CSE 573: INTRODUCTION TO COMPUTER VISION AND IMAGE PROCESSING

(FALL 2018)

Dr. Junsong Yuan

University at Buffalo

PROJECT 3 REPORT

Submitted by-

ANIRUDDHA SINHA (UB Person No - 50289428)

Date: December 3, 2018

TASK 1 – MORPHOLOGICAL IMAGE PROCESSING

Aim

Using morphology operations, denoise the given images. Also, find the boundaries of the objects in the image.

Image Morphology – A brief overview

In image processing, morphological image processing deals with operations on images where the shapes and boundaries of the objects in the image are taken into account. In morphology, we do not give importance to the pixel values at a particular location in an image, rather to the orientation of pixels in the image.

Morphological image processing requires images to be binary such that there are only two discrete values – 0 and 1 that can easily differentiate the image features.

The morphological operations in image processing are carried out using a particular shape-type mask known as the **structuring element**. The element is placed at all pixels of the image and it tries to morph the image in the same shape as itself.

Characteristics of a structuring element –

- It is a matrix smaller than the main image.
- It is a binary image, that is a collection of 0s and 1s placed together.
- The placement of 0s and 1s in the matrix identifies the type of shape that is finally going to be morphed on the main image.
- The structuring element has an origin, which can be any particular pixel in the image over which the results of the calculations are saved. It is also possible to have an origin outside the image.

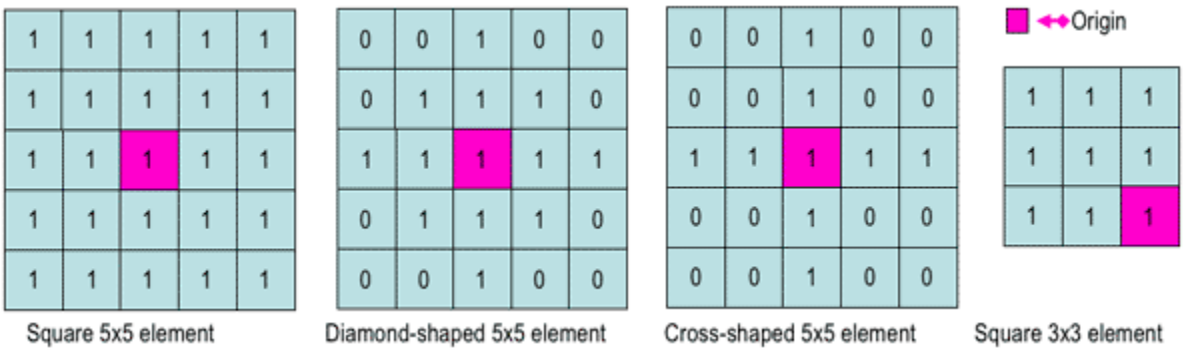

Fig.1.1 Examples of structuring elements
(https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm)

Morphology works on two concepts – **fit** and **hit.** The structuring element is said to *fit* the source image, when all pixels in the element that are 1 have corresponding image pixels also equal to 1. The structuring element is said to *hit* the image if for at least one pixel of the element set equal to 1, there is a corresponding match with the main image having an element equal to 1.

Erosion, Dilation, Opening and Closing:

Erosion of a binary image is the subtraction of pixels in an image by a structuring element. An erosion operation produces a new binary image of an image with 1s only at those locations (origins) in the new binary image where the structuring element **fits** the input image, otherwise replace those locations with 0s.

Dilation of a binary image is the addition of pixels in an image by a structuring element. A dilation operation produces a new binary image having 1s at all those locations (origins) where the structuring element **hits** the input image, else leave the main input image as it is.

Opening operation is a combination of erosion and dilation. While opening an image, we erode the input image with a structuring element and then dilate the eroded image with either the same structuring element or a new element.

Closing operation is just the opposite of opening, where we first dilate the image and then erode it.

Denoising an image:

An image can be denoised by performing a combination of opening and closing. The order of operations does not matter, that is we may first perform opening and then closing, or vice-versa and still obtain the same results.

Implementation

a) **Part 1** – **Remove the noise in the image**
   Input Image:

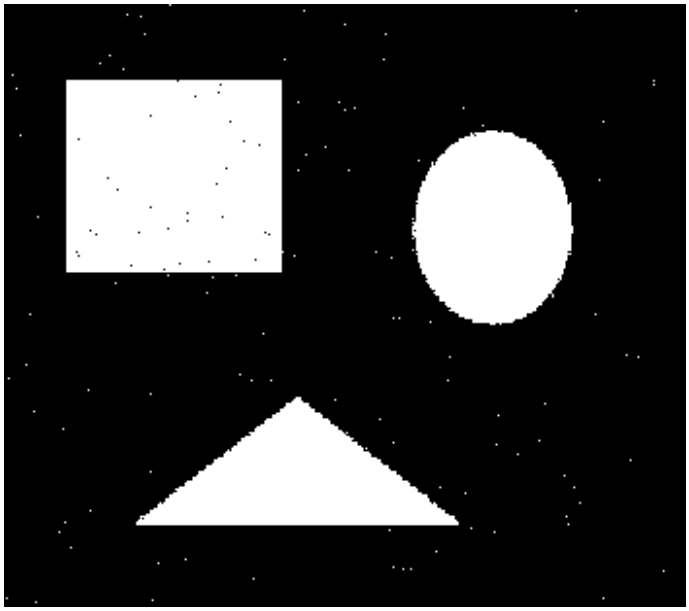

Fig.1.2 Input Image

*** Before performing any morphological operations on the image, we first **convert it to Binary**, by using the **threshold of 127**, i.e. pixels having intensities less than 127 are converted to **0**, while those above 127 are converted to **255 (1).** We use the method `cv2.threshold` from OpenCV to achieve this.

1st operation: **OPENING + CLOSING**

Structuring Element: Size = (3 x 3)

255	**255**	**255**
255	**255**	**255**
255	**255**	**255**

Table 1.1 Structuring element for Opening and Closing

- Step 1.1: **Erosion**. Results after erosion:

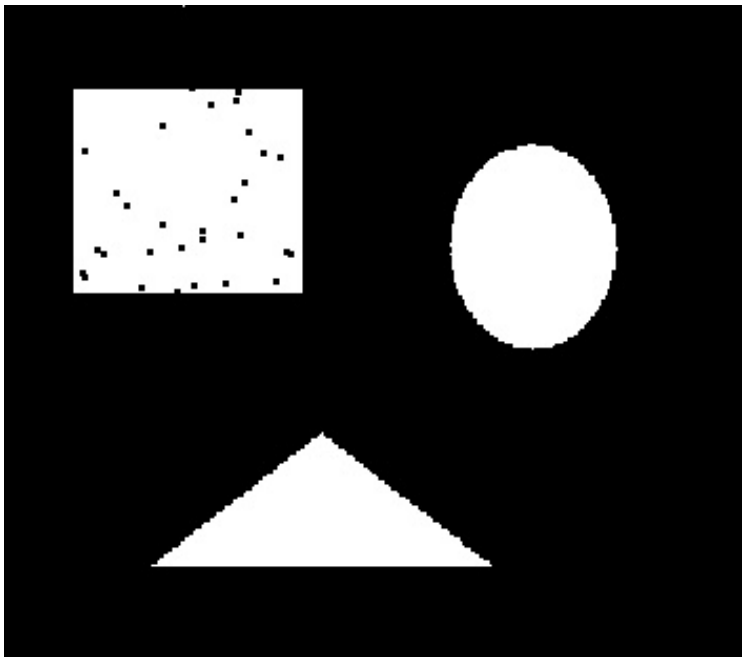

Fig 1.3. Results after eroding the input image

- Step 1.2: **Dilation.** Results after dilation of the eroded image in Step 1.1.

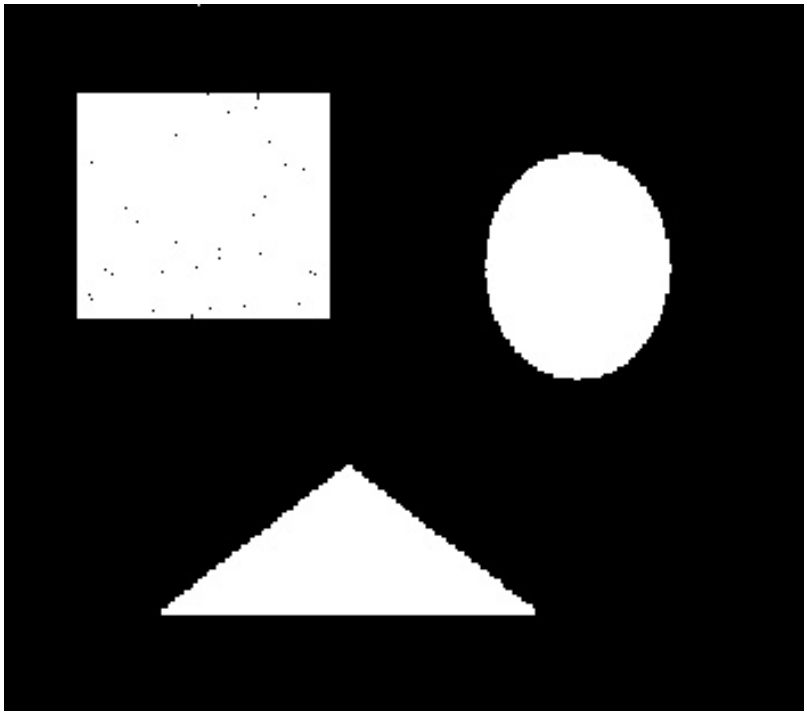

Fig 1.4. Results after dilating the image in Fig.1.3

This gives us the results after **opening** the image. Next, we **close** the image to remove the noise.

- Step 1.3: **Dilate** the image in Fig. 1.4. Results:

Fig.1.5 Results after dilating the opened image in Fig.1.4

- Step 1.4: **Erode** the intermediate image in Step 1.3. Results:

Fig.1.6 Final image with no noise. Output image – **res_noise1.jpg**

The image in Fig.1.6 shows the results of Opening + Closing that gives us the image with no noise.

2nd Operation: **CLOSING + OPENING (reverse)**

Structuring Element used: Same as Table 1.1. *Size* = (3 x 3)

255	**255**	**255**
255	**255**	**255**
255	**255**	**255**

Table 1.2 Structuring element for Closing and Opening

- Step 2.1: **Dilation**. Results after dilation:

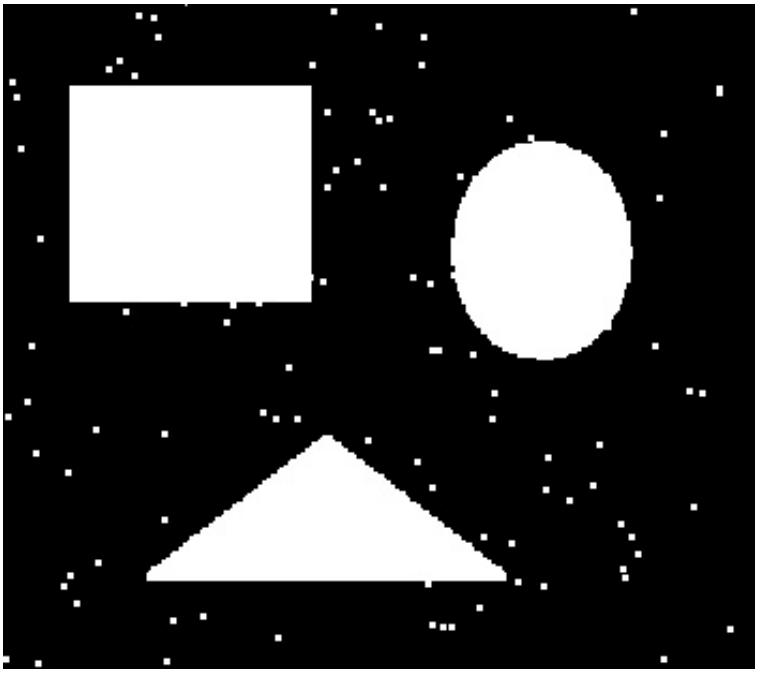

Fig 1.7. Results after dilating the input image

- Step 2.2: **Erosion.** Results after erosion of the dilated image in Step 2.1.

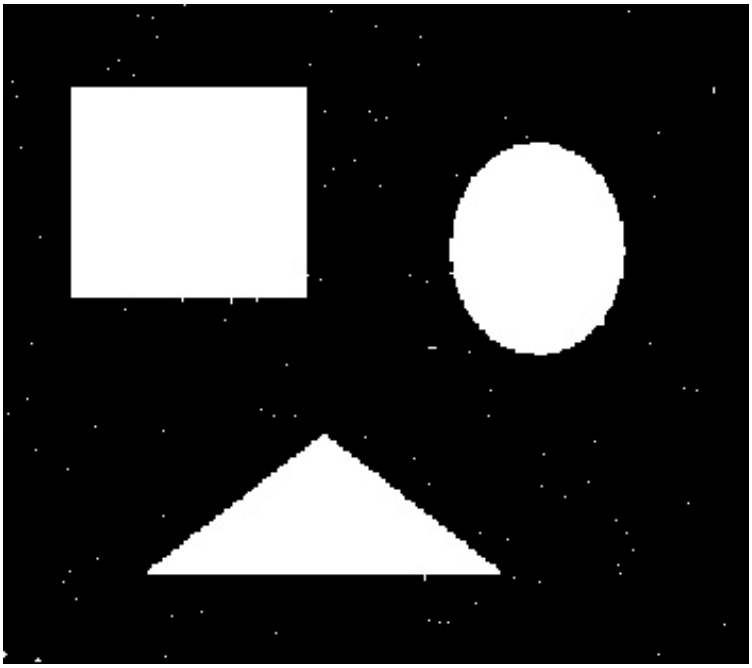

Fig 1.8. Results after eroding the image in Fig.1.3

This gives us the results after **closing** the image. Next, we **open** the image to remove the noise.

- Step 2.3: **Erode** the image in Fig. 1.8. Results:

Fig.1.9 Results after eroding the closed image in Fig.1.8

- Step 2.4: **Dilate** the intermediate image in Step 2.3. Results:

Fig.1.10 Final image with no noise. Output image – **res_noise2.jpg**

The image in Fig.1.10 shows the results of Closing + Opening that gives us the image with no noise.

b) Part 2 – <u>Comparison of the two results</u>

After having performed both combination of operations to remove the noise, i.e., opening + closing (Fig.1.6) and closing + opening (Fig.1.10), we can observe that **both images are the same, and do not have any noise in them.**

c) Part 3 – <u>Boundary Extraction</u>

In image processing, the backgrounds of objects in a binary image can be obtained by subtracting the eroded version of the image from its original binary image, i.e.

Boundary = Binary Image – Eroded Binary Image

Thus, for this we –

- First, convert the original image to binary.
- Then, we subtract the noise removed image from the binary image. This gives us the boundaries of the objects in the images.

Results after **subtracting first noise removed image from the binary image** –

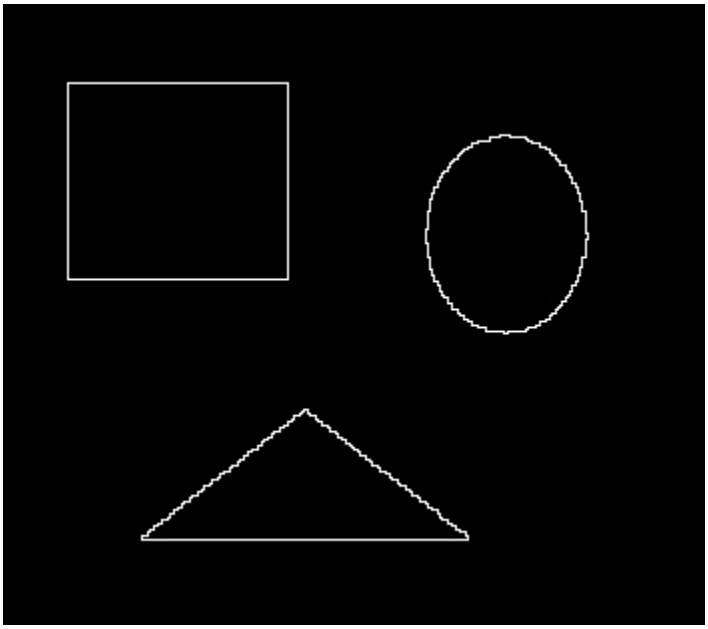

Fig.1.11 Boundary of the image after subtraction of res_noise1 from original image. Output **– res_bound1.jpg**

Results after **subtracting second noise removed image from the binary image** –

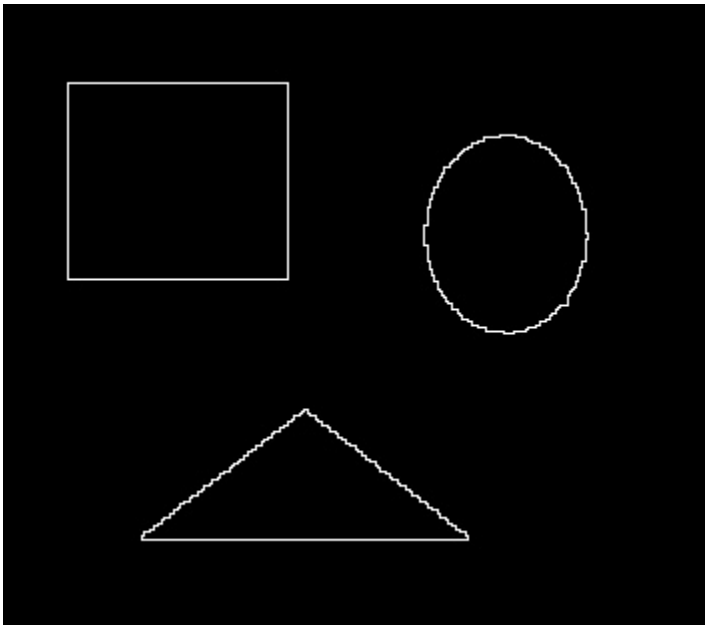

Fig.1.12 Boundary of the image after subtraction of res_noise2 from original image. Output – **res_bound2.jpg**

It is evident that the results of the images from the two methods of boundary subtraction are the same.

TASK 2 – POINT DETECTION AND IMAGE SEGMENTATION

Aim

Perform point detection on an image.

Also, use the concept of segmentation and thresholding to find an optimal threshold for segmenting the foreground from the background for a given grayscale image.

Point Detection – A brief overview

As the name suggests, point detection is a technique that helps us to detect points in an image. The process of detection is very similar to the techniques of correlation and convolution, where we take a mask and place it over the image and keep generating pixels wherever a point is detected.

Here, we perform correlation of a **kernel** with the image. The kernel can be any **odd-sized square matrix**, in which **the sum of all elements is zero**. In general, the origin of the kernel is a negative value that is numerically equal to the sum of all the other elements in the kernel.

An example of a 5x5 kernel is as shown below –

-1	**-1**	**-1**	**-1**	**-1**
-1	**-1**	**-1**	**-1**	**-1**
-1	**-1**	**24**	**-1**	**-1**
-1	**-1**	**-1**	**-1**	**-1**
-1	**-1**	**-1**	**-1**	**-1**

Table 2.1 A simple kernel for point detection

Thus, after correlation, we obtain –

$$R = \sum_{i=1}^{25} w_i z_i$$

- The above formula returns the weighted difference between the center point and its neighbours.
- We can easily detect a point at the origin of the kernel if the absolute value of R is greater than a non-negative threshold, i.e.

$$| R | \geq T$$

Image Segmentation

Segmentation is a technique in image processing that helps us to isolate objects from each other in an image when they are one over another.

Segmentation subdivides an image into its constituent regions or objects. Segmentation can be done by using two basic properties of image intensities –

- **Similarity –** divide an image into similar parts based on a predefined criterion like **thresholding.**
- **Discontinuity –** divide an image based on sudden changes in intensities like gradients, etc. Methods in this include **Point, Line and Edge Detection, etc.**

Implementation

POINT DETECTION

Input Image:

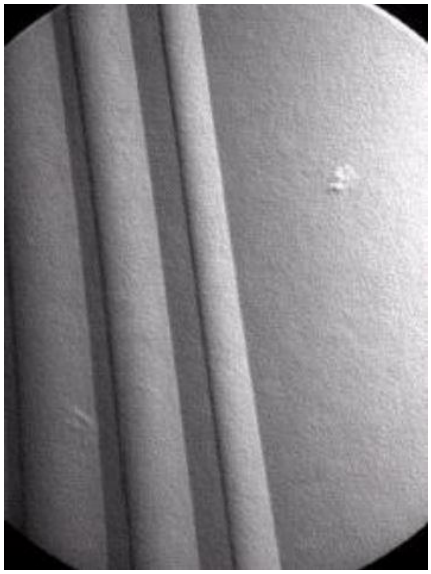

Fig.2.1 Input Image for point detection

Step 1: We first initialize the following kernel for correlation with the given image:

Kernel:

-1	**-1**	**-1**	**-1**	**-1**
-1	**-1**	**-1**	**-1**	**-1**
-1	**-1**	**24**	**-1**	**-1**
-1	**-1**	**-1**	**-1**	**-1**
-1	**-1**	**-1**	**-1**	**-1**

Table 2.2 Kernel used for point detection

Step 2: Next, we correlate the kernel with the given input image.

Step 3: Finally, we set a threshold of **T = 2300,** and find the pixels that satisfy the formula –

$$| R | \geq T$$

This gives us the final output image with the detected points as below –

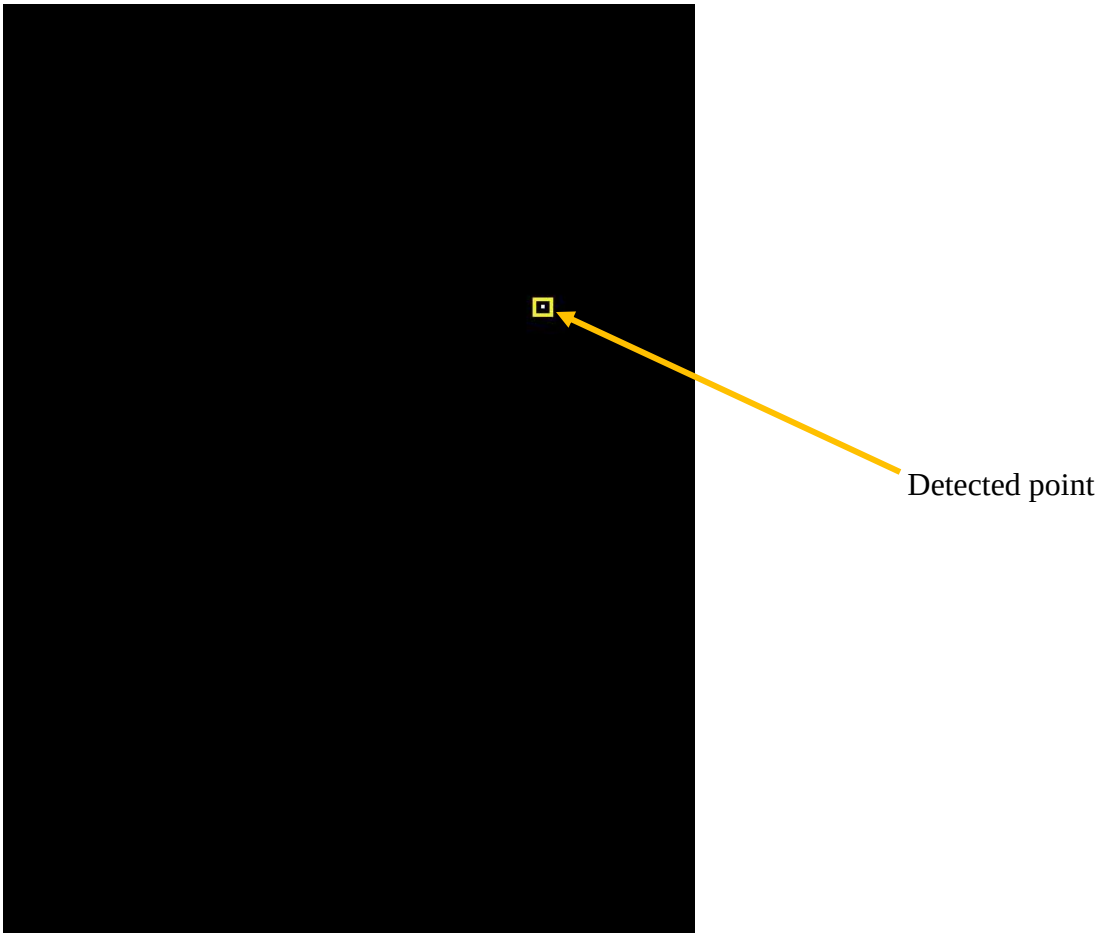

Fig.2.2 Point Detected Image **(point_detected.jpg)**

Coordinates of the detected point in the image (rows, column) = (249, 445)

*** The final output image is saved as **`point_detected.jpg`**.

IMAGE SEGMENTATION

Input Image:

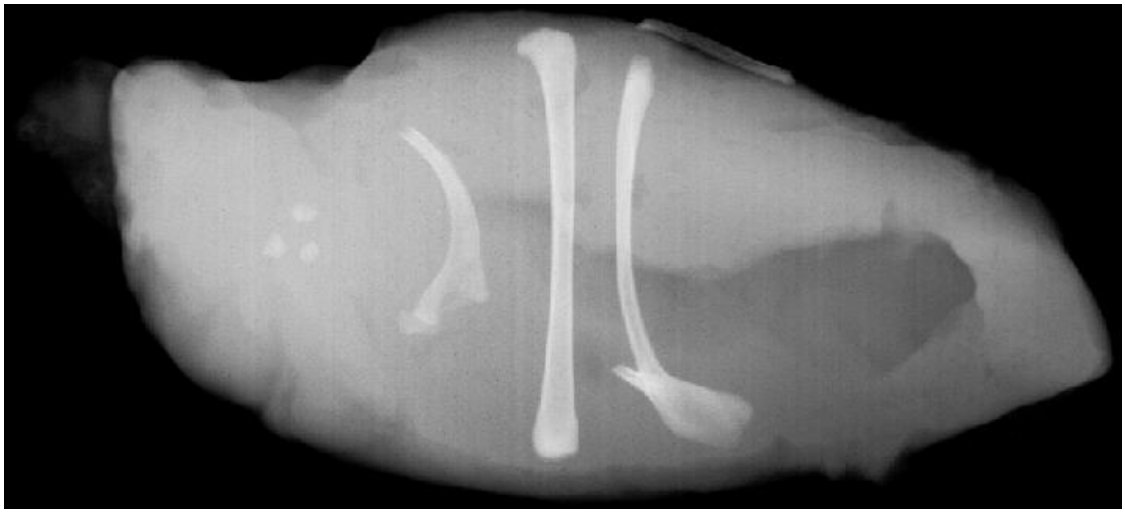

Fig.2.3 Input Image for image segmentation

The given input image is an X-Ray image of a chicken where we can see the bones. As is clear from the image, there are various backgrounds and foregrounds, for example the bones are in the foreground of the chicken's body (background) and the chicken's body is in the foreground of the image frame (black background).

Our objective is to segment the chicken bones and draw bounding boxes around them.

Step 1: For obtaining the **best threshold** to segment the objects from the background, we implement the **Heuristic thresholding algorithm**. At the completion of the algorithm, the threshold obtained is–

T = 201

Step 2: Next, we threshold the original image with the obtained threshold and get the image in Fig.2.4 –

Fig.2.4 Segmented bones from the main image after heuristic thresholding. Final threshold = 201

Step 3: Finally, we draw bounding boxes around the segmented bones using **Template Detection**. The binary image having the segmented bones and bounding boxes is as shown in Fig.2.5, saved as **`bones_segmented.jpg`**.

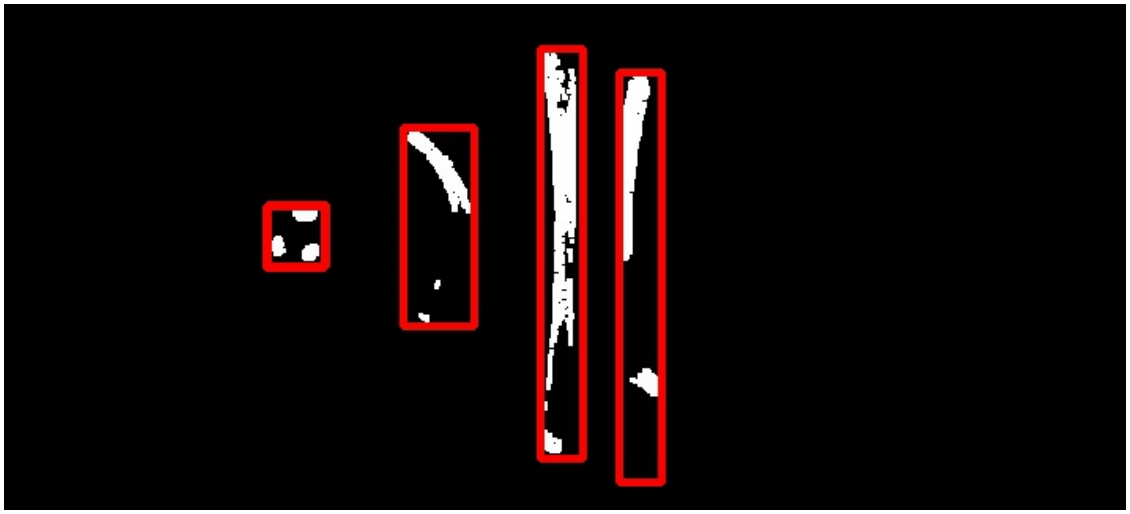

Fig.2.5 Bounding boxes on the segmented image (**bones_segmented.jpg**)

The bounding boxes coordinates from Left to Right, in terms of **image pixels' locations (rows, columns)** are –

Bones 1 - Box 1: (127, 167), (127, 203), (166, 167), (166, 203)

Bones 2 - Box 2: (78, 253), (78, 298), (204, 253), (204, 298)

Bones 3 - Box 3: (28, 340), (28, 367), (288, 340), (288, 367)

Bones 4 - Box 4: (43, 390), (43, 423), (252, 390), (252, 423)

The bounding boxes are also drawn on the main input image as shown in Fig.2.6. **The boxes do not contain all features of the bones completely in the main image** as some of the are features get removed after thresholding and segmentation.

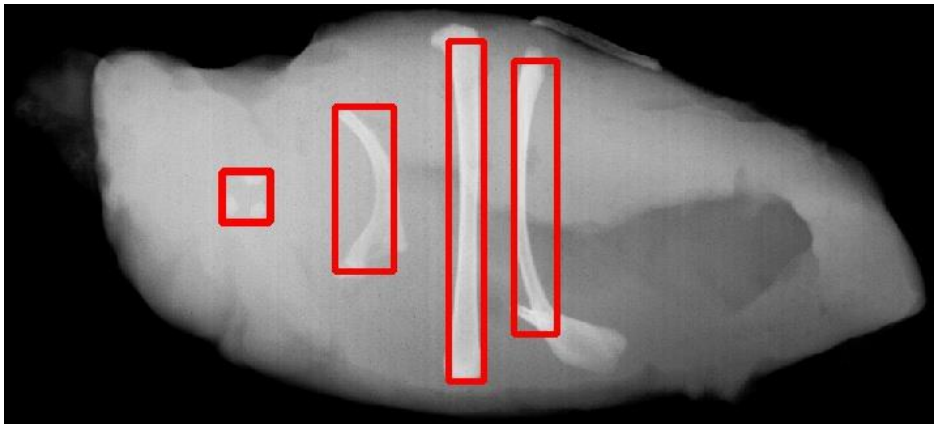

Fig.2.6 Bounding boxes on the main input images

TASK 3 – HOUGH LINE DETECTION AND HOUGH CIRCLE DETECTION

Aim

To implement Hough Transform on a given set of images and detect shapes like lines and circles in the image.

Line and Circle Detection using Hough Transform – Brief Overview

In the image space, it is difficult to robustly detect lines and circles, where the basic parameters such as the slope, intercept, radius, etc. of these shapes is not known from before. **Hough Transform** is a solution for the same that helps us in detecting these shapes.

- **Line Detection:**

The Hough Transform converts points in the XY space to lines in the mc-space, where **m** is the slope of the line and **c** is the intercept of the line.

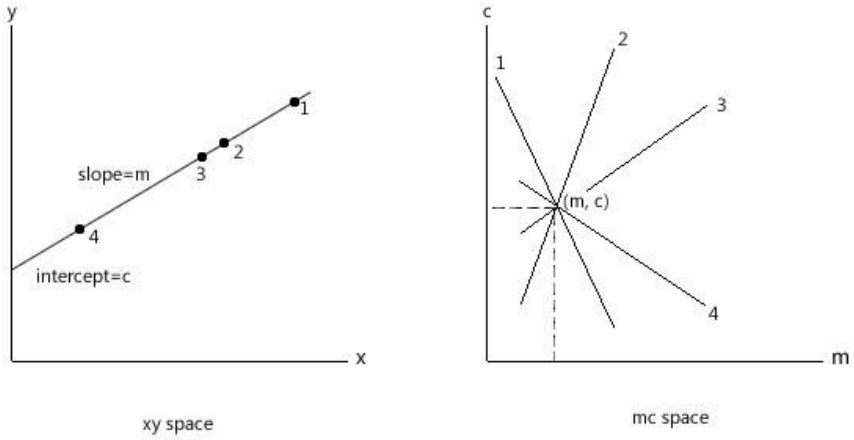

Fig. 3.1. An example of Hough Lines
(http://aishack.in/tutorials/hough-transform-basics/)

The points 1, 2, 3, 4 are represented as various lines in the mc-space. And the intersection of these lines is equivalent to the original line. [http://aishack.in/tutorials/hough-transform-basics/]

However, when **m = infinity**, i.e. for a vertical line it is not possible to store the line in mc-space. Hence, we need a new space to identify lines, which is given by the **ρθ space (normal form)**.

In normal form, any line in XY space can be represented as –

$$\rho = x_0 \cos\theta + y_0 \sin\theta, \qquad \text{…….. (1)}$$

where, **ρ** = distance of the normal from the origin (0,0) to the line

$\boldsymbol{\theta}$ = angle the normal makes with the x-axis. Θ ranges from -90 to +90

With these ideas, when we plot the (x,y) points in the **ρθ-space,** we will notice that each point represents a sinusoidal curve.

- **Circle Detection:**

The idea of circle detection using Hough Transform is very similar to that of line detection. For finding a circle, we need two main parameters –

- Location – (x, y)
- Radius – R

Thus, using the idea of Hough Transform, we can define a space – **ab space** where we have,

$$\boldsymbol{a = x - R\cos\theta}$$

$$\boldsymbol{b = y - R\sin\theta\ ,} \qquad \text{……..…….. (2)}$$

where, $\boldsymbol{\theta}$ ranges from 0 to 360^0.

Thus, with the above equations, we can easily detect the circles in the image.

Implementation

LINE DETECTION

Input Image:

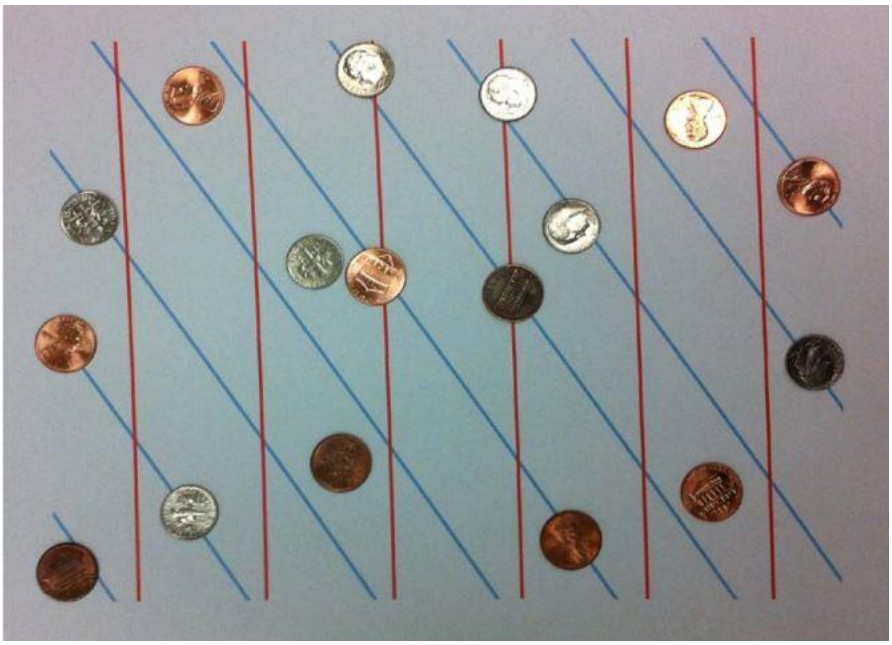

Fig. 3.2. Input image for Hough Transform

*** **To implement Hough transform, we performing *voting* on the image pixels and create an accumulator array - 2D for line detection and 3D for circle detection.**

- First, we perform **Edge Detection** on the given image by **using Sobel Edge Detection**. Hough transform works only on edge-detected binary images and not on normal images.

- <u>Creating the accumulator array:</u>
  The next objective is to loop over all pixels of the edge-detected image and create the accumulator array containing the **votes.** The votes tell us the occurrence of a particular pixel location (x, y) on various lines.

- The values of **ρ** range from – (length of the diagonal of the image) to + (length of the diagonal of the image), whereas $\boldsymbol{\theta}$ ranges from -90^0 to $+90^0$. Hence, we initialize a accumulator array having size = **(2*length of diagonal X 180)** with only zeros.

- We consider all non-zero pixels in the edge-detected image and do voting by taking a range of $\boldsymbol{\theta}$ values from -90^0 to $+90^0$, i.e. 180 values in total. For every $\boldsymbol{\theta}$ on a particular location (x,y) in the image, we calculate the corresponding **ρ** value by using the formula in (1).

- For every ρ calculated and θ, we update the corresponding (ρ, θ) in the accumulator by adding 1s every time.

- Again, for the same image pixel, we take another θ value, calculate ρ and update the value in the accumulator. After this, we keep getting votes every time.

- At the end, we will get an image having points as sinusoidal graphs, as shown in Fig. 3.3.

- Next, we extract all those locations (ρ, θ) from the accumulator, where the votes are between a particular threshold. The **threshold chosen here is in a range of 110 and 170**.

- Finally, we filter the red lines and the blue lines by finding the $\boldsymbol{\theta = -2^0}$ **values for red-lines and** $\boldsymbol{\theta = -36^0}$ **values for blue-lines.**

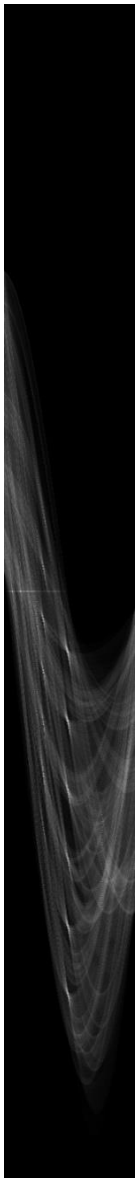

Fig. 3.3. Accumulator array

- The filtered points are sent to a function that converts (ρ, θ) points to (x,y) coordinates in the main image and draw the lines using the method `cv2.line()`.

For Bonus part (circle-detection):

- Similar to creating the (ρ, θ) accumulator array for lines detection, here, we create a 3D array – that saves the **(a,b,R) values for every (x,y) image pixel using equation (2).**

- For Hough circle transform, our angles range from 0^0 to 360^0.

- Since the **radius is not already known to us**, we have taken the **radii to be in a range of (22, 23, 24, 25, 25).**

- Thus, size of our accumulator array is **(image height X image width X range of angles)**.

- Now, we start our voting in the same way as line detection. For every image pixel (x,y) of the **Sobel edge-detected image**, we check the non-zero pixels and for every radius value, we perform voting on all angles from 0^0 to 360^0 using equation (2).

- **Increment** every corresponding **(a,b,R) pixel in the accumulator by +1.** This gives us an accumulator array that looks as shown in Fig. 3.4 -

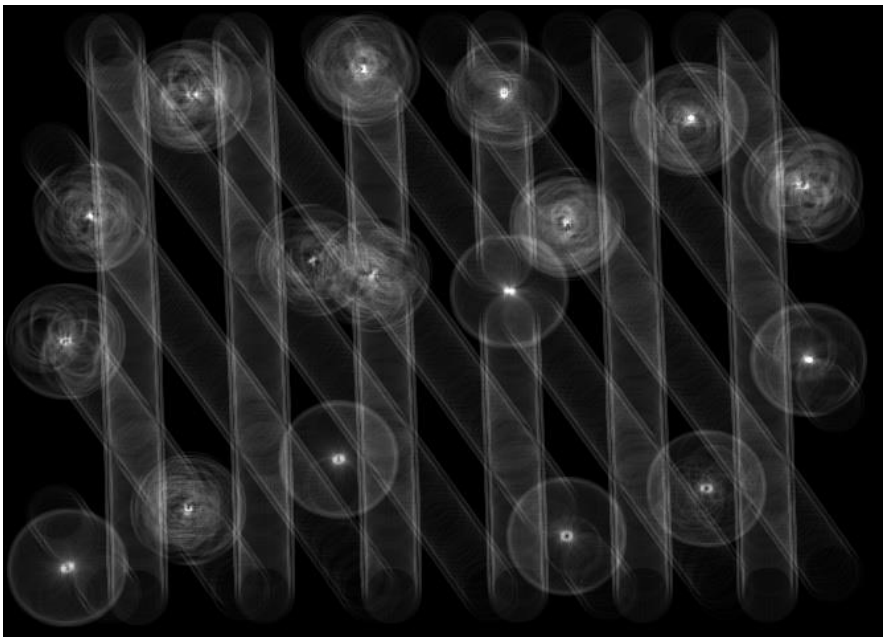

Fig. 3.4. Accumulator array for circles. White pixels represent the centers of every circle.

- We extract the centers of the circles by extracting the **(a,b,R)** pixels above a particular threshold. **Threshold chosen = 290**.
- Finally, we draw the circles on these centers with the corresponding R values using the inbuilt openCV method `cv2.circle()`.

Results

Detected Red Lines:

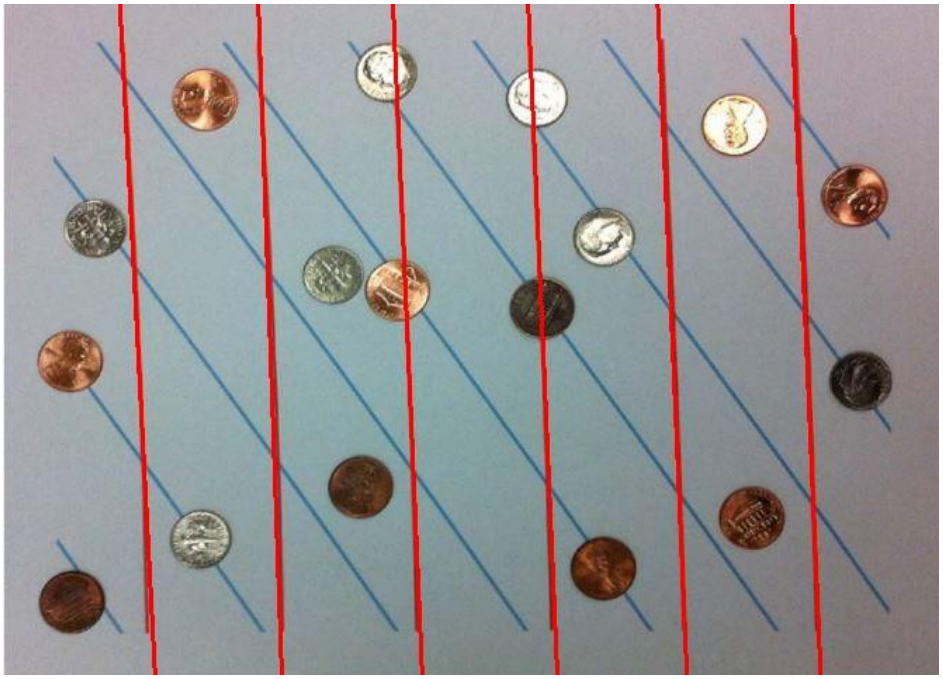

Fig. 3.5. Detected red-lines (red_line.jpg)

Number of detected Red Lines = 6

Detected Blue Lines:

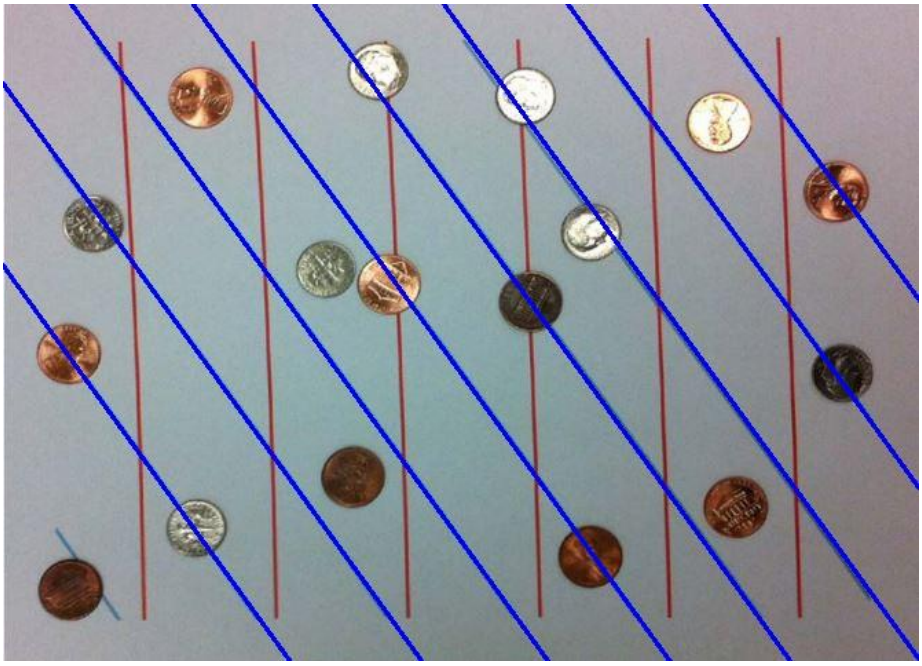

Fig. 3.5. Detected blue-lines (blue_lines.jpg)

Number of detected Blue Lines = 8

Bonus Results:

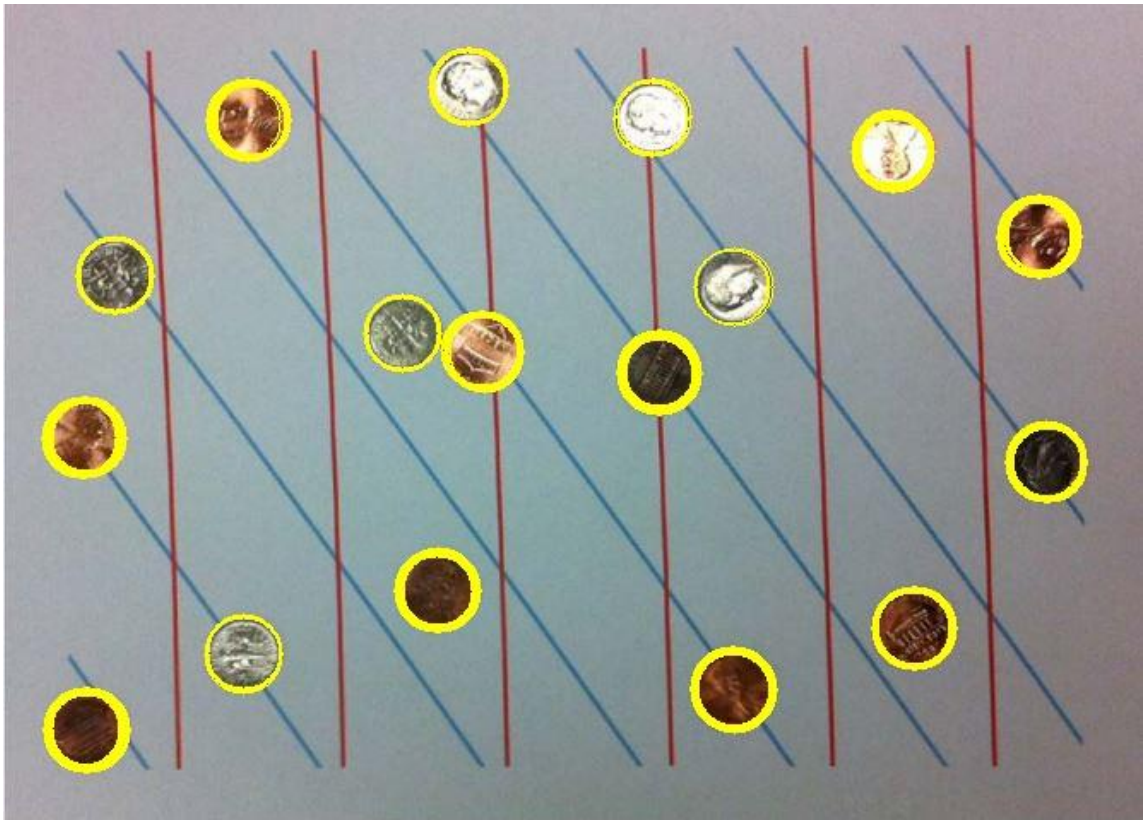

Fig. 3.6. Detected circles (circle.jpg)

Number of detected Circles = 17

References

- https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm
- https://www.mathworks.com/help/images/morphological-dilation-and-erosion.html
- https://www.wavemetrics.com/products/igorpro/imageprocessing/morphologicaloperations
- http://www.aishack.in/tutorials/mathematical-morphology/
- http://www.aishack.in/tutorials/circle-hough-transform/
- http://www.aishack.in/tutorials/hough-circles-opencv/
- http://aishack.in/tutorials/hough-transform-normal/