Vision Module development for Automated Membrane Experiment System

MSE492 Final Report

Haoning Yuan
1006702720

Background

- Reverse Osmosis (RO)
  - Water purification techniques
  - Removes contaminants including:
    - Impurities
    - Ions/Molecules
    - Large particles
  - Filtration by semi-permeable membrane

  - Important applications in:
    - Seawater desalination
    - Drinking water purification

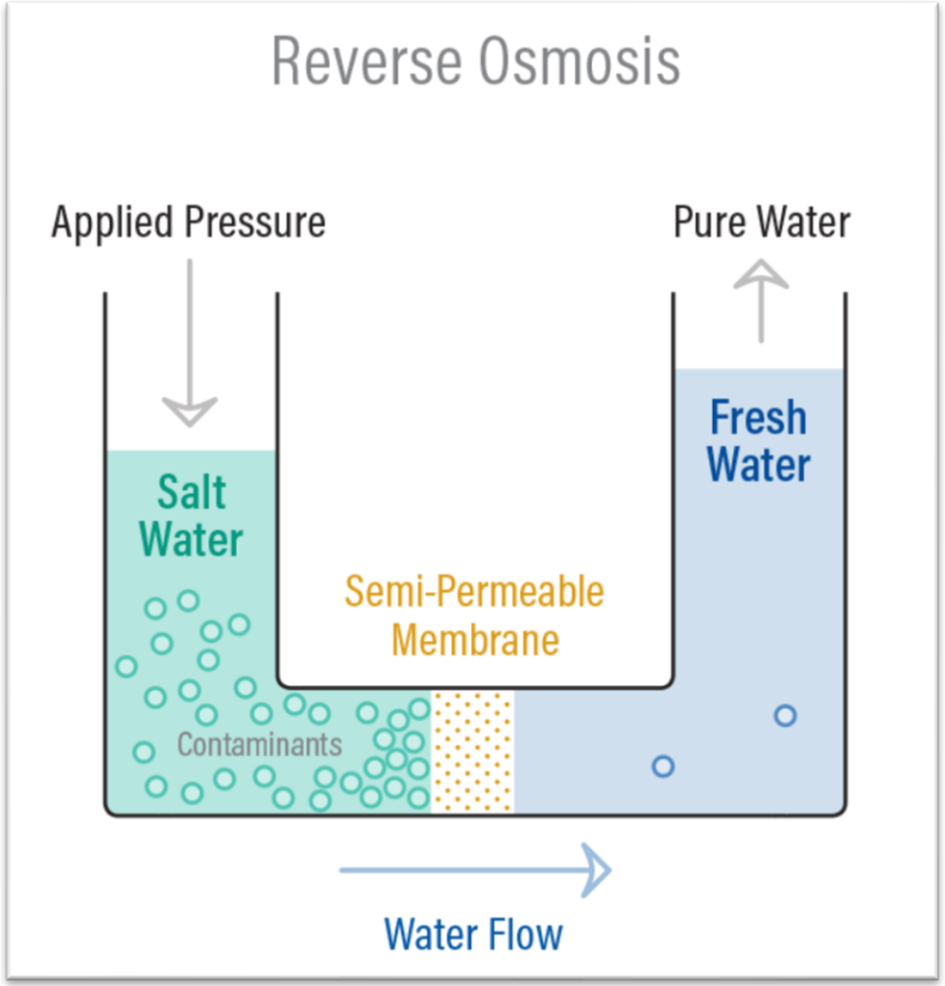


Figure 1. Reverse Osmosis schematic diagram [1]

Automated membrane experiment system

- Automated membrane making & testing
  - Liquid mixing
  - Blade casting
  - Compression testing
- System contains:
  - Opentrons Liquid Handling Robot
  - xArm Robot Arm
  - Compression Tester

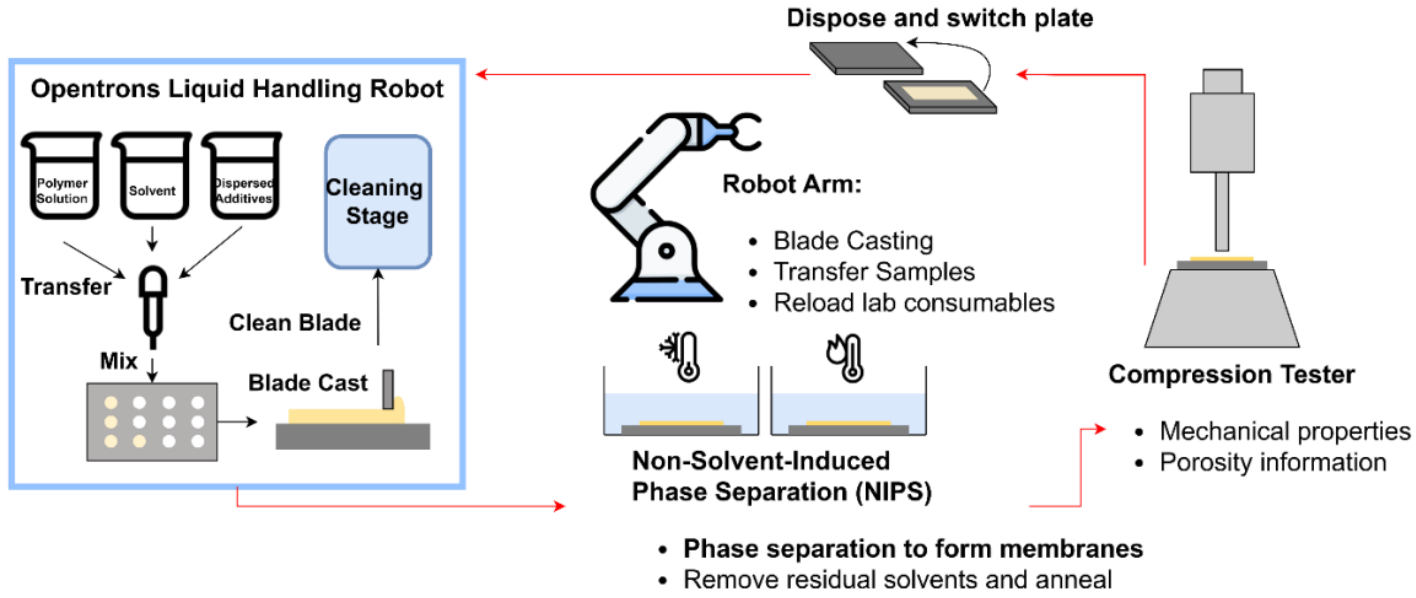


Figure 2. Reverse Osmosis schematic diagram [2]

Current Problem

- Defect on membrane made by robot
  - Wrinkles
  - Damaging
  - Tearing
  - Folding
- Will cause error in property measuring at compression testers

Figure 3. Image of defected membrane

Project Objective

1. Ensure the testing only can be conduct on uniform (no defect) region

2. Classify the quality of a membrane (flawless/defected)

Vision Module

- To achieve project objectives, a **Vision Module** is developed
- This vision module can:
  - Take photo of a membrane sample
  - Process the photo to get
    - Quality of membrane
    - Location for compression testing

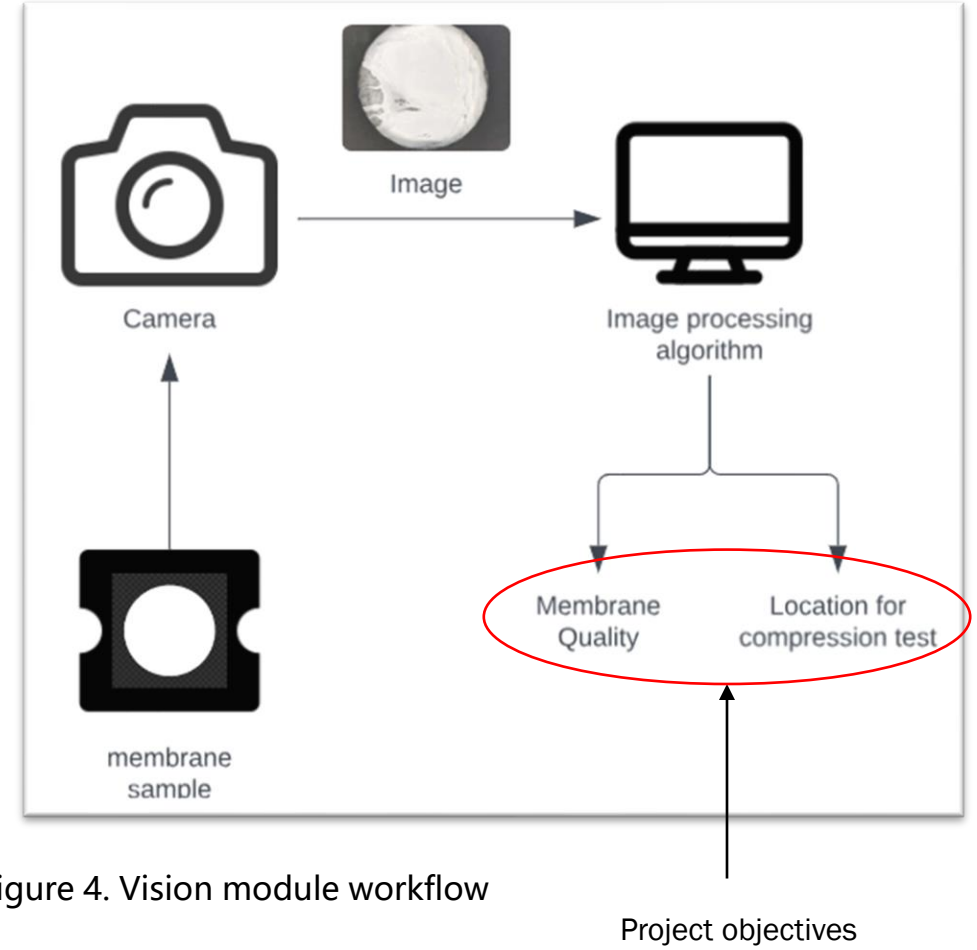


Figure 4. Vision module workflow

Module Objectives

1. Image Segmentation Algorithm

Objectives:

- Determine if a sample's quality is good enough for testing
- Identify a specific region for testing

2. Physical Camera Module

Objectives:

- Take clear photos of samples
- Interactable with robot arm

3. Arm-Module Interaction

Objectives:

- Robot arm programming

Module Objectives

Objectives:

1. Image Segmentation Algorithm

- Determine if a sample's quality is good enough for testing
- Identify a specific region for testing

2. Physical Camera Module

Objectives:

- Take clear photos of samples
- Interactable with robot arm

3. Arm-Module Interaction

Objectives:

- Robot arm programming

Image Segmentation

Literature Review

- Image Segmentation:
  - Divide an image into different regions

1. Thresholding
   - Convert into gray-scale image
   - Divide regions by a threshold value
2. Watershed Segmentation
   - Use gray-scale, treat brightness as topographic landscape
   - Simulate water flow to determine boundaries
3. Deep Learning
   - Model based on Convolution Neural Network (CNN)
   - Learn how to divide image by labelled data

- Otsu's method (traditional thresholding)
  - Search a threshold value that minimize intra-class variance
  - Divide the image by foreground and background signal
- Local Thresholding Technique
  - Divide image into smaller regions and apply different threshold values for every region.
  - The threshold is determined by local characteristic

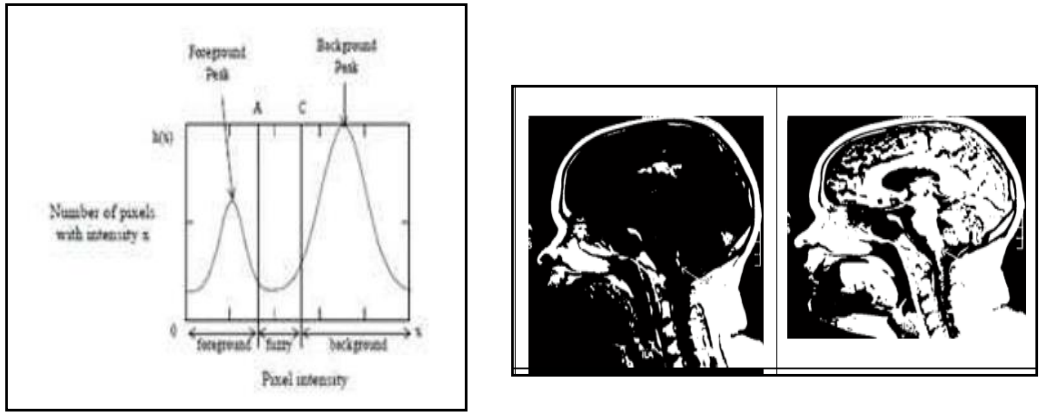


Figure 5&6. Thresholding segmentation examples [3]

Image Segmentation

Literature Review

- Image Segmentation:
  - Divide an image into different regions

1. Thresholding
   - Convert into gray-scale image
   - Divide regions by a threshold value
2. Watershed Segmentation
   - Use gray-scale, treatbrightness as topographic landscape
   - Simulate water flow to determine boundaries
3. Deep Learning
   - Model based on Convolution Neural Network (CNN)
   - Learn how to divide image by labelled data

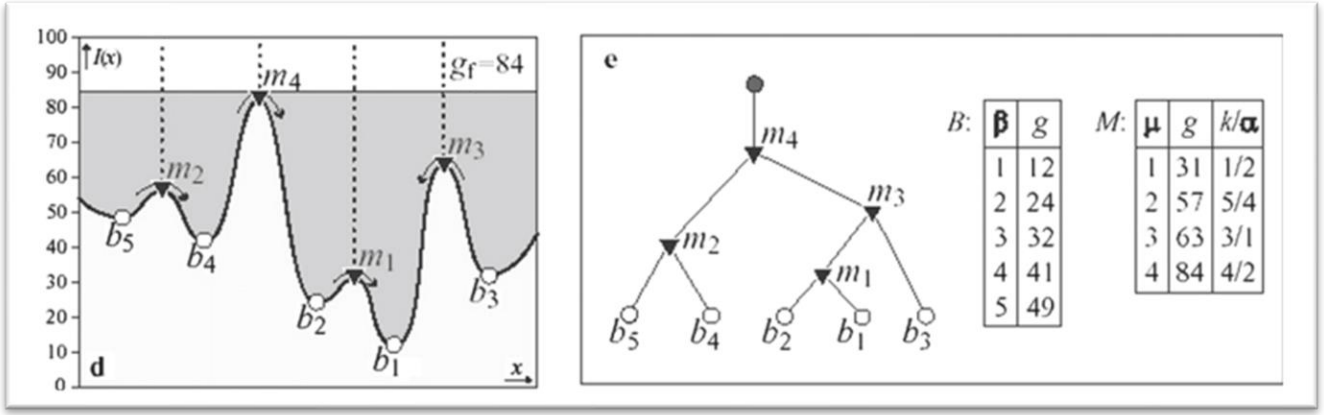


Figure 7. Watershed segmentation mechanism [4]

Brighter Region = Ridges

Darker Region = Valley

Gradient Change = Watersheds

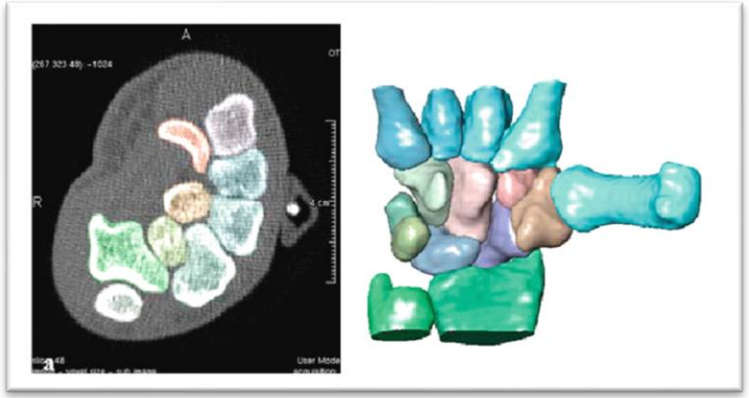


Figure 8. Image processed by watershed segmentation [4]

Image Segmentation

Literature Review

- Image Segmentation:
  - Divide an image into different regions

1. Thresholding
   - Convert into gray-scale image
   - Divide regions by a threshold value
2. Watershed Segmentation
   - Use gray-scale, treat brightness as topographic landscape
   - Simulate water flow to determine boundaries
3. Deep Learning
   - Model based on Convolution Neural Network (CNN)
   - Learn how to divide image by labelled data

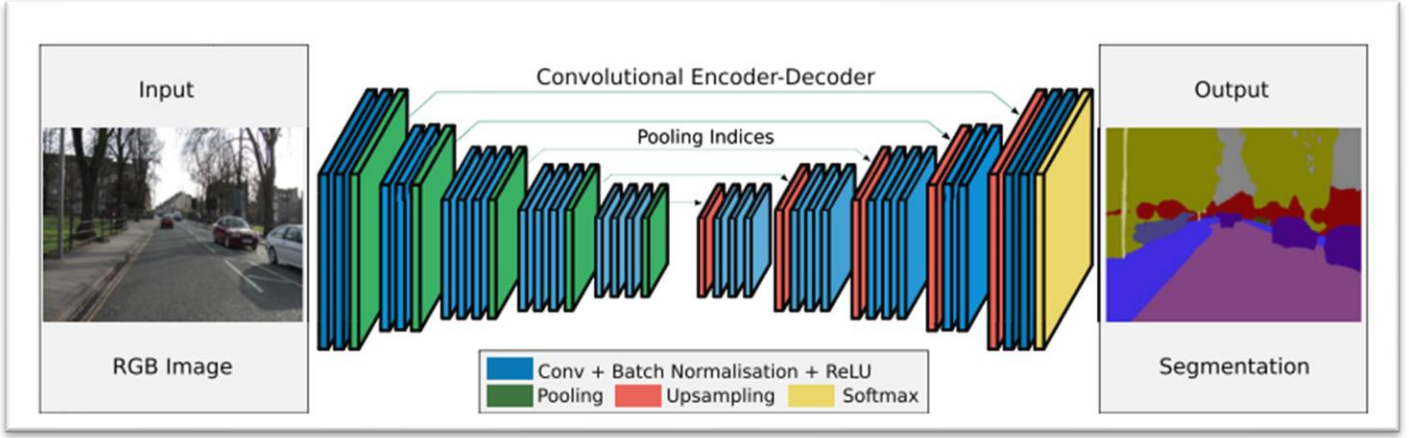


Figure 9. SegNet architecture [5]

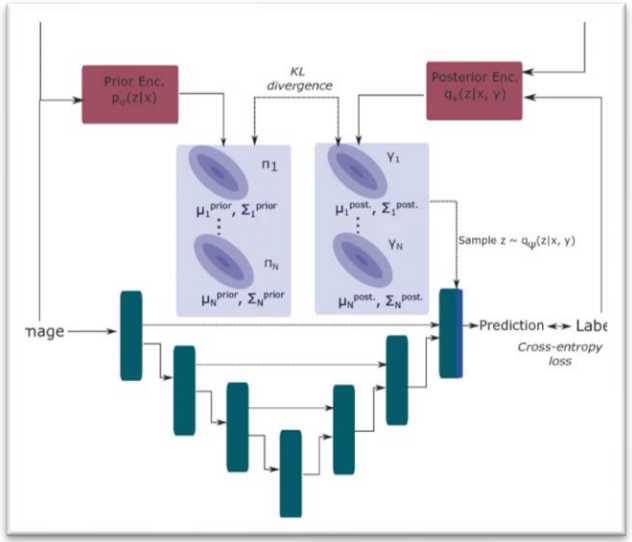


Figure 10. U-Net architecture [6]

Pros: Accurate, Powerful

Cons: Requires large amount of labelled data

Image Segmentation

Research Methodology

- Thresholding + Watershed Segmentation
- Use python to implement processing algorithm
- Convert to gray-scale image
  - Value 255 = pure white (brightest)
  - Value 0 = pure black (darkest)

- Use brightness difference to do **Edge Detection**

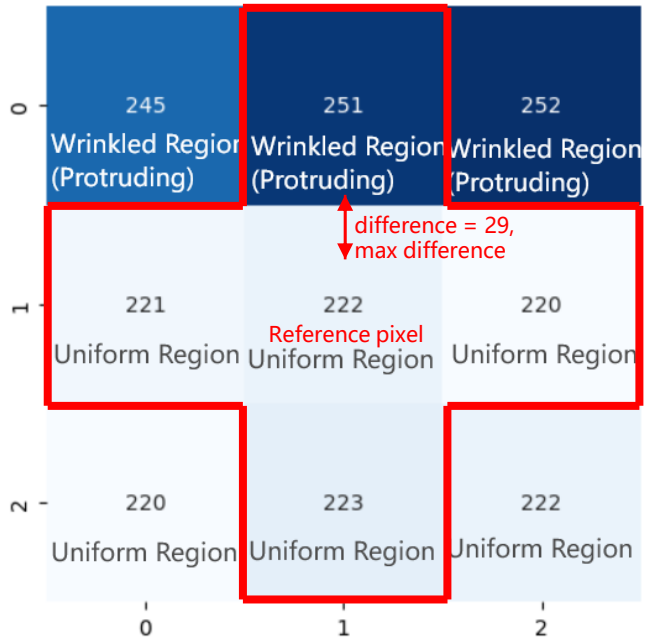


Figure 11. 3x3 pixel examples for edge detection

- Set a threshold difference value
- For every 4 pixels around the reference pixel:
  - Calculate max brightness difference within these 5 pixels

  - If the max difference exceed the threshold difference, it will be considered as edge, and turned in to black color (brightness=0)
  - For any pixel less than 150, turn into 0
  - For any pixel greater than 150, turn into 255

Image after brightness difference processing

Figure 12. Original Image with gray scale

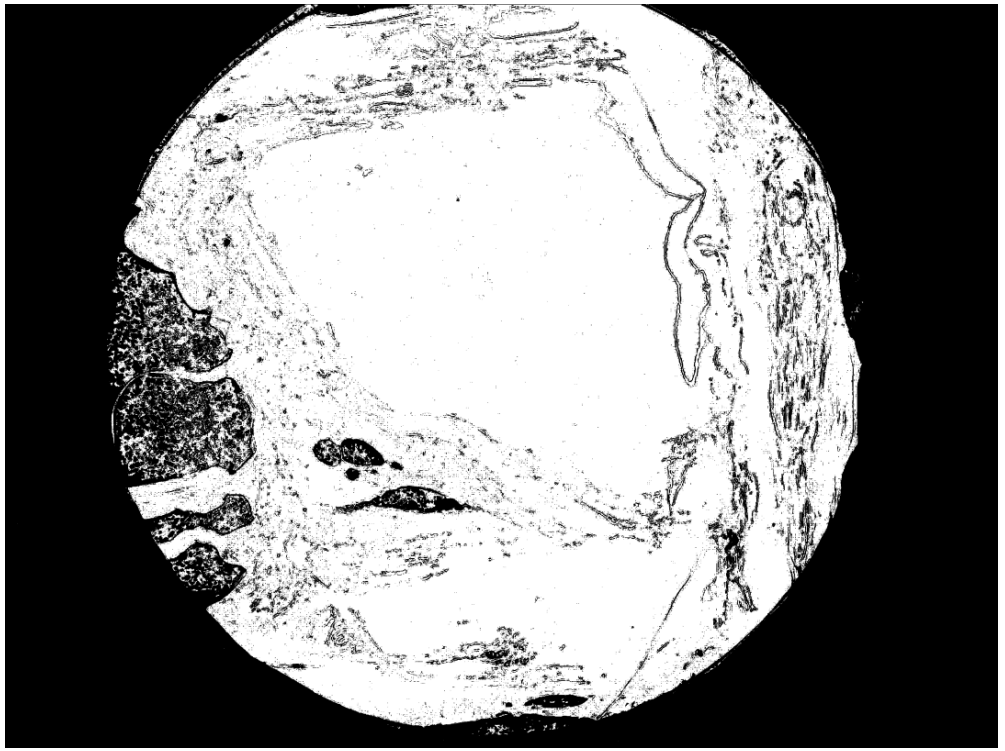


Figure 13. Brightness difference processed image

Image after brightness difference processing

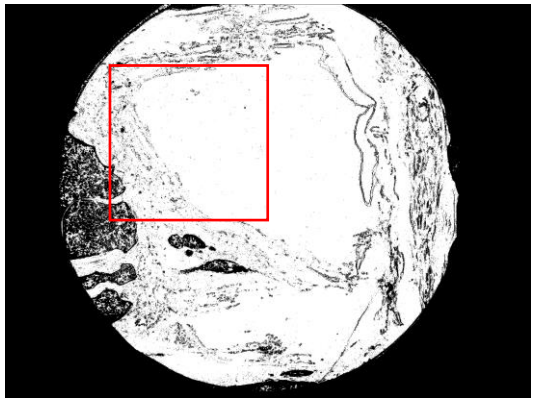


Figure 13. Brightness difference processed image

As there are many noise and discontinuity in the processed image, further denoising must performed for further processing.

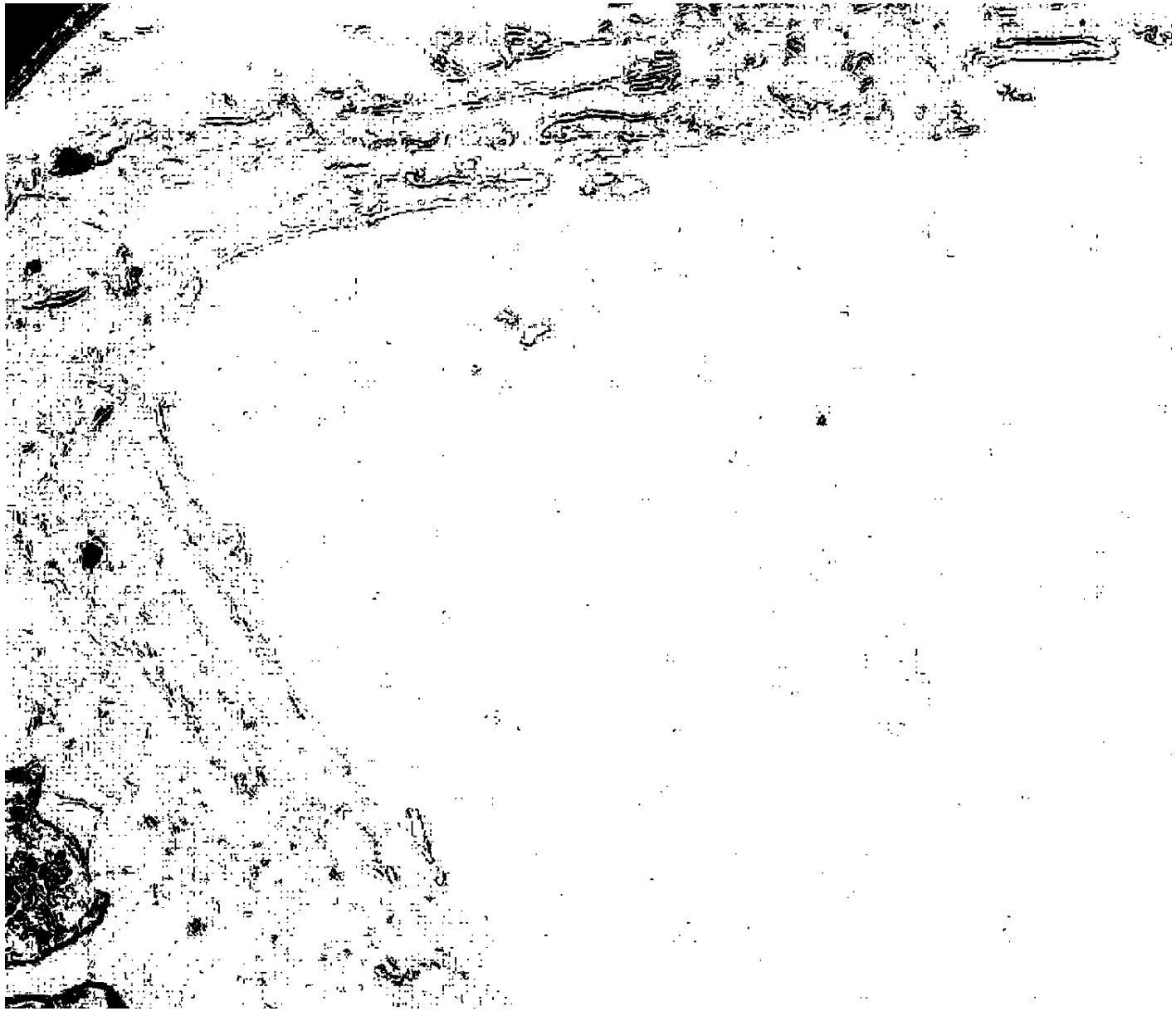


Figure 13-1. Enlarged processed image

Denoising

- To denoise:
  - Use a kernel to slide through each pixel
  - If the black pixel's ratio is greater than a specific threshold
    Reference pixel ➔ Black pixel
  - If ratio is less than specific threshold
    Reference pixel ➔ White pixel

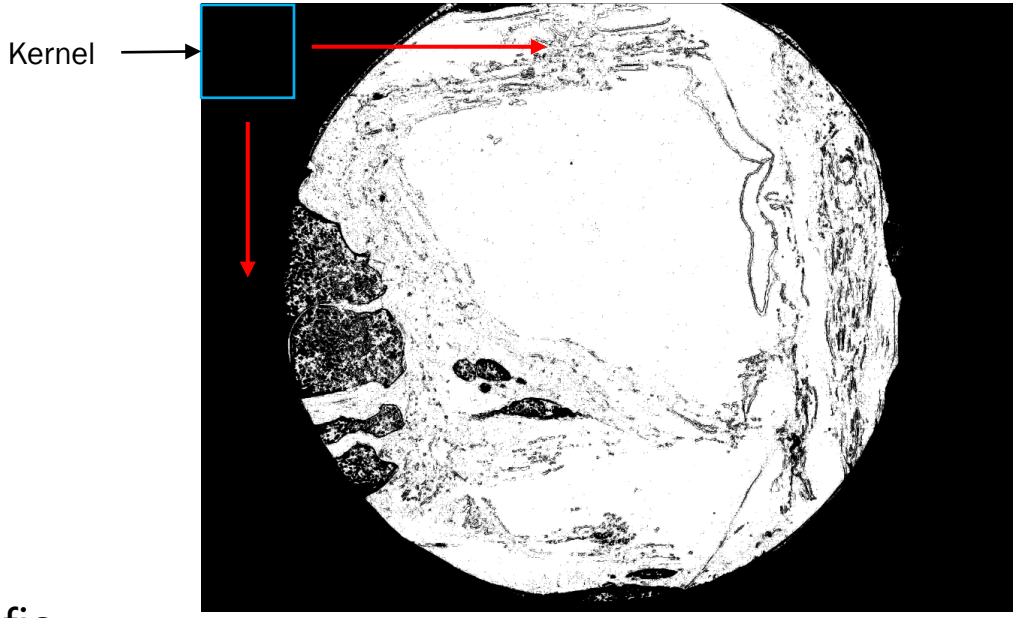
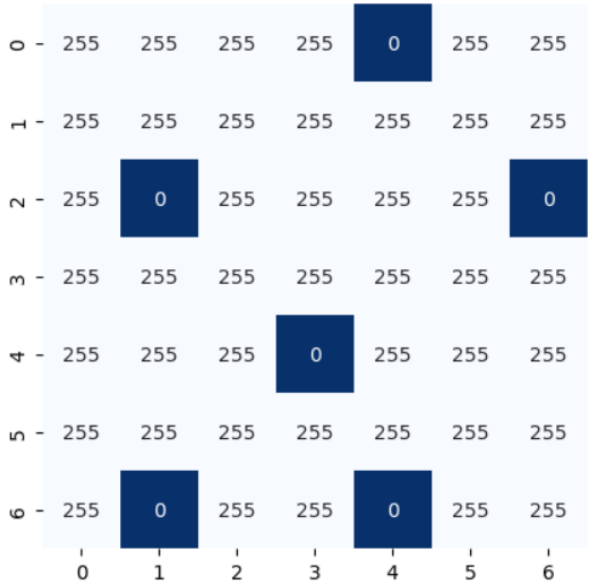


Figure 13. Brightness difference processed image



Threshold = 40%

Percentage of black pixel = 12.2%

Percentage smaller than threshold
12.2%<40%

This pixel should be white

Image after denoising

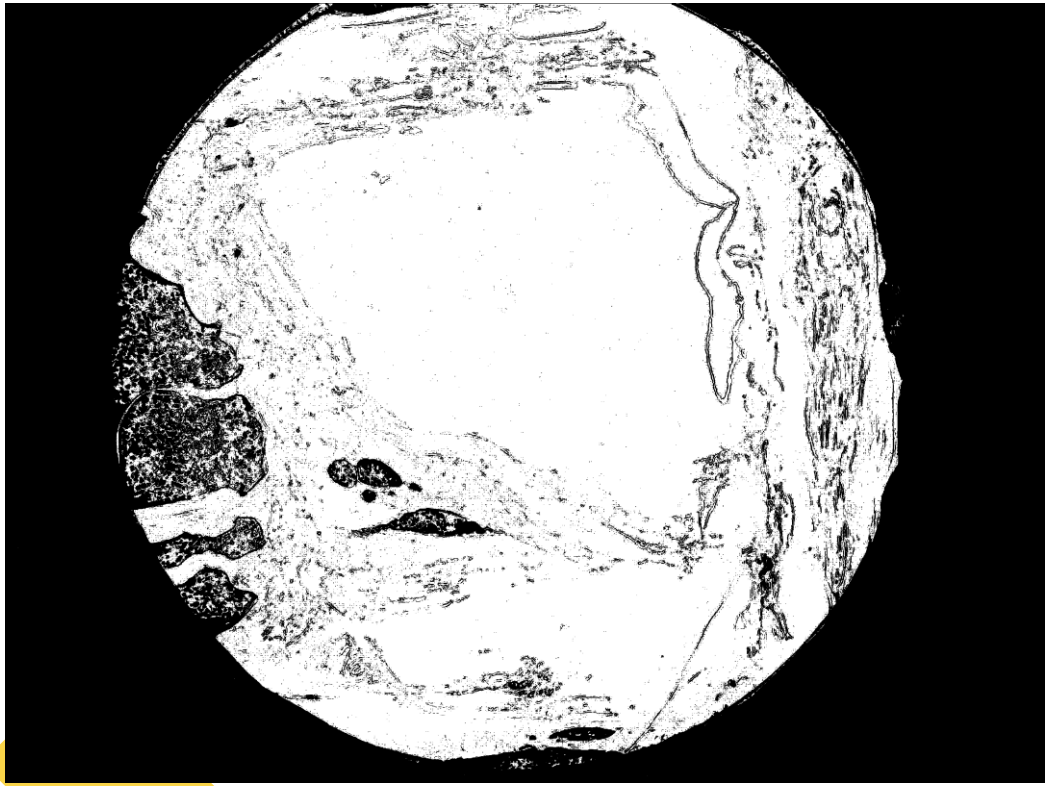


Figure 13. Brightness difference processed image

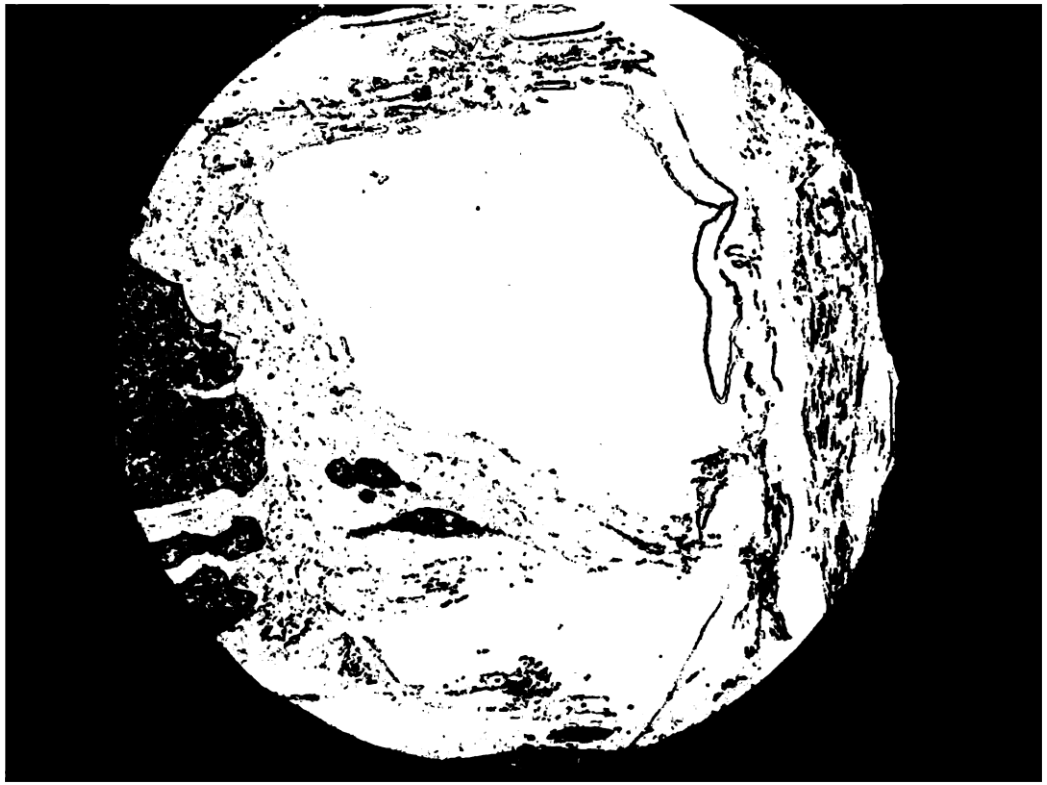


Figure 14. Denoised image (after brightness processed)

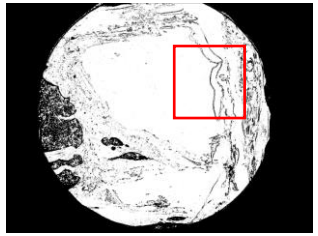


Image after denoising

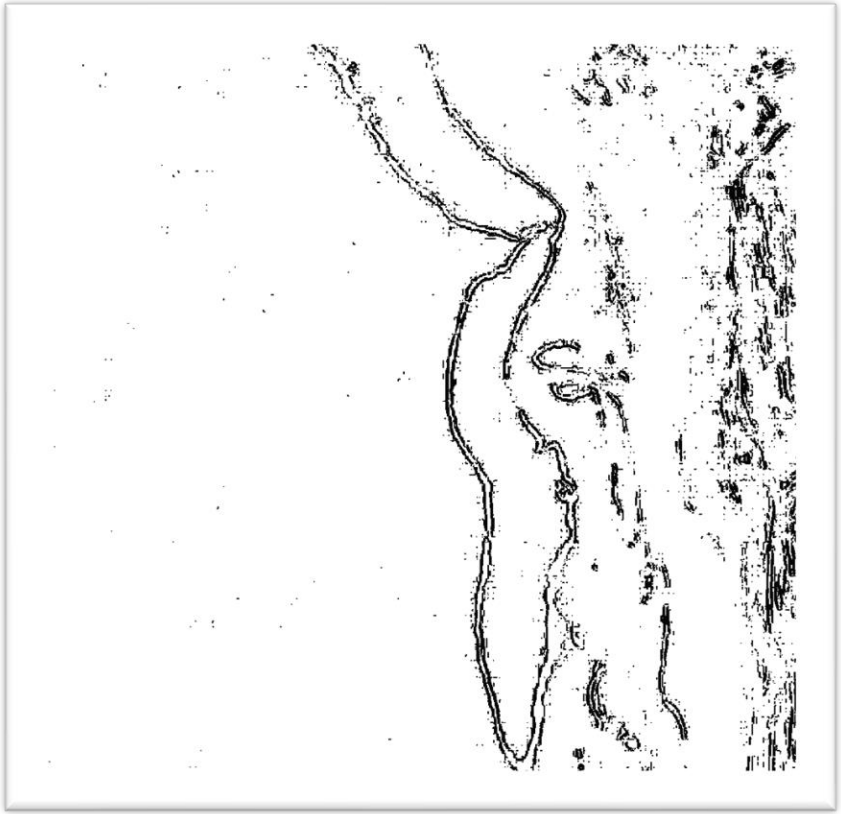


Figure 13-2. Brightness difference processed

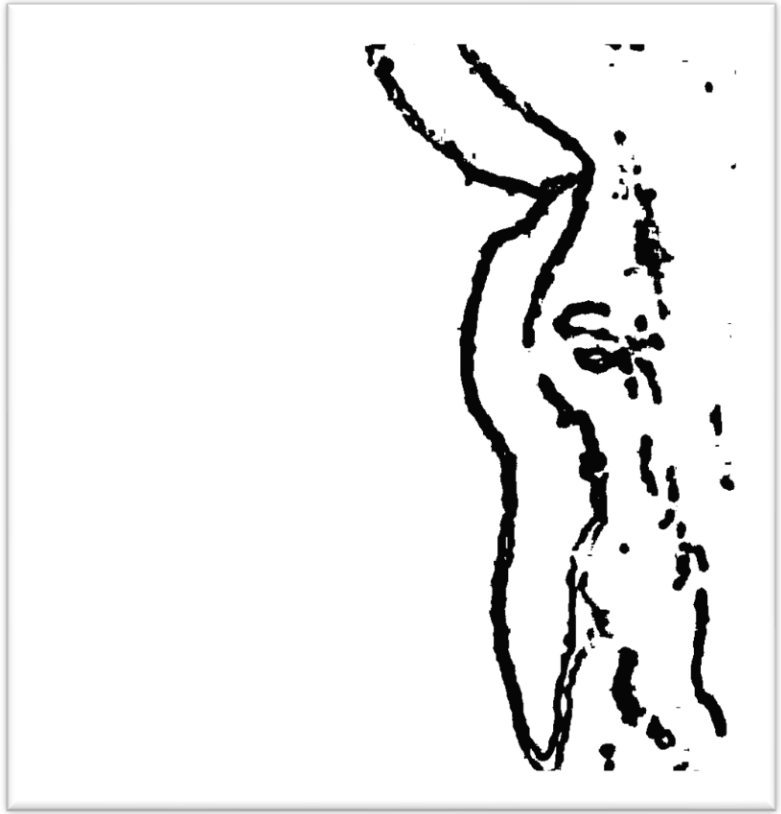


Figure 14-2. Denoised image (after brightness processed)

Image after denoising

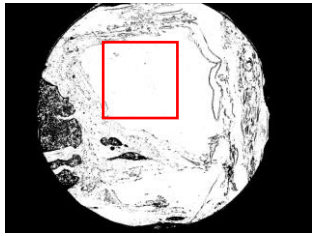
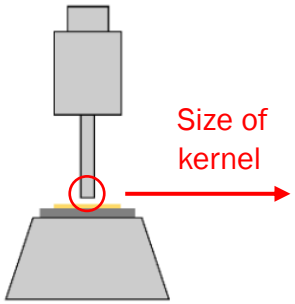


Figure 13-3. Brightness difference processed

Figure 14-3. Denoised image (after brightness processed)

Testing position locating



- If no black pixel inside the kernel

  ➔All pixels inside kernel are white (good region)

  ➔This region capable doing compression test

Gray regions = Regions capable for testing

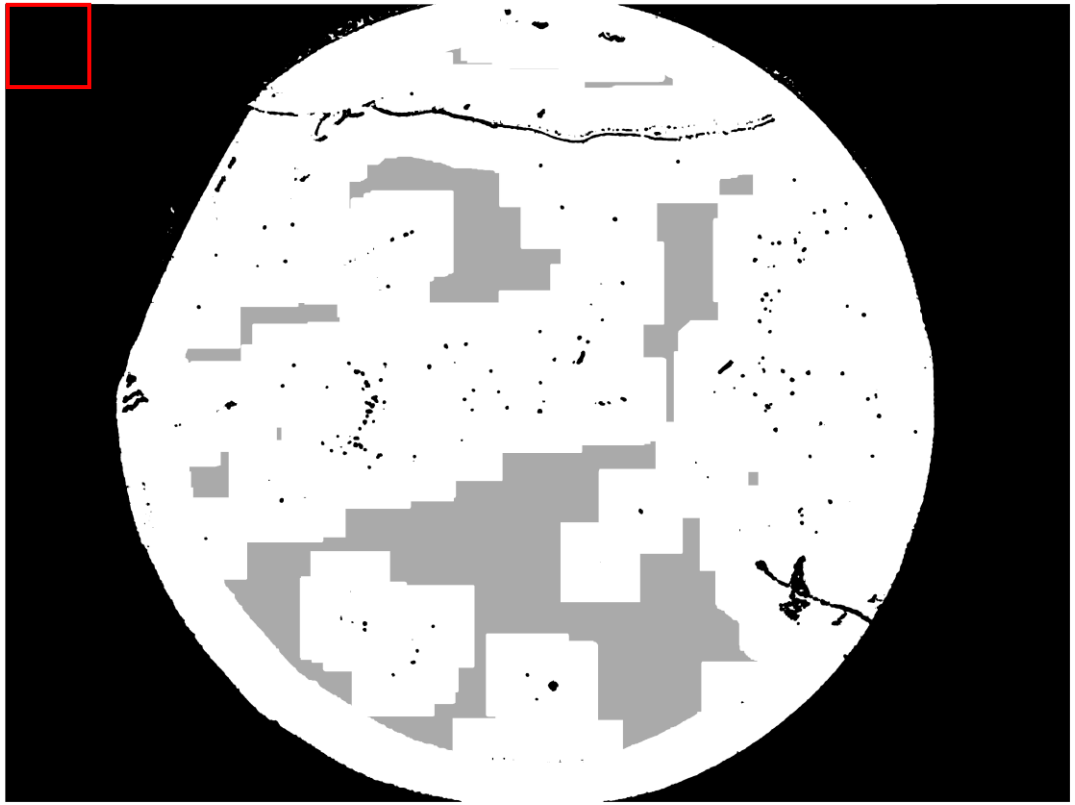


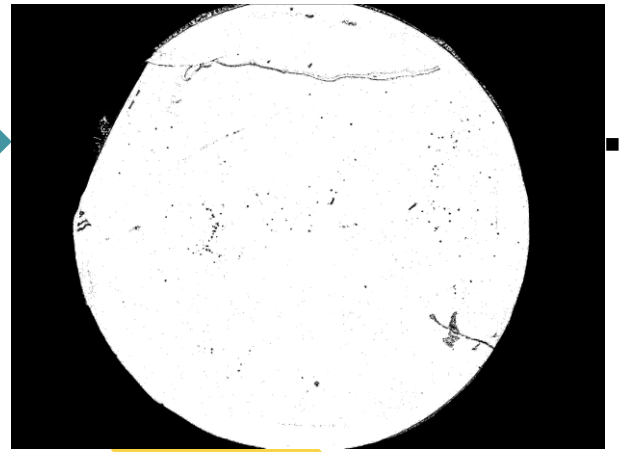
Figure 15. Membrane testing region labelled

- Image after processing
- Gray regions indicating regions where there are no black pixels near it in the range of the size of the kernel

Vision Detection Algorithm

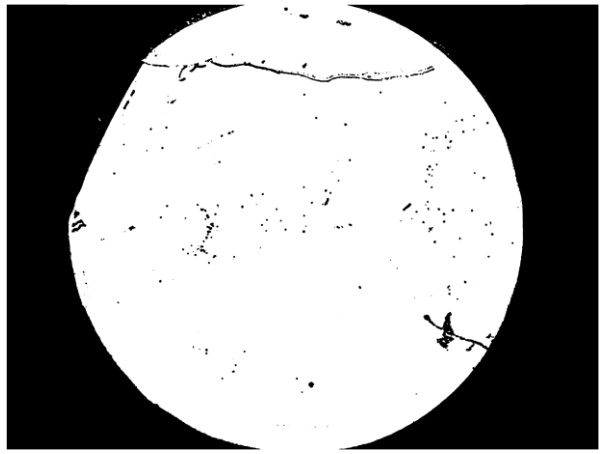
- Original Image (gray scale)

30-40 seconds



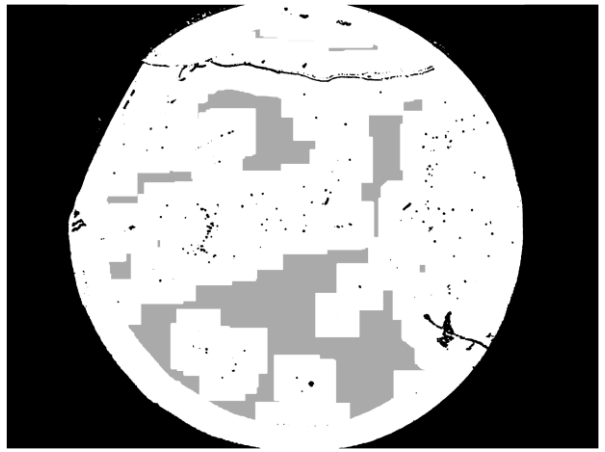
- Brightness difference processing

15-20 minutes



- Image denoising

15-20 minutes



- Testing position locating

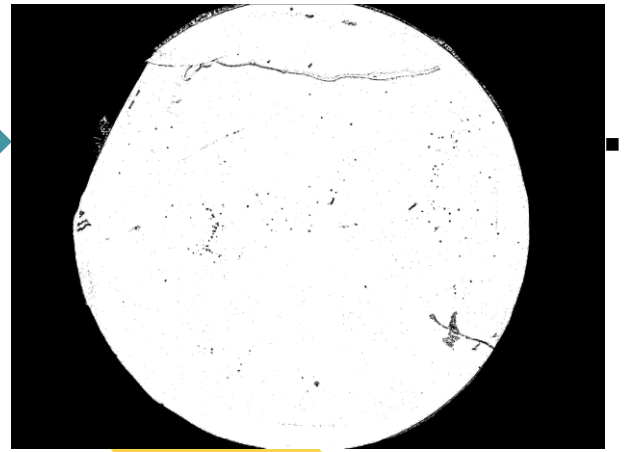
Vision Detection Algorithm

Accelerate algorithm using method such as:

- Integral Image
- Convolution algorithm

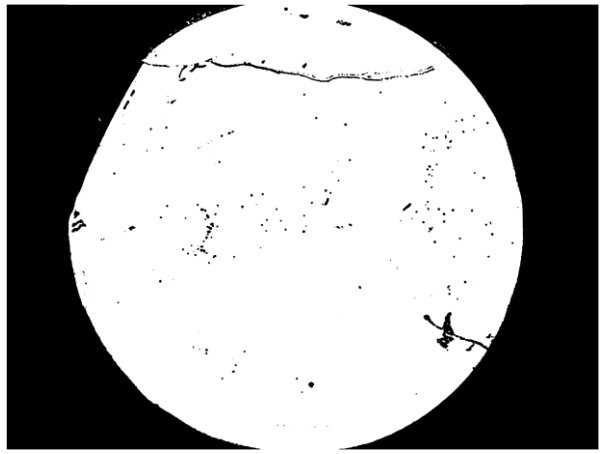
- Original Image (gray scale)

30-40 seconds



- Brightness difference processing

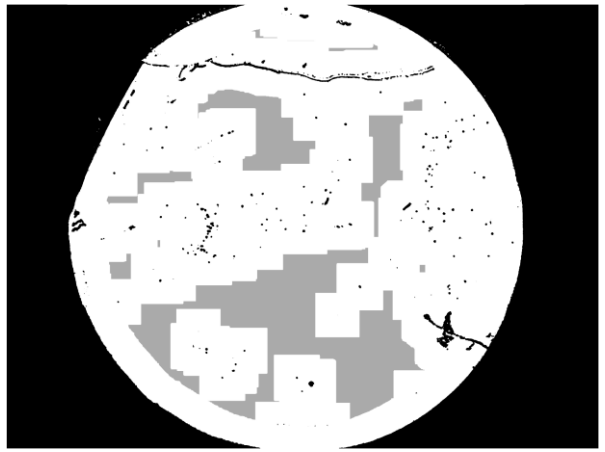
~~15-20 minutes~~ 30 seconds



- Image denoising

~~>20 minutes~~

30 seconds



- Testing position locating

Module Objectives

1. Image Segmentation Algorithm

Objectives:

- Determine if a sample's quality is good enough for testing
- Identify a specific region for testing

2. Physical Camera Module

Objectives:

- Take clear photos of samples
- Interactable with robot arm

3. Arm-Module Interaction

Objectives:

- Robot arm programming

Physical Camera Module

- The physical module for picture taking
- Able to take clear picture of the membrane
  - The picture should not be affected by outer lighting
- Using phone to take photo (or potentially a camera)
- Interactable with robot arm
  - Should be designed for membrane-serving coupon

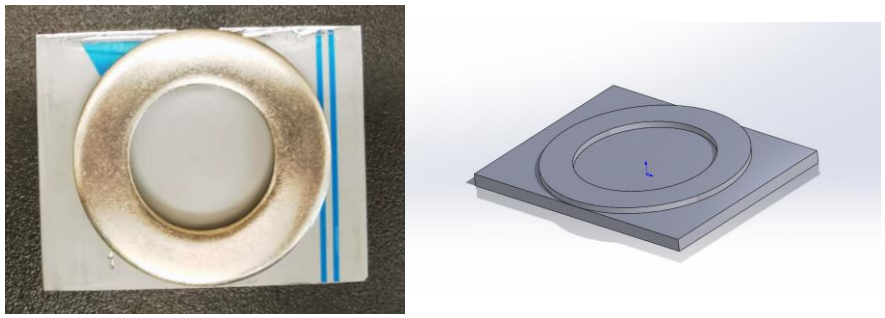


Figure 16&17. Membrane serving coupon

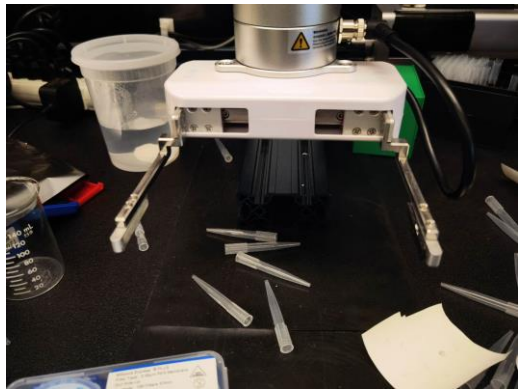


Figure 18. xArm robot arm for automatic processing

Physical Camera Module

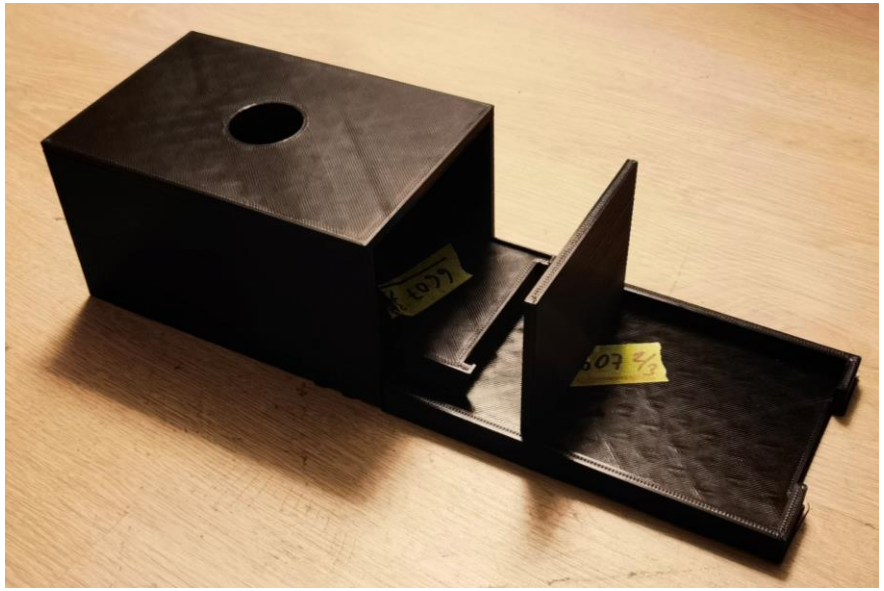
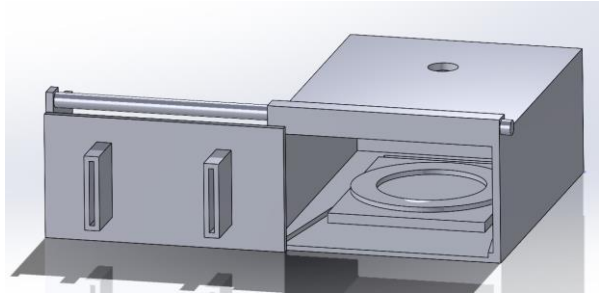


Figure 19&20. Physical Camera Module

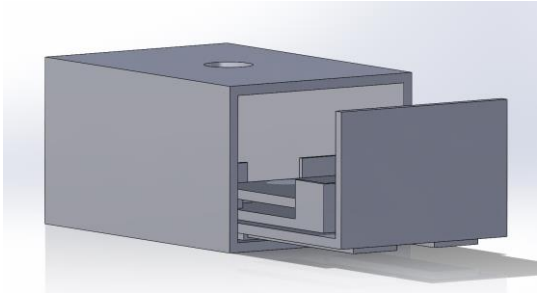
- Use a box to ensure the photo is taken in an environment with consistent lighting

Box Implementation

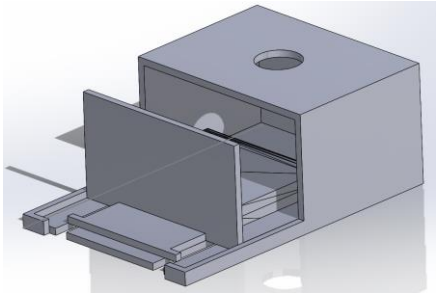
- A total of 6 versions of iterations are designed



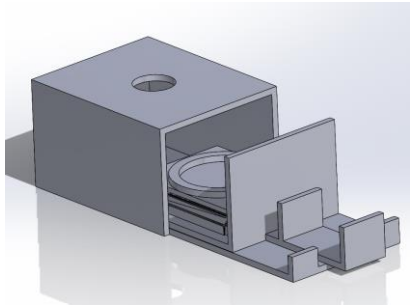
V1



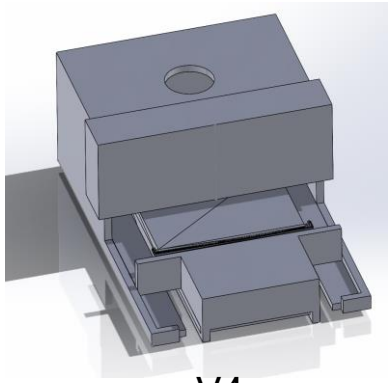
V2



V3



V5

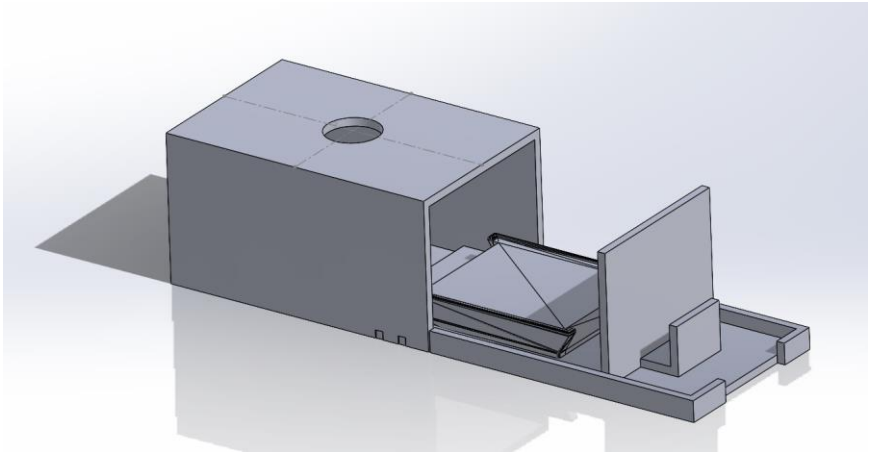


V4

3D printing used for box fabrication at Myhal Digital Fabrication Facility

Material: PLA

Printer: MK3



V6 (currently using)

Module Objectives

1. **Image Segmentation Algorithm**

Objectives:

- Determine if a sample's quality is good enough for testing
- Identify a specific region for testing

2. **Physical Camera Module**

Objectives:

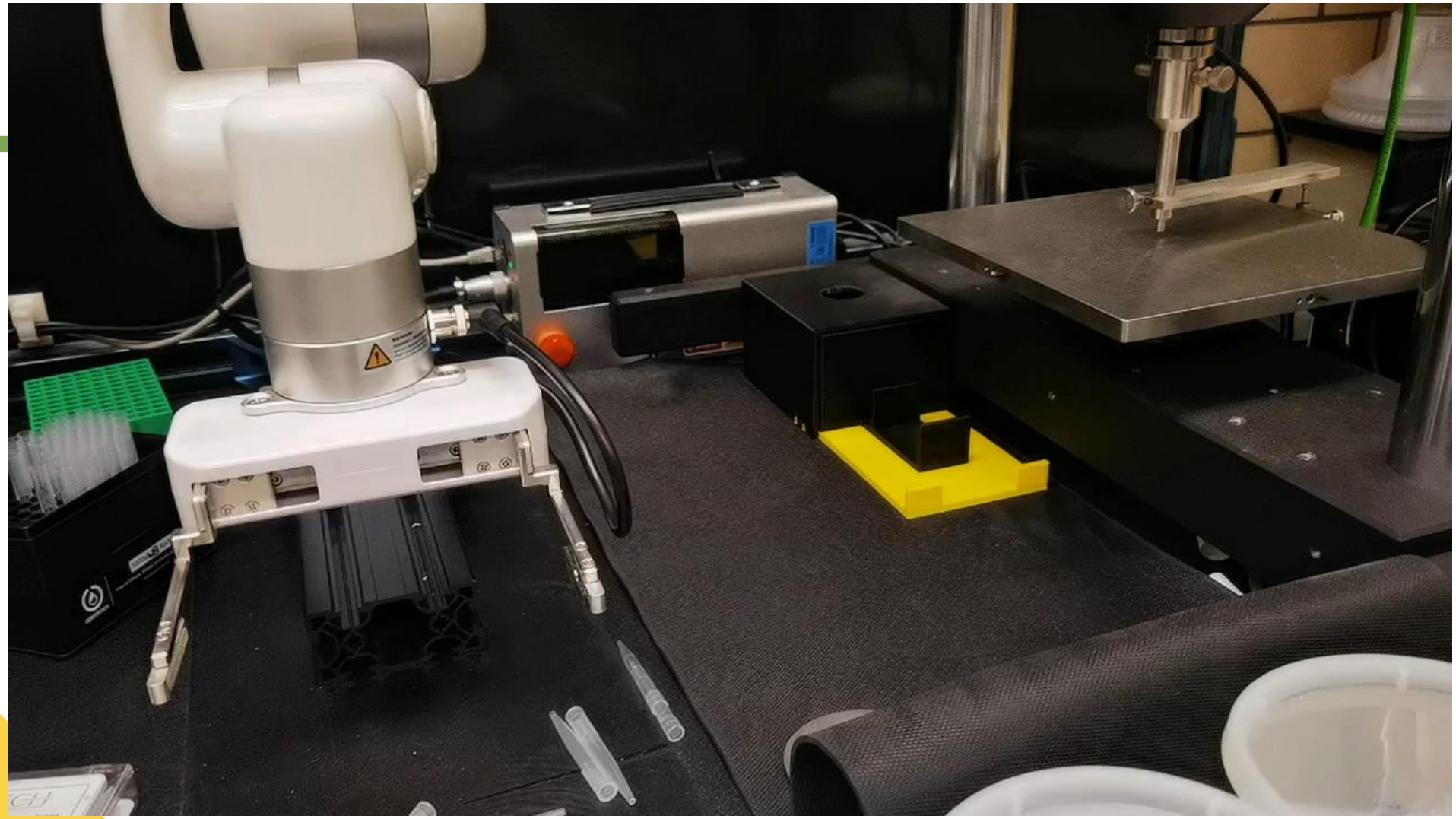
- Take clear photos of samples
- Interactable with robot arm

3. **Arm-Module Interaction**

Objectives:

- Robot arm programming

Box-Arm interaction



Future Work

- Robot arm programming
- Data collection
- Algorithm Improvement

Thank you!

Any Questions?

Reference

- [1] A. L. LLC, “Puretec industrial water: Deionized water services and reverse osmosis systems,” Reverse Osmosis | Puretec Industrial Water, https://puretecwater.com/reverse-osmosis/what-is-reverse-osmosis (accessed Nov. 24, 2023).
- [2] H. Wang, High-Throughput Experimentation for HighPressure Reverse-Osmosis Membranes,
- [3] S. N and V. S, “Image segmentation by using thresholding techniques for medical images,” *Computer Science & Engineering: An International Journal*, vol. 6, no. 1, pp. 1–13, 2016. doi:10.5121/cseij.2016.6101
- [4] B. Preim and C. Botha, “Image Analysis for Medical Visualization,” *Visual Computing for Medicine*, pp. 111–175, 2014. doi:10.1016/b978-0-12-415873-3.00004-3
- [5] V. Badrinarayanan, A. Kendall, and R. Cipolla, “SegNet: A deep convolutional encoder-decoder architecture for image segmentation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017. doi:10.1109/tpami.2016.2644615
- [6] I. Bhat, J. P. Pluim, and H. J. Kuijf, “Generalized probabilistic U-Net for medical image segementation,” *Uncertainty for Safe Utilization of Machine Learning in Medical Imaging*, pp. 113–124, 2022. doi:10.1007/978-3-031-16749-2_11

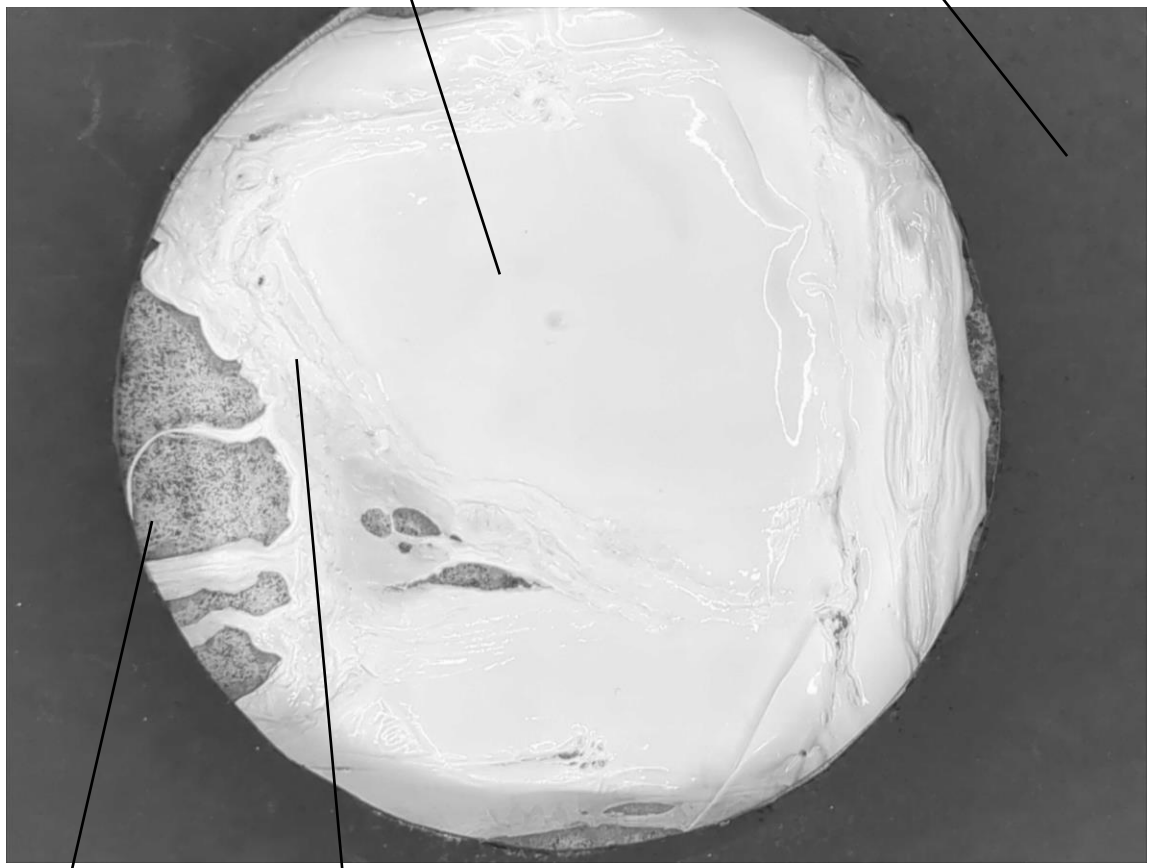
Back up slides

Python Script Implementation

- Use cv2 module to process image
- Use process image in gray scale (ignore any RGB color)
  - Value 255 = pure white (brightest)
  - Value 0 = pure black (darkest)

```
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)
```

- Brightness value for different regions are overlapping
- Cannot use a single thresholding value to reach desired objective
- **Use the brightness value difference between each region to divide out each region**

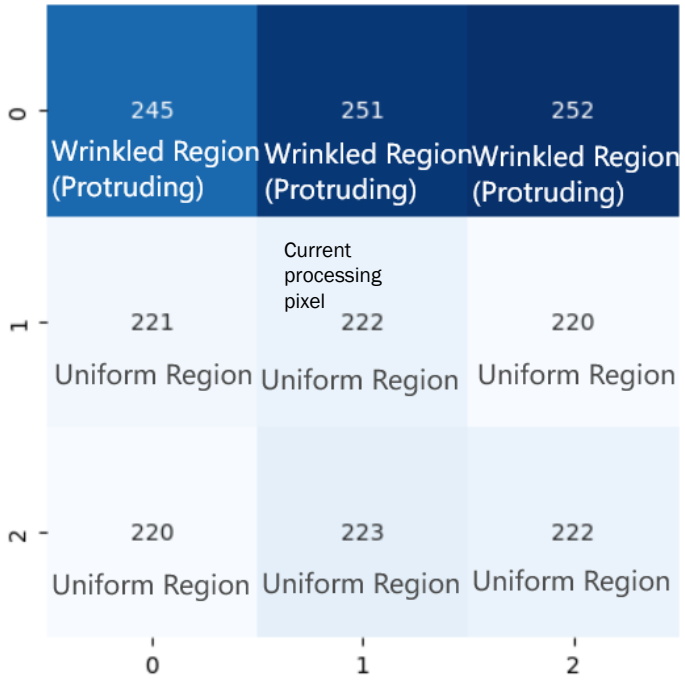


Edge detection using brightness difference

- Processing pixel by pixel using for loop

```
threshold_difference = 4
for i in tqdm(range(len(brightness_color_diff))): # for each row
    for j in range(len(brightness_color_diff[i])): # for each column
        current = int(image_with_padding[i][j])
        top = int(image_with_padding[i-1][j])
        bot = int(image_with_padding[i+1][j])
        left = int(image_with_padding[i][j-1])
        right = int(image_with_padding[i][j+1])
        top_diff = abs(current - top)
        bot_diff = abs(current - bot)
        left_diff = abs(current - left)
        right_diff = abs(current - right)
        diff = max(top_diff, bot_diff, left_diff, right_diff)
        if diff > threshold_difference or brightness_color_diff[i][j] <= 150:
            brightness_color_diff[i][j] = 0
        if brightness_color_diff[i][j] > 150:
            brightness_color_diff[i][j] = 255
```

- Tunable parameter: threshold_difference



- Calculate the brightness difference between current pixel's to its top, left, right and bottom pixel.
- Pick the max brightness difference
- If the max difference exceed the threshold difference, it will be considered as edge, and turned in to black color (brightness=0)
- For any pixel less than 150, turn into 0
- For any pixel greater than 150, turn into 255

Boundary Issue and solution

- Add padding to original image to avoid boundary error

```
                #j   0  1  2  3  4      # i
k_image = np.array([[ 1, 2, 3, 4, 5],   # 0
                    [ 6, 7, 8, 9,10],   # 1
                    [11,12,13,14,15],   # 2
                    [16,17,18,19,20]])  # 3

                   #x    0   1   2   3   4   5   6     # y
k_padded = np.array([[ 0,  0,  0,  0,  0,  0,  0],   # 0
                     [ 0,  1,  2,  3,  4,  5,  0],   # 1
                     [ 0,  6,  7,  8,  9, 10,  0],   # 2
                     [ 0, 11, 12, 13, 14, 15,  0],   # 3
                     [ 0, 16, 17, 18, 19, 20,  0],   # 4
                     [ 0,  0,  0,  0,  0,  0,  0]])  # 5
```

```
def generate_padding(image, border_size=1):
    # Get the dimensions of the original image
    height, width = image.shape

    # Create a new image with the desired size
    padded_image = np.zeros((height + 2 *
border_size, width + 2 * border_size),
dtype=np.uint8)

    # Copy the original image to the center of
the new image
    padded_image[border_size:border_size+height,
border_size:border_size+width] = image

    return padded_image
```

- When encountering image's corner and edges, there won't be any extra pixel outside the range for calculation (e.g. at right edge of the image, there won't be a pixel at right side of it for brightness difference calculation)
- Solution: Add padding, filling zeroes around the image to avoid error
- As the edge of the image are usually background and unusual information, it is ok to fill the padding with 0

Denoising using kernel and black pixel ratio thresholding

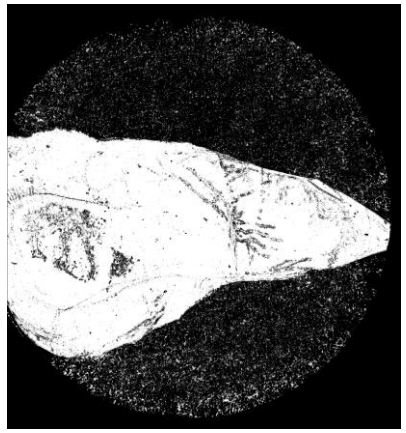
- Code implementation

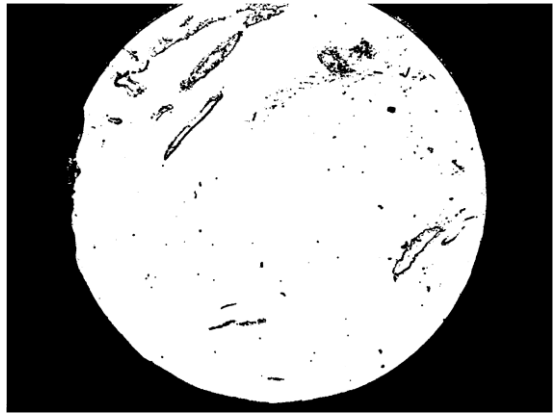
```python
kernel_n = 9
border_size = kernel_n // 2
threshold = 0.4

test_image_1 = image_test1.copy()
test_image_2 = generate_padding(test_image_1, border_size=border_size)
for i in tqdm(range(len(image_test1))):
    for j in range(len(image_test1[i])):
        y = i + border_size
        x = j + border_size
        kernel = test_image_2[y-border_size:y+border_size+1, x-border_size:x+border_size+1]
        black_pixel_count = 0
        for row in kernel:
            for element in row:
                if element == 0:
                    black_pixel_count += 1
        black_percentage = black_pixel_count / (kernel_n * kernel_n)
        if black_percentage > threshold:
            test_image_1[i][j] = 0
        else:
            test_image_1[i][j] = 255

```

- Setting a kernel sliding through each pixel
- Count the number of black pixel inside the kernel
- Use the ratio to determine whether this pixel should be black or white.

- Tunnable parameters:
  kernel_n (kernel size, must be odd)
  threshold (threshold for black pixel ratio)

Image after denoising

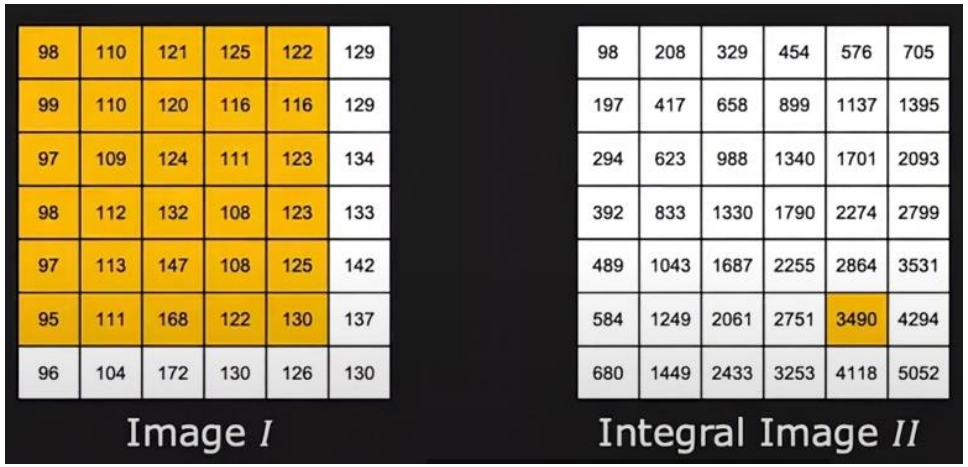




- After the image is denoised, we can assume the following points:
  - All white region (pixel with brightness = 255) are considered as region without defect
  - All black region (pixel with brightness = 0) are considered as region with defect or non-interested region

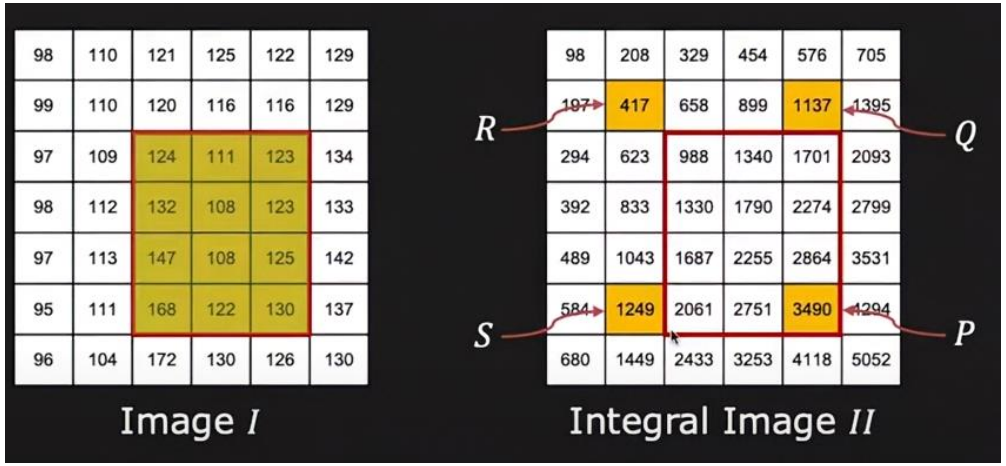
- To fulfill the object: identify region for compression testing, find continuous white region that is large enough for compression testing

Integral Image Algorithm



- First add up each column, then add up each row

```
def calculate_integral_image(image):
    integral_image = np.cumsum(np.cumsum(image, axis=0), axis=1)
    return integral_image
```



- Sum inside the square = P-S-Q+R

```
# Iterate over each pixel in the original image
for y in range(border_size, image.shape[0] - border_size):
    for x in range(border_size, image.shape[1] - border_size):
        # Calculate the sum of the kernel area using the integral image
        total = integral_image[y + border_size, x + border_size]            # P
        total -= integral_image[y + border_size, x - border_size - 1]       # -S
        total -= integral_image[y - border_size - 1, x + border_size]       # -Q
        total += integral_image[y - border_size - 1, x - border_size - 1]   # +R
        # Calculate the number of black pixels (assuming black is 0 and white is
255)
        black_pixel_count = (kernel_n**2) - total / 255
        # Apply threshold
        if black_pixel_count > threshold_value:
            output_image[y, x] = 0
```

Convolution Algorithm

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

- Convolution Definition

```
input_array = np.array([[1, 2, 3],
                        [4, 5, 6],
                        [7, 8, 9]])
kernel = np.array([[1, 0, -1],
                   [1, 0, -1],
                   [1, 0, -1]])
input_array_paddeled = np.array([[0, 0, 0, 0, 0],
                                 [0, 1, 2, 3, 0],
                                 [0, 4, 5, 6, 0],
                                 [0, 7, 8, 9, 0],
                                 [0, 0, 0, 0, 0]])
```

```
# rotate the kernel by 180 degrees
print(np.rot90(kernel, 2))
```

✓ 0.0s

```
[[-1  0  1]
 [-1  0  1]
 [-1  0  1]]
```

```
output_array = convolve2d(input_array, kernel, mode='same', boundary='fill', fillvalue=0)
output_array
```

✓ 0.0s

```
array([[  7,   4,  -7],
       [ 15,   6, -15],
       [ 13,   4, -13]])
```

```
# output_array[0][0] =
0*1 + 0*1 + 0*1 + 0*0 + 1*0 + 4*0 + 0*-1 + 2*-1 + 5*-1
```

✓ 0.0s

```
-7
```

Convolution Algorithm

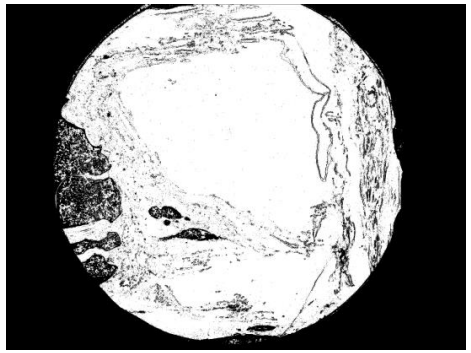
- However, if using for testing position locating, the convolve2d code took nearly 30 minutes for processing
- This could be resulted because the large kernel size (301 pixels), so the convolution algorithm took even longer time for processing

```
kernel_n = 301 # must be odd
kernel = np.ones((kernel_n, kernel_n))
result_array = convolve2d(image_test2, kernel, mode='same', boundary='fill', fillvalue=0)
# multiply result_array by 1/225
white_pixel_count = result_array / 255
black_pixel_ratio = (81 - white_pixel_count) / 81

image_test3 = image_test2.copy()
for i in tqdm(range(len(image_test3))):
    for j in range(len(image_test3[i])):
        if black_pixel_ratio[i][j] == 0:
            image_test3[i][j] = 170
```

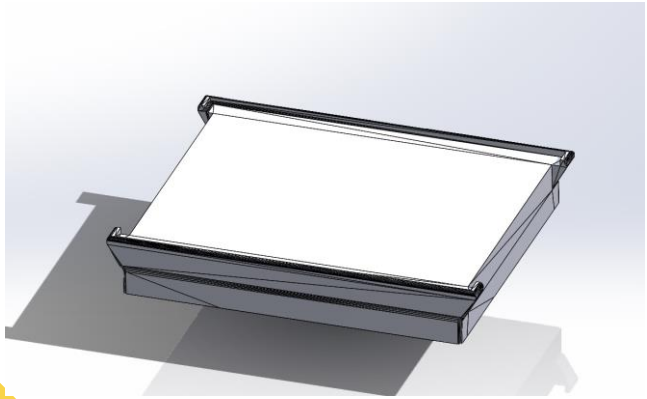
✓ 27m 31.3s

```
100%|██████████| 3024/3024 [00:03<00:00, 983.67it/s]
```

Box-Arm interaction



e.g. pixel (2000, 900) is capable for compression testing



Convert to relative position on coupon

e.g. (20,20) relative to coupon position is capable for compression testing

Move the coupons to position (230+20, 135+20) for compression testing