Assignment 3 – 282762 Robotics and Automation

Report

Machine Vision

Contents

1. Introduction

This report deals with the development of a Python-based computer vision system, implemented through the script mv_apples_main.py, which is designed to detect and count apples from a provided dataset. Python and OpenCV have been utilized to achieve such.

The primary objective was to create an algorithm that could accurately count the number of apples in an image and indicate their locations. The project utilizes standard machine vision pipeline by following the image extraction, preprocessing, and feature extraction, and outputting order. The developed program processes images from a specified "data" sub-directory autonomously, without user interaction.

The methodology involved several key steps. Image extraction was performed on a dataset consisting of 30 validation images, including both green and red apples. Preprocessing steps, such as extending image borders and applying YUV colour thresholding, were used for enhancing the accuracy of apple detection. Then feature extraction was implemented using the HoughCircle Transform to identify circular shapes representing apples.

1.1. Background Theory

Machine Vision is defined as the ability for robotics to “see” and process visual data for various processes. It is applied across almost all industries for various applications such as visual inspection, process control, automated measurements or in our case counting apples in a tray. The application usually allows robots and systems to operate autonomously and make decision based on provided visual data. [1, 2]

OpenCV (Open-Source Computer Vision Library) is an open-source software library that provides extensive tools and functions for real-time computer vision and image processing, widely used in academic research and commercial applications. [3]

2. Methodology

2.1. Overview

The project consists of the python script mv_apples_main.py and a dataset of 30 images of apples. The script allows the user to analyse either a single or collection of images to detect apples within the dataset. The general pipeline of computer vision applications has been followed to find each apple:

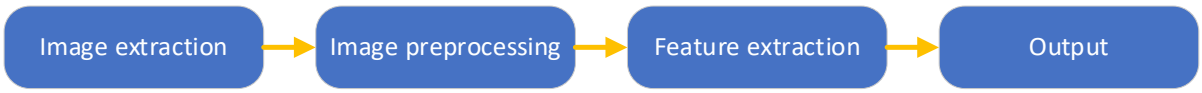


Figure 1 - Machine Vision Pipeline

2.2. Image Extraction

The provided main dataset consist of 30 validation images including both green, red and mixed batches of apples. The other datasets provided have not been used as per instructions as they are just variations of the main dataset.

2.2.1. Dataset Directory Structure

The file directory structure is as follows:

Assignment 3 – Machine Vision /dataset

/validation

/0.PNG … 30.PNG

/opencv_ws

/my_apples_main.py

To test for a single image, the following code has been used where the filepath is defined and then saved into an image using im.read. A second copy of the original image is saved to later output the circles onto a non-processed image.

```
    # Load the pre-segmented black and white image
    filepath = os.getcwd() + "/dataset/validation/17.png"
    img = cv.imread(filepath, cv.IMREAD_COLOR)
    original_image = img
```

For error handling a short statement has been implemented to output an error when the image could not be read from the directory.

```
if img is None:
        print(f"Error: Unable to read the image at '{filepath}'")
        return
```

In the single image ‘mode’, the image does not get saved to disc as it is mostly for development purposes.

The second method is iterating through all 30 images to read, process and write.

```
# Directory where images are located
image_dir = "C:/Users/nikla/OneDrive/Documents/Mechatronics/282.762
Robotics/Assesment 3 - Machine Vision/dataset/validation"

# Iterate through images "1.png" to "30.png"
for i in range(30):
    filename = os.path.join(image_dir, f"[4].png")

    # Load the image
    img = cv.imread(filename, cv.IMREAD_COLOR)
    original_image = img

    if img is None:
        print(f"Error: Unable to read the image at '{image_dir}'")
        continue
```

2.3. Image Preprocessing

For any circle detection method to work, the image needs to be pre-processed. This overall approach is similar across the different methods, but certain processing tools may work better with one or the other. For example, the blob methods require downscaling the image to a lower resolution for the blobs to be detected properly. After testing different methods with several image processing techniques, the following image processing structure has been found to work successfully with the chosen Hough transformation:

Figure 2 - Image Preprocessing Pipeline

2.3.1. Extend Image

The *extend_image* function adds a 200px white border around the whole image. This has been added after discovering that the Hough Transform has issues finding circles that are partially extending over the edge of the image, in our case a cut of apple. Figure 3 is an example of the difference of before and after the extension.

Figure 3 - Results before extension (left) and Results after extension (right)

The original image and extension size are passed into the extend_image function.

```
def extend_image(image, extension_pixels):
```

The new dimensions are then determined by extracting the current size using image.shape and adding the margins to both vertical and horizontal.

```
    # Determine new dimensions
    height, width = image.shape[:2]
    new_height = height + 2 * extension_pixels
    new_width = width + 2 * extension_pixels
```

A blank white image with the extended white pixels is then created using np.ones.

```
    # Create a new blank image with white pixels
    extended_image = np.ones((new_height, new_width, 3), dtype=np.uint8) * 255
```

The following code then offsets the original image based on the extension applied and pastes it onto the new blank image to create the final image for this stage. The extended image is then returned to main.

```
    # Calculate position to paste original image
    x_offset = extension_pixels
    y_offset = extension_pixels

    # Paste the original image onto the new blank image
    extended_image[y_offset:y_offset + height, x_offset:x_offset + width] =
image
```

```
return extended_image
```

2.3.2. YUV Colour Thresholding

The thresholding of the image is the most impactful step for the Hough Transform as it separates the important aspects of the image to be analysed in a binary image. In this case the YUV has been chosen after trying RGB, HSV and YUV based on how it interacts with the given datasets colour range. Once converted into YUV space, the image is masked based on lower and upper limits and then converted into a binary image using the cv.threshold function. A processed image for each step can be found under Appendix B – YUV Conversion to Appendix G – Binary Image with Median Blur and Gaussian Blur.

The structure of the threshold_yuv function is as follows:

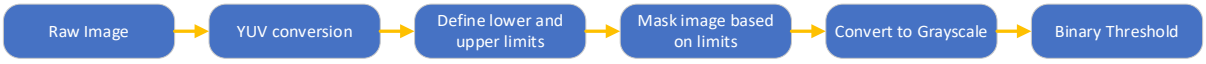


Figure 4 - threshold_yuv Pipeline

The image if first converted into YUV colour space.

```
    # Convert image to YUV colour space
    img_yuv = cv.cvtColor(img, cv.COLOR_BGR2YUV)
```

After converting the image into YUV space, the limits have been determined using imagej.exe to find the correct threshold to separate the apples from the background.

	Y	U	V
Lower	0	121	0
Upper	255	153	138

```
    # Define the lower and upper bounds for YUV channels based on the
specified ranges
    lower = np.array([0, 121, 0], dtype=np.uint8)
    upper = np.array([255, 153, 138], dtype=np.uint8)
```

A mask is then created using cv.inRange and the previously created limits. The image is then converted to grayscale to be converted to a binary image using the thresholding function.

```
    # Create a mask using inRange function to threshold YUV image
    mask = cv.inRange(img_yuv, lower, upper)
    # Apply the mask to original image
    result = cv.bitwise_and(img, img, mask=mask)
    # Convert the result to grayscale for thresholding
    result_gray = cv.cvtColor(result, cv.COLOR_BGR2GRAY)
    _, binary_image = cv.threshold(result_gray, 1, 255, cv.THRESH_BINARY)
```

2.3.3. Blur in detect_circles

In the detect_circles function, the last bit of image processing is done. This has been implemented to enhance the edge detection during the feature extraction. First the image is converted to grayscale (required by HoughCircles) and then a median and a gaussian blur are applied. The blur parameters have been tuned on the given dataset. The values in the code are values showing the highest success rate. Exemplary images of the different blurring stages can be found under Appendix F – Binary Image with Median Blur and Appendix G – Binary Image with Median Blur and Gaussian Blur.

```
    # Ensure input image is single-channel (grayscale)
    gray = np.where(img > 0, 255, 0).astype(np.uint8)
    mblur = cv.medianBlur(gray, 5)
    gblur = cv.GaussianBlur(mblur, (3, 3), 2)
```

2.4. Feature Extraction

The features of interest are the circle shaped apples. Several methods such as HoughCircles [5], SimpleBlobDetection [6] and findContours [7] have been evaluated.

HoughCircles and SimpleBlobDetection have shown the most promising results. While SimpleBlobDetection worked reliably after thresholding and downscaling the image for most apples, it struggled with apples that have white spots inside (predominately from the label on the apples).

Figure 5 - Example of apple with label

Morphological transformations [8] have been applied to manipulate the image but no ideal ratio has been found. Blob detection would have been pursued further if HoughCircles would have not given promising results.

HoughCircles has shown the most promising results. The tuned parameters for the given dataset are:

- dp = 0.95
- minDist=75
- param1=50
- param2=10
- minRadius=75
- maxRadius=100

The most impact on finding circles are param2 and dp. Param2 determines how strong the edges in an image should be to be considered part of a circle. Decreasing param2 leads to lowering the threshold and increasing the chance of a weaker edge to be recognised as a circle. Dp specifically affecting the resolution of the accumulator used in the algorithm, meaning a higher dp leads to larger steps are taken during the Hough transform, which lead to inaccurate and more results due to lower resolution. [5]

The feature extraction pipeline is as follows:

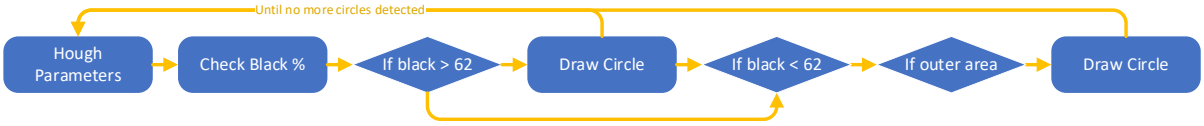


Figure 6 - HoughCircles Detection Pipeline

First the HoughCircles function is called with the set parameters.

```
    # Apply Hough Circle Transform with adjusted parameters
    circles = cv.HoughCircles(
        gblur,
        cv.HOUGH_GRADIENT,
        dp=0.95,
        minDist=75,
        param1=50,
        param2=10,
        minRadius=75,
        maxRadius=100
    )
```

Then the bounding of the circle is calculated within a for loop that iterates through until no more circles have been found. After the region of interest (ROI) has been extracted using the bounding box.

```
  if circles is not None:
        circles = np.uint16(np.around(circles))

        # Draw detected circles on the original image
        draw = original_image.copy()  # Use original image directly for
drawing
        for idx, circle in enumerate(circles[0, :], start=1):
            center = (circle[0], circle[1])
            radius = circle[2]

            # Calculate the bounding box for the circle ROI
            x1 = max(0, circle[0] - radius)
            y1 = max(0, circle[1] - radius)
            x2 = min(original_image.shape[1], circle[0] + radius)
            y2 = min(original_image.shape[0], circle[1] + radius)
            # Extract ROI around the circle (considering boundary conditions)
            roi = gray[y1:y2, x1:x2]
```

In the next step the ratio of black to white pixels within a circle is calculated. As the input image is a pre-processed binary representation of the original image, every apple should be predominantly black (Appendix D – Thresholded Binary image). The only exception to this are partial apples. As the image has been extended using white pixels, the black to white ratio would be far lower. To mitigate such, the script detects if a circles radius exceeds the boundaries of the original image by looking at

its centre position, radius and relative edge of the frame and adjusts the threshold of the black to white ratio to 10%. If either of the two is satisfied, it draws a red circle with green centre point, as well as its number, black percentage and radius.

```
            # Calculate percentage of black pixels (apples) within the ROI
            black_percentage = np.mean(roi < 128) * 100  # Assuming threshold
of 128 for binary

            # Draw circle only if majority of pixels are black (apples)
            if black_percentage > 62:
                # Draw the full circle in red
                cv.circle(draw, center, radius, (0, 0, 255), 3)

                # Draw the center
                cv.circle(draw, center, 2, (0, 255, 0), 3)

                # Annotate with black pixel percentage
                cv.putText(
                    draw,
                    f"Circle {idx}: | Black Pixels: {black_percentage:.2f}% |
Radius: {radius:.2f}",
                    (circle[0] - radius, circle[1] - radius - 20),
                    cv.FONT_HERSHEY_SIMPLEX,
                    0.4,
                    (0, 0, 255),
                    1
                )

            if 10 < black_percentage < 62:
                img_height, img_width = img.shape[:2]
                min_radius = 50
                if (center[0] + radius > img_width - extension_pixels or
center[0] - radius < extension_pixels or center[1] + radius > img_height -
extension_pixels or center[1] - radius < extension_pixels):
                    cv.circle(draw, center, radius, (0, 0, 255), 3)

                    # Draw the center
                    cv.circle(draw, center, 2, (0, 255, 0), 3)

                    # Annotate with black pixel percentage
                    cv.putText(
                        draw,
                        f"Circle {idx}: | Black Pixels:
{black_percentage:.2f}% | Radius: {radius:.2f}",
                        (circle[0] - radius, circle[1] - radius - 20),
                        cv.FONT_HERSHEY_SIMPLEX,
                        0.4,
                        (0, 0, 255),
                        1
```

```
                    )
```

2.5. Output

Depending on if the single or multi image mode is selected the output is either displayed onscreen or saved to file respectively.

The following code shows the calling of the discussed functions to process and analyse the image and display the image with circles detected. If no circles have been detected it outputs a prompt saying so without displaying the image.

```
    # Threshold and process the image
    binary_result = threshold_yuv(img)

    # Detect circles in the processed binary image and draw on the original
image
    result_image = detect_circles(binary_result, original_image)

    if result_image is not None:
        # Display the result (convert to BGR before display)
        cv.imshow("Circles Detected", result_image)
        cv.waitKey(0)
        cv.destroyAllWindows()
    else:
        print("No circles detected or error occurred.")
```

In the batch image mode, the program iterates through the set number of images and executes the code like the single image mode. The main difference is that no images are displayed but rather saved to the origin file directory with a suffix. Prompts to update the user on the status and execution of the program have also been implemented.

```
# Iterate through images "0.png" to "29.png"
for i in range(30):
    filename = os.path.join(image_dir, f"{i}.png")

    # Load the image
    img = cv.imread(filename, cv.IMREAD_COLOR)
    original_image = img

    if img is None:
        print(f"Error: Unable to read the image at '{image_dir}'")
        continue

    img = extend_image(img, extension_pixels)
    original_image = extend_image(original_image, extension_pixels)

    # Threshold and process the image
    binary_result = threshold_yuv(img)
```

```
    result_image = detect_circles(binary_result, original_image)
    # Save the resulting image with circles detected
    output_filename = os.path.join(image_dir, f"{i}_circle.png")
    cv.imwrite(output_filename, result_image)

    print(f"Processed and saved {output_filename}")

print("Processing complete.")
```

3. Results

3.1. Preparation and Processing

To analyse the accuracy of the model, the apples have been counted manually as a reference to compare against the detected apples. Two different methods have been chosen to evaluate the accuracy. Both methods separate complete to partial apples but in method one every partial apple is counted as partial whereas method two only treats apples that are less than half in frame as partial. This has been done to understand both best case scenario where all apples are fully in frame and a compromise for the current dataset where apples may be cut off. Examples of partial apples can be found under Appendix H – Partial Apple Examples.

3.2. Model Evaluation

The results are based on the provided dataset of 30 images. For methods one, the calculated accuracy for complete apples is at 100.0% and for partial apples at 68.4% (see Appendix I – Results Table Full/Partial Apples (Method 1) for a breakdown of each image).

Method two provided an accuracy of 98.2% for every apple that is over 50.0% in the frame and a 55.9% accuracy on apples cut off more than 50.0% (see Appendix J – Results Table Full/Partial Apples (Method 2) for a breakdown of each image).

The detection of complete apples for the provided dataset is very good with 100% and still very high at 98.4% including partial apples that are over 50.0% in frame. The percentage of detected partial apples of 68.5% and 55.9% respectively may seem low at first but this includes all partial apples even if they show as a fraction of a complete apple in the image. Furthermore, this can be avoided by setting up the image processing station to only take full images of a tray in an industrial application. In that case the developed model seems very promising based on the 100.0%/98.4% accuracy.

The worst performing images were 13.PNG and 15.PNG with an accuracy of 90.9%. As seen those images are very similar and one apple that is partially cut could not be detected.

Figure 7 - Result 13.PNG

Figure 8 - Result 15.PNG

Analysing the results further a pattern can be seen where the program struggles detecting a single red apple in a batch of several as seen in Figure 7 and Figure 8. Further tuning on the thresholding and parameters may lead to a higher yield in also detecting such.

Regardless of those outliers the overall accuracy is within the expected range for the project scope. When looking at images that have a better framing, an accuracy of 100% has been observed as seen in Figure 9 - Result 21.PNG and Figure 10 - Result 3.PNG. The output of all images can be found under Appendix K – Results of all Images.

Figure 9 - Result 21.PNG

Figure 10 - Result 3.PNG

To understand the computational load of the program, 5 consecutive runs have been performed and the processing time of each has been recorded. This test has been performed on a modern desktop computer with a four-year-old mid-high-range CPU (AMD Ryzen 5800x) [4]. On average the batch of 30 images took 3.3 seconds or about 0.1 seconds per image.

When looking at the processing speed of the program, it seems reasonable to assume that less powerful devices such as a Raspberry Pi should be able to process the data in an adequate time in a production environment. Regardless this would need to be verified in field and adjustments may be made depending on the outcome.

4. Conclusion

This project consists of a Python script, mv_apples_main.py, to detect apples in a dataset of 30 images using computer vision techniques. The methodology included image extraction, preprocessing, and feature extraction. Key steps like image border extension and YUV colour thresholding were essential for accurate detection.

The model, utilizing the Hough Transform, achieved a 100% accuracy for complete apples and 98.2% for apples over 50% in the frame. Detection accuracy for partial apples was lower, highlighting a need for improvement during the image acquisition process.

The processing speed of 0.1 seconds per image indicates potential for real-time use, even on less powerful devices like the Raspberry Pi, though further testing is required.

In conclusion, the apple detection system is highly accurate and efficient, making it suitable for industrial applications. Future work could enhance partial apple detection and refine parameters for diverse apple batches as well as improving the image framing for the highest apple detection rate possible.

The project offered a great insight into Machine Vision and OpenCV. The different image processing techniques and feature detection functions allowed to get me a good understanding of the theory behind it and the actual application of such in an industrial setting.

5. References

[1] J. Lasky, "Computer vision," ed: Salem Press Encyclopedia of Science, 2019.

[2] "What Is Machine Vision?" © Intel Corporation. https://www.intel.com/content/www/us/en/manufacturing/what-is-machine-vision.html (accessed 15/05, 2024).

[3] "OpenCV About." OpenCV. https://opencv.org/about/ (accessed 15/05, 2024).

[4] "AMD Ryzen 7 5800X Desktop Processor." 2024 Advanced Micro Devices, Inc. https://www.amd.com/en/products/processors/desktops/ryzen/5000-series/amd-ryzen-7-5800x.html (accessed 13/05, 2024).

[5] "Hough Circle Transform." OpenCV. https://docs.opencv.org/4.x/da/d53/tutorial_py_houghcircles.html (accessed 14/05, 2024).

[6] "cv::SimpleBlobDetector Class Reference." OpenCV. https://docs.opencv.org/3.4/d0/d7a/classcv_1_1SimpleBlobDetector.html (accessed 14/05, 2024).

[7] "Contours : Getting Started." OpenCV. https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html (accessed 14/05, 2024).

[8] "Morphological Transformations." OpenCV. https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html (accessed 15/04, 2024).

Appendix

Appendix A – Block Diagram High Level Architecture Interaction

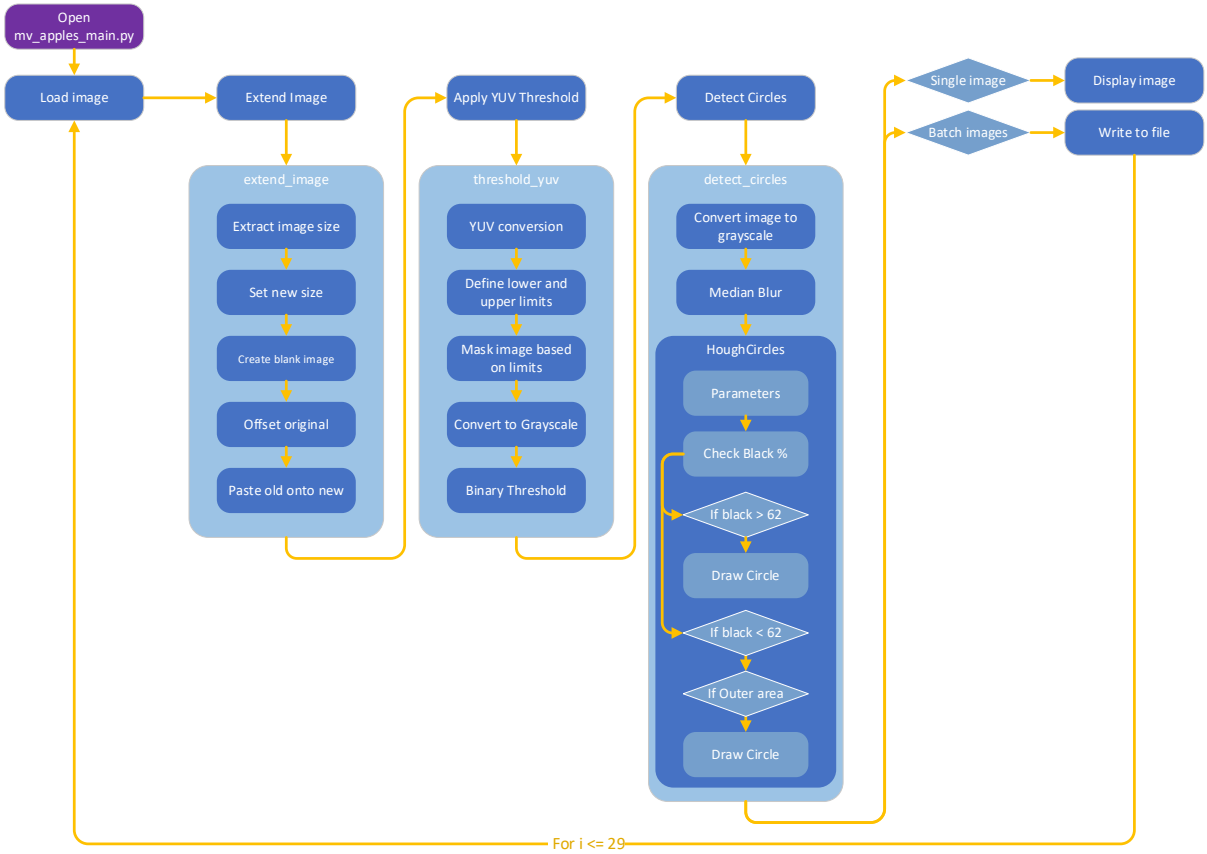


Figure 11 - Block Diagram High Level Architecture

Appendix B – YUV Conversion

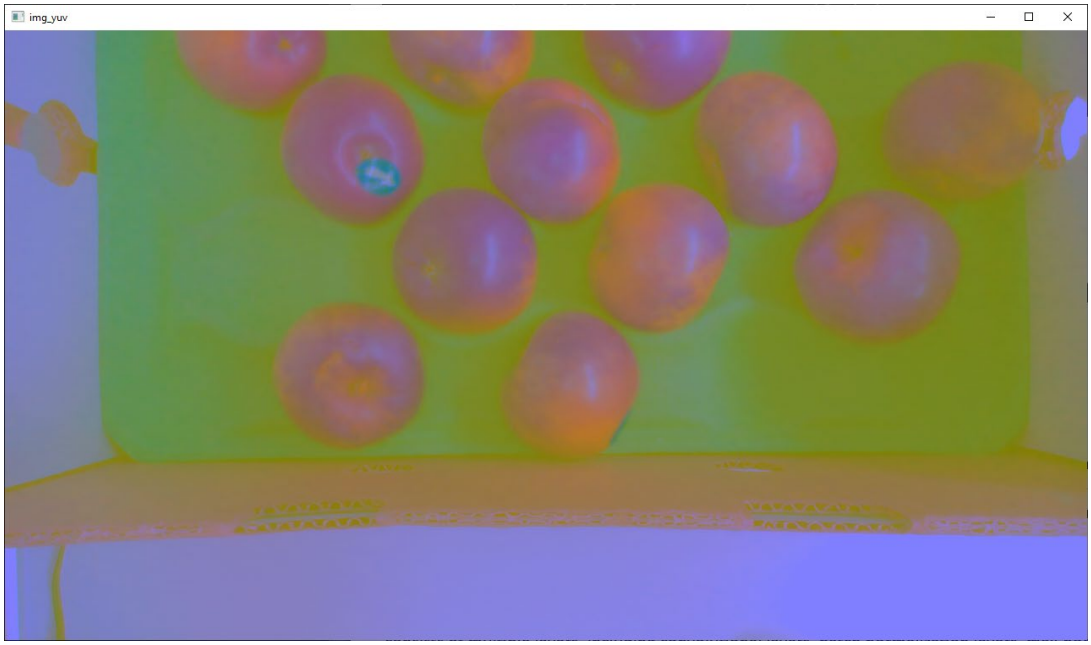


Figure 12 - YUV Conversion Image

Appendix C – Mask based on Colour Space Limits

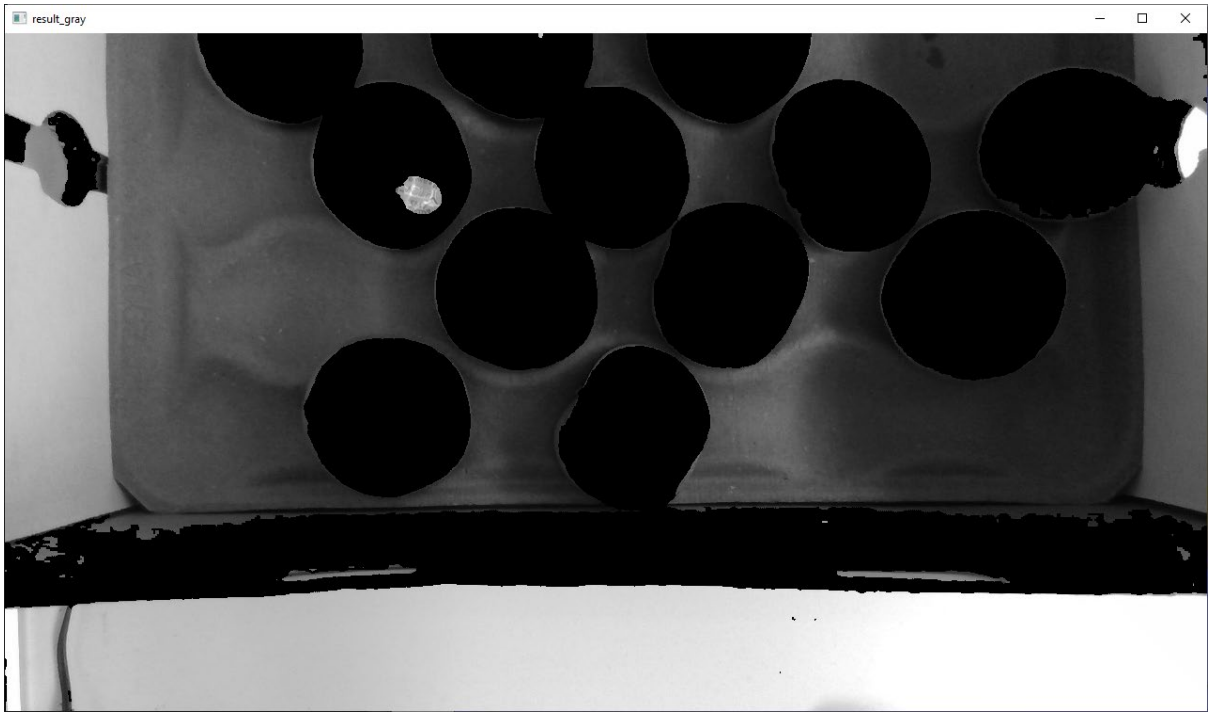


Figure 13 - Masked Colour Range Image

Appendix D – Thresholded Binary image

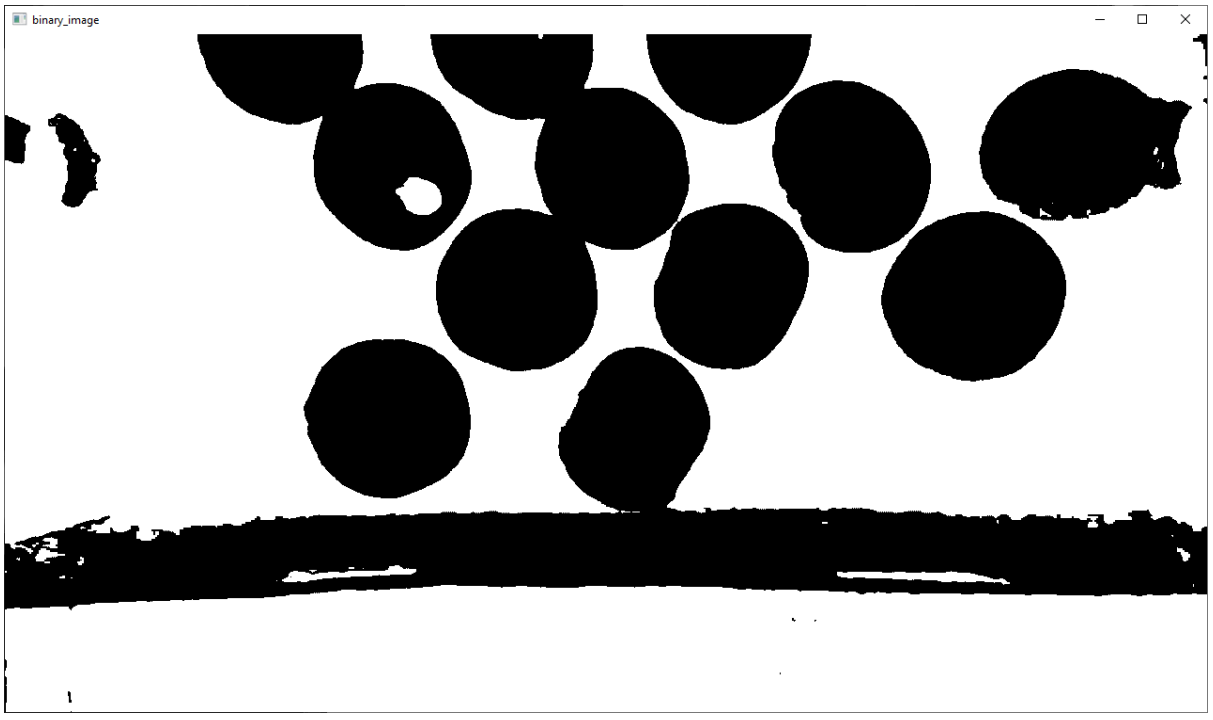


Figure 14 - Thresholed Binary Image

Appendix E – Extended Binary image

Figure 15 - Extended Binary Image

Appendix F – Binary Image with Median Blur

Figure 16 - Binary Image with Median Blur

Appendix G – Binary Image with Median Blur and Gaussian Blur

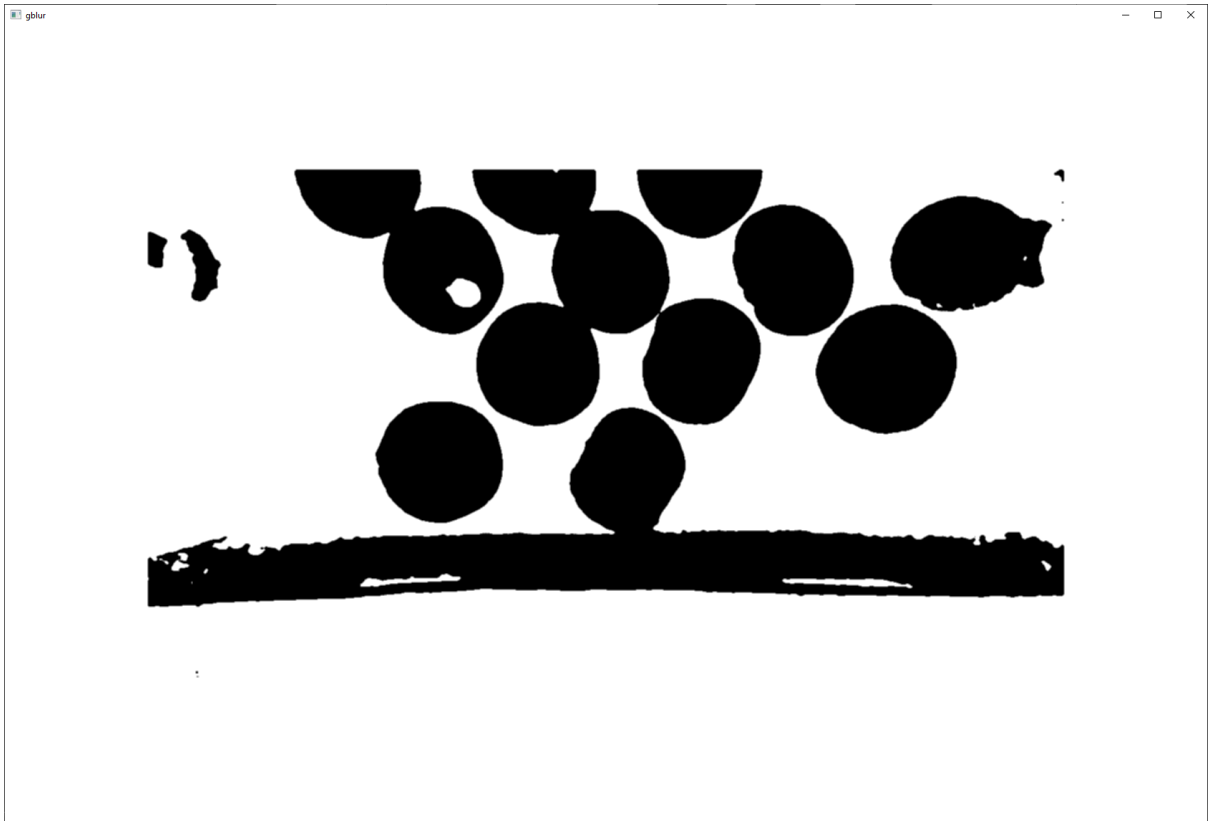


Figure 17 - Binary Image with Median and Gaussian Blur

Appendix H – Partial Apple Examples

Figure 18 0 Partial Apples Example Image

Appendix I – Results Table Full/Partial Apples (Method 1)

image	expected apples	found apples	cut off apples	found cut off apples	% of found apples	% of found cut apples
0	10	10	4	3	100	75
1	12	12	2	1	100	50
2	8	8	4	4	100	100
3	11	11	3	3	100	100
4	12	12	2	2	100	100
5	9	9	3	3	100	100
6	9	9	3	3	100	100
7	9	9	3	3	100	100
8	8	8	4	4	100	100

9	11	11	3	2	100	66.66666667
11	9	9	4	3	100	75
12	13	13	2	1	100	50
13	10	10	2	0	100	0
14	13	13	2	1	100	50
15	10	10	2	0	100	0
16	13	13	2	1	100	50
17	10	10	2	0	100	0
18	9	9	6	4	100	66.66666667
19	9	9	4	3	100	75
20	16	16	2	0	100	0
21	13	13	3	3	100	100
22	10	10	6	5	100	83.33333333
23	16	16	2	2	100	100
24	14	14	4	3	100	75
25	10	10	7	5	100	71.42857143
26	12	12	6	4	100	66.66666667
27	13	13	4	2	100	50
28	16	16	2	2	100	100
29	14	14	5	4	100	80
				Average	100	68.44006568

Appendix J – Results Table Full/Partial Apples (Method 2)

image	expected apples	found apples	cut off apples	found cut off apples	% of found apples	% of found cut apples
0	11	11	3	2	100	66.66666667
1	12	12	2	1	100	50
2	11	11	1	1	100	100
3	13	13	1	1	100	100
4	14	14	1	1	100	100
5	9	9	3	3	100	100
6	9	9	3	3	100	100
7	9	9	3	3	100	100
8	9	9	3	3	100	100
9	11	11	3	2	100	66.66666667
11	12	11	1	1	91.66666667	100
12	13	13	2	1	100	50
13	11	10	1	0	90.90909091	0
14	13	13	2	1	100	50
15	11	10	1	0	90.90909091	0
16	13	13	2	1	100	50
17	11	11	1	0	100	0
18	15	13	0	0	86.66666667	
19	13	12	0	0	92.30769231	
20	16	16	2	0	100	0
21	16	16	0	0	100	
22	16	15	0	0	93.75	
23	18	18	0	0	100	
24	14	14	4	3	100	75
25	14	14	3	1	100	33.33333333
26	16	16	2	0	100	0

27	15	15	2	0	100	0
28	18	18	2	2	100	100
29	17	17	1	0	100	0
				Average	98.14514508	55.90277778

Appendix K – Results of all Images


Circle 14: | Black Pixels: 17.49% | Radius: 76.00
Circle 6: | Black Pixels: 76.34% | Radius: 88.00
Circle 11: | Black Pixels: 84.23% | Radius: 86.00
Circle 4: | Black Pixels: 75.02% | Radius: 92.00
Circle 10: | Black Pixels: 82.49% | Radius: 92.00
Circle 7: | Black Pixels: 77.85% | Radius: 92.00


Circle 16: | Black Pixels: 78.42% | Radius: 97.00
Circle 2: | Black Pixels: 75.15% | Radius: 88.00
Circle 5: | Black Pixels: 83.66% | Radius: 86.00
Circle 12: | Black Pixels: 74.97% | Radius: 86.00
Circle 14: | Black Pixels: 73.80% | Radius: 99.00
Circle 3: | Black Pixels: 84.19% | Radius: 96.00
Circle 9: | Black Pixels: 94.42% | Radius: 88.00
Circle 8: | Black Pixels: 88.05% | Radius: 94.00
Circle 13: | Black Pixels: 89.54% | Radius: 92.00
Circle 1: | Black Pixels: 78.62% | Radius: 99.00

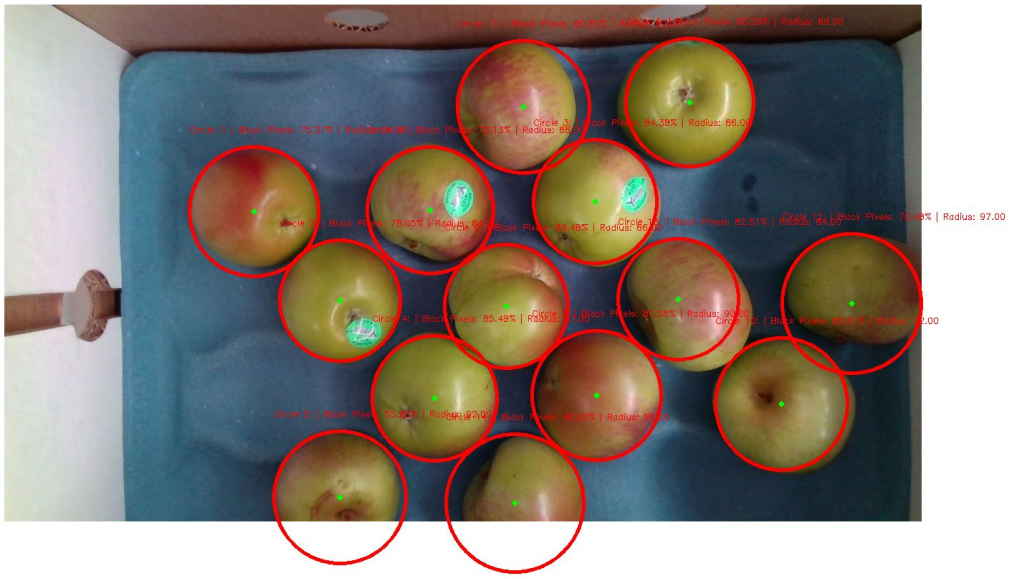

Circle 10: | Black Pixels: 52.90% | Radius: 91.00
Circle 15: | Black Pixels: 73.15% | Radius: 95.00
Circle 9: | Black Pixels: 76.61% | Radius: 99.00


Circle 16: | Black Pixels: 78.62% | Radius: 90.00
Circle 1: | Black Pixels: 65.01% | Radius: 93.00
Circle 17: | Black Pixels: 83.10% | Radius: 92.00
Circle 13: | Black Pixels: 83.23% | Radius: 93.00


Circle 12: | Black Pixels: 26.98% | Radius: 89.00
Circle 13: | Black Pixels: 35.69% | Radius:
Circle 5: | Black Pixels: 87.98% | Radius: 90.0
Circle 12: | Black Pixels: 91.75% | Radius: 84.00
Circle 13: | Black Pixels: 85.01% | Radius:
Circle 5: | Black Pixels: 71.70% | Radius: 90.00


Circle 13: | Black Pixels: 14.95% | Radius: 92.00
Circle 3: | Black Pixels: 75.22% | Radius: 96.00
Circle 2: | Black Pixels: 68.26% | Radius: 94.00
Circle 6: | Black Pixels: 72.03% | Radius: 94.00
Circle 8: | Black Pixels: 70.40% | Radius: 96.00
Circle 4: | Black Pixels: 78.51% | Radius: 89.00
Circle 7: | Black Pixels: 83.35% | Radius: 79.00

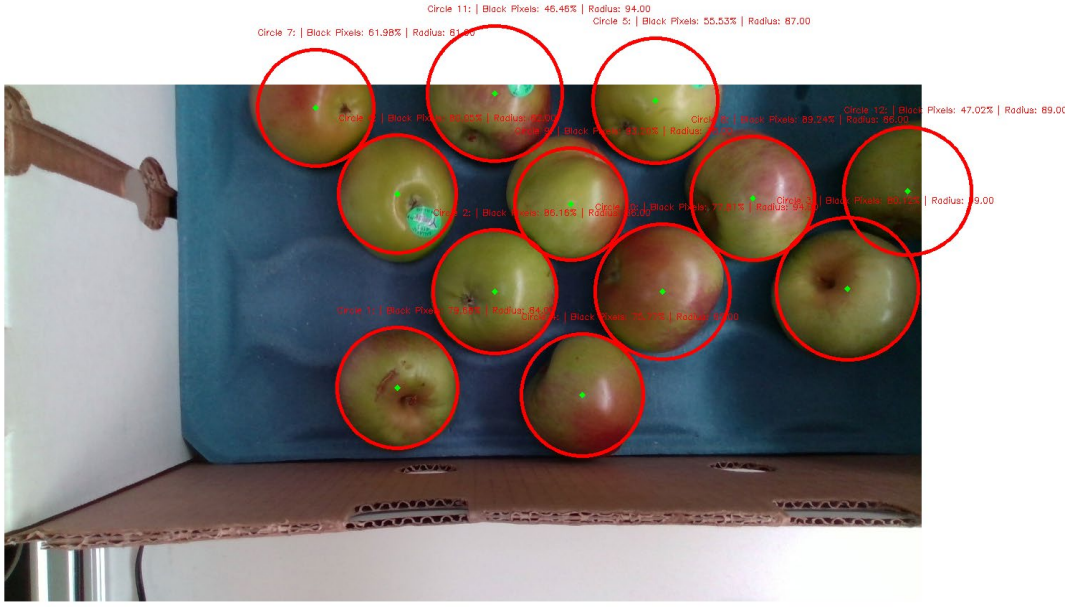

Circle 5: | Black Pixels: 62.30% | Radius: 92.00
Circle 6: | Black Pixels: 62.22% | Radius: 89.00
Circle 4: | Black Pixels: 66.91% | Radius: 87.00
Circle 12: | Black Pixels: 34.05% | Radius:
Circle 7: | Black Pixels: 81.97% | Radius: 87.00
Circle 1: | Black Pixels: 83.65% | Radius: 79.00
Circle 2: | Black Pixels: 84.85% | Radius: 84.00
Circle 3: | Black Pixels: 85.15% | Radius: 82.00




Circle 11: | Black Pixels: 39.18% | Radius: 98.00
Circle 5: | Black Pixels: 44.70% | Radius: 90.00
Circle 12: | Black Pixels: 53.01% | Radius: 77.00

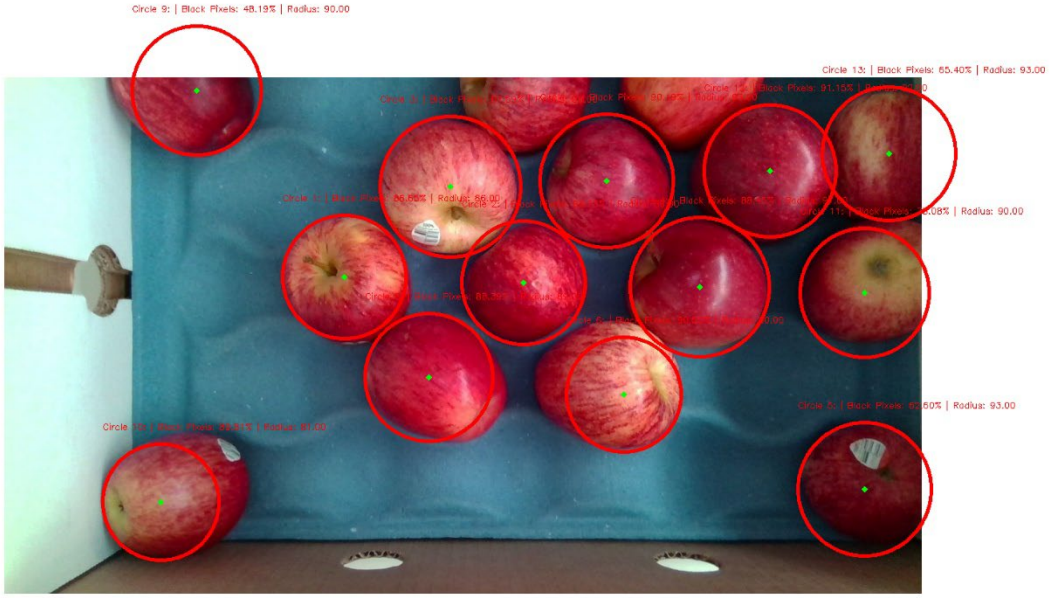

Circle 11: | Black Pixels: 38.00% | Radius: 92.00
Circle 12: | Black Pixels: 42.63% |
Black Pixels: 43.66% | Radius: 83.00


Circle 11: | Black Pixels: 49.10% | Radius: 95.00
Circle 7: | Black Pixels: 47.78% | Radius: 97.00
Circle 8: | Black Pixels: 77.92% | Radius: 95.00


Circle 11: | Black Pixels: 46.46% | Radius: 94.00
Circle 5: | Black Pixels: 55.53% | Radius: 87.00
Circle 7: | Black Pixels: 61.98% |


Circle 12: | Black Pixels: 26.98% | Radius: 89.00
Circle 13: | Black Pixels: 35.69% | Radius: 79.00
Circle 5: | Black Pixels: 87.98% | Radius: 90.00
Circle 8: | Black Pixels: 91.75% | Radius: 84.00
Circle 3: | Black Pixels: 85.01% | Radius: 94.00
Circle 11: | Black Pixels: 71.70% | Radius: 90.00


Circle 12: | Black Pixels: 53.33% | Radius: 86.0
Circle 5: | Black Pixels: 92.13% | Radius: 90.00
Circle 7: | Black Pixels: 95.18% | Radius: 91.00


Circle 9: | Black Pixels: 48.19% | Radius: 90.00
Circle 13: | Black Pixels: 65.40% | Radius: 93.00
Circle 5: | Black Pixels: 66.50% | Radius: 93.00


Circle 12: | Black Pixels: 53.33% | Radius: 86.0
Circle 7: | Black Pixels: 62.0 | Radius: 91.00


Circle 4: | Black Pixels: 80.90% | Radius: 88.00
Circle 6: | Black Pixels: 87.18% | Radius: 86.00
Circle 7: | Black Pixels: 76.43% | Radius: 87.00
Circle 16: | Black Pixels: 86.74% | Radius: 81.00
Circle 14: | Black Pixels: 79.50% | Radius: 98.00
Circle 13: | Black Pixels: 88.10% | Radius: 96.00
Circle 8: | Black Pixels: 78.92% | Radius: 98.00
Circle 11: | Black Pixels: 82.18% | Radius: 95.00


Circle 15: | Black Pixels: 67.27% | Radius: 98.00
Circle 6: | Black Pixels: 80.53% | Radius: 91.00
Circle 16: | Black Pixels: 65.89% | Radius: 97.00
Circle 12: | Black Pixels: 85.36% | Radius: 96.00
Circle 11: | Black Pixels: 81.40% | Radius: 97.00


Circle 5: | Black Pixels: 62.74% | Radius: 91.00
Circle 13: | Black Pixels: 73.59% | Radius: 89.00
Black Pixels: 69.08% | Radius: 86.00
Black Pixels: 63.54% | Radius: 85.00
Black Pixels: 58.48% | Radius: 80.00