Machine Vision Report

Jesse Taylor 16042846

Contents

1 Introduction

Machine vision, when effectively implemented, can be a powerful tool for data analysis, production work, and an increasingly wide variation of applications. In this assignment, a machine vision algorithm is written as an introduction to machine vision. By using a data set of example images that are similar to an industrial environment, the assignment aims to develop software development skills through the correct use of machine vision tools and libraries.
In this assignment, Python 3.8 was used with the Open-CV library to develop a machine vision algorithm to count the number of apples in an image, and locate each one. The program takes files from a 'data' sub-directory, and writes a copy of the image back to the same location, with the number of apples and their locations written onto the image. The file name also contains the number of apples, for marker and viewer convenience.
This report summarises the work done to complete this algorithm, and it first presents an overview of the developed algorithm, identifying its key stages and primary program loops. It then discusses the methodology used to develop the algorithm, including further details on the development progress, and increased detail for some sections of code. Finally, the report discusses the results of the algorithm, and concludes.

2 Algorithm

The library dependencies of the program are:
- Open-CV
- datetime
- glob

datetime is used to time operations and total program execution time, providing some interesting feedback. It is not strictly necessary. However, it comes with the default install of Python.
glob is used for extremely capable file processing and searching, and is used for finding source images in the ./data/ sub-directory that are usable by the program. It is necessary for program execution, and can be installed using the command: 'pip install glob2'. Opening and closing images are handled by their respective Open-CV functions.
This section describes the high level process that the algorithm follows to detect apples in an image. The algorithm has two main sections, a bounding box generator, and a decaying area check.

The algorithm high level flow is:
- Input Image is provided to the program
- Image is parsed for bounding box
- Image is resized to bounding box
- Resized image is parsed for apples
- Parsed apples is filled to make blobs
- Decaying area checks are performed, until everything has been processed
- Final results written to output image
- Program returns

2.1 Bounding Box Generator

In all of the provided reference images, the apples are placed on the same blue coloured tray. Assuming that no apples are outside the tray, and therefore outside the box, a bounding box is generated around the blue tray.
The bounding box is used to scale the original image, so that the HSV colour detection later used to detect apples will only run within the bounding box established here. In terms of the original image, the HSV apple colour detection is only running within the area of the blue apple tray.

The blue tray is detected by finding the the maximum and minimum locations for both the X and Y coordinates where the threshold values detected 'blue'. The result of this operation gives a (MinX, MinY) and (MaxX, MaxY) coordinate set, which is used to draw the bounding box on the output image, and to scale the input image for later processing.

2.2 Decaying Area Checks

With the working area defined above by the bounding box, the image can now be processed for the apples in the image. The image is processed, and tidied, then passed into a while loop. The while loop checks that there

Figure 1: Bounding Box detected on Reference Image 0

is still white in the thresholded image (See figure 11), and, if there is, erodes the white in the image by a user set element size, and performs the area filter.

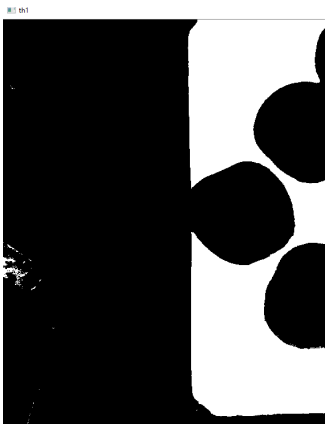
The area filter runs contour detection over the newly eroded image, and calculates the area and moments of each detected contour. If the area is within certain bounds, and the centroid of the contour is in a location of an apple, the location of the apple is drawn on the output image. The detected contour is also removed from the processing list, preventing it from being double processed.

In this way, objects detected by the thresholding operation are either detected as apples and removed, or eroded away. This results in, after several passes, an empty threshold image, and the loop ends

3 Methodology

3.1 Bounding Box Generator

The bounding box generator finds the maximum and minimum points where the blue apple tray is present. The blue apple tray is present in all of the reference images, and every apple is contained within its area. This means that once the area has been found, threshold bounds for apple detection can be less aggressive, as there is less areas for noise. The bounding box generator works by thresholding for a wide blue range, to detect the blue apple tray and nothing else, and then moving over the thresholded image to find the maximum and minimum points.



In order to pick up the areas of the blue apple tray that are shadowed by the walls of the cardboard box, the low bound for Value (Of Hue, Saturation, Value) is extremely low. This means that noise can be picked up from darker areas, such as the left of Reference Image 21 (Seen Left).
This widens out the bounding box, making it unusable for the program. To eliminate this noise, two techniques are used. First, a erode of a small element size is used, to eliminate any small packets of noise. Second, if the step size is greater than 1, it acts as a natural average, only picking up data points if they are more than [step] wide or high.

In an effort to increase programming skills, a new method (for me) of iterating over an array was used for bounding box detection. By modifying a standard 2-D for loop to use the next() function, I was able to speed up program execution by a factor of 2.
The code snippet below finds the first value every row; where step = 1, th1 is the original image thresholded for 'blue' values (See figure 10), and height and width

are the height and width of the image.

```
for i in range(0, height, step):
    next((j for j in range(0,width,step) if th1[i,j] == 255), None)
```

next() takes an iterable object or a generator as its first input, and an optional return value if to return if the iterator is exhausted (Moved through the entire object).
The 'generator' (first parameter) is equivalent to the below code snippet:

```
    for j in range(0, width, step):
        if th1[i,j] == 255:
            return j
```

Therefore, with the addition of some additional logic, and moving over the row backwards, the bounding box is found. The above snippet finds the minimum X value where blue is detected, moving over the row backwards finds the maximum X. Additionally, if any value is found, that means that there is 'blue' at that Y position. When compared with the current maximum and minimum values, max and min values for Y can also be found in the same step.

```
for i in range(0, height, step):
    currMin = next((j for j in range(0,width,step) if th1[i,j] == 255), None)
    if currMin != None:
        if currMin < minX:
            minX = currMin
        if i > maxY:
            maxY = i
        if i < minY:
            minY = i
    currMax = next((j for j in range(width-1,0,-1 * step) if th1[i,j] == 255), None)
    if currMax != None:
        if currMax > maxX:
            maxX = currMax
```

However, 2 * width(1280) * height(720) steps(1.8 million) are still needed to move through the image. For an image of the size of the references, this takes 1-2 seconds. The entire rest of the program takes approximately 0.1 seconds, so this is an incredibly slow part of the process by comparison. However, it can be improved.
By changing the step size passed to the algorithm, we can step through a much smaller amount of points, having to do less comparisons, for nearly equivalent accuracy. It is recommended to use an odd value for the step amount. See figure 12 for an example of the computational differences for different step sizes.

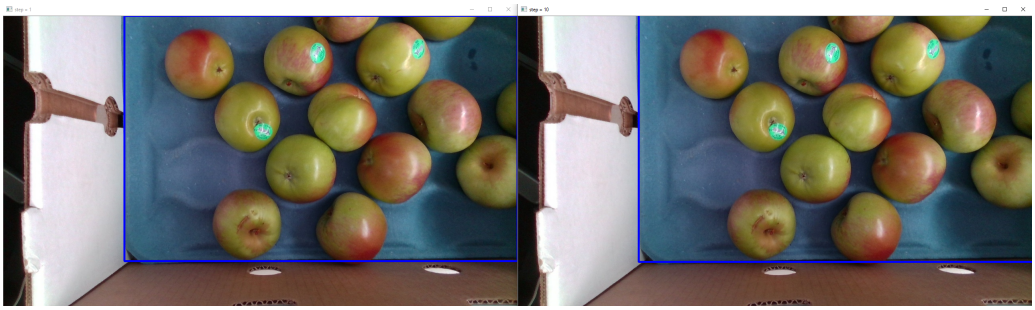


Figure 2: Side by side comparison of bounding boxes for step=1 and step=11

For figure 2, the detected Min/Max points were an average of 3 pixels off from the correct results, however, program execution was around 50 times faster. The bounding box detection happens in approximately 0.03s, making it usable with the rest of the other Open-CV functions in terms of speed.

Using a bounding box is necessary for apple detection, as the walls of the cardboard box have extremely similar HSV (Hue, Saturation, Value) values to the apples (See figure 9), meaning that many false positives generate on the walls of the box. To negate these false positives from occurring, some way of removing the walls from the cardboard box was necessary. After noticing that the apple tray was consistent across all the images, bounding box detection was implemented.
Using a bounding box also allows for lower bound values, meaning an increased detection rate for occluded and shadowed apples.

3.2 Apple Detection

The input image is stored and read into the program in BGR (Blue, Green, Red) format. A copy is generated and left in BGR for the output, and then the image is converted to HSV . Green is easy to determine bounds for, as its Hue is around 60. Therefore bounds can be estimated to be 50 to 70. However, red is centered on 0, meaning that the lower bound is estimated at -10, or 170. The inRange function from Open-CV does not implement -ve handling, so two masks are required.
Once the two masks are generated, they are added together (the equivalent of a bitwise OR), and initial threshold image is ready for processing. See figure 4 for an example of the combined masks, or figure 3 showing the green and red individual masks.

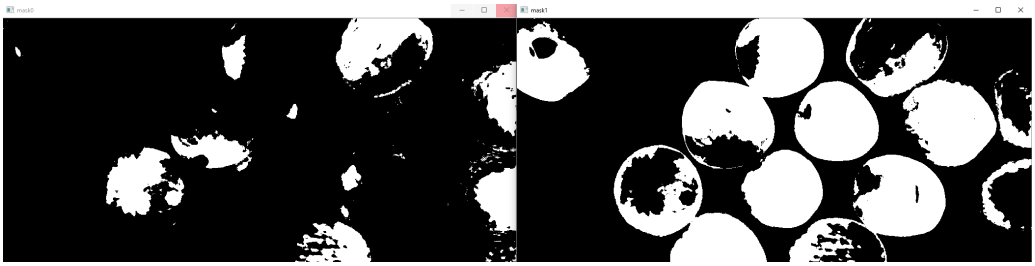


Figure 3: Green and Red masks for Reference Image 0

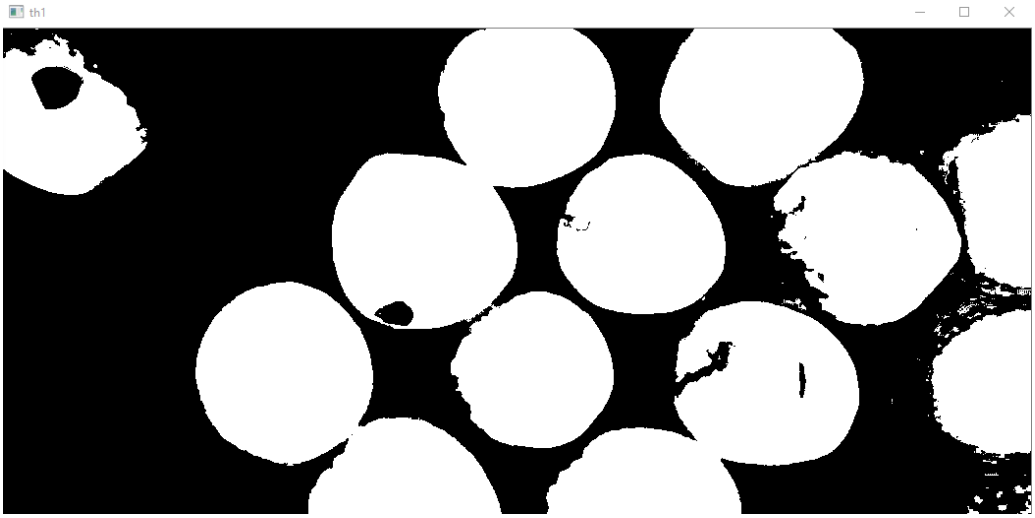


Figure 4: Combined masks for Reference Image 0

3.3 Hull and Contour Filling

In order to identify apples in high shadow areas, a low 'value' must be used for the lower bound of the image threshold. This introduces reasonably large levels of noise and disrupts the continuity of the detected contours, in worst cases resulting in contours within contours.
To decrease noise, and remove 'inception' contours, a contour detect is run, however, it uses the RETR_EXTERNAL flag to only return the external contours. The returned external contours are filled with white, removing any black spots within detected contours.
To smooth out the detected contours, the hull operation is used. If the area is less than the estimated area of one apple (To prevent the hull operation from working on multiple apples), the hull operation is applied. This detects concavity defects, and attempts to correct them, resulting in a smoother shape. This means that apples with stickers on their edges, or other shape defects resulting in odd shaped blobs, are corrected for smoother centroid generation, and linear area erosion. See figure 5 for an example of before and after the hull and contour fill operations are applied.
This process was implemented over using a dilation erosion combo is due to erosion's unwillingness to split connected blobs. A dilate operation connects two blobs, generally of blobs close to each other, or near the edge, and multiple erosion passes are unable to separate the two blobs again. This means that frequently, if the two blobs are not of equal size at the start of the operation, the smaller one is eroded until it is under the area limit, and the calculated centroid is generally somewhere in the middle of the two apples. Using the hull and contour fill process means that no size change occurs, meaning that close but separate blobs stay separate, and are able to be individually detected by the later parts of the algorithm

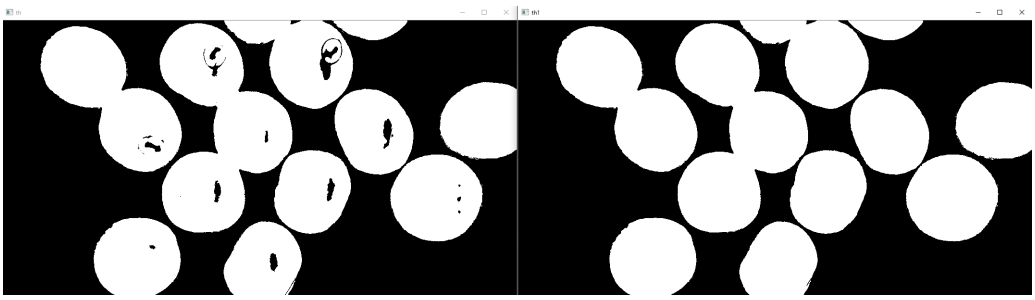


Figure 5: Before and after the hull and contour fill operations

3.4 Decaying Area Checks

Having generated a bounding box, and with a mask for likely apples, apple detection begins. The basic program loop is:

```
i = 0
while 255 in th1:
        i += 1
        th1 = cv2.erode(th1, element)
        # Area Filter Process
        # ---
        if i > 10:
            print('i > 10, breaking')
            break
```

i is a loop tracker, used for tracking how many times the while loop has run and to check that the program isn't running into infinity. In practise, the program always successfully completes before the break statement is called, or, before 10 iterations.
The use of the in statement in the while loop is used here to check if 255 (white) is still present in the thresholded image. If there is no 255 in the image, there is no white, and the in inbuilt function returns false. If white is present in the image, i is incremented, the image is eroded, and the area filter process begins.

```
contours, hierarchy = cv2.findContours(th1,cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
for x in range(len(contours)):
    cnt = contours[x]
    area = cv2.contourArea(cnt)
    M = cv2.moments(cnt)
    if area > (1000 - 200*passes) and area < 15000:
        if M['m00'] != 0:
            cx = int(M['m10']/ M['m00'])
            cy = int(M['m01']/ M['m00'])
            if mask[cy,cx] == 255:
                cv2.circle(output, (cx+minX,cy+minY), 5, getColour(x), 2)
                cv2.circle(output, (cx+minX,cy+minY), 80 , getColour(x), 2)
                cv2.drawContours(th1, contours, x , (0), -1)
                apples += 1
```

The area filter process starts by detecting all the contours on the passed input image. Then, the program performs the following steps on every contour. First, its area and moments are calculated using their respective Open-CV functions. In the early passes, blobs can be linked together, resulting in extremely large areas. This is why and upper bound is used, to prevent the 'detection' of groups of apples. As the amount of passes decreases, the area of each blob decreases as it is eroded. This results in some blobs having areas of less than 100 if they are on the edge of an image. This is why the lower bound for area decreases with the amount of passes, to allow the decreased area to be 'detected'. This is why a upper and lower bound is necessary for the area filtering.
Next, the centroid is calculated using formulas provided on the Open-CV documentation site.
The resulting centroid is evaluated to check that the detected contour is centered on a likely position of an apple, and, if it is, three draw operations are performed. First, the centroid of the contour is drawn on the output image. This is based on the assumption that the erode process happens evenly over the contour, meaning that the centroid of the eroded contour is the same as the centroid of the original apple. This is not always the case, as the presence of stickers throws off the centroid location during the erosion process, but it is usually pretty close.

Figure 6: Centroid drawn on each apple in Reference Image 1

Second, a larger circle describing the outline of the apple is drawn on the output image. The outline is centered on the centroid, and is a fixed radius, so it acts as an indicator rather than a accurate description of the outline of the apple. Observe the top 2 apples in figure 7 for an example.

Figure 7: Circles drawn on each apple in Reference Image 1

Finally, the contour that was detected is removed from the original threshold image, so that it is not double processed, and does not intrude on the detection of other objects in the image.
Finally, the sum of apples is increased by one.

Accuracy of circles drawn could be improved by rerunning thresholding and contour detection on the areas within the circles drawn on each apples location. (ie. within the circles drawn in figure 7) Or by multiplying by some factor of its area - centroid would move

Reference Image	Detected Apples	Apples
0	14	14
1	14	14
2	12	12
3	14	14
4	14	14
5	12	12
6	12	12
7	12	12
8	12	12
9	14	14
10	13	13
11	13	13
12	15	15
13	12	12
14	15	15
15	12	12
16	15	15
17	12	12
18	15	15
19	13	13
20	16	18
21	17	17
22	16	16
23	18	18
24	18	18
25	16	17
26	18	18
27	17	17
28	18	18
29	17	18

Table 1: Detected / Real apples for every image in the reference dataset

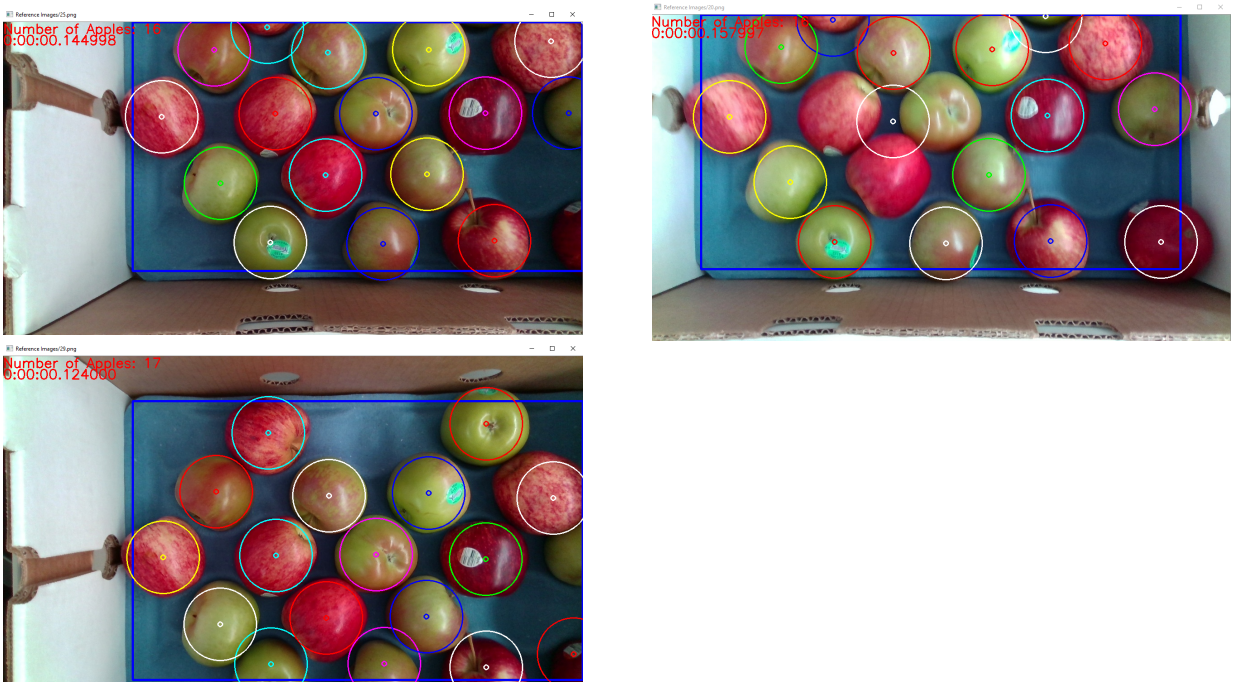


Figure 8: All incorrectly detected images

4 Results

As can be seen from figure 8, there are two unique issues with the reported algorithm, which shows itself over 3 images in the reference data set.

The first issue is seen in the left hand column of figure 8, on the right hand of the two images in the column. There are cases of red and green apples being so shadowed by the edge of the cardboard box and the lighting placement that their saturation and value are too low to be picked up by the threshold lower limits. This could be solved by lowering the lower bounds of the threshold, however, this results in significantly increased noise pickup.

The other unique issue is observed in the right hand column of figure 8. The central group of 4 similarly coloured apples are detected as a single object, and cannot be split by erosion before the area of the combined blob is under the upper area threshold. This could be solved by running the entire algorithm over the separate red and green masks, rather than the single combined one, however, apples with red and green colour content are often double processed.

There are also cases in the reference images where the apples are so far occluded that they are only shadows. For example, the lower left shadow in Reference Image 21. The algorithm does not detect the presence of a shadow as it does not have the correct colouration. For an 'apple' to be successfully detected, it must have colour present for the thresholding to pick it up, and be large enough that it is not classified as noise. Therefore, the quoted 'real' number of apples may be different to the markers metric. Each image executes in an average of 0.15 seconds, according to the datetime module. The algorithm does not distinguish between red or green apples, and does not rely on any pre-built machine learning models.

If the algorithm were to be tested on images outside the set used in the reference images, perhaps with a different lighting environment or similar, it is likely that the algorithm would under perform compared to the the results presented here. This is because the algorithm has been extremely tuned for use on the reference data set. This included creation of a rudimentary optimizer, and the use of a function to test single value changes against the entire data set.

Technically, the program is a blob within bounding box detector, as there is no feature recognition for apples. If a similar coloured object was placed in an equivalent bounding box, similar results could theoretically be obtained.

5 Conclusion

The objectives of this assessment was to develop an algorithm that can count and show the amount of apples in an image, using an appropriate programming language and machine vision tool set. This report has summarized the algorithm developed which best fulfills these requirements. The reported algorithm is not perfect, with a 437 out of 440 successful detection rate. The algorithm has a high likelihood of being over tuned for this set of example images, however, without more images to test against, it is difficult to tell.

The reported location of the center and outer diameter are estimations, however, they are accurate enough to be used for further processing. Once a location is found, a repeat threshold operation on a much smaller area centered on the estimated location of the apple could be performed, to find the exact outline using contours, and the centroid using contour moments. This would provide more accurate location data for robotic manipulators, at the cost of increased processing time.

Further improvements could be made by separately processing the red and green threshold images, which would allow for greater recognition of the apples colour and greater detection of occluded apples. In addition, a feature recogniser could be added, for the sake of saying the process includes machine learning, and some sort of trainer could be added for improved detection. However, with the 3 unique situations provided in the reference images, and only 440 points to train on, there would be no real value to adding any sort of machine learning without an external database.

This assignment was an excellent exercise that improved both my programming skills, and my experience with machine vision. There are still improvements that could be made, and the success of the algorithm is dependent on certain criteria. However, it can successfully detect shadowed apples, filter for noise, and provide an accurate count for the user 99.3% of the time.

6 Supporting Images

List of Figures

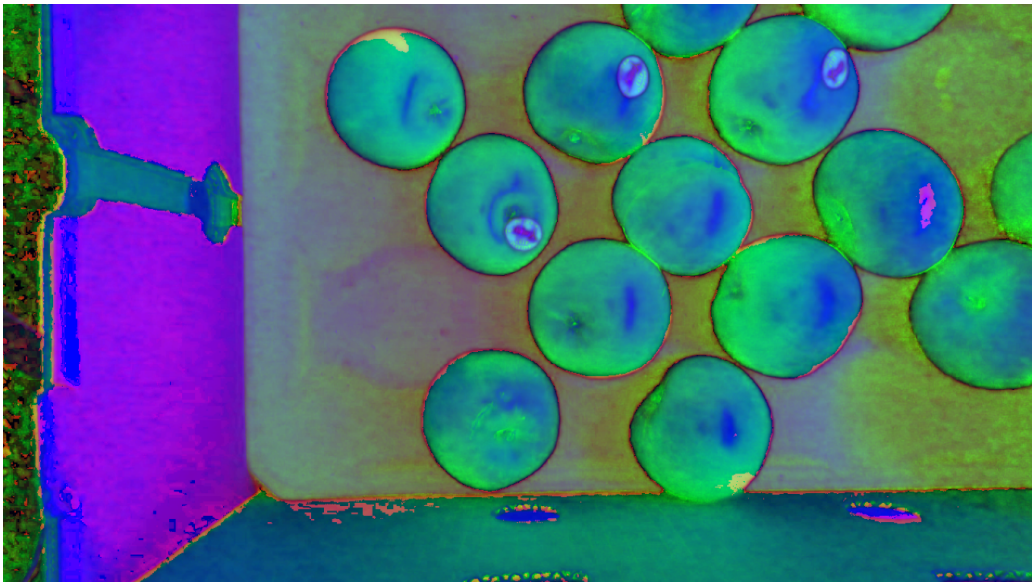


Figure 9: An RGB representation of the HSV values for Reference Image 0

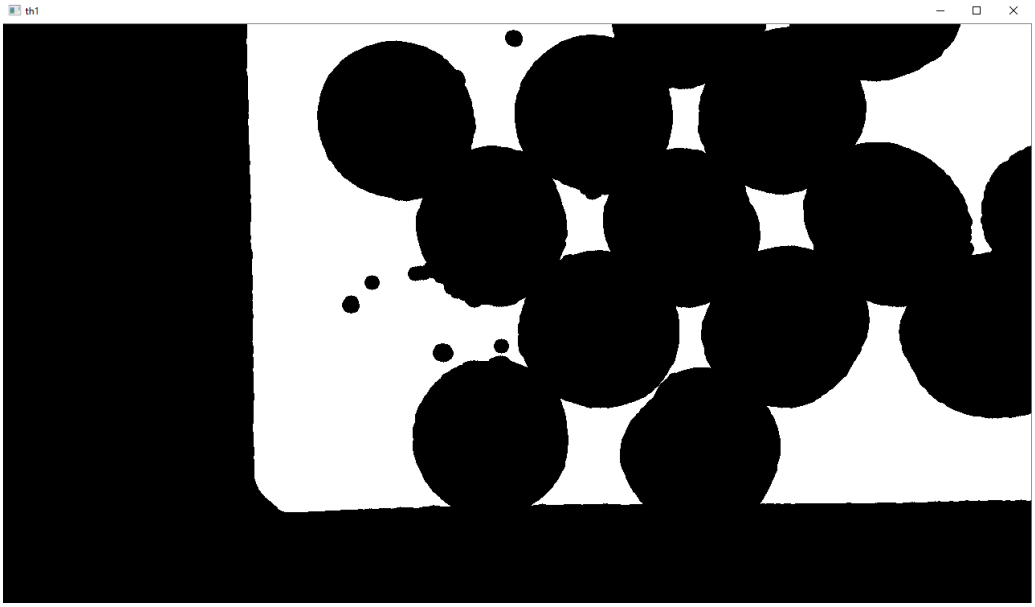


Figure 10: Thresholded image for bounding box detection after noise reduction (Reference image 0)

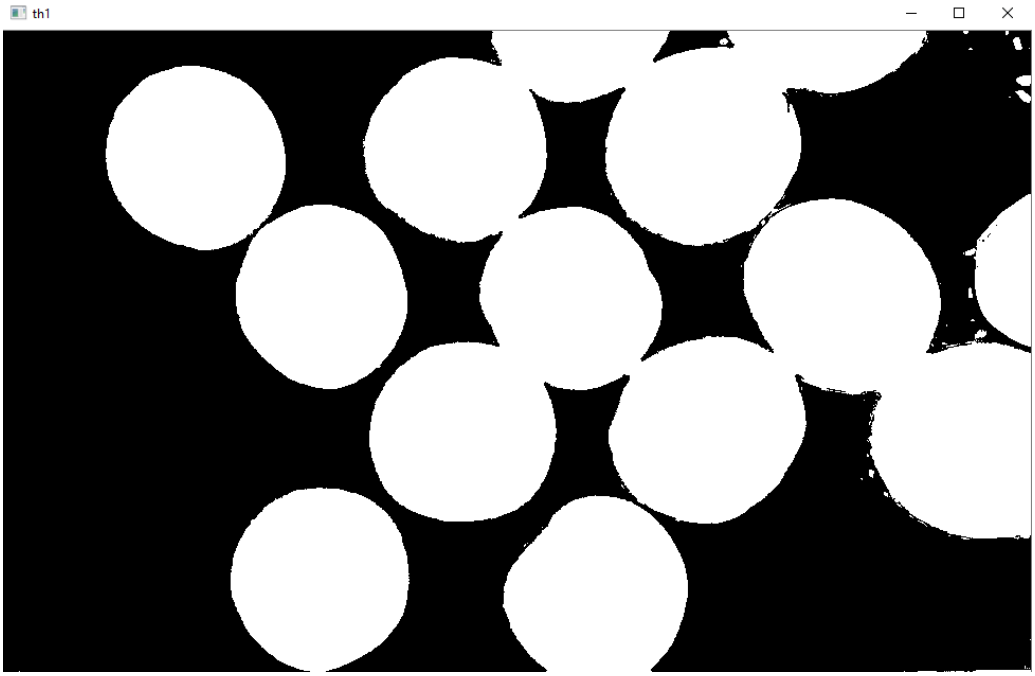


Figure 11: Thresholded image for apple detection after contour filling (Reference Image 0)

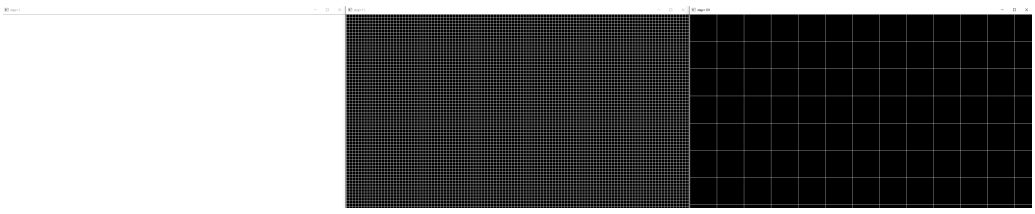


Figure 12: Side by side comparison of scanned lines for step=1, 11, 101