ELEC4630 Assignment 1

HENGJI ZHAO

47184521

Question 1: Street Signs Detection.

- **The target of the question**

This question aims to detect the street signs in 10 images. The 10 images are shown in Figure 1.1.

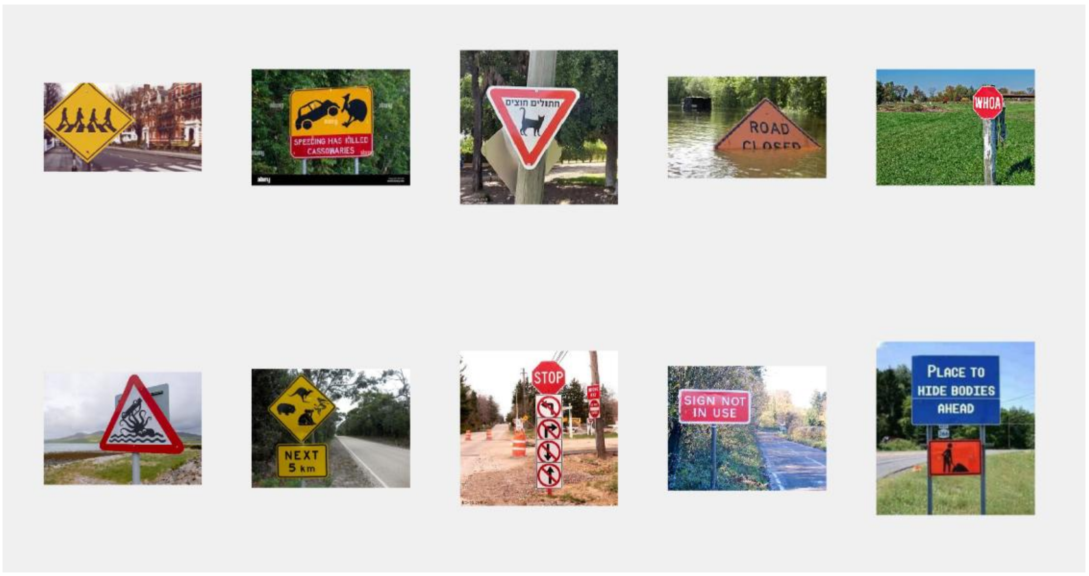
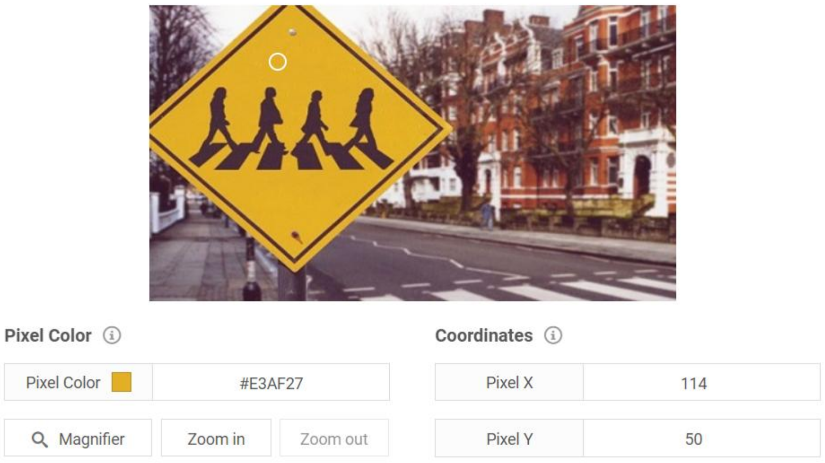


Figure 1.1: The street signs need to be detected.

To mark the signs in Figure 1.1, I need to apply multiple computer vision techniques to find out the street signs. Here I will introduce the computer vision techniques I applied to this question, after that I will describe how I found the solution for this question.

- **The computer vision techniques I applied.**

1. **RGB coordinate:** RGB coordinate is used to explain how an image is formed, we know that there are 3 primary colors they are Red, Green and Blue, these are the represented words for RGB. Each color has 256 levels ranging from 0 to 255. So, this coordinate can represent 16777216 colors (256*256*256) in total. In this question, I used a website named "RedKetchup" to help me find out the RGB values for me to determine the reasonable color range to set the threshold. The address of the pixel detect tool is https://redketchup.io/color-picker. Figure 1.2 shows the usage of the tool.



Pixel Color

Pixel Color 🟨	#E3AF27

Coordinates

Pixel X	114
Pixel Y	50

Figure 1.2: The usage of the pixel color detect tool.

In Figure 1.2, we find the pixel color we selected is #E3AF27, this is a hexadecimal number, the RGB values in decimal are 227, 175 and 39 respectively. The pixel is located at the little white circle. We can use this to help with the range of values.

2. **Threshold:** This method is a technique that can convert the grayscale images into a binary form, that is either white or black. For example, if we what to find the parts where the R value larger than 150, we can set the threshold of the red value 150, than use the MATLAB to find the places that R value larger than 150. In this question, because the colors of each sign are quite different. so we should apply multiple threshold values for different situations. In my code, I will create an array to store the threshold values.

3. **Erode and dilate**: These two techniques are similar because both these two methods would use a kernel matrix to do the convolution calculation, these two calculations are working on the binary images, in this question, we will do these techniques on the RGB filtered images. The difference is that for the erode calculation, the pixel from original image would be written as 1 only when all the pixel under the kernel matrix are all 1, this technique can effectively remove the noise around the objects we want to detect, because these noise are much smaller compare to the objects we want to detect. On the other hand, the dilate technique method has very similar calculation logic, a pixel would be written as 1 when at least 1 pixel is 1 under the kernel matrix. So this technique can be used to enlarge the object that we want to detect.
   Normally these two techniques would appear together because the after we applied the erode technique, though the noise removed, the target object would also be smaller. At this moment, we can use the dilate method to enlarge the eroded image.

This step can help us to do further work.

4. **Edge detection:** Just as the topic's name, this technique is used to detect the edge of the objects in an image. There are multiple methods to detect the edges. For example, like "Gradient-Based Method", "Sobel operator" and "Canny edge detector". In my method, I did not use the edge detection.

   Sobel operator: The Sobel operators are divided into two types, the horizontal and vertical operator respectively. Figure 1.3 illustrates the two operators, the figure is provided by the lecture 3 lesson slides which in page 16.

$$\underset{s_x}{\frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}} \qquad \underset{s_y}{\frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}}$$

Figure 1.3: The two types of Sobel operators.

   These two kinds of Sobel operators would do the kernel convolution calculations with the images, and then we do use the following formula to do the calculation, calculate the average value for the formula calculated value and then take the average value as a threshold value, if the added value larger than the threshold value we set it 1, otherwise we set that 0. The formula I mentioned is given below:

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

   G_x and G_y are the values of the two Sobel operators respectively.

- **Method introduction**

  At the start, we should read the images from the files, the images are in the Figure 1, here I do plot the figure here again.

  After we read the images, we firstly divide them into different groups according to their different colors, by observing the images, I divide then into RED, BLUE and YELLOW signs. Note that there are also some white signs in these images, I do not decide to divide them into one group that is because these white signs normally have red boundaries, so they can be divided into red signs, this step we should use the knowledge of RGB. The figure for its red channel is shown in Figure 1.4.

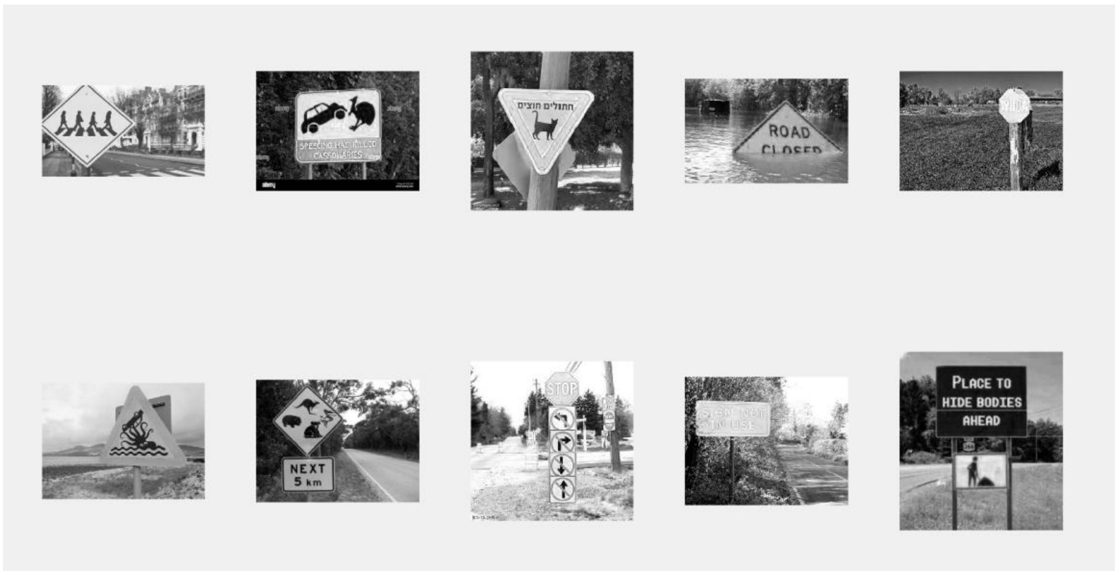


Figure 1.4: The images in red channel.

Similarly, we can plot the images in green and blue channels, the corresponding figure are shown in Figure 1.5 and Figure 1.6.

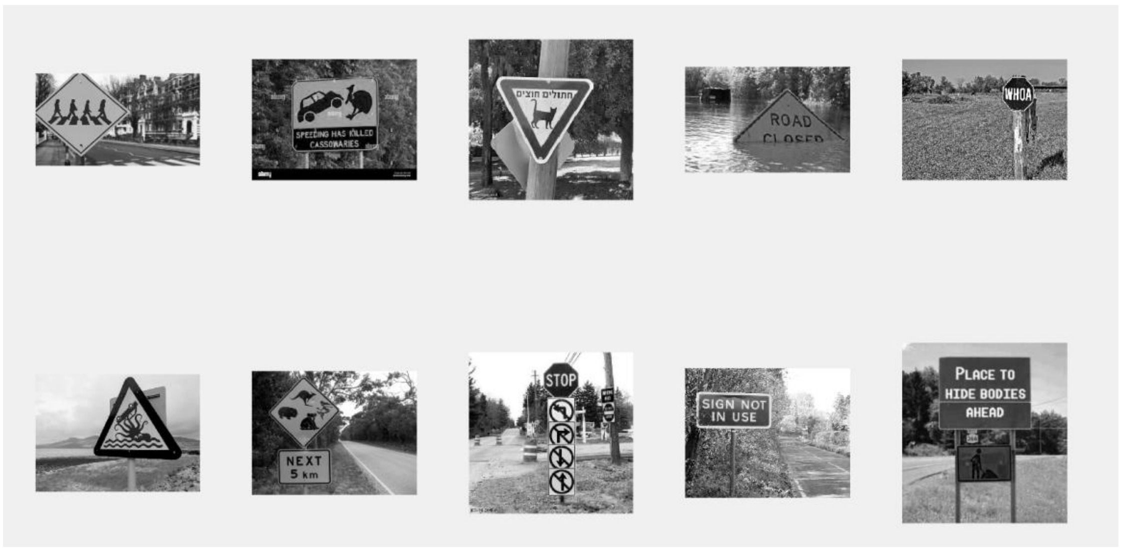


Figure 1.5: The images in green channel.

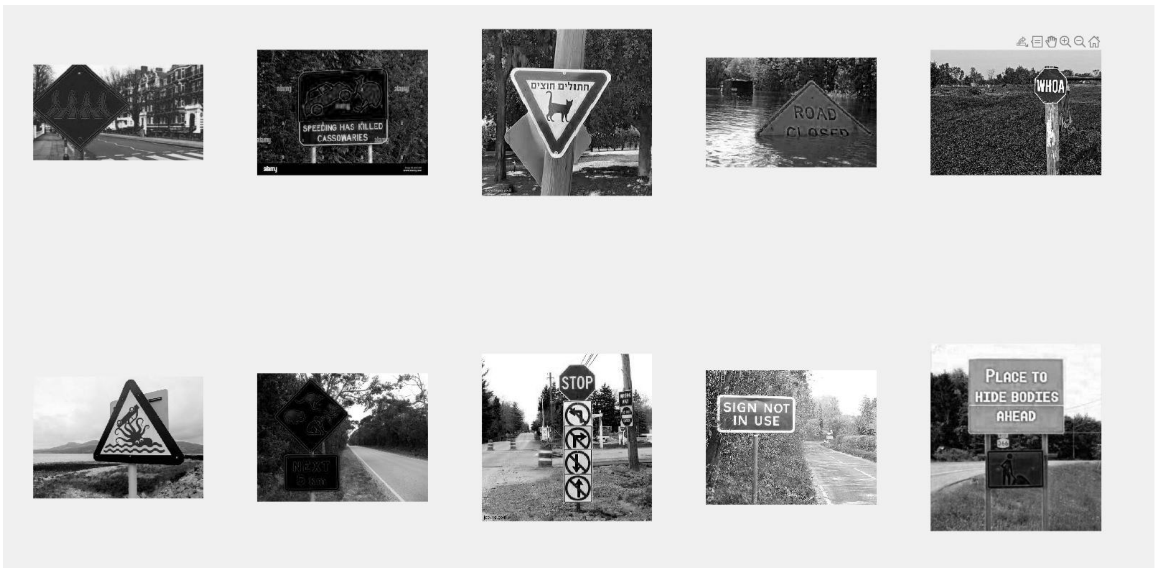


Figure 1.6: The images in blue channel.

Then, we should start to do the thresholding process, this step can help us find out which part includes the red sign. The result of the filtering process is shown in Figure 1.7.

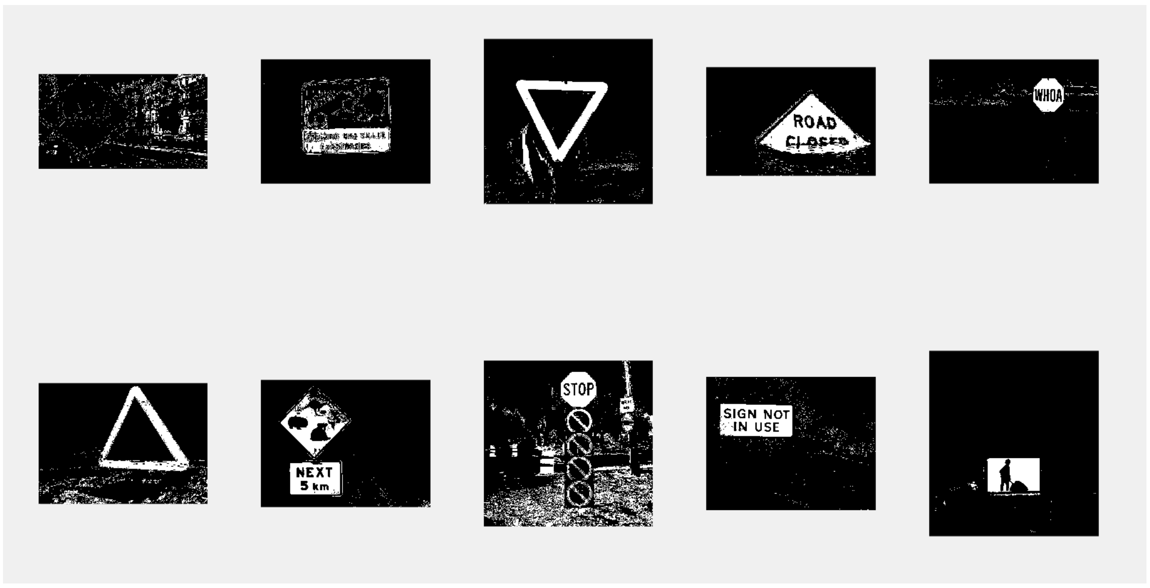


Figure 1.7: The images after the red RGB filtering.

According to the Figure 1.7, we can easily find that the images which include red signs have a every good detection performance. Then we should start to do the erode and the dilate processes for the red RGB filtered images. in this process, I would first do a dilate to enlarge some details of the images, this is because for the image6, we can find that there are a little missing of the structure of the triangle shape, I want to make the triangle to looks more “perfect”. The result of the first dilatation process is shown in Figure 1.8.

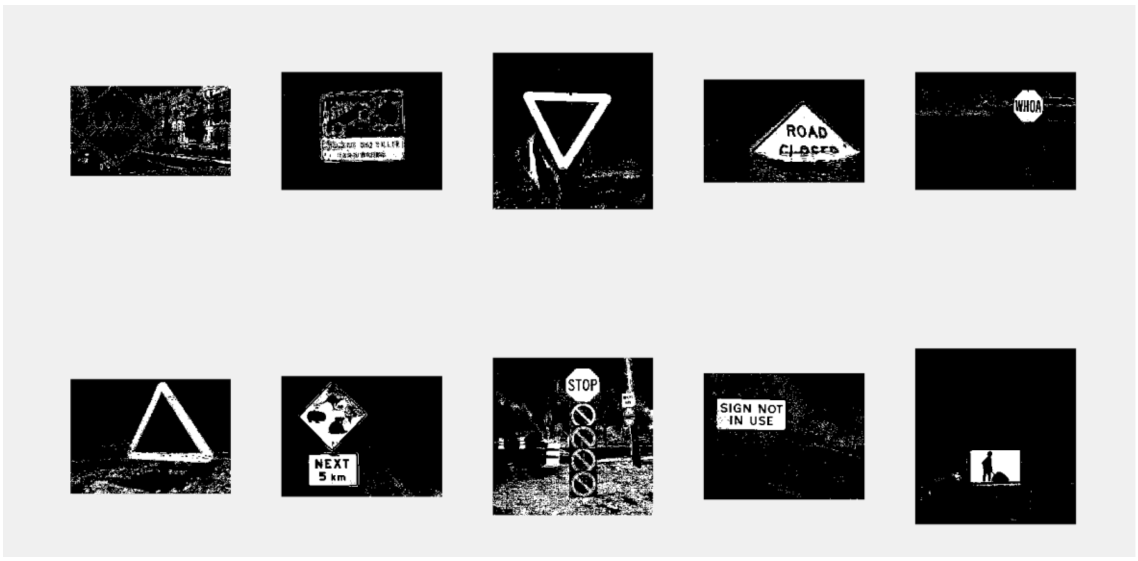


Figure 1.8: The first dilatation process for the red filtered images.

There is also a “unexpected gain”, that is the image4, if we see this image from our eyes, we might consider this image as a yellow sign, however, I think because of the shadow, the yellow sign taken into a “red” one.

Let’s go back to our images, we observed that the details of the images are all well fixed, then we should do an erode process. I just created a element with width of 2, so to do the erode process to remove the noise. The result after we applied the erode process is shown in Figure 1.9.

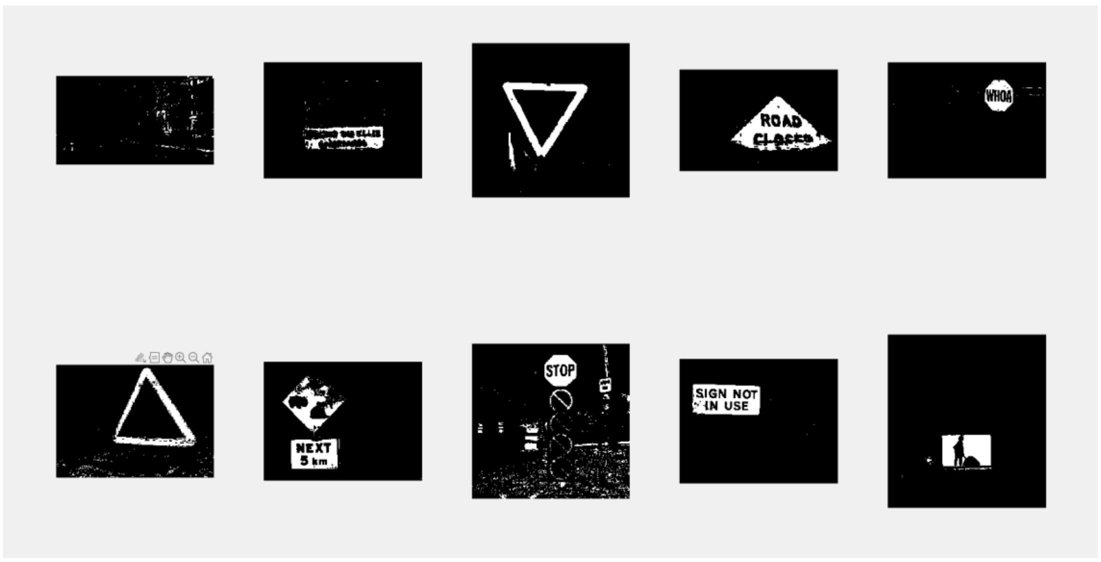


Figure 1.9: The images after the erode process.

We find the effect of the erode process is good, many noise are removed, however there are still some little noise appear on our images, if we plot the rectangle on the current images, of course our final images would be influenced. So, we need to set a threshold to filter the noise. Compared to the target objects we want to detect, the size of the noise can be filtered. In our code, I enlarge the size of the images 16 times, so when we design the minimal size of the filter, we should set that larger. Figure 1.10 is the result of the second dilatation process.

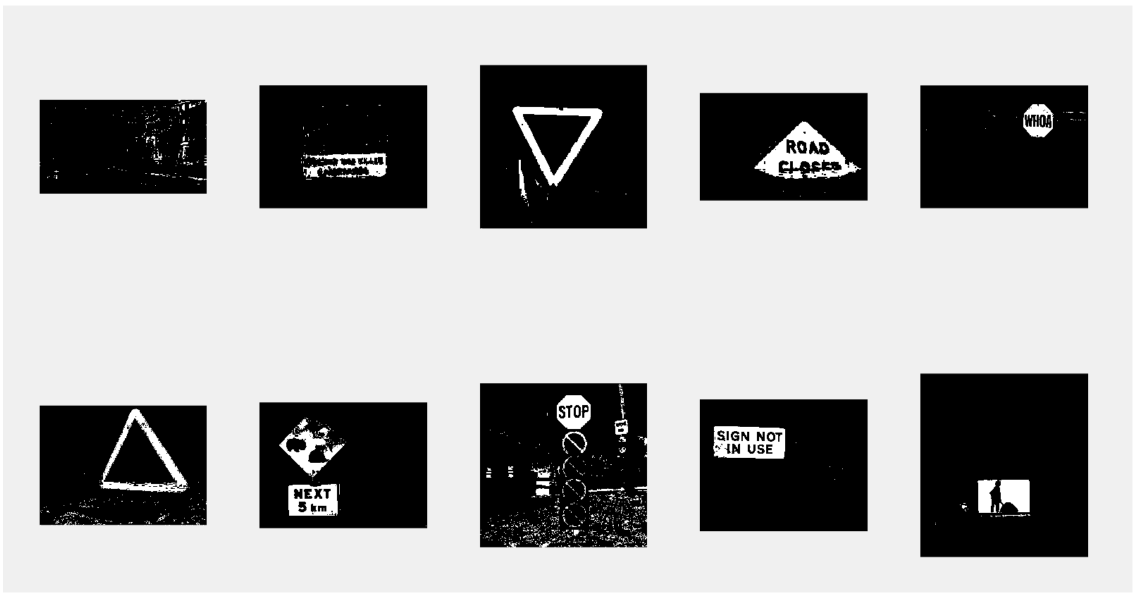


Figure 1.10: The result of the second dilatation.

Then we should do the filtering,

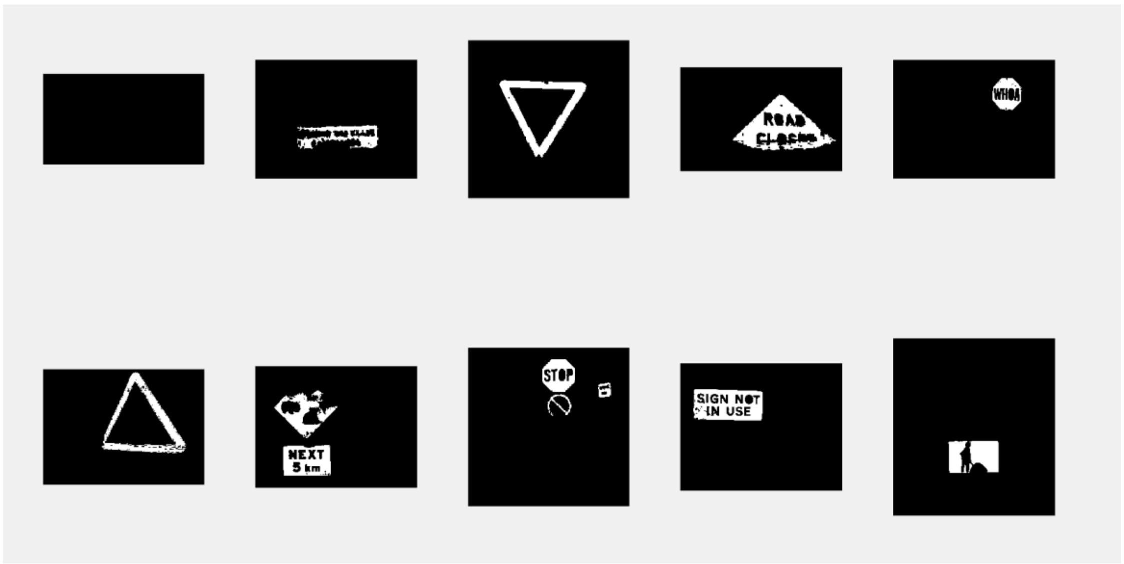


Figure 1.11: The result of the noise filtering.

In Figure 1.11, we find the remaining parts are all the signs we want to detect. We then fill the holes inside the images. The filled images are all shown in Figure 1.12.

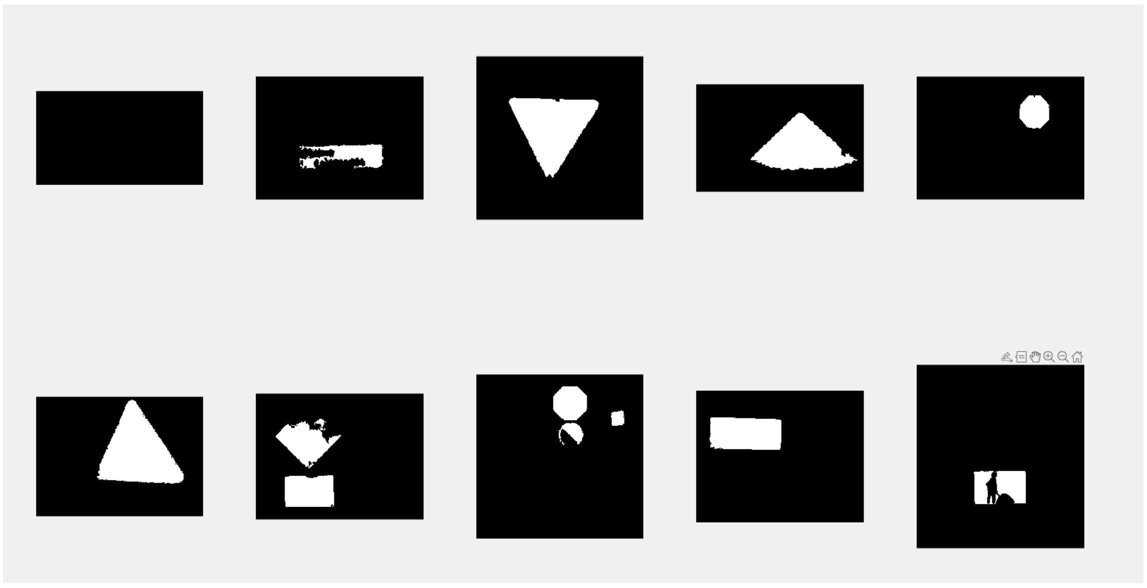


Figure 1.12: The filled images.

After this step , we find most of the signs we want to detect are all kept. But we still need to do a square size filter, that is for the image10, if we do not do this step, the final result for the image10 is shown in Figure 1.13.

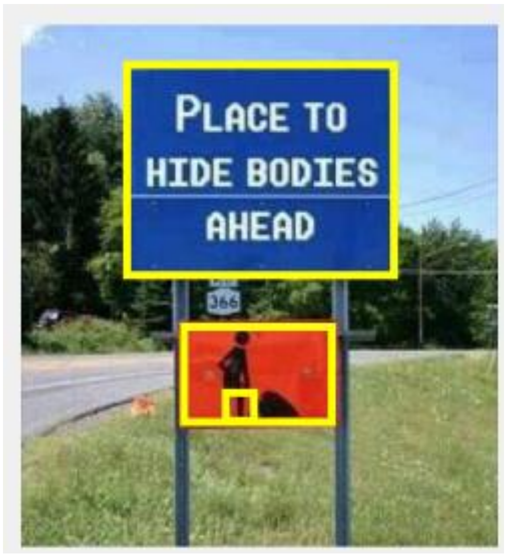


Figure 1.13: There is another small rectangle for the last image.

After one more square size filter, the result is shown in Figure 1.14.

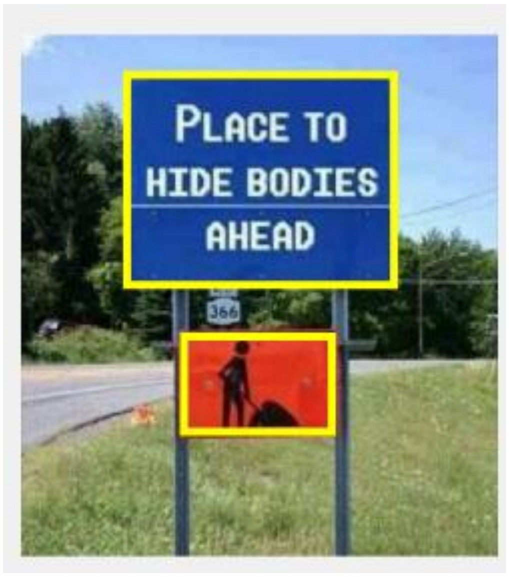


Figure 1.14: After we perform one more square size filter.

Now we finished the detection of the red street signs, then we start to do the analysis to the blue signs. After we doing the thresholding, the result should be like in Figure 1.15.

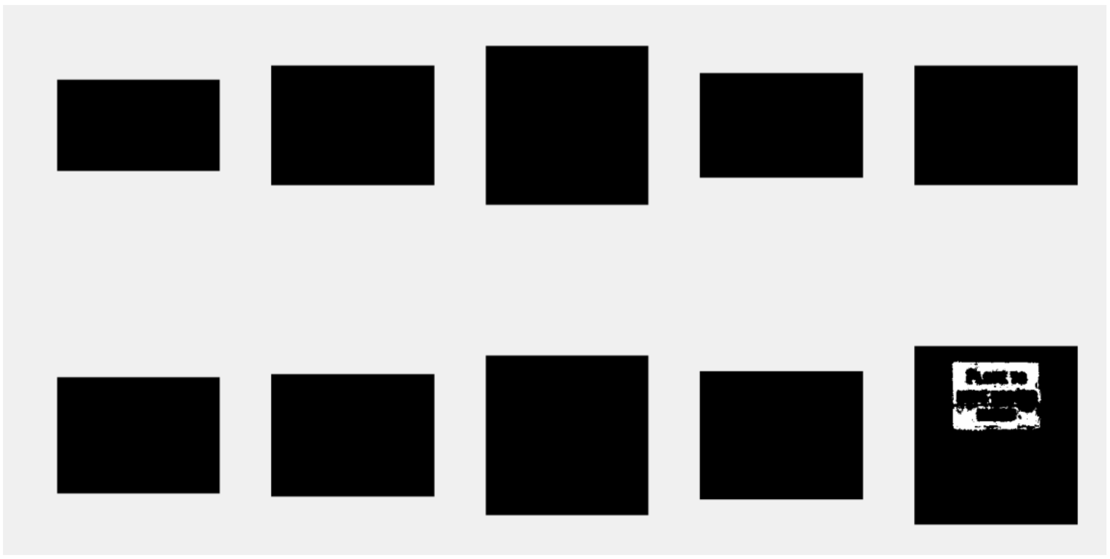


Figure 1.15: The image after the blue filtering.

We can find that for blue signs, there is only one sign in image10 among all the images. But we also find that there is a white line cross the single blue sign and divide the blue sign into two parts, so for detecting the blue sign better, we should try to “reconnect” the two parts, here we can use the dilation process, Figure 1.16 illustrates the result after the dilation process.

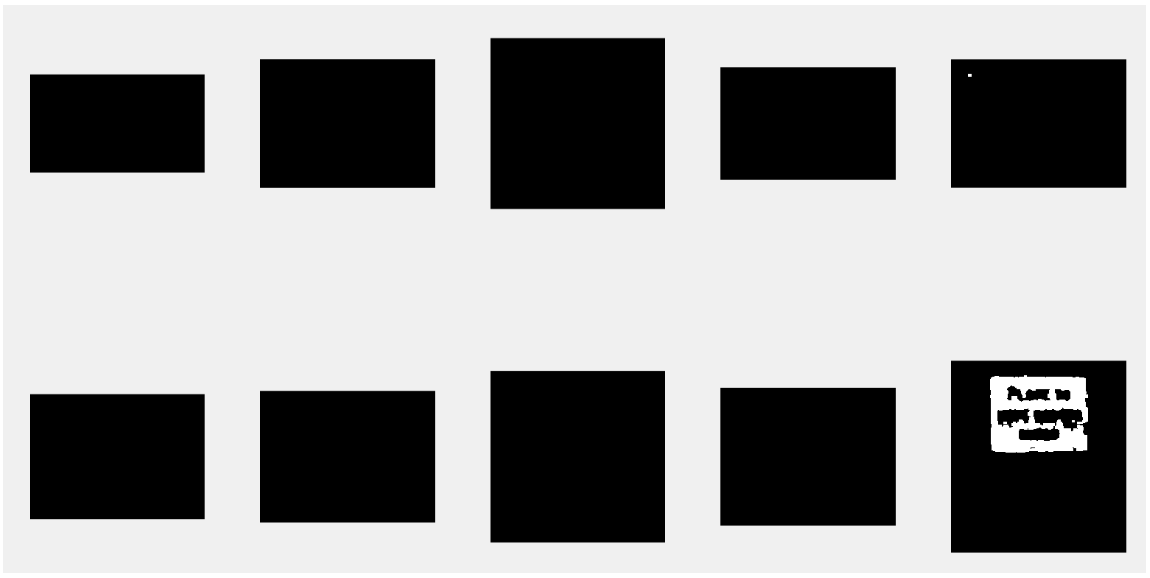


Figure 1.16: After doing the dilation process.

We can find that the two parts are connected by this dilation process, however, if we take a look of the image5, we find that there is a little white dot appears. We firstly fill the holes of the image10, and then use a size filter to filter the noise. Figure 1.17 would show the step of fill the holes, and Figure 1.18 would show the result of size filtering.

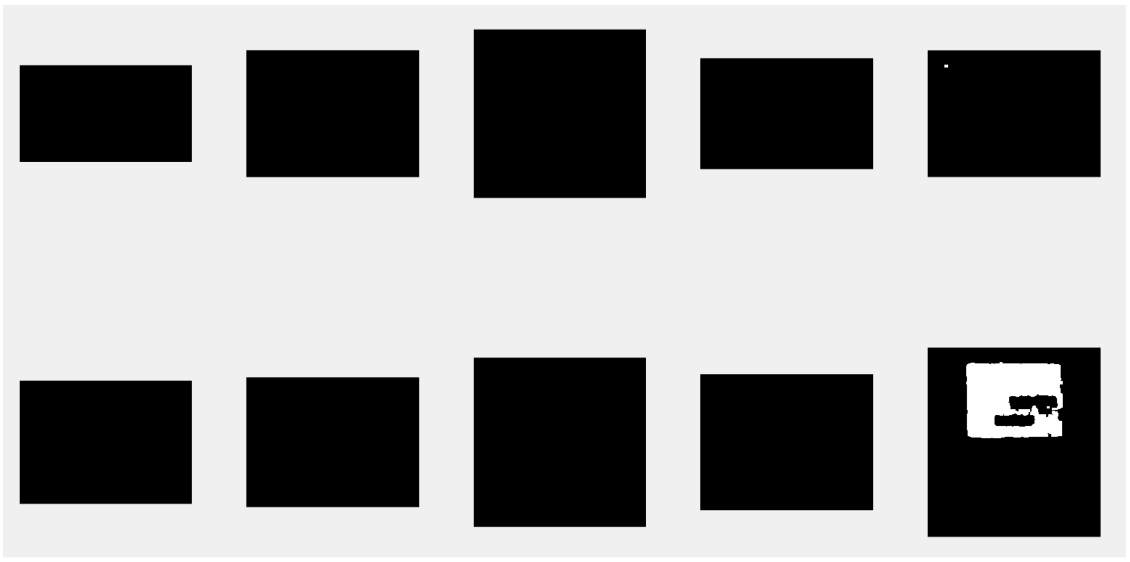


Figure 1.17: The step after we fill the holes.

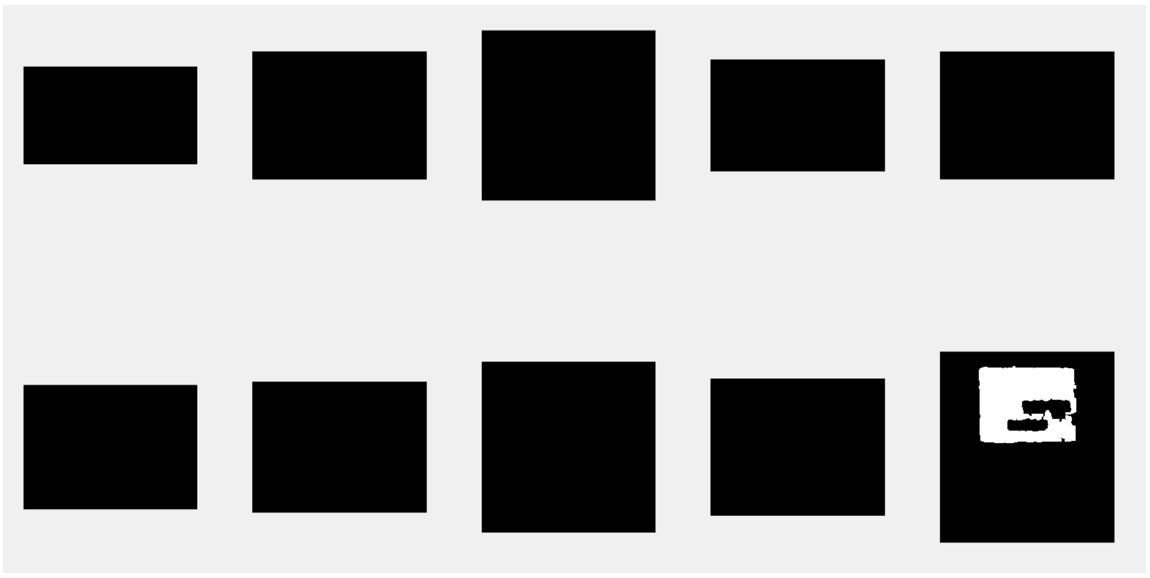


Figure 1.18: After we filtering the small size noise.

Here, we successfully removed the noise and reconnected the two parts of sign in image10.

Then we start to do the analysis for yellow signs. To detect the yellow signs, we should use the RGB threshold value to filter the yellow street signs. The results are shown in Figure 1.19.

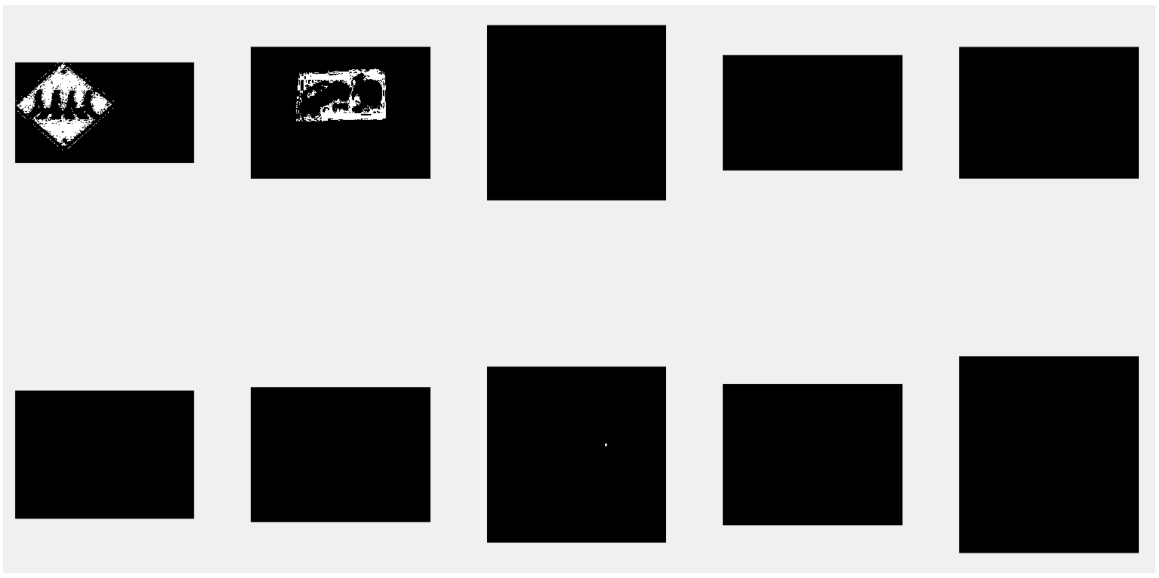


Figure 1.19: The images after yellow filtering.

After we the yellow threshold filtering, we firstly should do a dilation process, this can help us make the object we want to detect become more larger, the size of the structure element we using here is a square of 10x10, Figure 1.20 would show the images after the dilation process.

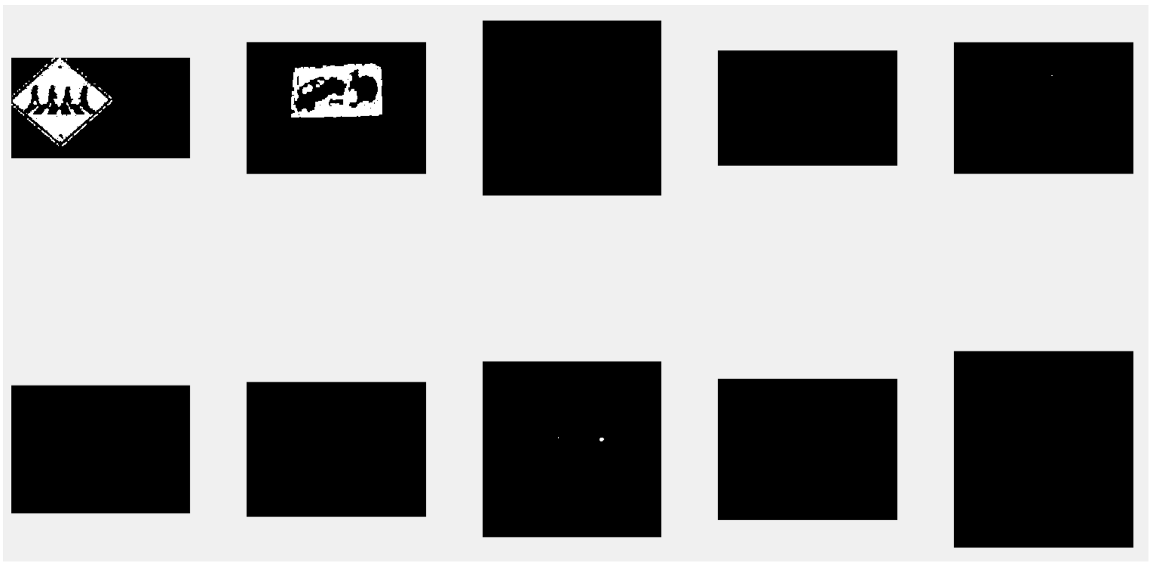


Figure 1.20: Yellow signs after the first dilation process.

After the first dilation process, we then do the erode process, this step can help us remove the noises, the results are shown in Figure 1.21.

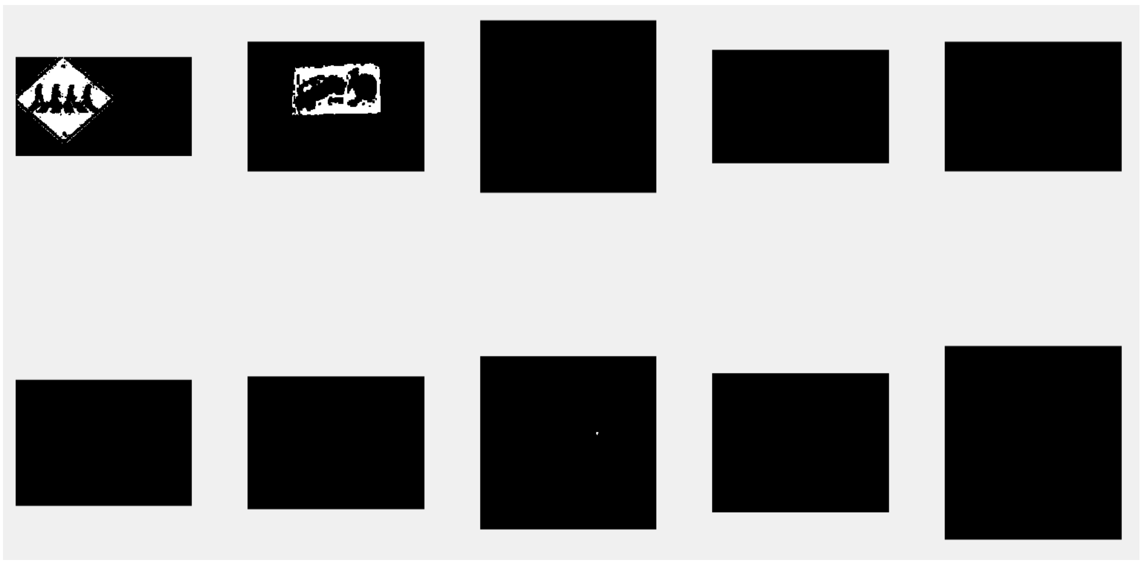


Figure 1.21: After we doing the erode process.

We find that some noise are removed from the image1 and image2, then we need another dilation process to recover the eroded images. The results are shown in image Figure 1.22.

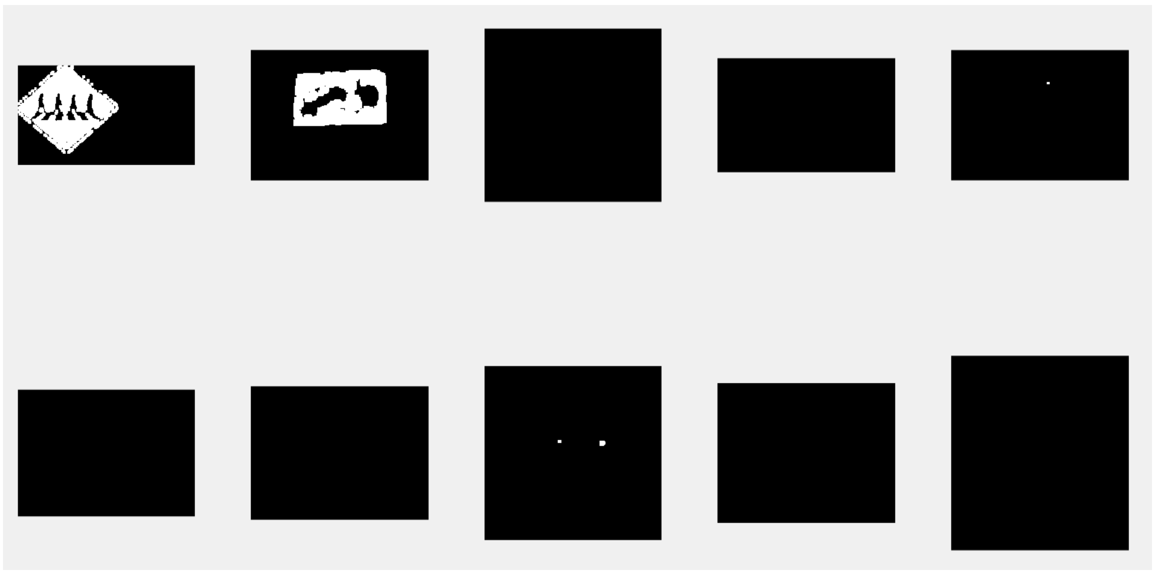


Figure 1.22: The result of second dilation.

Then, we fill the holes in the image1 and image2, this can make the target object as a whole. The result is shown in Figure 1.23.

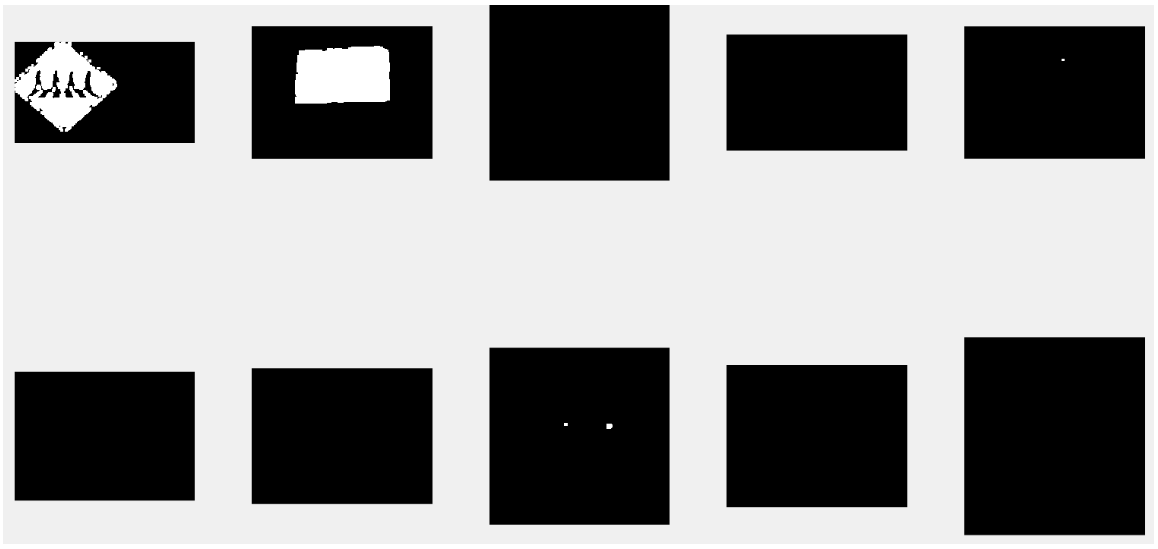


Figure 1.23: The image of filling the holes.

After this step, the detected images in image1 and image2 are a whole now, however, there are still some noises in other images, so we need a size filter to filter these noises. The result of the size filter shown in Figure 1.24.

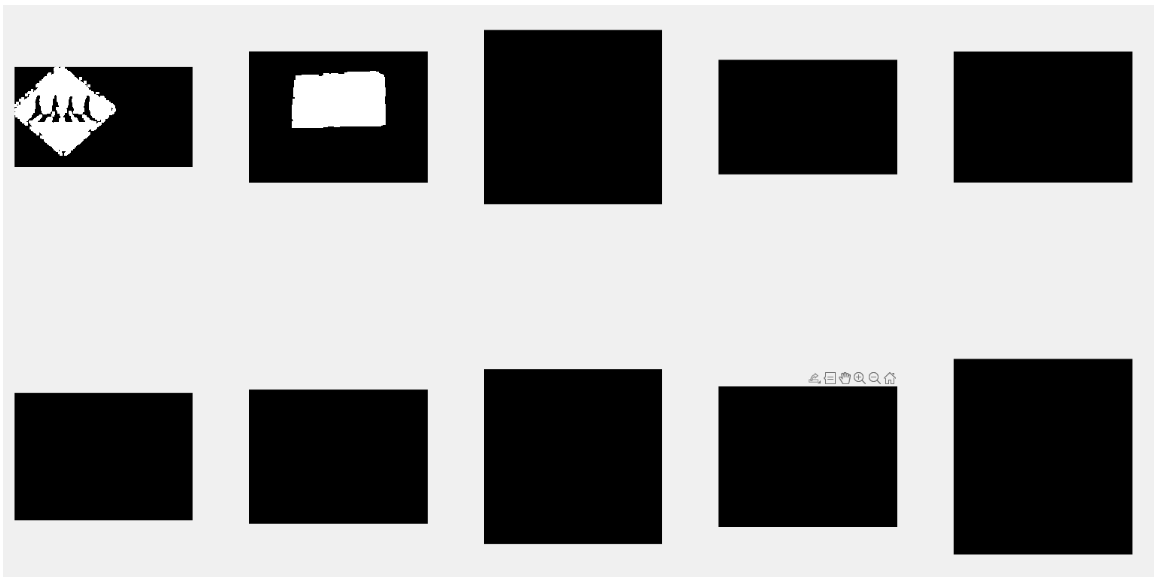


Figure 1.24: After the size filter.

Then we finished the steps for detecting all the street signs and transforming them into binary forms. It is shown in Figure 1.25.

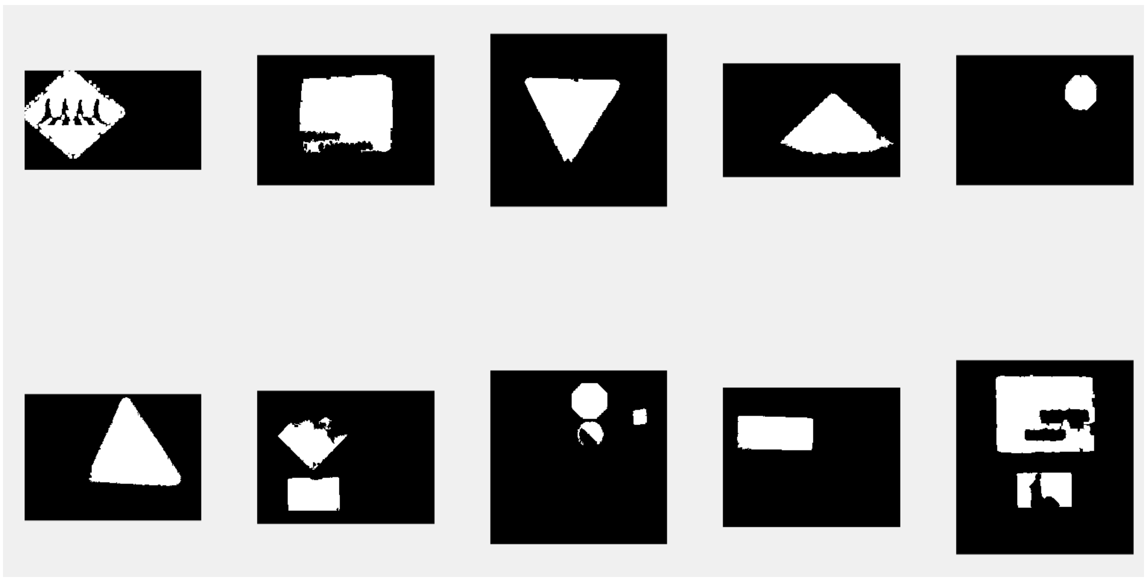


Figure 1.25: The final results of the detected images.

The last step is to draw rectangles for the corresponding original images. every rectangle would covers one street sign. The results are shown in Figure 1.26.

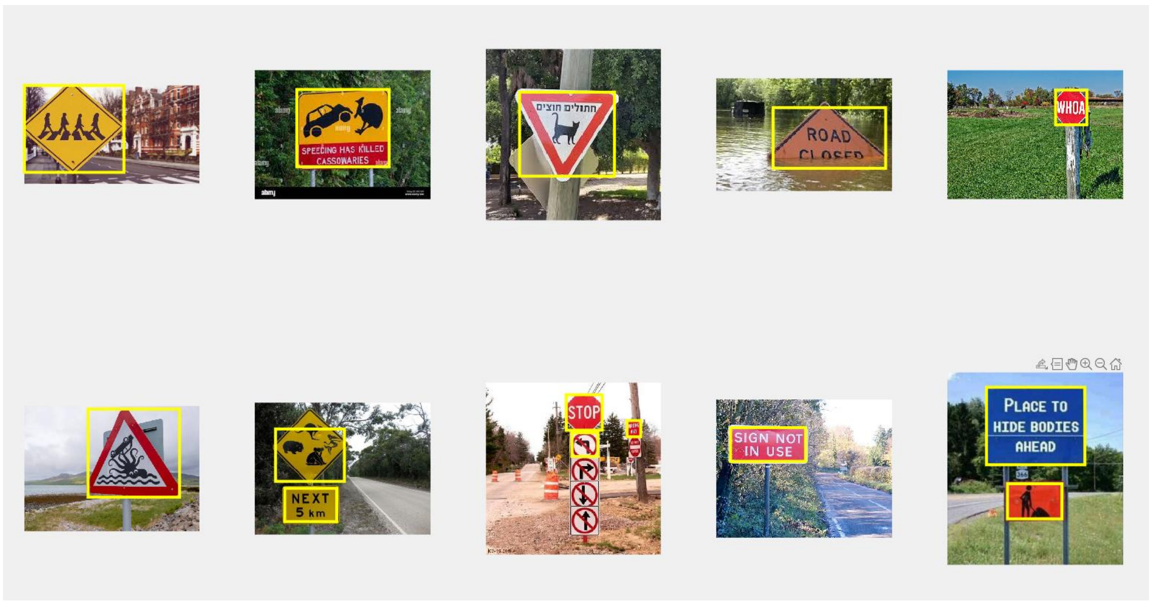
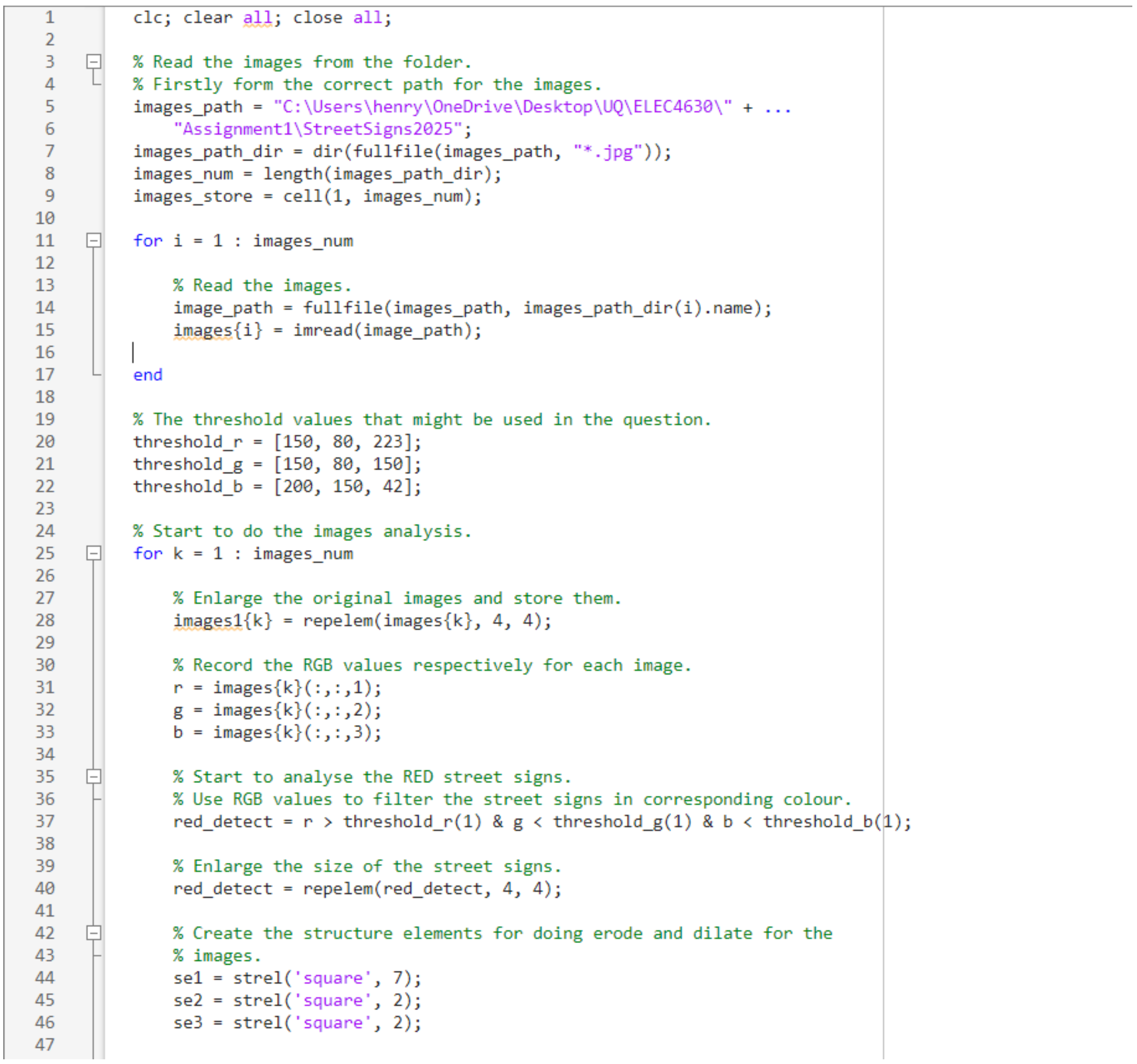


Figure 1.26: The final results of detecting the images.

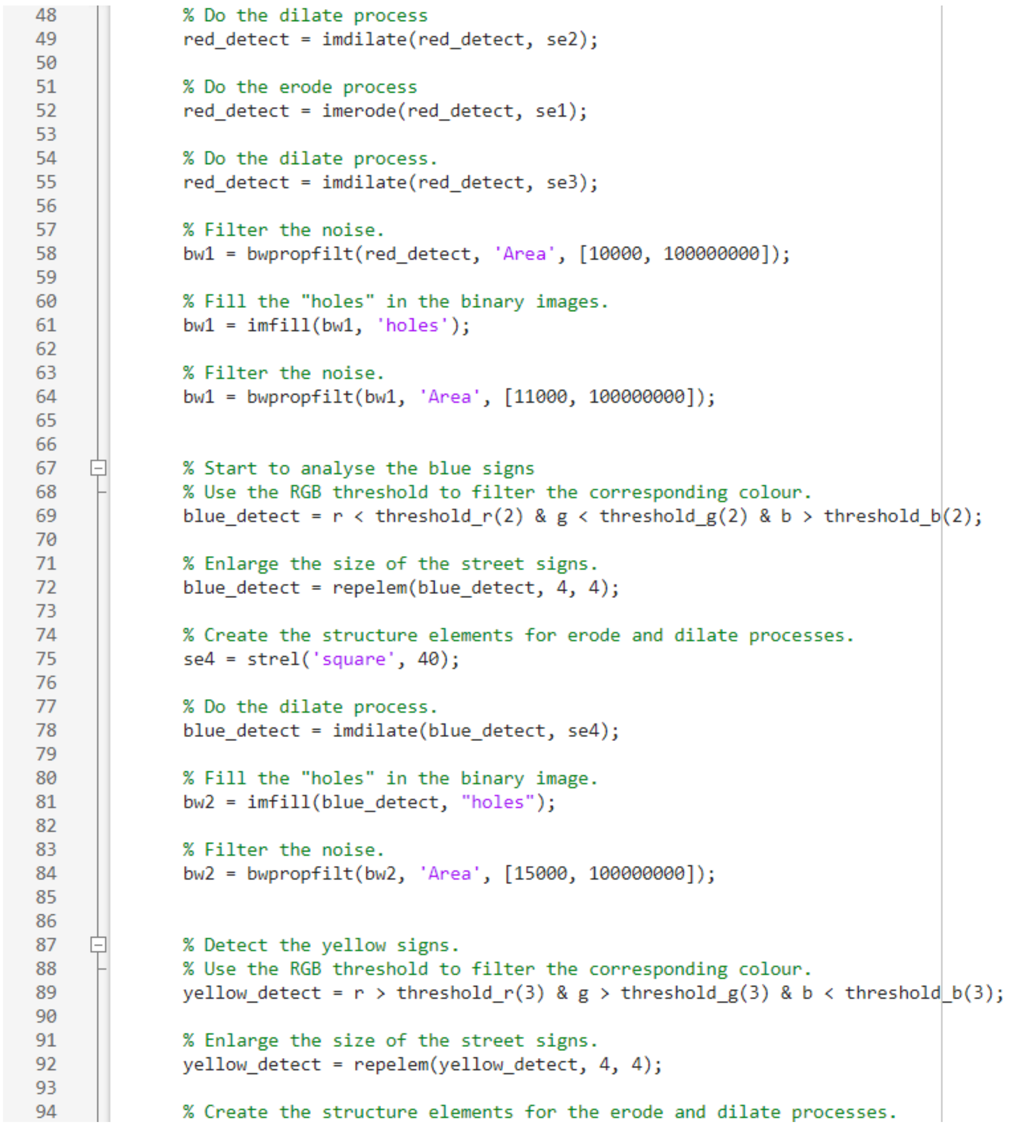
Here is the code for this question, there are two versions, the first version is the simple version, it finally would give you the results as it shown in Figure 1.26, the second version would plot every figure we plot before exclude Figure 1.13 and Figure 1.14.

The code part are too long so I took the whole screenshot for the code.



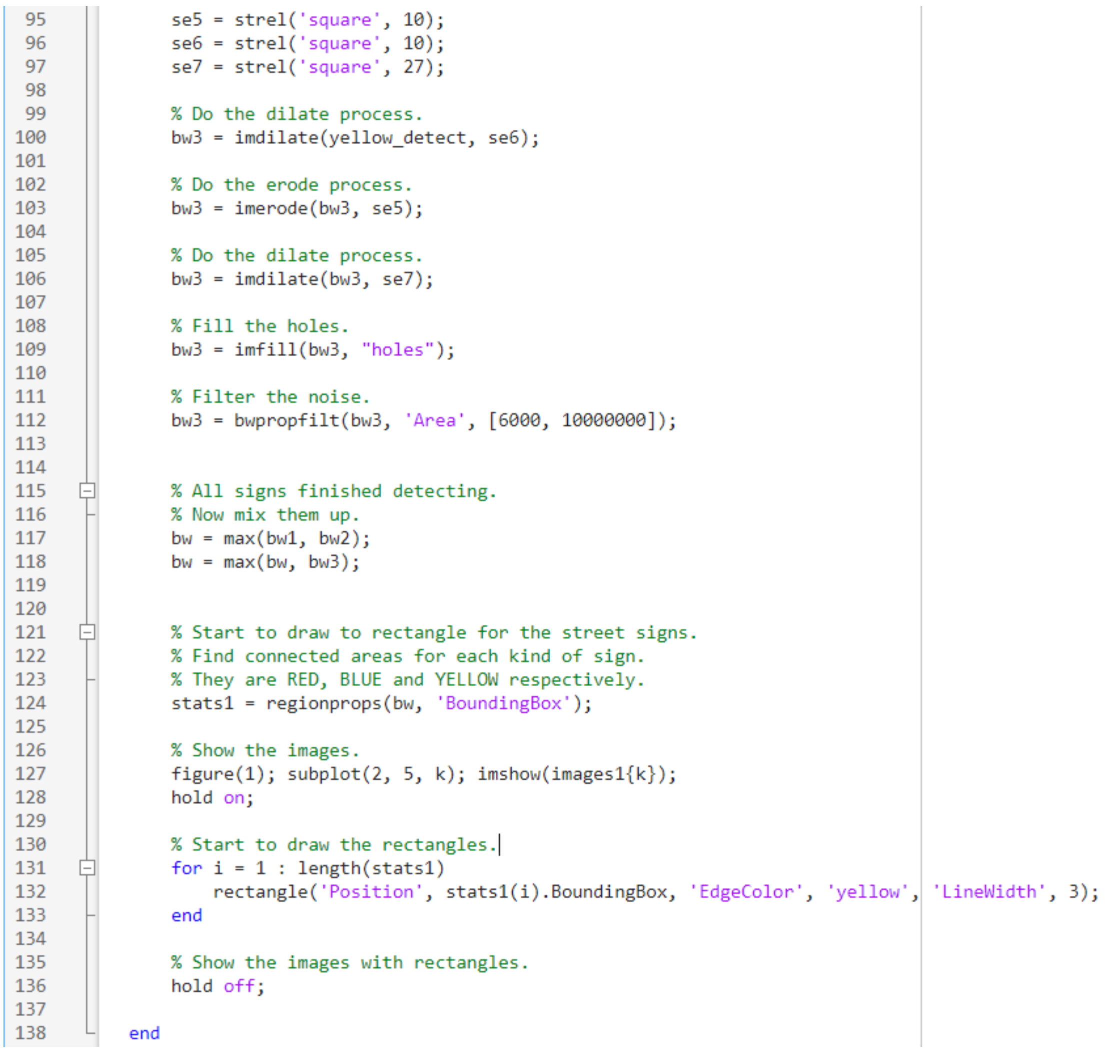
```
clc; clear all; close all;

% Read the images from the folder.
% Firstly form the correct path for the images.
images_path = "C:\Users\henry\OneDrive\Desktop\UQ\ELEC4630\" + ...
    "Assignment1\StreetSigns2025";
images_path_dir = dir(fullfile(images_path, "*.jpg"));
images_num = length(images_path_dir);
images_store = cell(1, images_num);

for i = 1 : images_num

    % Read the images.
    image_path = fullfile(images_path, images_path_dir(i).name);
    images{i} = imread(image_path);

end

% The threshold values that might be used in the question.
threshold_r = [150, 80, 223];
threshold_g = [150, 80, 150];
threshold_b = [200, 150, 42];

% Start to do the images analysis.
for k = 1 : images_num

    % Enlarge the original images and store them.
    images1{k} = repelem(images{k}, 4, 4);

    % Record the RGB values respectively for each image.
    r = images{k}(:,:,1);
    g = images{k}(:,:,2);
    b = images{k}(:,:,3);

    % Start to analyse the RED street signs.
    % Use RGB values to filter the street signs in corresponding colour.
    red_detect = r > threshold_r(1) & g < threshold_g(1) & b < threshold_b(1);

    % Enlarge the size of the street signs.
    red_detect = repelem(red_detect, 4, 4);

    % Create the structure elements for doing erode and dilate for the
    % images.
    se1 = strel('square', 7);
    se2 = strel('square', 2);
    se3 = strel('square', 2);

```

Figure 1.27: The code part 1.



```
    % Do the dilate process
    red_detect = imdilate(red_detect, se2);

    % Do the erode process
    red_detect = imerode(red_detect, se1);

    % Do the dilate process.
    red_detect = imdilate(red_detect, se3);

    % Filter the noise.
    bw1 = bwpropfilt(red_detect, 'Area', [10000, 100000000]);

    % Fill the "holes" in the binary images.
    bw1 = imfill(bw1, 'holes');

    % Filter the noise.
    bw1 = bwpropfilt(bw1, 'Area', [11000, 100000000]);


    % Start to analyse the blue signs
    % Use the RGB threshold to filter the corresponding colour.
    blue_detect = r < threshold_r(2) & g < threshold_g(2) & b > threshold_b(2);

    % Enlarge the size of the street signs.
    blue_detect = repelem(blue_detect, 4, 4);

    % Create the structure elements for erode and dilate processes.
    se4 = strel('square', 40);

    % Do the dilate process.
    blue_detect = imdilate(blue_detect, se4);

    % Fill the "holes" in the binary image.
    bw2 = imfill(blue_detect, "holes");

    % Filter the noise.
    bw2 = bwpropfilt(bw2, 'Area', [15000, 100000000]);


    % Detect the yellow signs.
    % Use the RGB threshold to filter the corresponding colour.
    yellow_detect = r > threshold_r(3) & g > threshold_g(3) & b < threshold_b(3);

    % Enlarge the size of the street signs.
    yellow_detect = repelem(yellow_detect, 4, 4);

    % Create the structure elements for the erode and dilate processes.
```

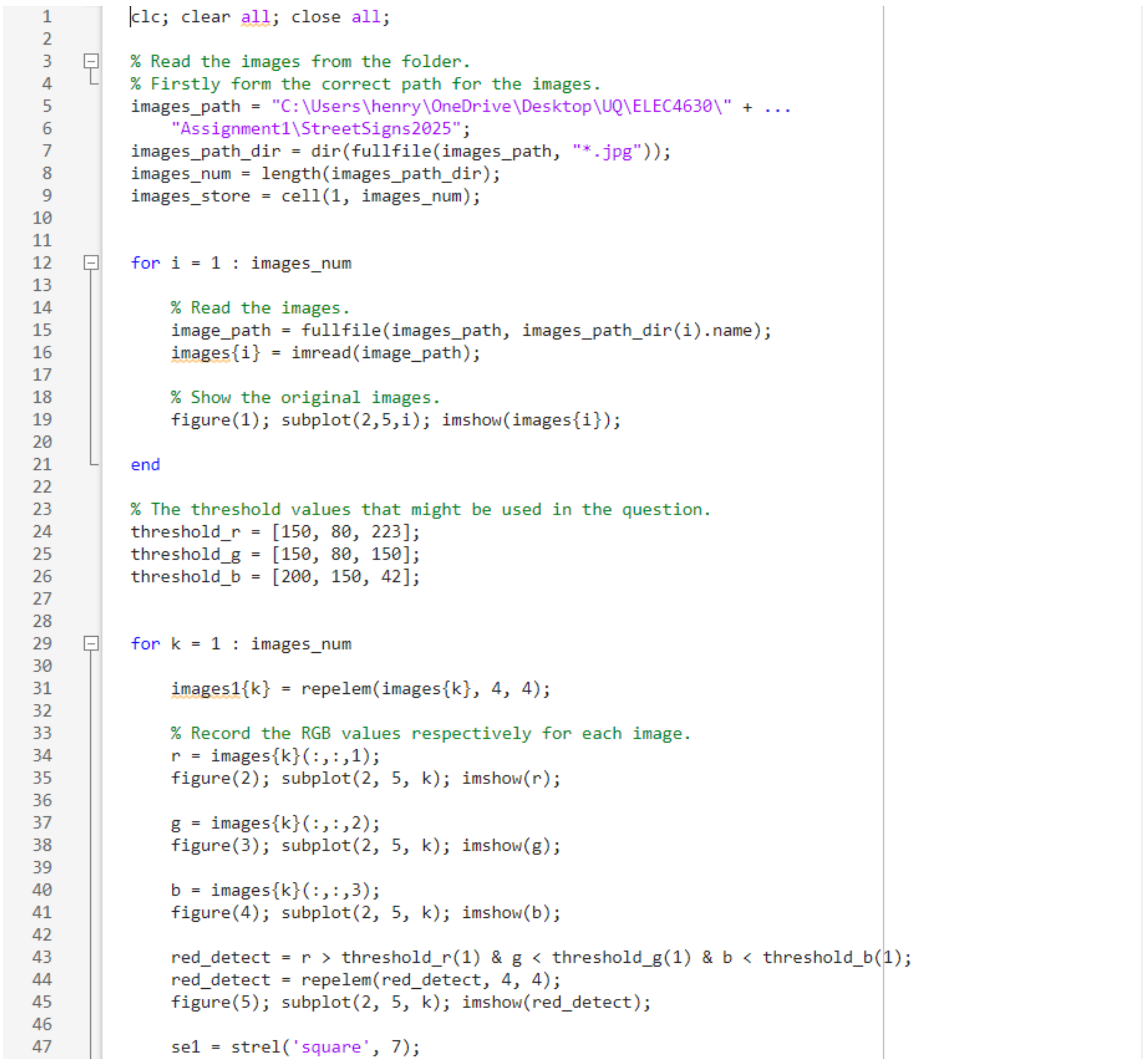
Figure 1.28: The code part 2.



```matlab
    se5 = strel('square', 10);
    se6 = strel('square', 10);
    se7 = strel('square', 27);

    % Do the dilate process.
    bw3 = imdilate(yellow_detect, se6);

    % Do the erode process.
    bw3 = imerode(bw3, se5);

    % Do the dilate process.
    bw3 = imdilate(bw3, se7);

    % Fill the holes.
    bw3 = imfill(bw3, "holes");

    % Filter the noise.
    bw3 = bwpropfilt(bw3, 'Area', [6000, 10000000]);


    % All signs finished detecting.
    % Now mix them up.
    bw = max(bw1, bw2);
    bw = max(bw, bw3);


    % Start to draw to rectangle for the street signs.
    % Find connected areas for each kind of sign.
    % They are RED, BLUE and YELLOW respectively.
    stats1 = regionprops(bw, 'BoundingBox');

    % Show the images.
    figure(1); subplot(2, 5, k); imshow(images1{k});
    hold on;

    % Start to draw the rectangles.
    for i = 1 : length(stats1)
        rectangle('Position', stats1(i).BoundingBox, 'EdgeColor', 'yellow', 'LineWidth', 3);
    end

    % Show the images with rectangles.
    hold off;

end
```

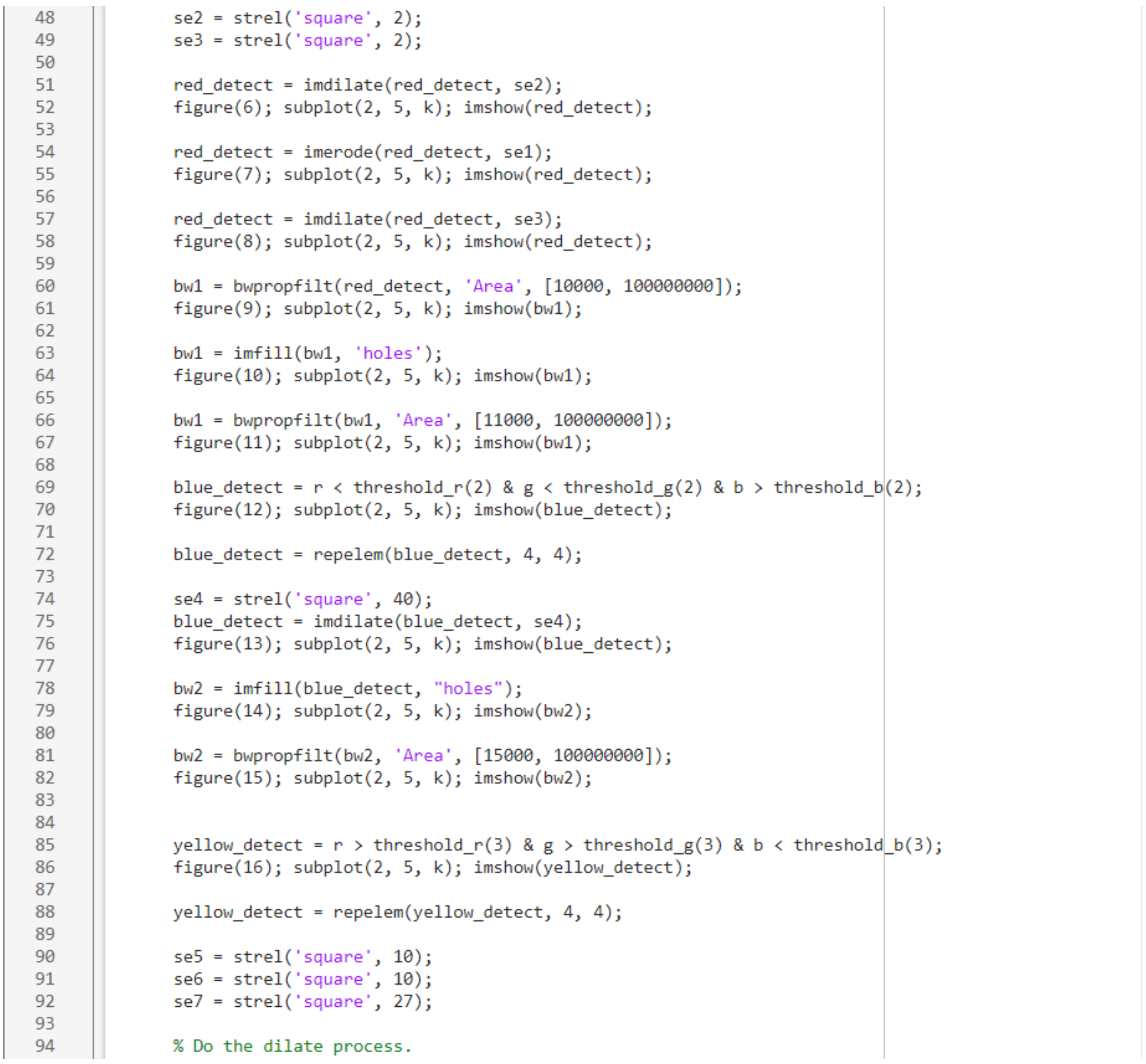
Figure 1.29: The code part 3.

These 3 screenshots illustrate the simple version of my work, it would only give you a final result. The second version is the same but it contains more figures, that is the only difference between these two versions.



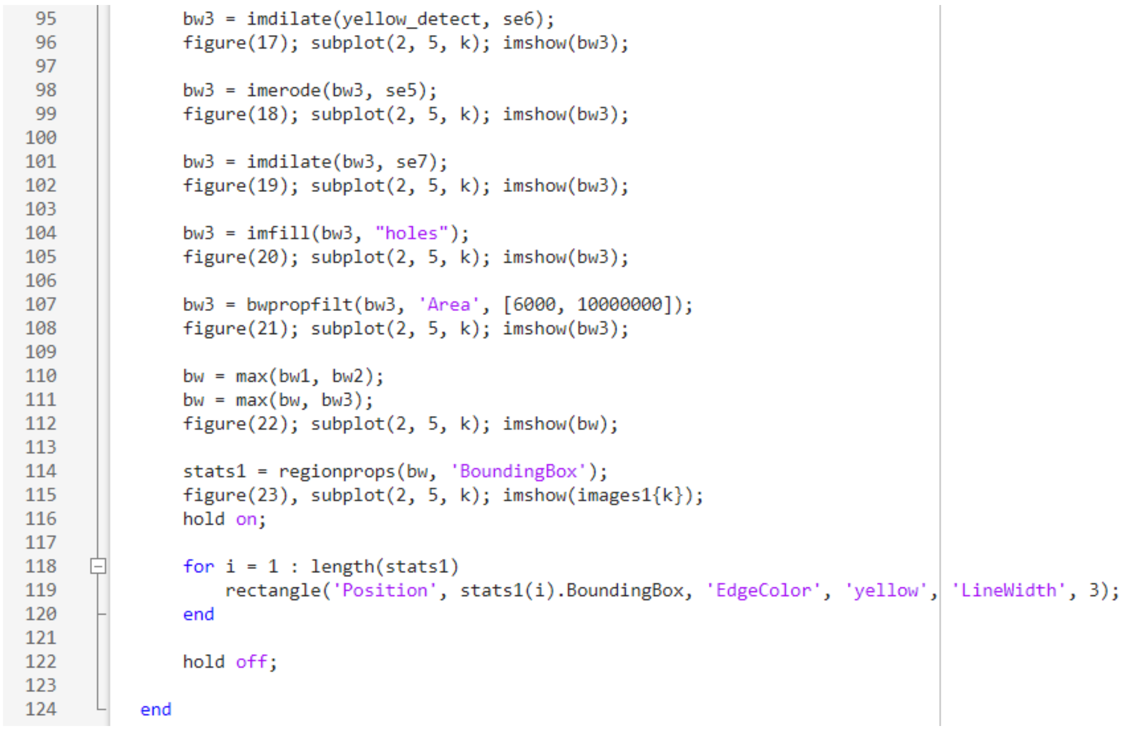
```
clc; clear all; close all;

% Read the images from the folder.
% Firstly form the correct path for the images.
images_path = "C:\Users\henry\OneDrive\Desktop\UQ\ELEC4630\" + ...
    "Assignment1\StreetSigns2025";
images_path_dir = dir(fullfile(images_path, "*.jpg"));
images_num = length(images_path_dir);
images_store = cell(1, images_num);


for i = 1 : images_num

    % Read the images.
    image_path = fullfile(images_path, images_path_dir(i).name);
    images{i} = imread(image_path);

    % Show the original images.
    figure(1); subplot(2,5,i); imshow(images{i});

end

% The threshold values that might be used in the question.
threshold_r = [150, 80, 223];
threshold_g = [150, 80, 150];
threshold_b = [200, 150, 42];


for k = 1 : images_num

    images1{k} = repelem(images{k}, 4, 4);

    % Record the RGB values respectively for each image.
    r = images{k}(:,:,1);
    figure(2); subplot(2, 5, k); imshow(r);

    g = images{k}(:,:,2);
    figure(3); subplot(2, 5, k); imshow(g);

    b = images{k}(:,:,3);
    figure(4); subplot(2, 5, k); imshow(b);

    red_detect = r > threshold_r(1) & g < threshold_g(1) & b < threshold_b(1);
    red_detect = repelem(red_detect, 4, 4);
    figure(5); subplot(2, 5, k); imshow(red_detect);

    se1 = strel('square', 7);
```

Figure 1.30: The figure version code part 1.



```
        se2 = strel('square', 2);
        se3 = strel('square', 2);

        red_detect = imdilate(red_detect, se2);
        figure(6); subplot(2, 5, k); imshow(red_detect);

        red_detect = imerode(red_detect, se1);
        figure(7); subplot(2, 5, k); imshow(red_detect);

        red_detect = imdilate(red_detect, se3);
        figure(8); subplot(2, 5, k); imshow(red_detect);

        bw1 = bwpropfilt(red_detect, 'Area', [10000, 100000000]);
        figure(9); subplot(2, 5, k); imshow(bw1);

        bw1 = imfill(bw1, 'holes');
        figure(10); subplot(2, 5, k); imshow(bw1);

        bw1 = bwpropfilt(bw1, 'Area', [11000, 100000000]);
        figure(11); subplot(2, 5, k); imshow(bw1);

        blue_detect = r < threshold_r(2) & g < threshold_g(2) & b > threshold_b(2);
        figure(12); subplot(2, 5, k); imshow(blue_detect);

        blue_detect = repelem(blue_detect, 4, 4);

        se4 = strel('square', 40);
        blue_detect = imdilate(blue_detect, se4);
        figure(13); subplot(2, 5, k); imshow(blue_detect);

        bw2 = imfill(blue_detect, "holes");
        figure(14); subplot(2, 5, k); imshow(bw2);

        bw2 = bwpropfilt(bw2, 'Area', [15000, 100000000]);
        figure(15); subplot(2, 5, k); imshow(bw2);


        yellow_detect = r > threshold_r(3) & g > threshold_g(3) & b < threshold_b(3);
        figure(16); subplot(2, 5, k); imshow(yellow_detect);

        yellow_detect = repelem(yellow_detect, 4, 4);

        se5 = strel('square', 10);
        se6 = strel('square', 10);
        se7 = strel('square', 27);

        % Do the dilate process.
```

Figure 1.31: The figure version code part 2.



```
        bw3 = imdilate(yellow_detect, se6);
        figure(17); subplot(2, 5, k); imshow(bw3);

        bw3 = imerode(bw3, se5);
        figure(18); subplot(2, 5, k); imshow(bw3);

        bw3 = imdilate(bw3, se7);
        figure(19); subplot(2, 5, k); imshow(bw3);

        bw3 = imfill(bw3, "holes");
        figure(20); subplot(2, 5, k); imshow(bw3);

        bw3 = bwpropfilt(bw3, 'Area', [6000, 10000000]);
        figure(21); subplot(2, 5, k); imshow(bw3);

        bw = max(bw1, bw2);
        bw = max(bw, bw3);
        figure(22); subplot(2, 5, k); imshow(bw);

        stats1 = regionprops(bw, 'BoundingBox');
        figure(23), subplot(2, 5, k); imshow(images1{k});
        hold on;

        for i = 1 : length(stats1)
            rectangle('Position', stats1(i).BoundingBox, 'EdgeColor', 'yellow', 'LineWidth', 3);
        end

        hold off;

    end
```

Figure 1.32: The figure version code part 3.

- **Some revisions about this method:**
  To this question, the method I applied successfully detects most of the street signs. However, this method is highly related to the color of the street signs, if the quality of the photo taken was not good or the color of the environment is similar to the sign color, my method might not work. Currently, this method only works on these 10 images, to other street sign photos, this might not work. For example, like Figure 33.

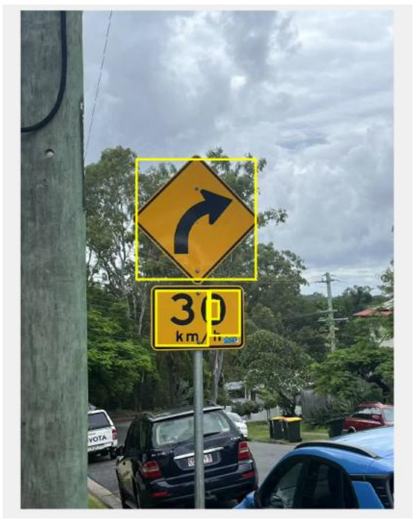


Figure 1.33: Real life test.

The street sign in Figure 1.33 is a photo I took near my home which located in Indooroopilly, and it was took by my iPhone 13, because the size of the image is too large for the parameters I set in the question code, so my code seems not work very well in this image, Figure 34 shows the binary image for this real life street sign.

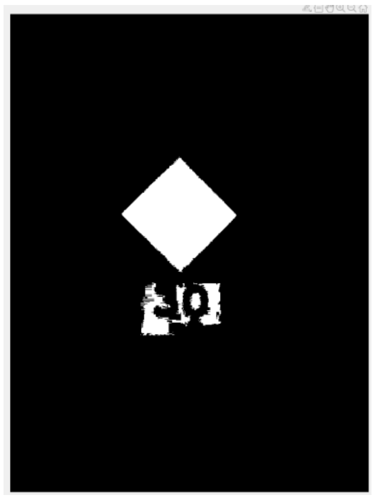


Figure 1.34: Binary image for the real-life street sign.

We see the square sign is considered as a whole but the rectangle sign is divided into 3 parts, so in the final result, the square sign successfully detected but the rectangle failed.

At the same time, this method is not working on some small signs in image8, this is because these signs are compared small or the color are close to the environment color, so these colors are considered as noises and removed by the erode function.

I think the main problem of my solution is that it highly related to different conditions of the street signs photo, we take the example in Figure 33, if we observe that we will find there is a small sticker on it, the color of the sticker is quite different from the street sign, so my solution does not recognize it as a part of the street sign, then the rectangle street sign is recognized as 3 parts, so my method requires the street signs' photos have not been polluted.

Question 2: Power Cable & Pantograph Analysis.

The target for this question is to analyze a video and mark out the area where the pantograph and power cable touched. The frames in video are like in Figure 2.1.

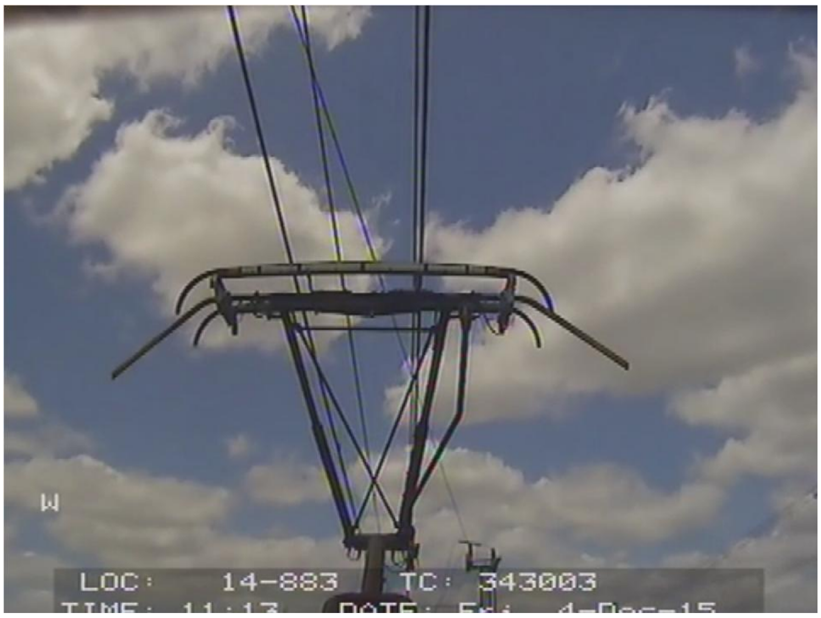


Figure 2.1: The pantograph of the frame.

For this question, the main techniques I will apply are Hough Transform and Template matching, Hough Transform I will introduce in this next question. In this question, I will introduce the Template matching and Hough Transform I will introduce in the next question.

- **TEMPLATE MATCHING.**

For template matching, we have the formula to calculate it:

$$E[i,j] = \sum_{m}\sum_{n}(f[m,n] - t[m-i,n-j])^2 \qquad (2-1)$$

Our goal is to minimize the result of the formula. Where in this formula, the parameter $t[m-i,n-j]$ is the template and the $f[m,n]$ is the image. We can expand the formula like the formula (2-2).

$$E[i,j] = \sum_{m}\sum_{n}\left[f^2[m,n] + t^2[m-i,n-j] - 2f[m,n]t[m-i,n-j]\right] \qquad (2-2)$$

So, our problem becomes how to maximize the formula in (2-3).

$$R_{tf}[i,j] = \sum_m \sum_n f[m,n]t[m-i,n-j] \qquad (2-3)$$

We call it cross-correlation. In MATLAB, we can use the function normxcorr2() to do the template matching. In my solution, to detect the pantograph, I can take the template of the pantograph, the pantograph appears on every frame, so, just take the pantograph template from a random frame is fine. After this step, we can find the location of pantograph for every frame.

- **SOLUTION TO THE QUESTION.**

Firstly, we read the video from the folder, the frames are just like that in Figure 2.1. We notice that the color for both power cable and the pantograph are both black, so, we can use the RGB threshold value to filter the frames. The result of the RGB filter is shown in Figure 2.2.

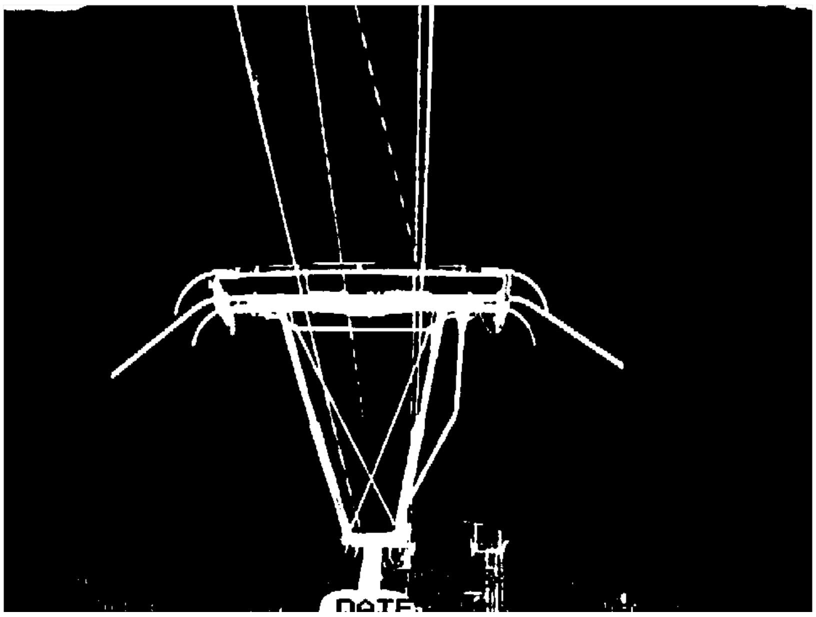


Figure 2.2: The RGB filtered frame of the video.

We observe the frame shows in Figure 2.2, the pantograph and the power cable are very obvious, for most of the frames, the two items we need are all like this.

The next step for us is to extract the template of the pantograph, the pantograph template I selected is the pantograph in the 370th frame, for much more accurate detection of the location of the pantograph, I choose to extract the template from the RGB filtered binary frame. The template I extracted is shown in Figure 2.3.

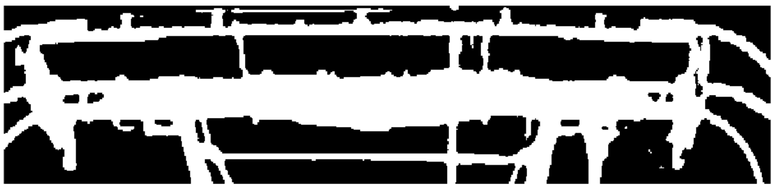


Figure 2.3: The template of the pantograph.

This step can help us find the location of the pantograph, the result of the detection is shown in Figure 2.4.

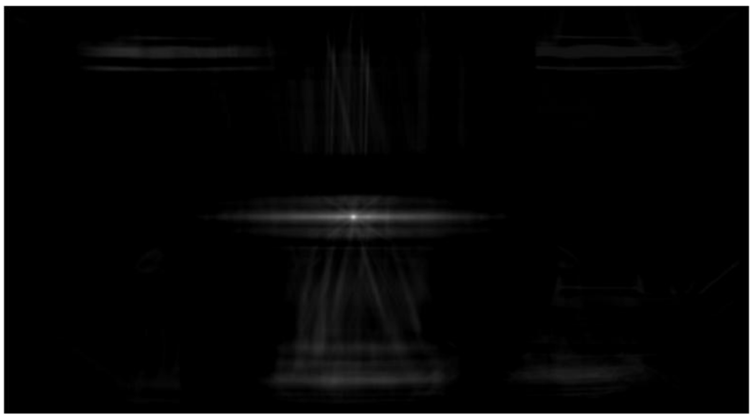


Figure 2.4: The result of the template matching.

From the Figure 2.4, we find the performance of the center place is much more obvious than the other places, that is the place that the computer think the most suitable place where the pantograph located. We can get the location data from the cross-correlation calculation. We can use the result to find the pantograph in the original frame image. The result of the original frame detection shown in Figure 2.5.

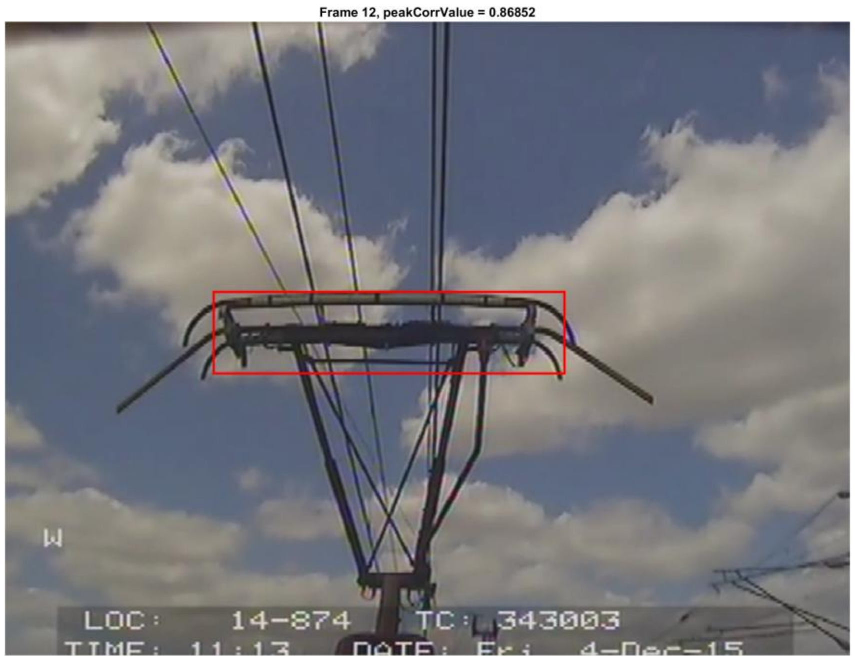

Frame 12, peakCorrValue = 0.86852
LOC: 14-874 TC: 343003
TIME: 11:13 DATE: Fri 4-Dec-15

Figure 2.5: Find the pantograph in the original frame.

We can find in Figure 2.5 that the location of the rectangle fits the pantograph in the original frame very well. The next step for us is to find the power cable and do the Hough Transform for the cable.

We notice that the power cable appears in the upper place of the pantograph, so, when we doing this part, we can just focus on the upper right part. The place I am about to do the Hough Transform Detection is shown in Figure 2.6.

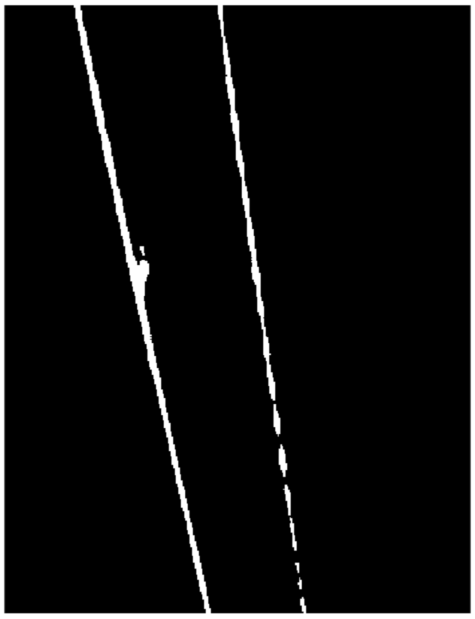


Figure 2.6: The area we are about to do the power cable detection.

The area data of the cable detection is [374.5, 2.5, 347, 453]. Also, we find in this area, sometimes there are 2 or more cables appear in this area, but normally, the power cable is the largest in this area, so we need to use the function bwareafilt() to find the largest cable appears in the image. The result is shown in Figure 2.7.

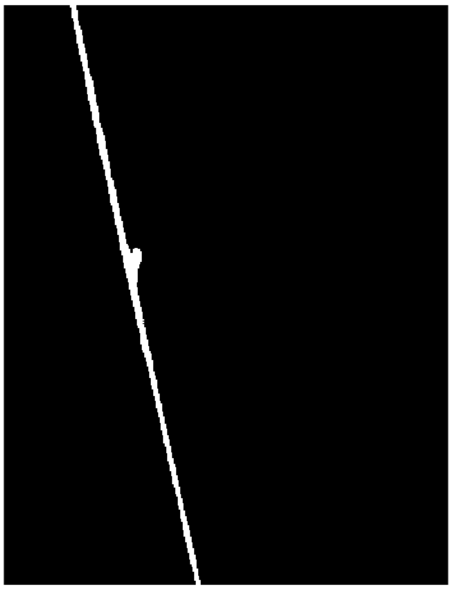


Figure 2.7: The power cable detection.

So, for our solution, we can use Hough Transform to detect the straight cable. The result of the pantograph detection and power cable detection are shown in the Figure 2.8.

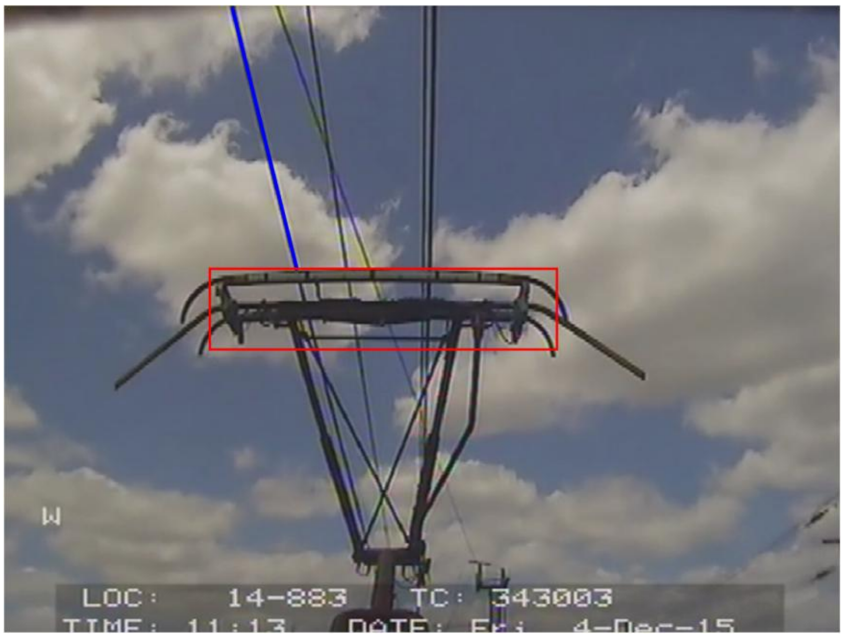


Figure 2.8: The detection of the pantograph and power cable.

Here, our task is to mark the cross point of the blue straight line of power cable and the red rectangle contains pantograph, that point is where the power cable touches the pantograph. However, in some frames, like the Figure 2.9, the Hough Transform detected line does not touch the red rectangle.

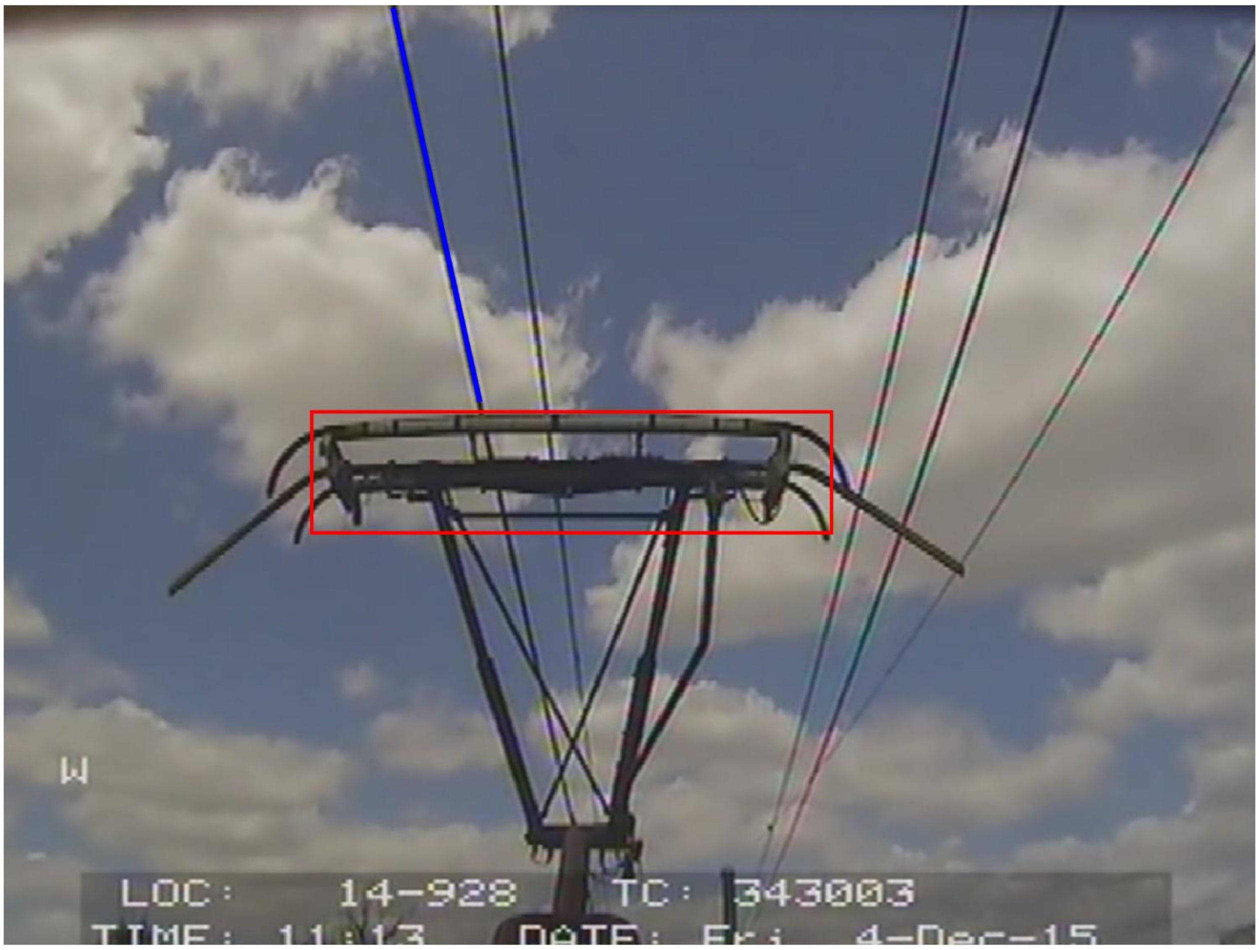


Figure 2.9: The frame where the Hough Transform detected line does not touch the pantograph.

I know the Figure 2.9 is not very obviously, so I make the image larger, to prevent this situation, we should extend the blue line, it should be like Figure 2.10.

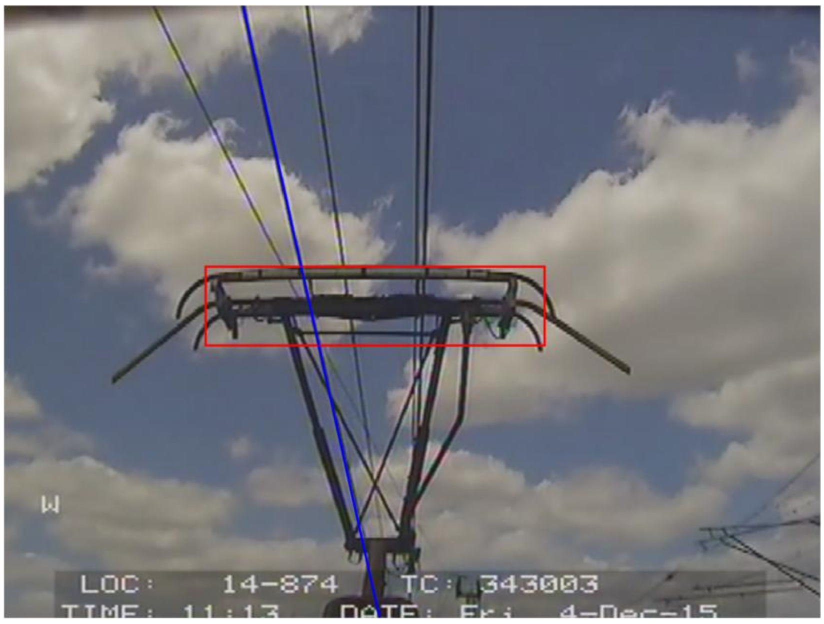


Figure 2.10: Extend the blue line of Hough Transform power cable.

After this step, we can make sure that the power cable straight line can have a cross point with the upper border of the rectangle. We can mark the touch point for the upper border of the red rectangle, the result of marking the touch point is shown in Figure 2.11.

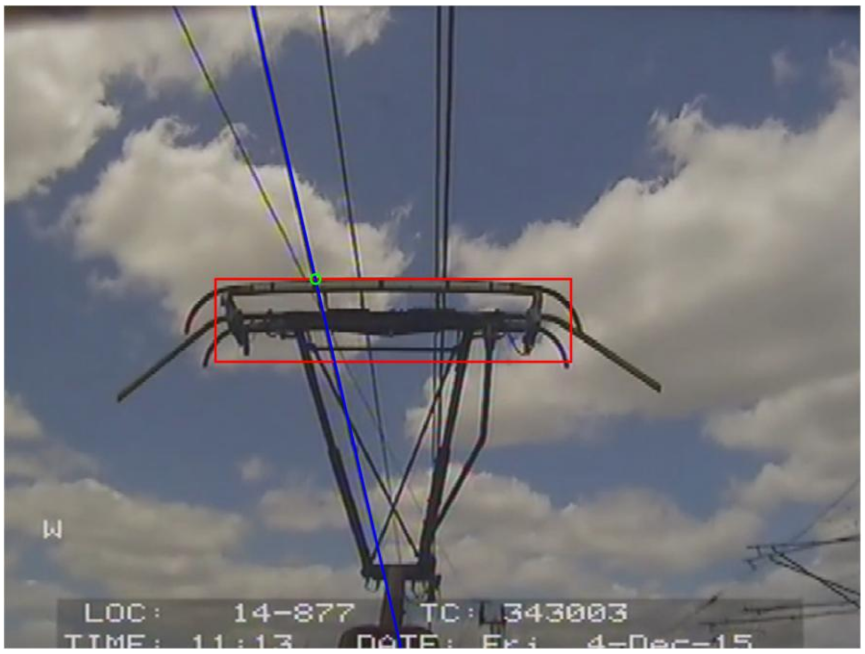


Figure 2.11: The touch point marked in the original image.

Now, the target of the question to detect the touch point for the pantograph is basically finished. The upper border of the red rectangle can be recognized as the pantograph. For a better vision effect, we can just plot the upper border of the red rectangle, just like it shows in Figure 2.12.

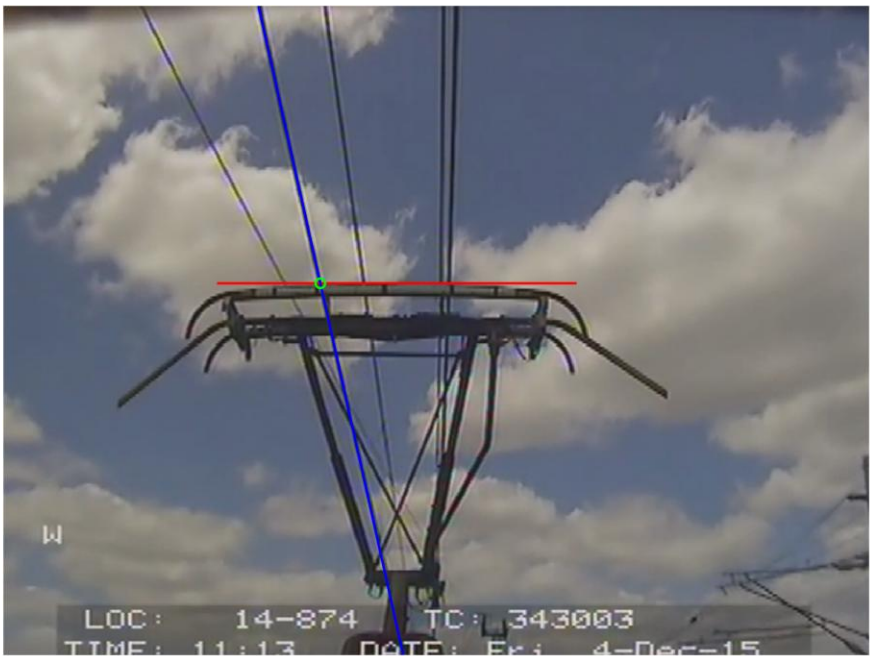
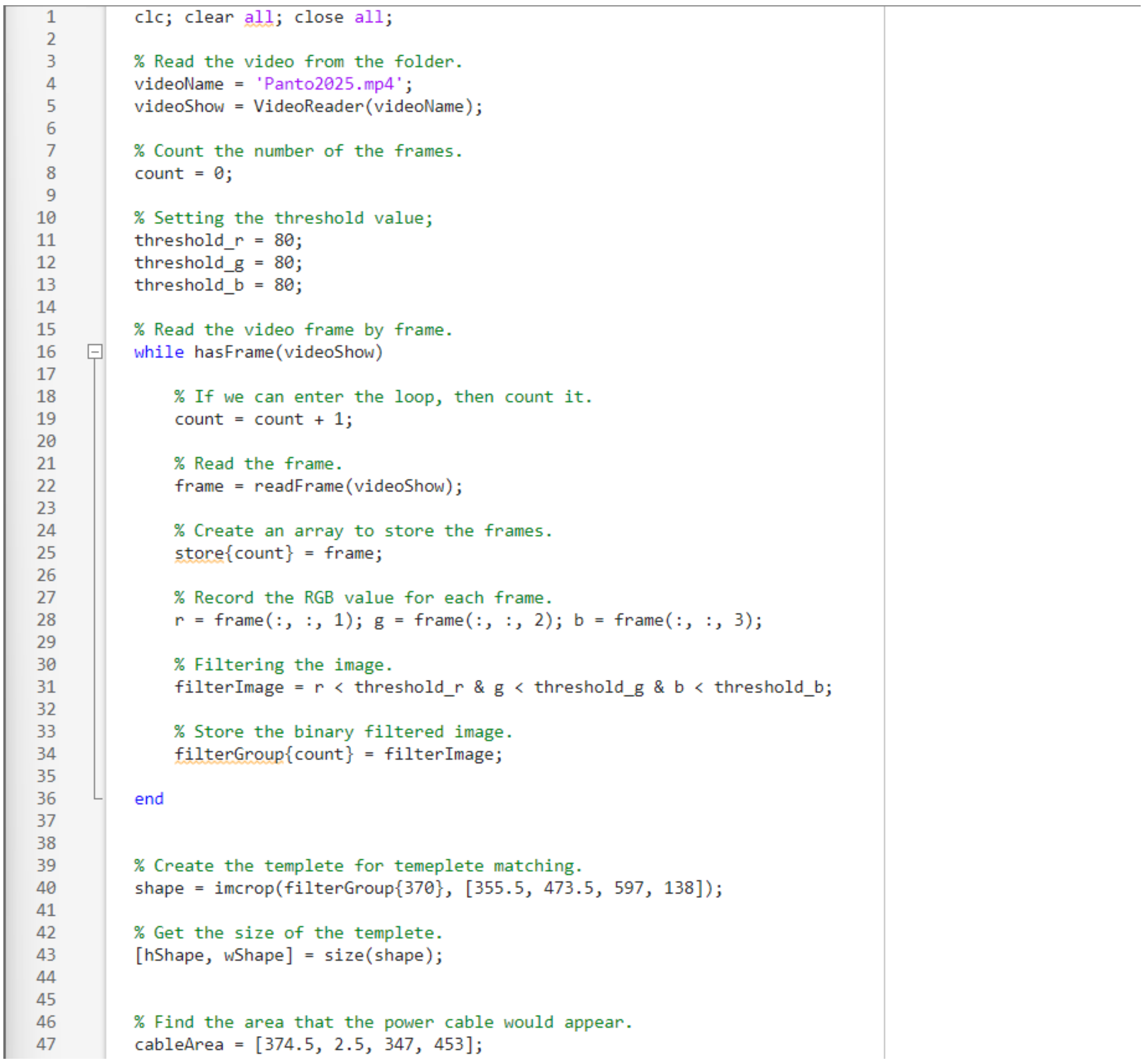


Figure 2.12: Only keep the upper border of the pantograph.

Both rectangle version of the pantograph detection and the upper border of rectangle version I all kept in my code, depending on which version we want.

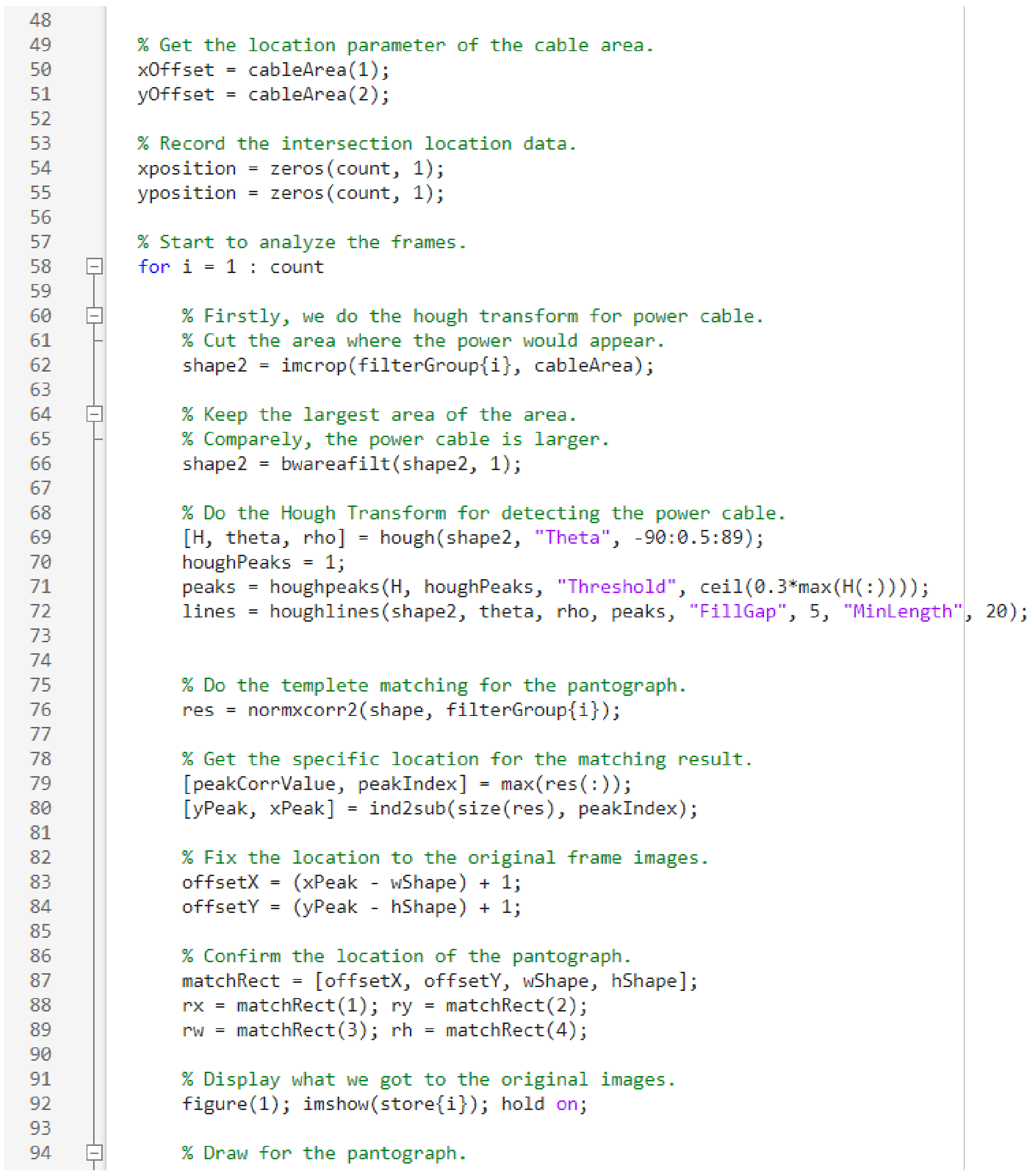
Then, we display the code for the solution. Figure 2.13 to Figure 2.17 shows the overall code.

- **CODE DISPLAY.**



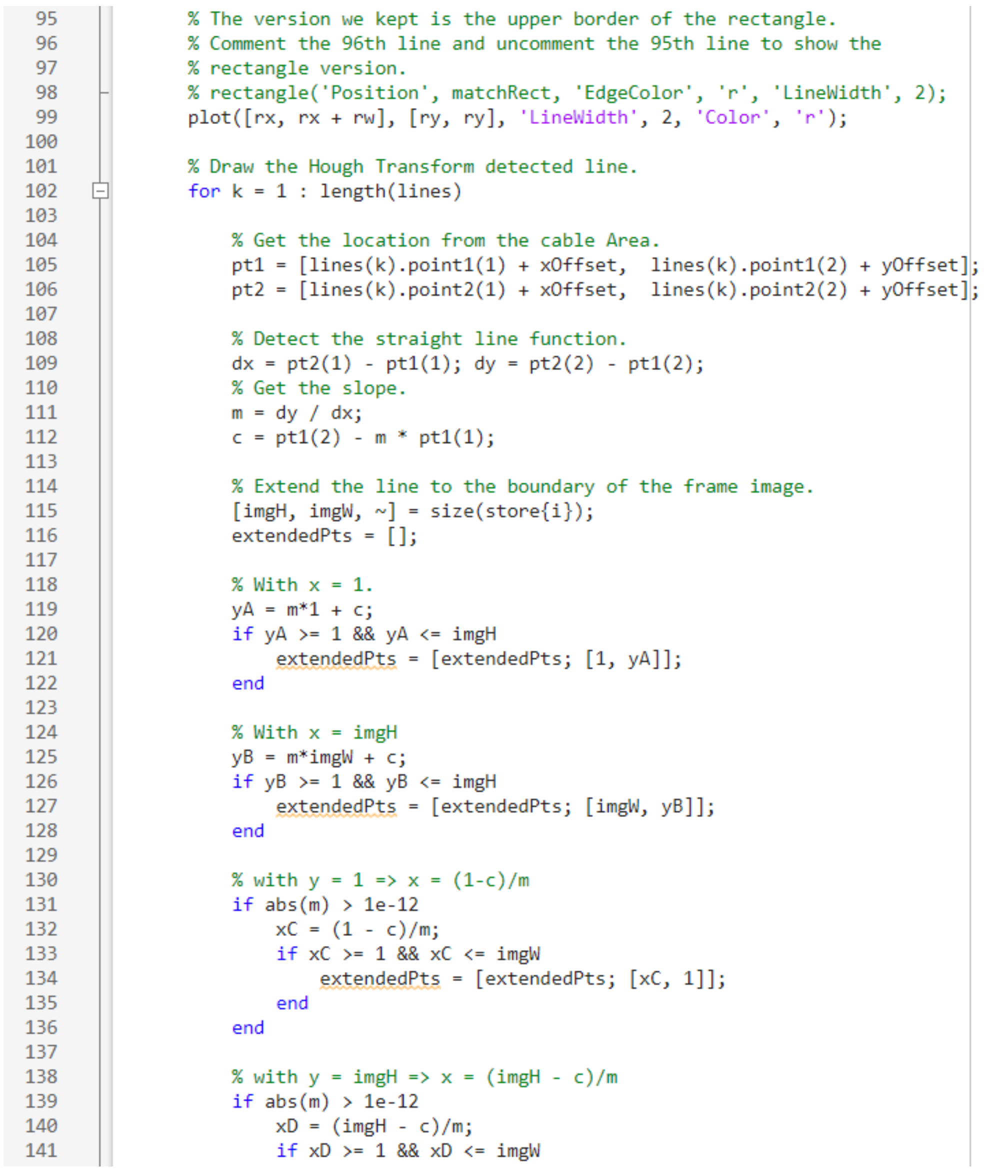
```
clc; clear all; close all;

% Read the video from the folder.
videoName = 'Panto2025.mp4';
videoShow = VideoReader(videoName);

% Count the number of the frames.
count = 0;

% Setting the threshold value;
threshold_r = 80;
threshold_g = 80;
threshold_b = 80;

% Read the video frame by frame.
while hasFrame(videoShow)

    % If we can enter the loop, then count it.
    count = count + 1;

    % Read the frame.
    frame = readFrame(videoShow);

    % Create an array to store the frames.
    store{count} = frame;

    % Record the RGB value for each frame.
    r = frame(:, :, 1); g = frame(:, :, 2); b = frame(:, :, 3);

    % Filtering the image.
    filterImage = r < threshold_r & g < threshold_g & b < threshold_b;

    % Store the binary filtered image.
    filterGroup{count} = filterImage;

end


% Create the templete for temeplete matching.
shape = imcrop(filterGroup{370}, [355.5, 473.5, 597, 138]);

% Get the size of the templete.
[hShape, wShape] = size(shape);


% Find the area that the power cable would appear.
cableArea = [374.5, 2.5, 347, 453];
```

Figure 2.13: The code display part 1.



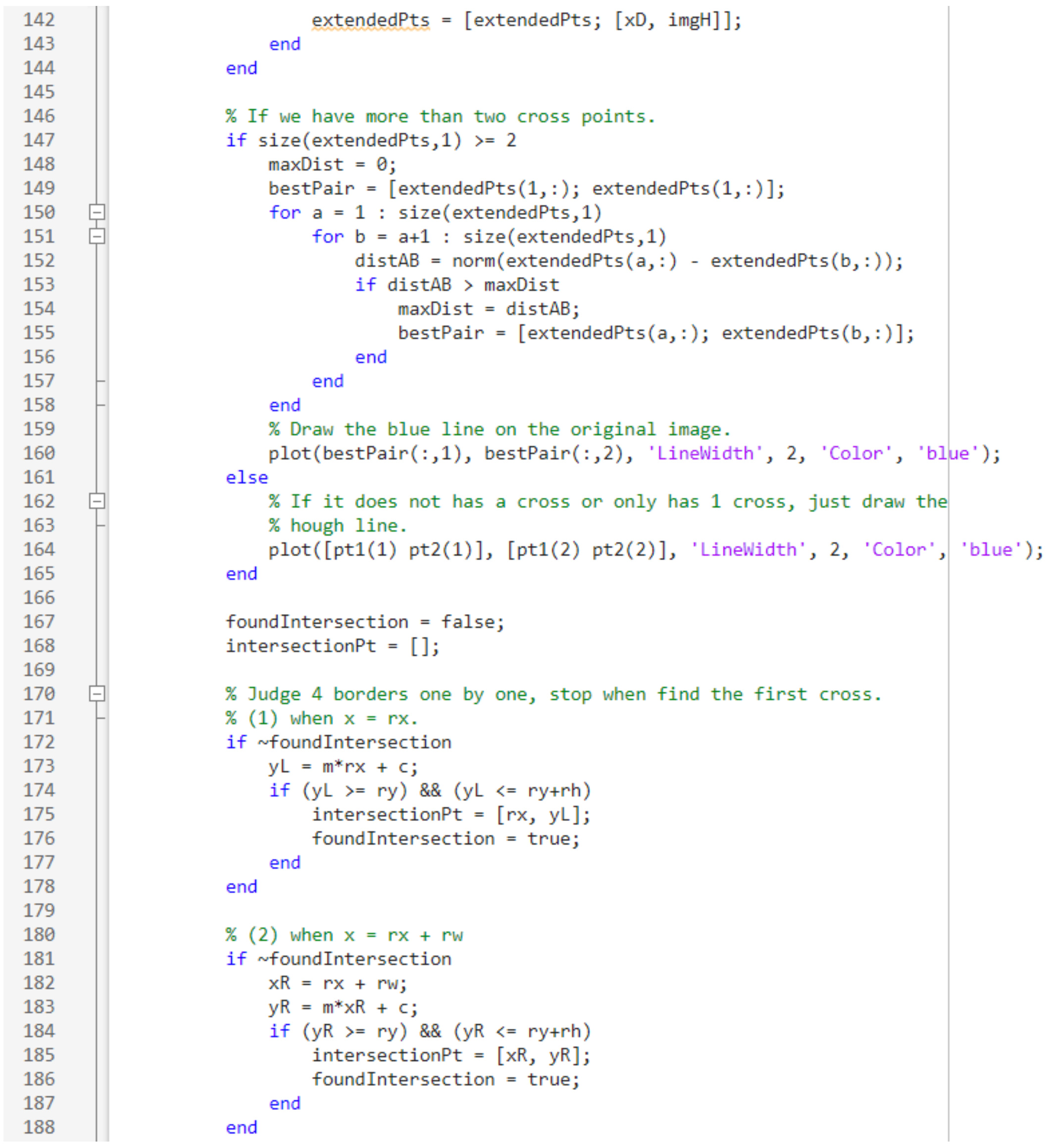
```matlab

% Get the location parameter of the cable area.
xOffset = cableArea(1);
yOffset = cableArea(2);

% Record the intersection location data.
xposition = zeros(count, 1);
yposition = zeros(count, 1);

% Start to analyze the frames.
for i = 1 : count

    % Firstly, we do the hough transform for power cable.
    % Cut the area where the power would appear.
    shape2 = imcrop(filterGroup{i}, cableArea);

    % Keep the largest area of the area.
    % Comparely, the power cable is larger.
    shape2 = bwareafilt(shape2, 1);

    % Do the Hough Transform for detecting the power cable.
    [H, theta, rho] = hough(shape2, "Theta", -90:0.5:89);
    houghPeaks = 1;
    peaks = houghpeaks(H, houghPeaks, "Threshold", ceil(0.3*max(H(:))));
    lines = houghlines(shape2, theta, rho, peaks, "FillGap", 5, "MinLength", 20);


    % Do the templete matching for the pantograph.
    res = normxcorr2(shape, filterGroup{i});

    % Get the specific location for the matching result.
    [peakCorrValue, peakIndex] = max(res(:));
    [yPeak, xPeak] = ind2sub(size(res), peakIndex);

    % Fix the location to the original frame images.
    offsetX = (xPeak - wShape) + 1;
    offsetY = (yPeak - hShape) + 1;

    % Confirm the location of the pantograph.
    matchRect = [offsetX, offsetY, wShape, hShape];
    rx = matchRect(1); ry = matchRect(2);
    rw = matchRect(3); rh = matchRect(4);

    % Display what we got to the original images.
    figure(1); imshow(store{i}); hold on;

    % Draw for the pantograph.
```

Figure 2.14: The code display part 2.



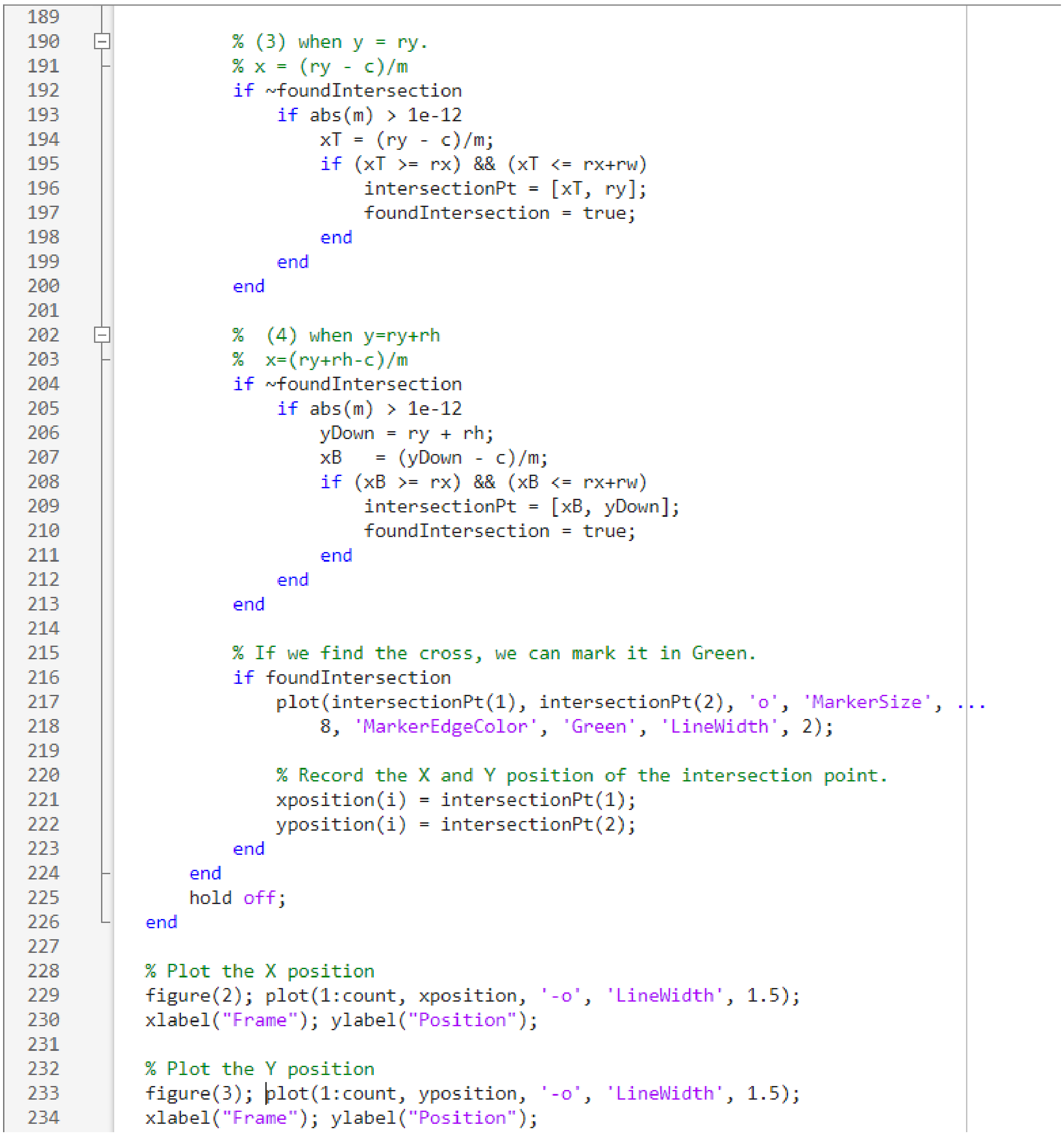
```
 95        % The version we kept is the upper border of the rectangle.
 96        % Comment the 96th line and uncomment the 95th line to show the
 97        % rectangle version.
 98        % rectangle('Position', matchRect, 'EdgeColor', 'r', 'LineWidth', 2);
 99        plot([rx, rx + rw], [ry, ry], 'LineWidth', 2, 'Color', 'r');
100
101        % Draw the Hough Transform detected line.
102        for k = 1 : length(lines)
103
104            % Get the location from the cable Area.
105            pt1 = [lines(k).point1(1) + xOffset,  lines(k).point1(2) + yOffset];
106            pt2 = [lines(k).point2(1) + xOffset,  lines(k).point2(2) + yOffset];
107
108            % Detect the straight line function.
109            dx = pt2(1) - pt1(1); dy = pt2(2) - pt1(2);
110            % Get the slope.
111            m = dy / dx;
112            c = pt1(2) - m * pt1(1);
113
114            % Extend the line to the boundary of the frame image.
115            [imgH, imgW, ~] = size(store{i});
116            extendedPts = [];
117
118            % With x = 1.
119            yA = m*1 + c;
120            if yA >= 1 && yA <= imgH
121                extendedPts = [extendedPts; [1, yA]];
122            end
123
124            % With x = imgH
125            yB = m*imgW + c;
126            if yB >= 1 && yB <= imgH
127                extendedPts = [extendedPts; [imgW, yB]];
128            end
129
130            % with y = 1 => x = (1-c)/m
131            if abs(m) > 1e-12
132                xC = (1 - c)/m;
133                if xC >= 1 && xC <= imgW
134                    extendedPts = [extendedPts; [xC, 1]];
135                end
136            end
137
138            % with y = imgH => x = (imgH - c)/m
139            if abs(m) > 1e-12
140                xD = (imgH - c)/m;
141                if xD >= 1 && xD <= imgW
```

Figure 2.15: The code display part 3.



```
                extendedPts = [extendedPts; [xD, imgH]];
            end
        end

        % If we have more than two cross points.
        if size(extendedPts,1) >= 2
            maxDist = 0;
            bestPair = [extendedPts(1,:); extendedPts(1,:)];
            for a = 1 : size(extendedPts,1)
                for b = a+1 : size(extendedPts,1)
                    distAB = norm(extendedPts(a,:) - extendedPts(b,:));
                    if distAB > maxDist
                        maxDist = distAB;
                        bestPair = [extendedPts(a,:); extendedPts(b,:)];
                    end
                end
            end
            % Draw the blue line on the original image.
            plot(bestPair(:,1), bestPair(:,2), 'LineWidth', 2, 'Color', 'blue');
        else
            % If it does not has a cross or only has 1 cross, just draw the
            % hough line.
            plot([pt1(1) pt2(1)], [pt1(2) pt2(2)], 'LineWidth', 2, 'Color', 'blue');
        end

        foundIntersection = false;
        intersectionPt = [];

        % Judge 4 borders one by one, stop when find the first cross.
        % (1) when x = rx.
        if ~foundIntersection
            yL = m*rx + c;
            if (yL >= ry) && (yL <= ry+rh)
                intersectionPt = [rx, yL];
                foundIntersection = true;
            end
        end

        % (2) when x = rx + rw
        if ~foundIntersection
            xR = rx + rw;
            yR = m*xR + c;
            if (yR >= ry) && (yR <= ry+rh)
                intersectionPt = [xR, yR];
                foundIntersection = true;
            end
        end
```

Figure 2.16: The code display part 4.



```matlab

        % (3) when y = ry.
        % x = (ry - c)/m
        if ~foundIntersection
            if abs(m) > 1e-12
                xT = (ry - c)/m;
                if (xT >= rx) && (xT <= rx+rw)
                    intersectionPt = [xT, ry];
                    foundIntersection = true;
                end
            end
        end

        %  (4) when y=ry+rh
        %  x=(ry+rh-c)/m
        if ~foundIntersection
            if abs(m) > 1e-12
                yDown = ry + rh;
                xB   = (yDown - c)/m;
                if (xB >= rx) && (xB <= rx+rw)
                    intersectionPt = [xB, yDown];
                    foundIntersection = true;
                end
            end
        end

        % If we find the cross, we can mark it in Green.
        if foundIntersection
            plot(intersectionPt(1), intersectionPt(2), 'o', 'MarkerSize', ...
                8, 'MarkerEdgeColor', 'Green', 'LineWidth', 2);

            % Record the X and Y position of the intersection point.
            xposition(i) = intersectionPt(1);
            yposition(i) = intersectionPt(2);
        end
    end
    hold off;
end

% Plot the X position
figure(2); plot(1:count, xposition, '-o', 'LineWidth', 1.5);
xlabel("Frame"); ylabel("Position");

% Plot the Y position
figure(3); plot(1:count, yposition, '-o', 'LineWidth', 1.5);
xlabel("Frame"); ylabel("Position");
```

Figure 2.17: The code display part 5

In Figure 2.18 and Figure 2.19. I plot the (X, Y) coordinate changes as the frame processing, which can illustrate the change of the touch point of pantograph and the power cable.

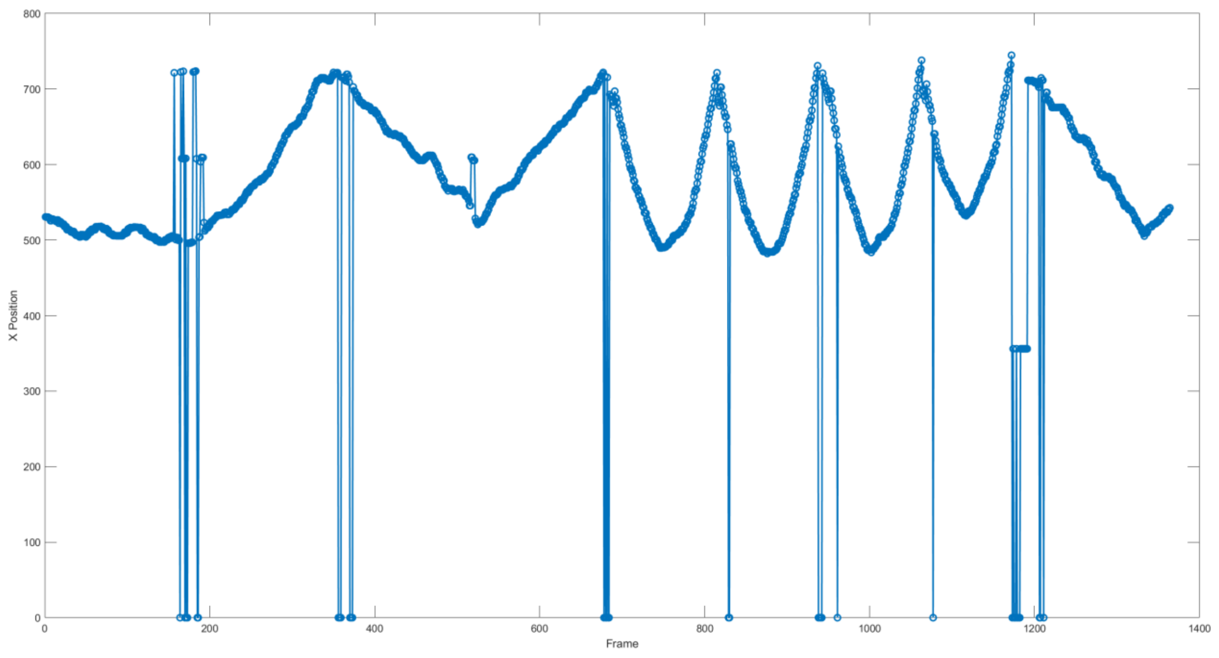


Figure 2.18: Frame & X position.

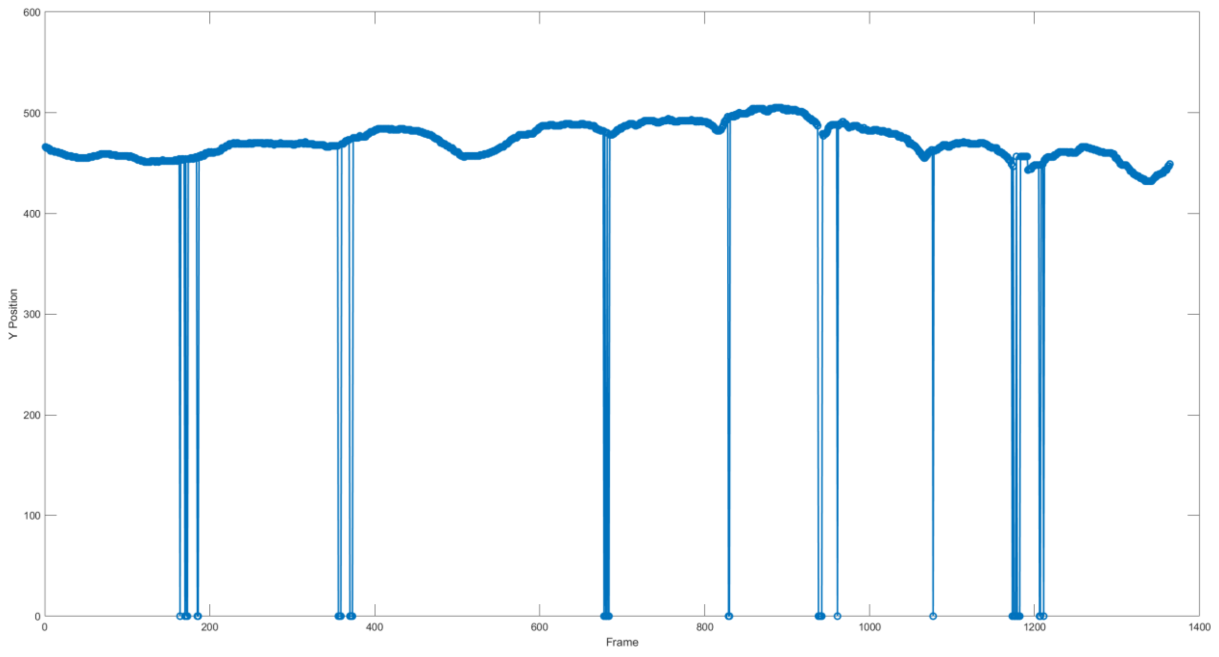


Figure 2.19: Frame & Y position.

From Figure 2.18 and Figure 2.19, we find that for both X and Y are normal for most of the frames. And for both X and Y outliers are all appear at the same time.

- **REVIEW OF THE QUESTION (PROBLEMS)**

The code helps us detect the touch point on the pantograph and the power cable successfully on most of the frames, however, in some cases, the detection failed, for example, like it shows in Figure 2.20.

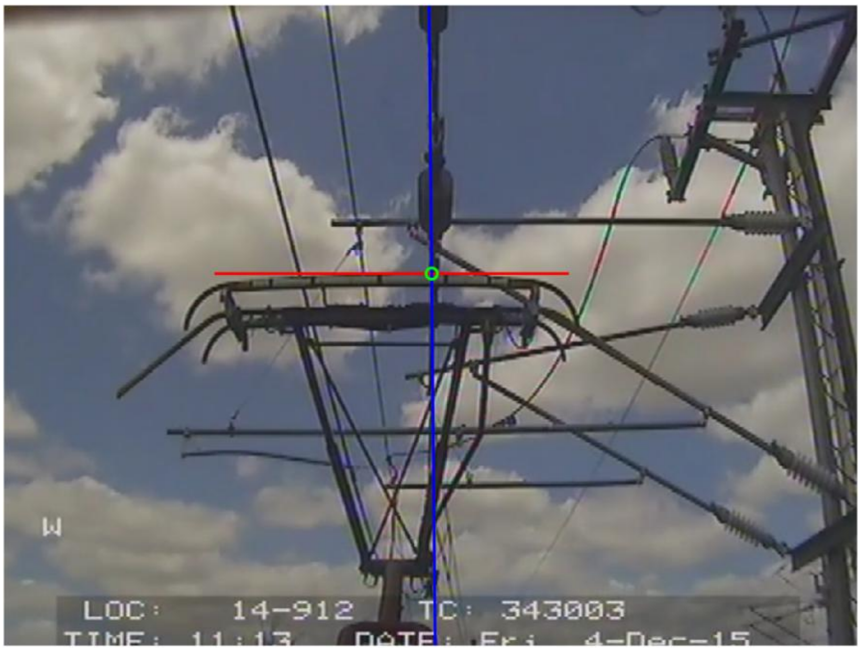


Figure 2.20: The failed detection of the power cable.

The reason why this situation would appear is easy, because I only detect the power cable in selected area, and recognize the largest cable in that area as the power cable, so when there is a larger cable or the structure like it shows in the right side of the Figure 2.20 appears, the system would recognize that as the power cable, though sometimes it looks strange. So, the drawback of my method is: it can not prevent the possible some other interfering objects that suddenly appear unless the objects are smaller than the power cable after RGB filtering.

Samly, I said the power cable detection only appears in a specific area, so as the video progresses, sometimes the cable will run out of the specific area, which would make the system cannot detect the cable. This kind of failure is shown in Figure 2.21.

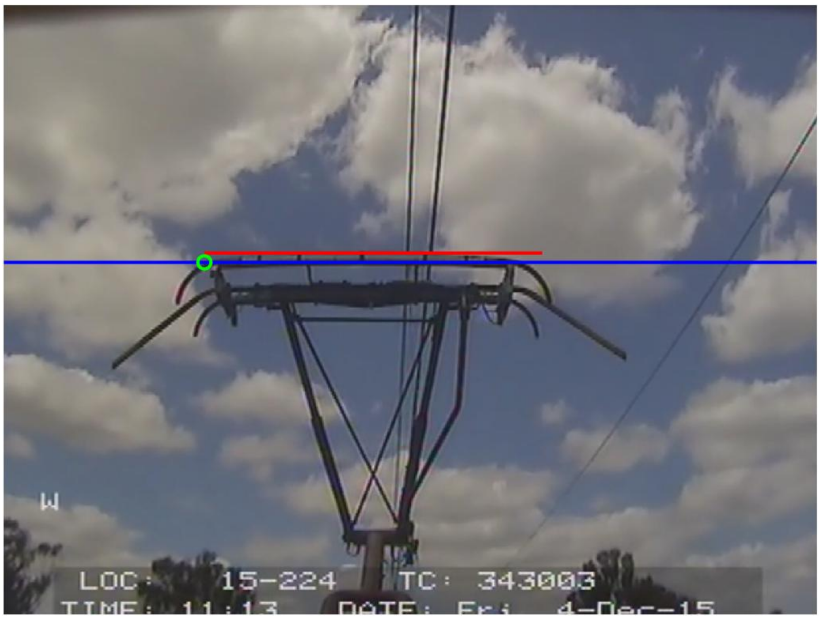


Figure 2.21: The second kind of failed detection.

These are two largest problems that I came across in this question solution. Which currently is hard for me to deal with.

Question 3： Mr. Bean line & circle detection.

● The target of the question

In this question, we are required to use the Hough Transform to detect the stick and the white lines in the provided Mr. Bean's image, the image is shown in Figure 3.1.

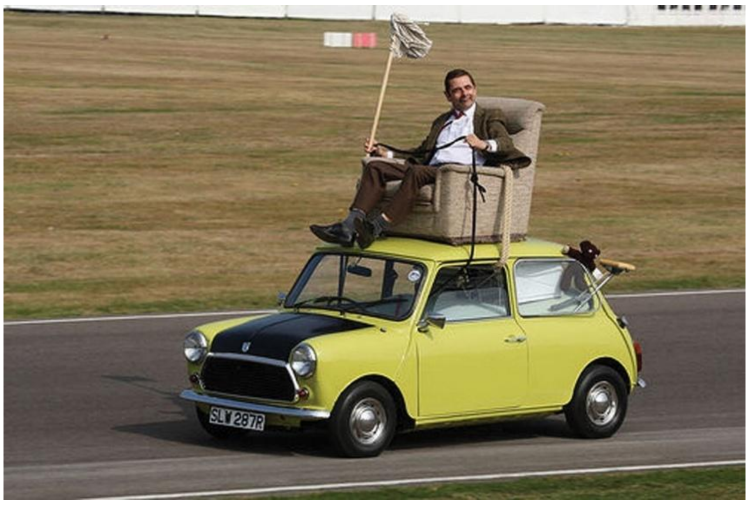


Figure 3.1: Mr. Bean2025.jpg

To complete this question, we should understand what Hough Transform is.

Hough Transform: Hough Transform is a method used to detect lines and circles in image processing. Normally, after we finished the convert the original RGB images to binary images, the value of the binary images are only 0 and 1, however, those points which value is 1 has its coordinate, so we can use a function expression like $y = kx + b$ in cartesian coordinate system or $\rho = x * cos(\theta) + y * \sin(\theta)$ in polar coordinate system to express those points might be in a line. In our Hough Transform in MATLAB, we prefer to use the expression in polar coordinate system.

For any point, there are many different function expressions can pass through these points, all these expressions have different parameters, so when we transform original function expression from image space to parameter space, we can find that in parameter space, all those parameters are also follow a function, this is one point. For other points, we can also transform the original function expression from image space to parameter space, after this step, if these points are indeed on a straight line, then there will be an intersection in the parameter space, we just need to watch the corresponding coordinate data, then use the data to find the function expression for the points in image space. This is Hough transform for detecting lines.

For detecting circles, it is almost the same as that line detection, for any points located in the same circle, the circle can be expressed as a function: $(x_i - a)^2 + (y_i - b)^2 = r^2$, where the a and b are the horizontal and vertical coordinates of the center of the circle respectively, and r is the radius of the circle. We note that there are 3 parameters in circle detection, so, for successfully detecting circles, we should give the radius for detecting, the radius can be a specific value and it can also be a range.

- **Method Introduction.**

In this part, I would introduce standard line detection and circle detection respectively, firstly, I would introduce Standard line detection.

Firstly, we read the image from the folder, the image is shown in Figure 35, we do not show that again.

The second step is to convert the image from RGB to grayscale, this step can help simplify the algorithm and reduce the complexity of computing, the gray image is shown in Figure 3.2.

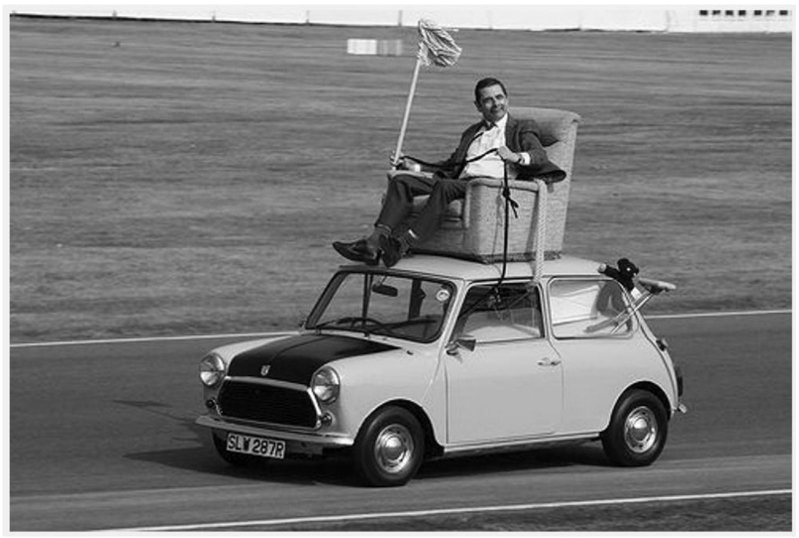


Figure 3.2: The grayscale of Mr. Bean.

After we converted it, here we use Canny edge detector to find the edge of the image, at the same time we can convert the image into a binary image. Here we set a threshold range for a better detection. The image after the Canny edge detection is shown in Figure 3.3.

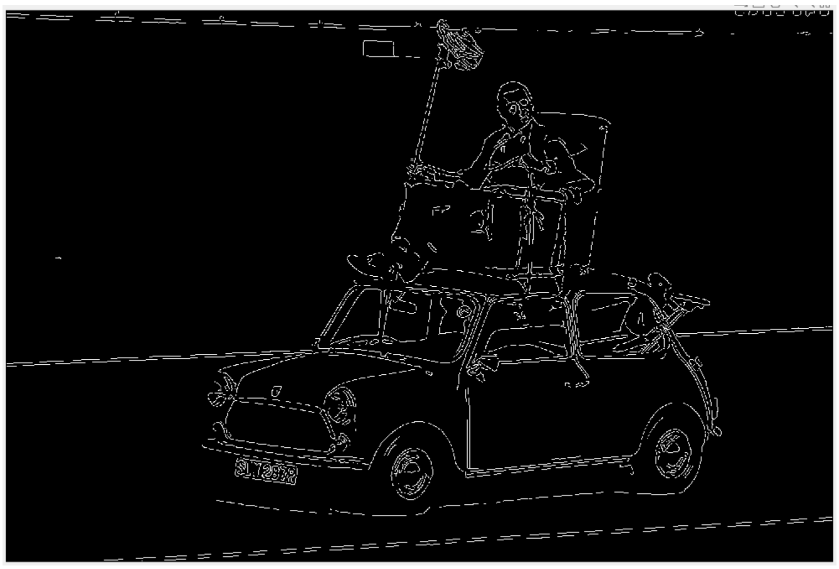


Figure 3.3: Mr. Bean after the Canny edge detection.

From the Figure 3.3, we can find that our target object are all kept and the noise are filtered a lot, this is because we set a threshold for this detection, if we do not set a threshold for this image, the result of the detection would be like Figure 3.4, which contains a great amount of noise that make us cannot do any analysis.

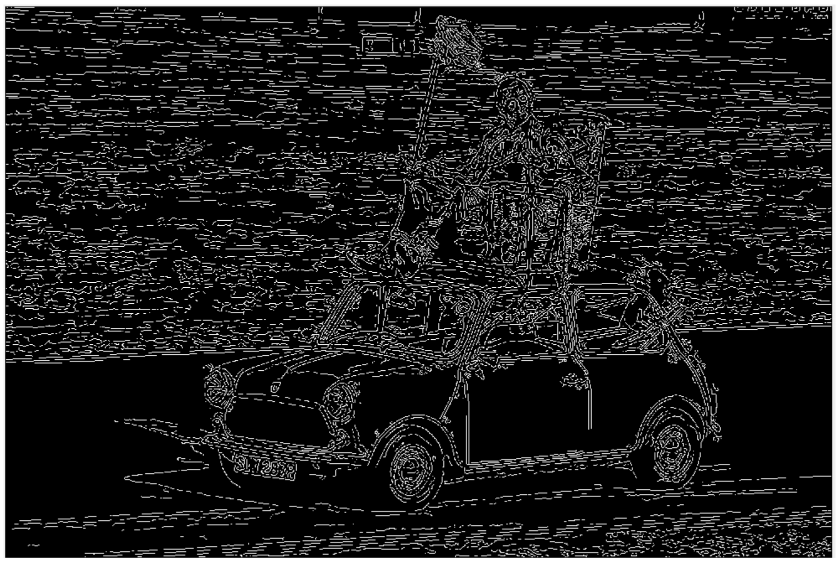


Figure 3.4: Edge detection without threshold.

After we get the canny edge detected image, we can start to calculate the Standard

Hough Transform (SHT) of the binary image. This step is using the polar coordinate system $\rho = x * cos(\theta) + y * \sin(\theta)$. The graph of parameter is shown in Figure 3.5.

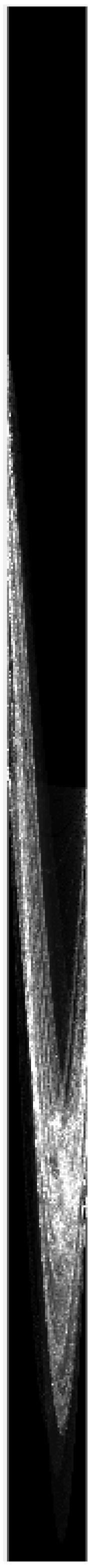


Figure 3.5: Parameter space image.

Figure 3.6 shows an image more readable for parameter space.

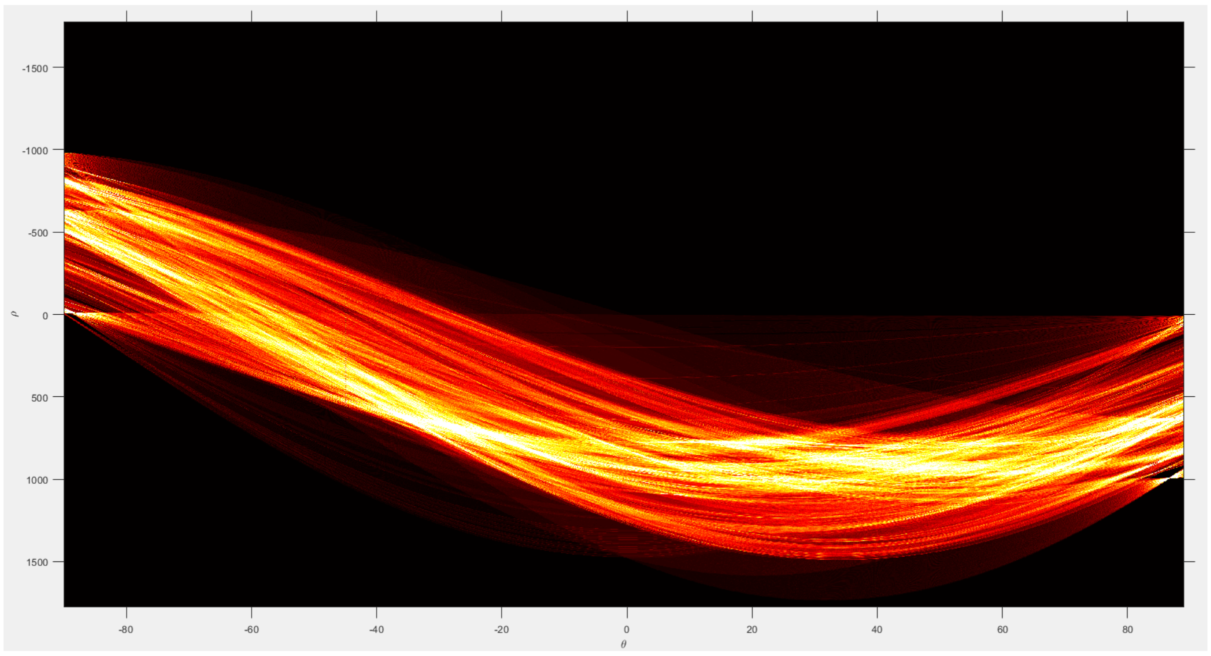


Figure 3.6: Parameter space image.

By this image, we can plot the peak of the parameter space. In this question, I set the maximum of the number of peaks to 1000. If the peak number reaches 1000, it would be plotted on the image, the results are shown in Figure 3.7.

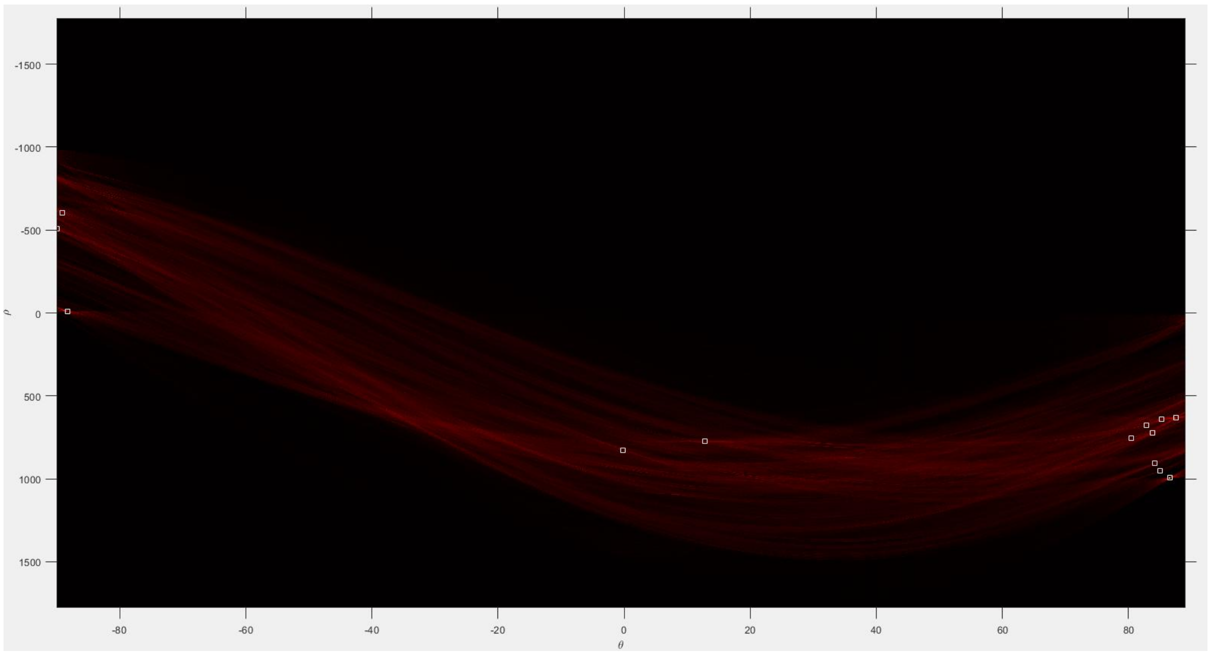


Figure 3.8: The peak number detection.

After this step, we can draw the lines on the original image. The result of standard line detection is shown in Figure 3.9, and the lines are drawn in blue color.

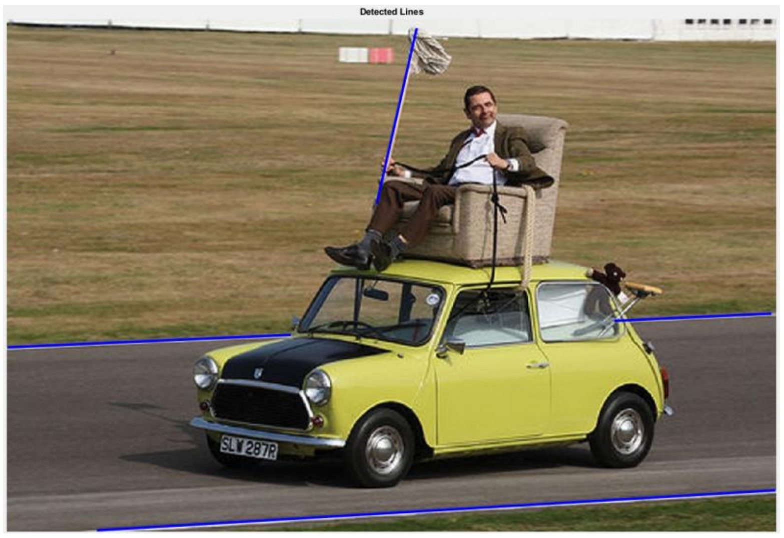


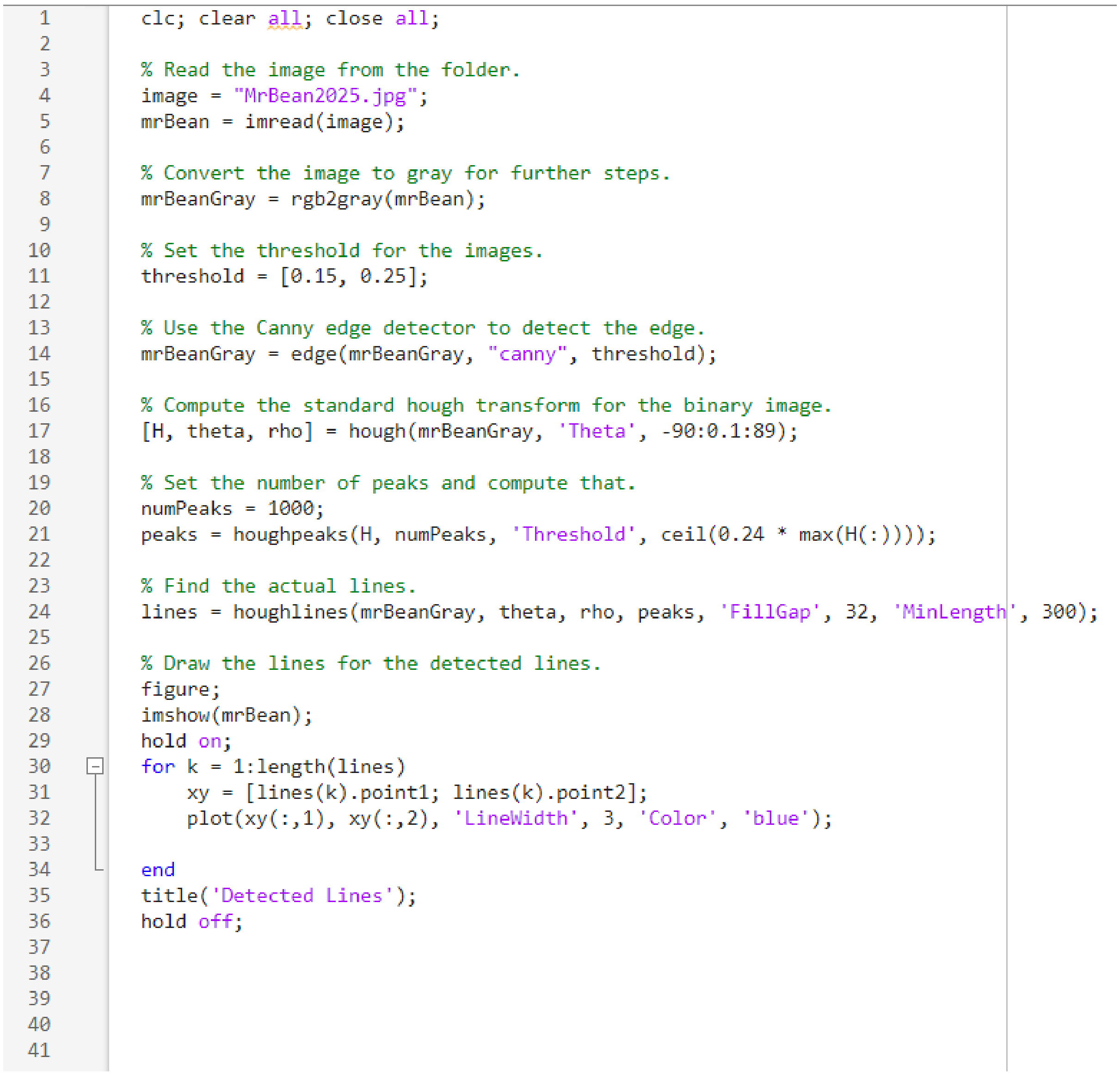
Figure 3.9: The final result of standard line detection.

By observing the Figure 3.9, we find that the lines required us to detect are all marked, however, this also has some drawbacks, that is the stick Mr. Bean grabbed. In this picture, Mr. Bean’s knees were bent at an angle that formed a straight line with the stick, this lead my code consider his knees as a part of the stick, to the other parts, the code works well.

- **Code**

The code I divided into 2 versions, the first only has the final results and the second version contains the Figures from Figure 3.1 to Figure 3.9.

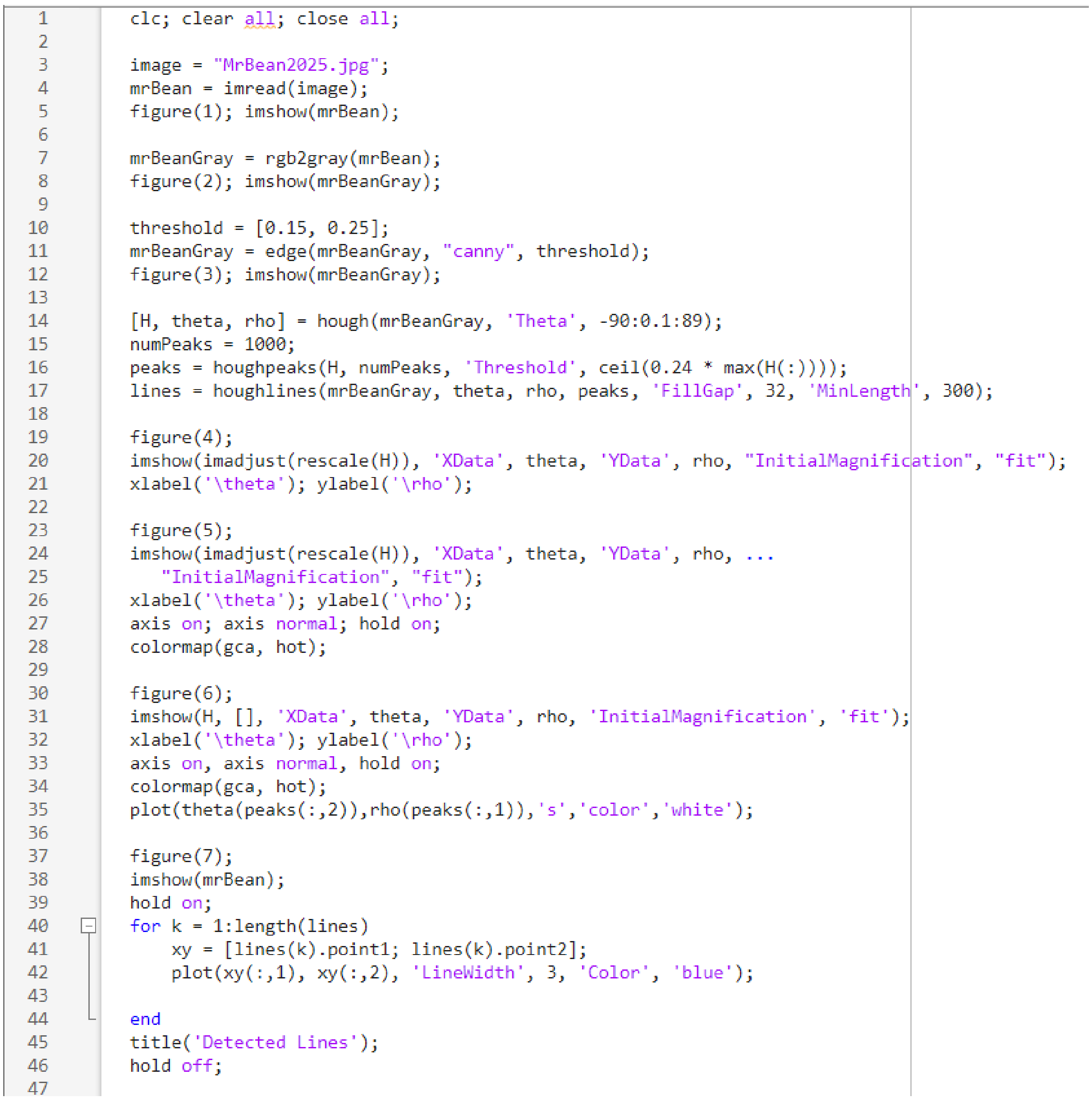
The first simple version is shown in Figure 3.10:



```matlab
clc; clear all; close all;

% Read the image from the folder.
image = "MrBean2025.jpg";
mrBean = imread(image);

% Convert the image to gray for further steps.
mrBeanGray = rgb2gray(mrBean);

% Set the threshold for the images.
threshold = [0.15, 0.25];

% Use the Canny edge detector to detect the edge.
mrBeanGray = edge(mrBeanGray, "canny", threshold);

% Compute the standard hough transform for the binary image.
[H, theta, rho] = hough(mrBeanGray, 'Theta', -90:0.1:89);

% Set the number of peaks and compute that.
numPeaks = 1000;
peaks = houghpeaks(H, numPeaks, 'Threshold', ceil(0.24 * max(H(:))));

% Find the actual lines.
lines = houghlines(mrBeanGray, theta, rho, peaks, 'FillGap', 32, 'MinLength', 300);

% Draw the lines for the detected lines.
figure;
imshow(mrBean);
hold on;
for k = 1:length(lines)
    xy = [lines(k).point1; lines(k).point2];
    plot(xy(:,1), xy(:,2), 'LineWidth', 3, 'Color', 'blue');

end
title('Detected Lines');
hold off;
```

Figure 3.10: The simple version of Mr. Bean Standard Line Detection.

The Figure Version is shown in Figure 3.11:



```
clc; clear all; close all;

image = "MrBean2025.jpg";
mrBean = imread(image);
figure(1); imshow(mrBean);

mrBeanGray = rgb2gray(mrBean);
figure(2); imshow(mrBeanGray);

threshold = [0.15, 0.25];
mrBeanGray = edge(mrBeanGray, "canny", threshold);
figure(3); imshow(mrBeanGray);

[H, theta, rho] = hough(mrBeanGray, 'Theta', -90:0.1:89);
numPeaks = 1000;
peaks = houghpeaks(H, numPeaks, 'Threshold', ceil(0.24 * max(H(:))));
lines = houghlines(mrBeanGray, theta, rho, peaks, 'FillGap', 32, 'MinLength', 300);

figure(4);
imshow(imadjust(rescale(H)), 'XData', theta, 'YData', rho, "InitialMagnification", "fit");
xlabel('\theta'); ylabel('\rho');

figure(5);
imshow(imadjust(rescale(H)), 'XData', theta, 'YData', rho, ...
   "InitialMagnification", "fit");
xlabel('\theta'); ylabel('\rho');
axis on; axis normal; hold on;
colormap(gca, hot);

figure(6);
imshow(H, [], 'XData', theta, 'YData', rho, 'InitialMagnification', 'fit');
xlabel('\theta'); ylabel('\rho');
axis on, axis normal, hold on;
colormap(gca, hot);
plot(theta(peaks(:,2)),rho(peaks(:,1)),'s','color','white');

figure(7);
imshow(mrBean);
hold on;
for k = 1:length(lines)
    xy = [lines(k).point1; lines(k).point2];
    plot(xy(:,1), xy(:,2), 'LineWidth', 3, 'Color', 'blue');

end
title('Detected Lines');
hold off;

```

Figure 3.11: The code with Figure demos.

Then we goes to the circle detection, to the circle detection, the steps until using threshold canny edge detection are the same, after we detected the edge for the image, we should use the MATLAB function “imfindcircles” to detect the circles in the image, for this question, we set the radius range for the image from 12 to 50. The detection of the circle result is shown in Figure 3.12.

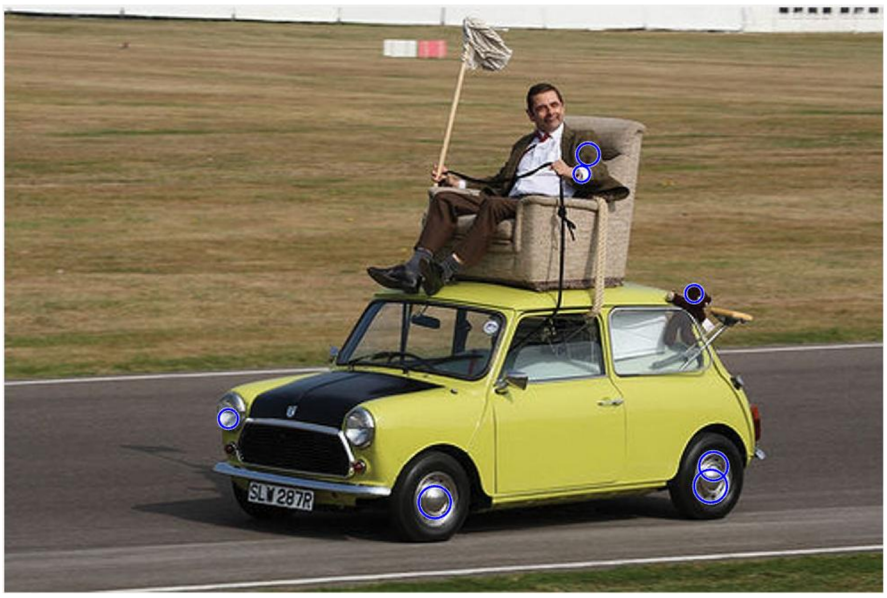


Figure 3.12: The circle detection of Mr. Bean.

We find that the circle detection for the two wheels is all included in the image, so we just need to eliminate the other circles we do not want, then the answer would be what we want. Figure 3.13 shows the final results.

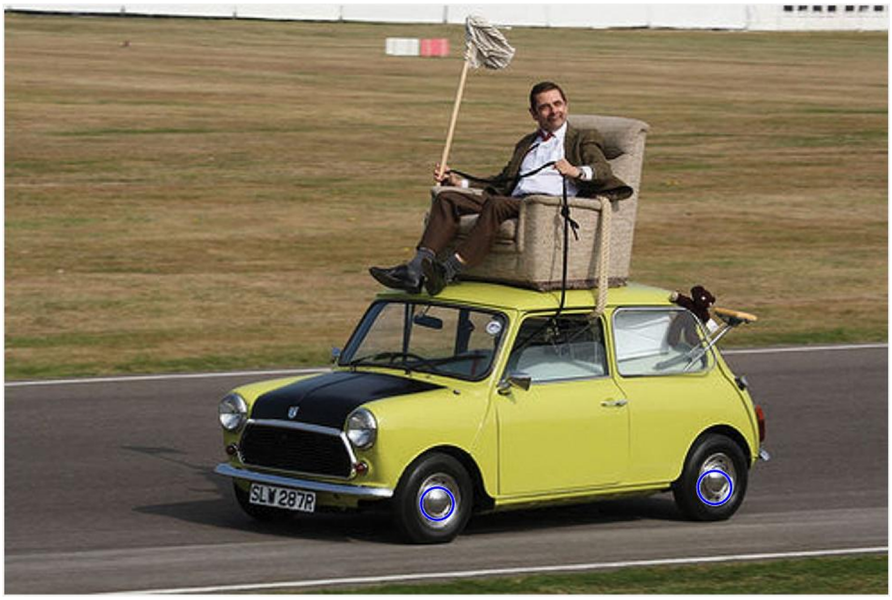
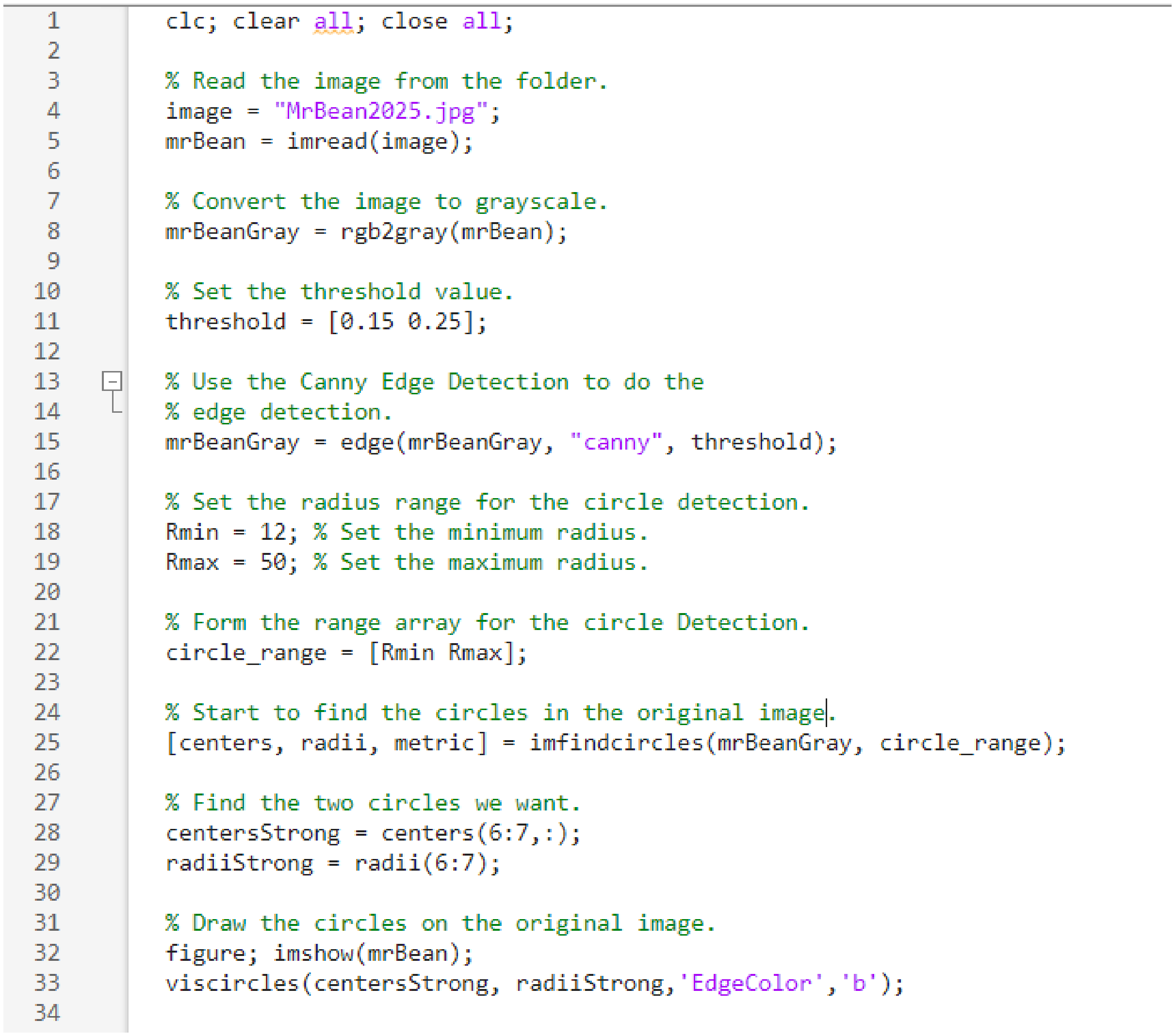


Figure 3.13: The two circles detection of the Mr. Bean image.

In Figure 3.13, we marked the two circles we want in blue color .

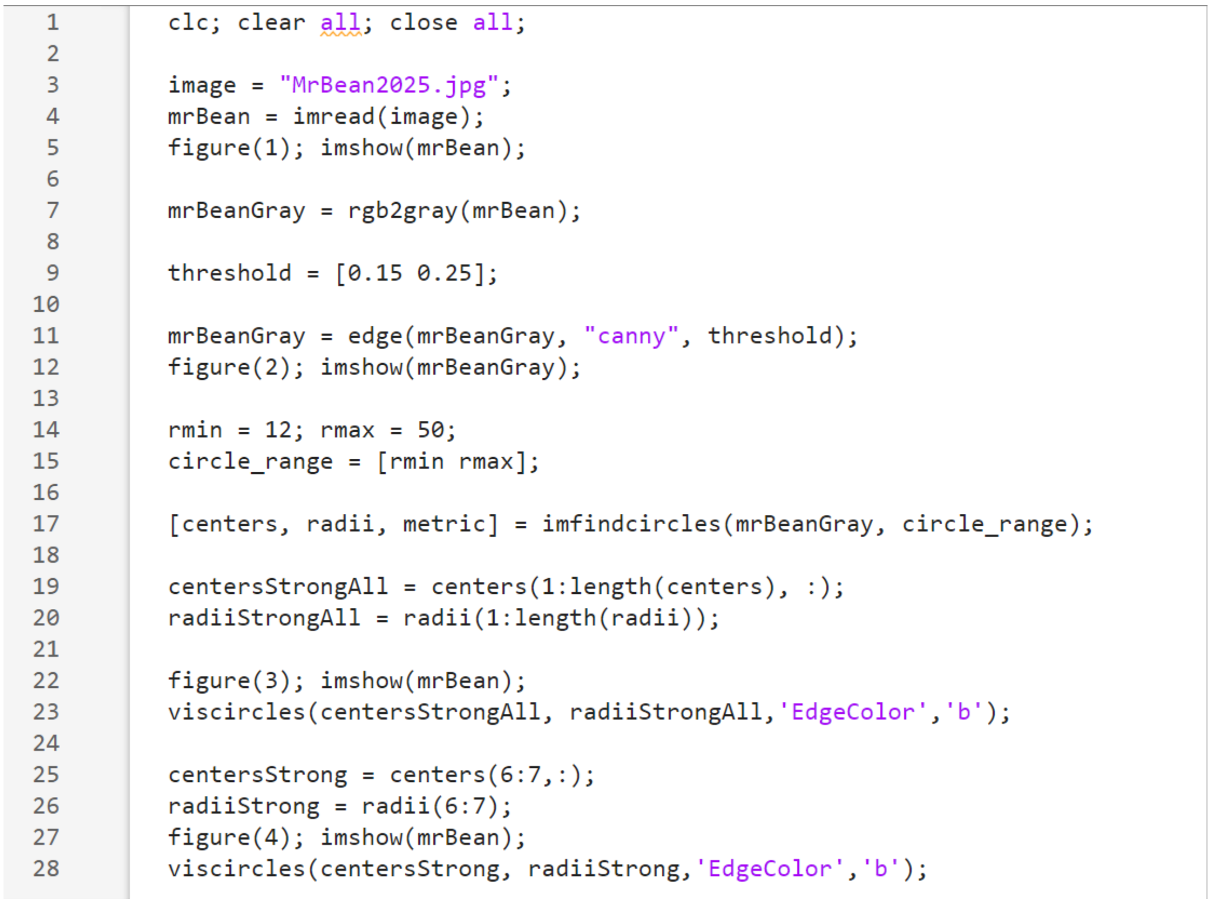
Then I would show the code for the circle detection, also it has two versions, the first version only plots the final result and the second version plots all the figures involved in this question. The first version is shown in Figure 3.14.



```
clc; clear all; close all;

% Read the image from the folder.
image = "MrBean2025.jpg";
mrBean = imread(image);

% Convert the image to grayscale.
mrBeanGray = rgb2gray(mrBean);

% Set the threshold value.
threshold = [0.15 0.25];

% Use the Canny Edge Detection to do the
% edge detection.
mrBeanGray = edge(mrBeanGray, "canny", threshold);

% Set the radius range for the circle detection.
Rmin = 12; % Set the minimum radius.
Rmax = 50; % Set the maximum radius.

% Form the range array for the circle Detection.
circle_range = [Rmin Rmax];

% Start to find the circles in the original image.
[centers, radii, metric] = imfindcircles(mrBeanGray, circle_range);

% Find the two circles we want.
centersStrong = centers(6:7,:);
radiiStrong = radii(6:7);

% Draw the circles on the original image.
figure; imshow(mrBean);
viscircles(centersStrong, radiiStrong,'EdgeColor','b');

```

Figure 3.14: The simple version only plots final results.

The second version plots all the figures involved in the circle detection, the code in shown in Figure 3.15.



```matlab
clc; clear all; close all;

image = "MrBean2025.jpg";
mrBean = imread(image);
figure(1); imshow(mrBean);

mrBeanGray = rgb2gray(mrBean);

threshold = [0.15 0.25];

mrBeanGray = edge(mrBeanGray, "canny", threshold);
figure(2); imshow(mrBeanGray);

rmin = 12; rmax = 50;
circle_range = [rmin rmax];

[centers, radii, metric] = imfindcircles(mrBeanGray, circle_range);

centersStrongAll = centers(1:length(centers), :);
radiiStrongAll = radii(1:length(radii));

figure(3); imshow(mrBean);
viscircles(centersStrongAll, radiiStrongAll,'EdgeColor','b');

centersStrong = centers(6:7,:);
radiiStrong = radii(6:7);
figure(4); imshow(mrBean);
viscircles(centersStrong, radiiStrong,'EdgeColor','b');
```

Figure 3.15: The Figure version.

Question 4: Consider a color image of 128x128 pixels

a) For an image contains 128x128 pixels, if we consider it in RGB coordinate system, there will be 3 bytes for each pixel, so that is 24 bits for a pixel, so for a single pixel, it can be 2^{24} colors. To an image with 128x128 pixels, it has 16384 (2^{14}) pixels, so there will be $(2^{24})^{2^{14}} = (2^{24})^{16384} = 2^{393216}$. Therefore, for a 128x128 pixels image, the cardinality for it should be 2^{393216}.

b) We can consider a single RGB pixel as a 3-dimensional coordinate because of its 3 channels of color, then for 16384 pixels, it will have $16384 \times 3 = 49152$ dimensions, the range for a single dimension is 0~255, we should use 8 bits for representing this dimension. So, the dimensionality for all possible 128x128 color image is 49152 and 8 bits are required for each dimension.