Figure 24: money_extreme after dilating

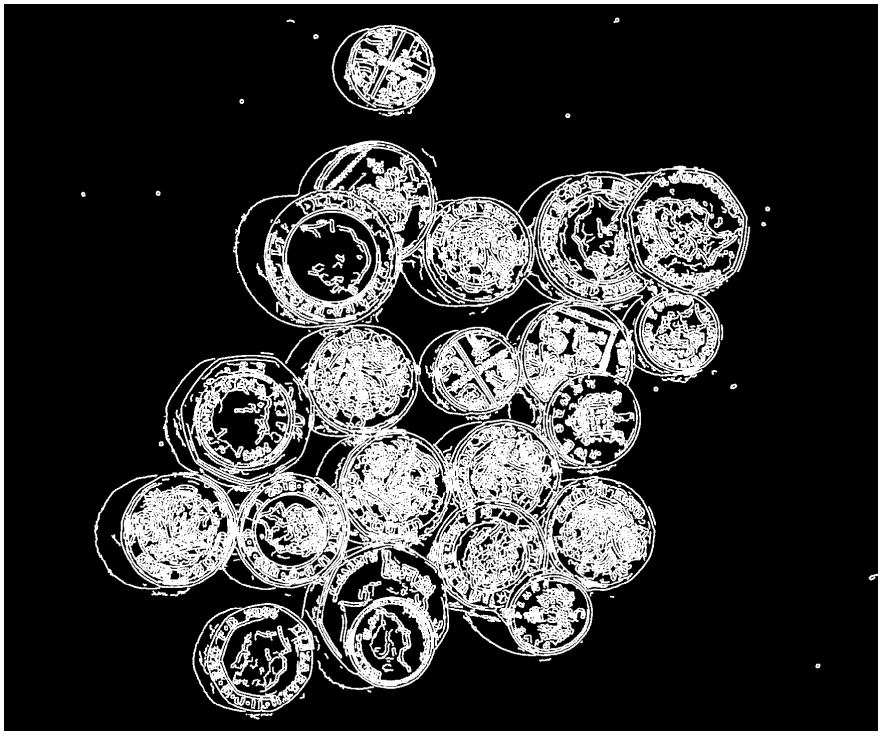


Figure 25: money_extreme after clearing border

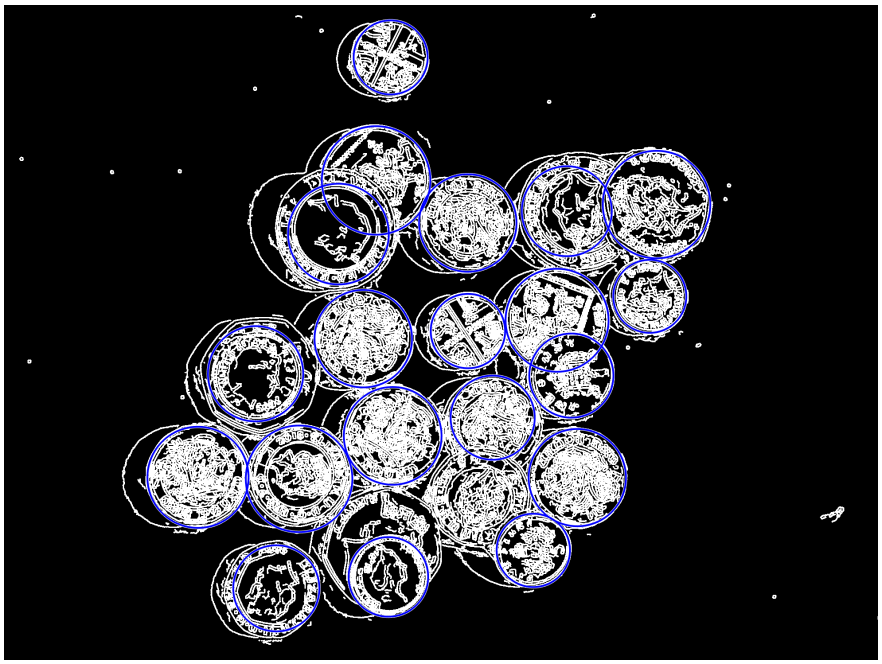


Figure 26: money_extreme results

The `money_extreme` gives me a result 20 rather than 23. Overall, except this one, other instances all gave correct results 23. The reason for this failure might be: there are too many overlapping coins and coins are locating near each other. This situation would bring more difficulties to distinguish one from others and also edge detection would not perform that well.

7 Q6

To determine coins' values, I used `money_easy` as a standard metric. For example, in `money_easy`, a 2-pound coin normally has a radius larger than 200 pixels and a 5-penny coin normally has a radius less than 150 pixels. Similarly, other types of coins can have their own radius range and I hard-coded those rules into code.

Since in different figures, such as in `money_medium`, the ratio or scale of figures can be different. In other words, same 2-pound coins might look "smaller" in another picture. To overcome this problem, I recorded the maximum radius occurring in `miney_easy` which is 209.7. In the current picture, find the maximum radius and then make a division to get the ratio. Afterwards, the current `radii` array provided by `imfindcircles` would be divided by that ratio, mapping to the original `money_easy` scale. In this way, the accuracy was increased compared to the case without introducing "ratio".

```
%   scale to the money_easy case
    curMax = max(radii);
    ratio = curMax/defaultMax;
    radii = radii / ratio;
```

Results In `money_easy`, the coin values: 14.09 pounds. In `money_medium`, the coin values: 15.12 pounds. In `money_hard`, the coin values: 13.12 pounds. In `money_very_hard`, the coin values: 11.58 pounds. In `money_extreme`, the coin values: 11.81 pounds.

Evaluation The difficulty here is to determine the true edge of coins rather than their "shadow" edge. Another problem is for non-circular coins like 20p, their recognizable circle structure would be smaller than their genuine structure, which would lead to confusion with other types of coins. For example, a 50p coin is easily recognized as a 2-pound coin.

In results mentioned above and images below, we can see as recognition difficulty increases, more wrong classifications are made even though "scaling to money_easy" is applied. But overall, in `money_easy` and `money_medium`, the classification predictions are good.

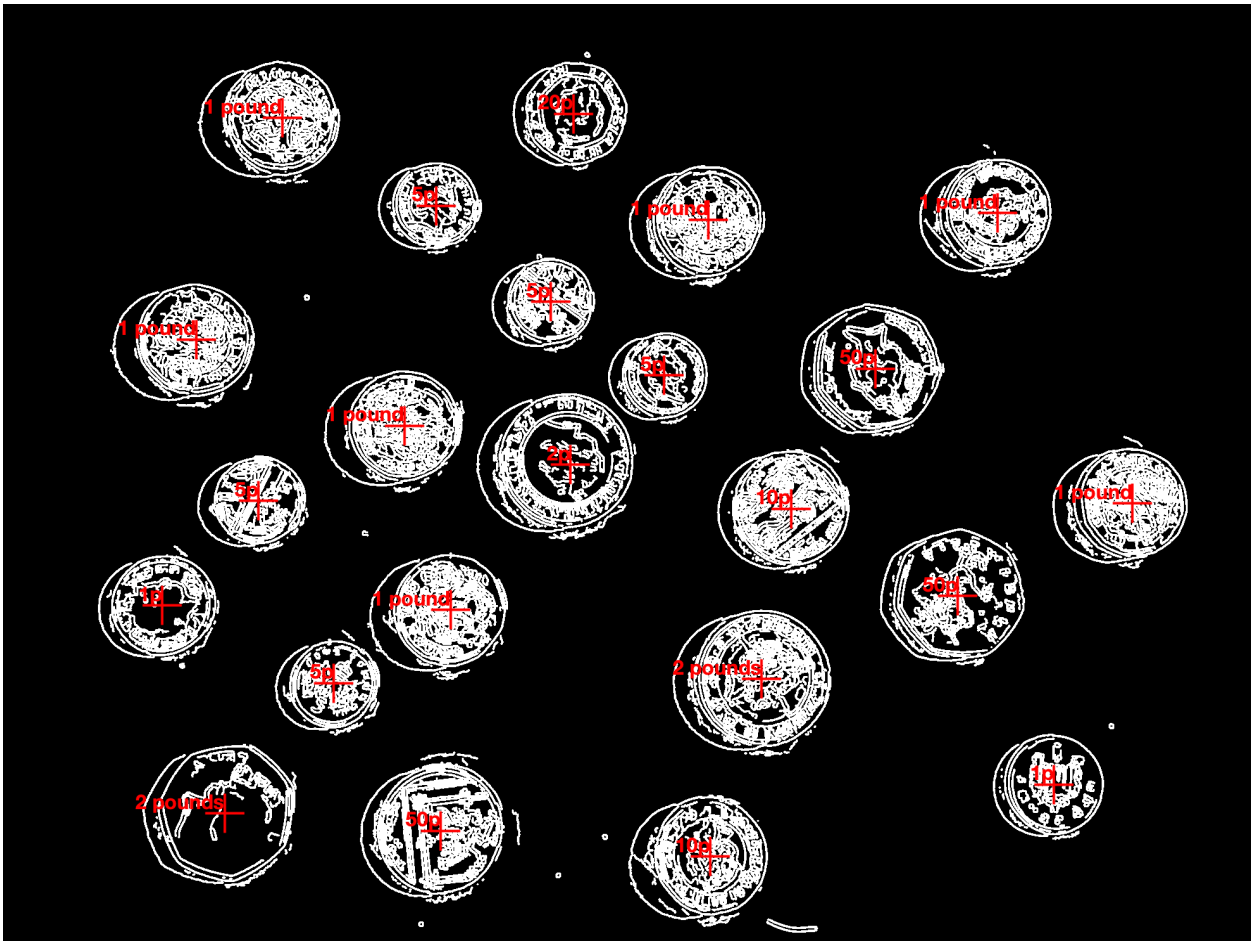


Figure 27: money_easy recognition

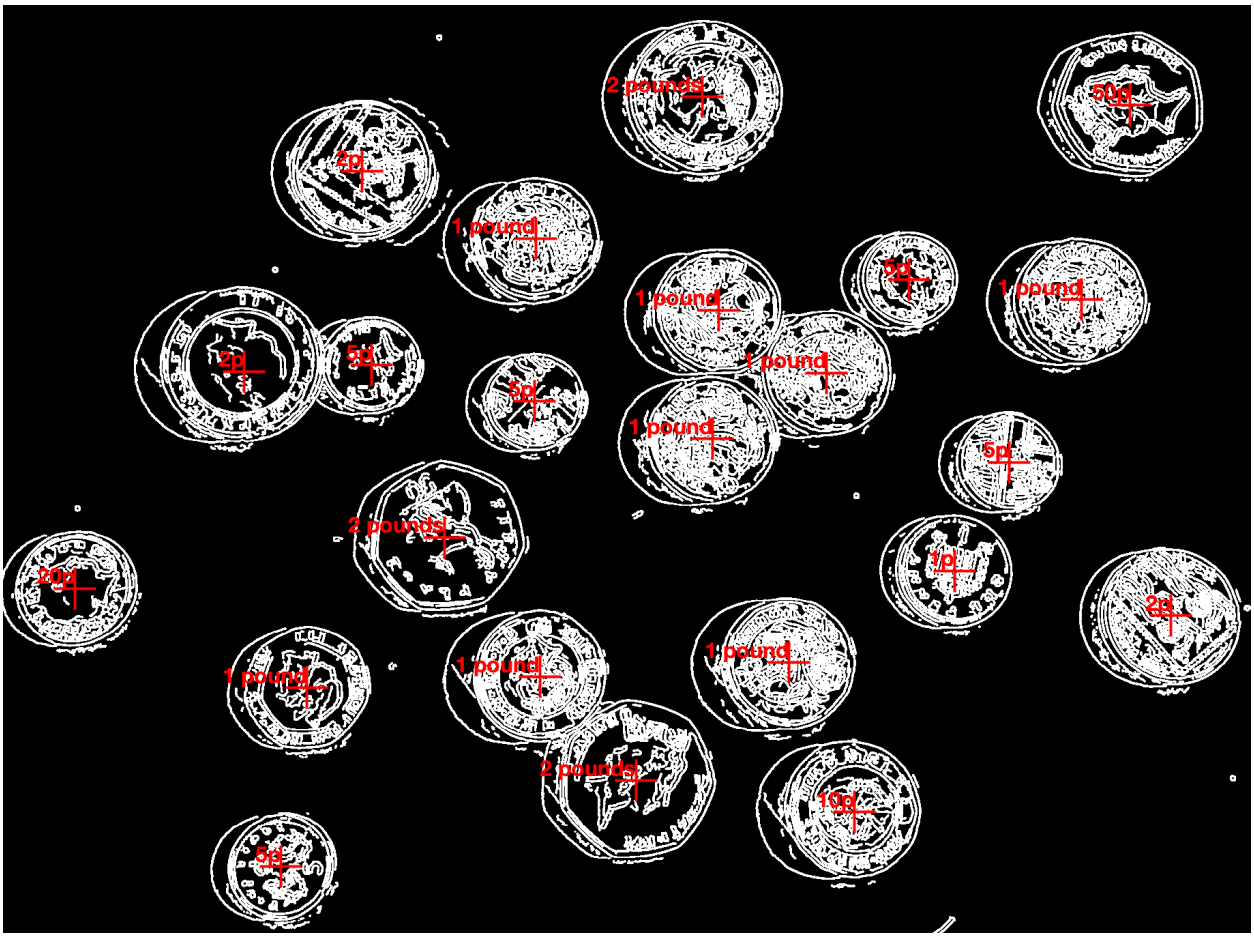


Figure 28: money_medium recognition

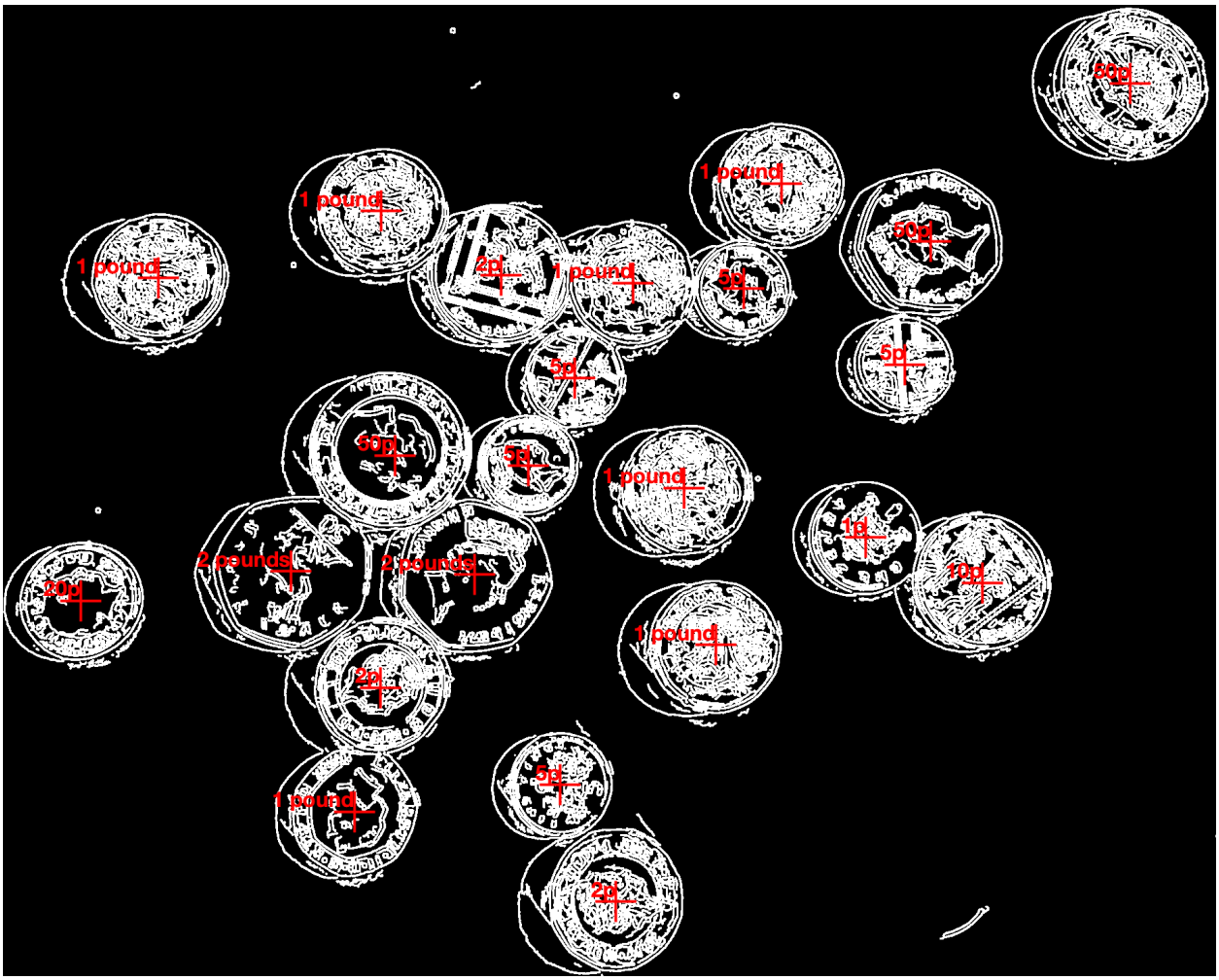


Figure 29: money_hard recognition

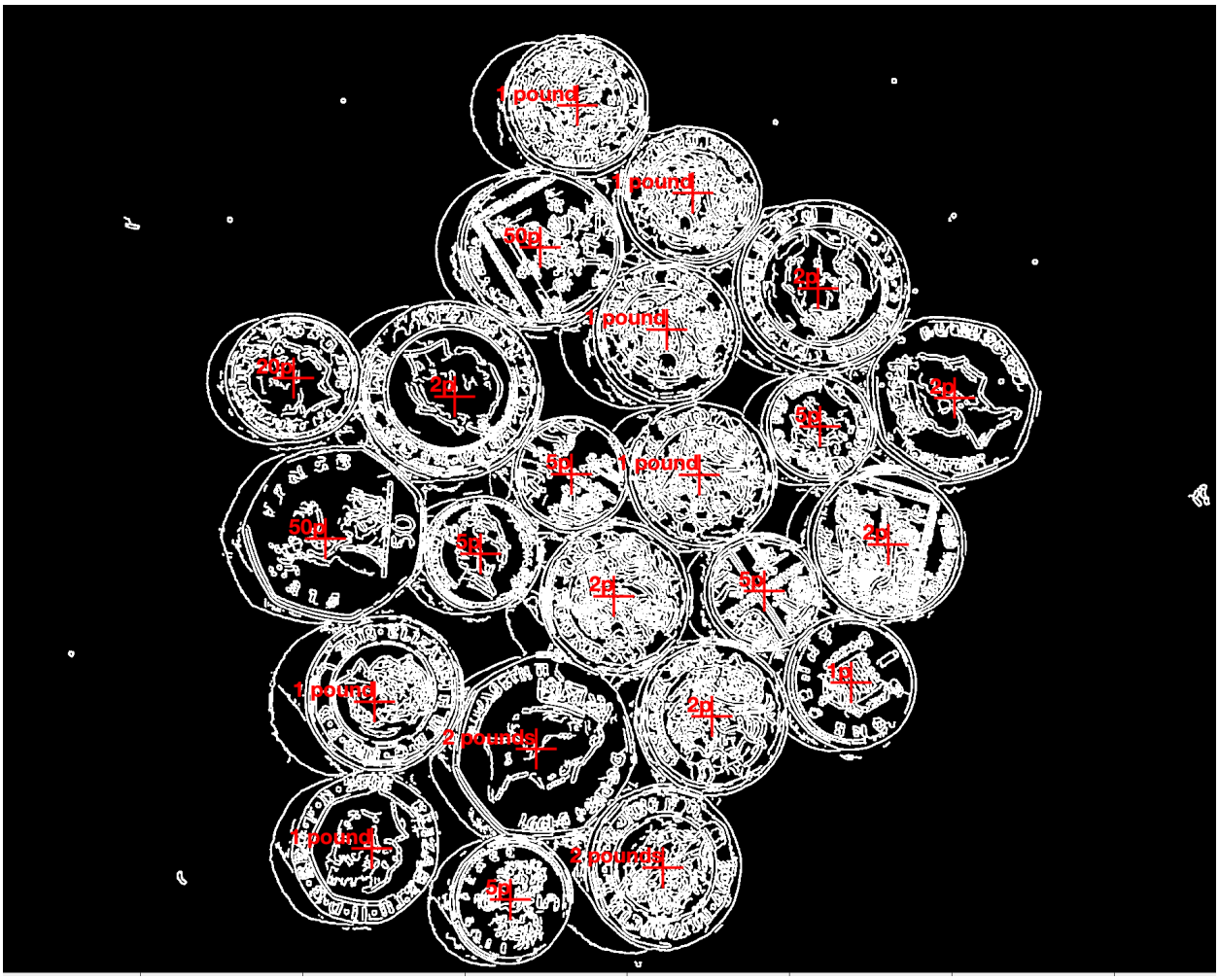


Figure 30: money_very_hard recognition

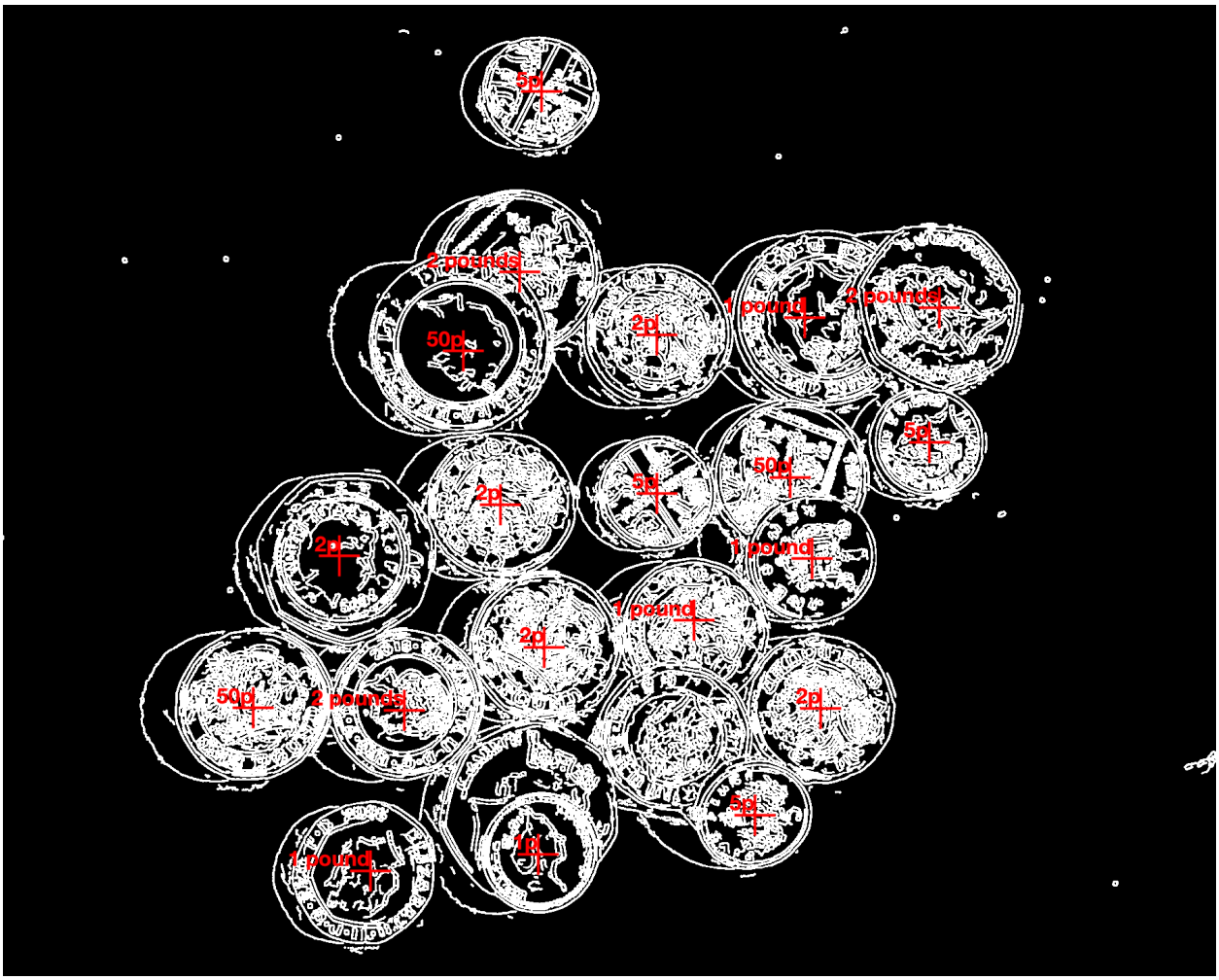


Figure 31: money_extreme recognition