Wholly owned by UTAR Education Foundation
(Co. No. 578227-M)
DU012(A)

UEMH4334 Machine Vision

Practical 2 Report

2024

Student Name	Ng Yen Keat
Student ID	2202489
Program	MH
Practical Group	P1
Marks	

Table of Contents

0.0 Equipment and Materials

Item Description	*Item category	Quantity estimation (e.g. per set/group of student)
MATLAB (Image Processing Toolbox)	*S*	*1*
Python (Anaconda) + OpenCV	*S*	*1*
Visual Studio (Visual Studio Code) + OpenCV	*S*	*1*
MVTec Halcon	*S*	*1*

*Item category	
SP	Sample or specimen
C	Consumable
CH	Chemical
W	Labware, glassware, tool, and components
E	Equipment
S	Software

1.0 Introduction

This lab covers the application of object and feature-based methods to isolate the object of interest from various images using image processing techniques. The main objective of this lab is to understand and utilize different methods of object recognition to identify objects and differentiate their advantages and disadvantages when dealing with other types of objects. From this lab, I have learnt the importance of proper lighting setup, object arrangement, focal length and aperture size in making the image processing method more effective and efficient to recognize the object and thus able to easily remove the background without affecting or distorting the object edge pixels.

2.0 Theory/Hypothesis

The Sobel operator uses 3 x 3 convolutional kernels to detect changes in intensity in the x, y-directions, computing the image intensity gradient at each point. It is sensitive to high-frequency noise and is often equipped with a Gaussian filter to remove the noise. The configuration makes it suitable for general edge detection tasks. Scharr operator is the enhancement of the Sobel operator; therefore, it will produce a finer and more accurate result by using a different weighing kernel. Robert's operator is one of the earliest edge detection methods, and it uses a small, simple kernel to compute the gradient in a diagonal direction. As a result, it is used for quick and straightforward edge detection tasks and is more susceptible to noise, producing fragment images in noisy images. The Prewitt operator is similar to the Sobel operator with a slightly different mask and does not emphasize any pixels close to the centre; thus, it would produce very similar results to the Sobel. Canny edge detection is a multi-stage detection where it first uses the Gaussian blur to filter out the white noise, secondly calculates the intensity gradient in each direction, thirdly implements the non-maximum suppression to retain the local maxima in the gradient direction, and lastly uses double thresholding method to recognize strong and weak edge to classify the object edge which improves its edge detection accuracy and makes it highly effective detect for edges in low noise. Laplacian of the Gaussian method relies on the second derivative of the image intensity, which can easily highlight the area of rapid change to detect the blob and can detect and respond to the edges in all directions as it isotropic. Otsu's method is an automatic thresholding technique where that automatically calculates the optimal threshold by minimizing the intra-class variance or maximizing the inter-class variance. It is helpful for

images which have bimodal histograms to separate into the foreground and background clusters using a threshold, but it may not perform well in non-uniform lighting images.

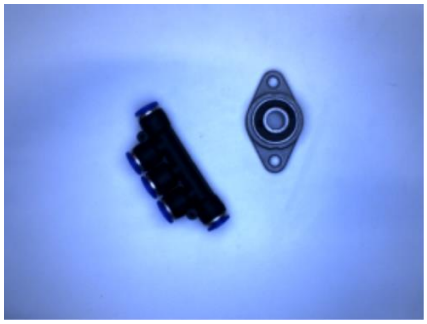
Next, the adaptive mean thresholding method is based on the mean of the local neighbourhood minus a constant to find the threshold value, which can easily find the object under varying light conditions compared to the global thresholding method. The adaptive Gaussian thresholding method of a threshold value for each region is determined by the weighted sum of the pixel intensities within the neighbourhood, and it may produce a smoother image than adaptive mean thresholding. Niblack's algorithm is a local thresholding method based on the calculation of the local mean and local standard deviation of the image intensity, and it is effective for high-contrast images. Sauvola's method is an improvement over Niblack's method and will perform better in terms of varying light conditions, binarization of images, and noise reduction. Bernsen's method sets its threshold value to the midrange of the grey value in the local window; thus, it can work well for images with moderate contrast and varying illumination. Lastly, the k-mean clustering method will effectively distinguish the boundary between the object and the background based on different pixel intensities, but it may struggle with the different intensities on the detail of the object.

3.0 Results

3.1) Images taken from lab (samples)

Picture 1

Picture 2



Picture 3

Picture 4

3.2) Image Segmentation Techniques

3.2a) Gradient-based method

a1) Sobel Operator + global thresholding

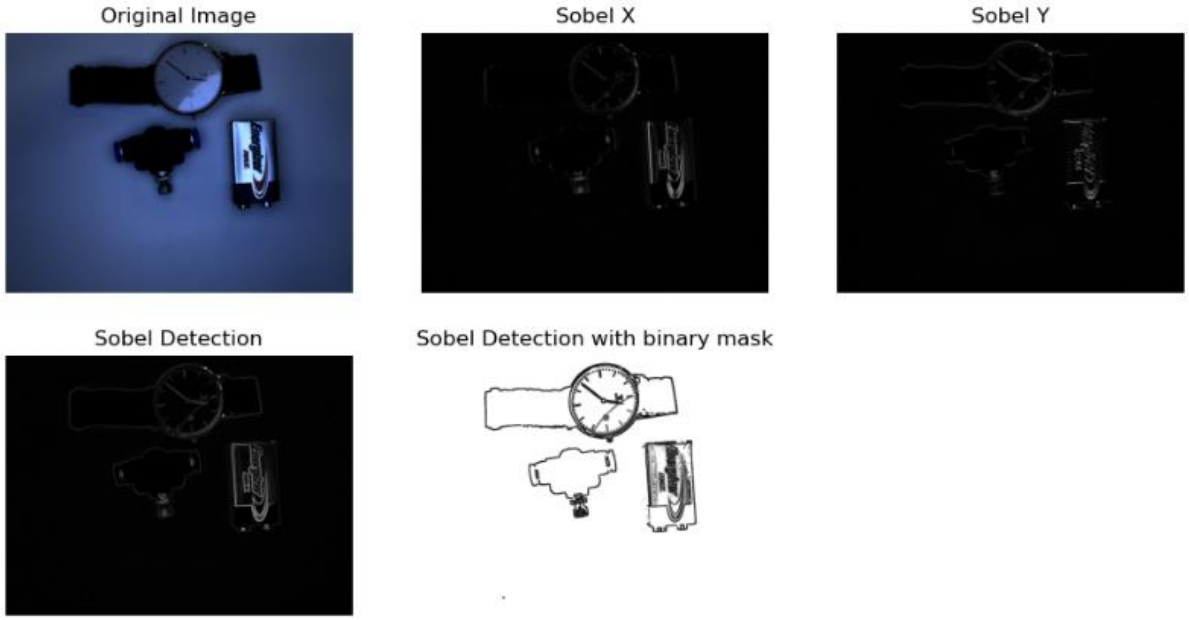
Code

```python
import numpy as np
import cv2
from PIL import Image
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic2.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Apply Gaussian smoothing (optional)
blurred_image = cv2.GaussianBlur(gray_image, (3, 3), 0)

# Sobel operators
Gx = cv2.Sobel(blurred_image, cv2.CV_64F, 1, 0, ksize=3)
Gy = cv2.Sobel(blurred_image, cv2.CV_64F, 0, 1, ksize=3)

# Gradient magnitude
G = np.sqrt(Gx**2 + Gy**2)

# Normalize to range 0-255
Gx = np.uint8(255 * np.abs(Gx) / np.max(Gx))
Gy = np.uint8(255 * np.abs(Gy) / np.max(Gy))
G = np.uint8(255 * G / np.max(G))

# Threshold the magnitude image to get binary mask
_, mask = cv2.threshold(G, 17, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Dilate the mask to ensure the entire object is covered
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)

# Invert the mask to get the background
mask_inv = cv2.bitwise_not(mask)

# Create a white background image
white_background = np.ones_like(gray_image) * 255

# Use the mask to extract the object from the original image
object_only = cv2.bitwise_and(gray_image, gray_image, mask=mask)

# Use the inverse mask to extract the background from the white background image
background_only = cv2.bitwise_and(white_background, white_background, mask=mask_inv)

# Combine the object and the white background
final_image = cv2.add(object_only, background_only)
```

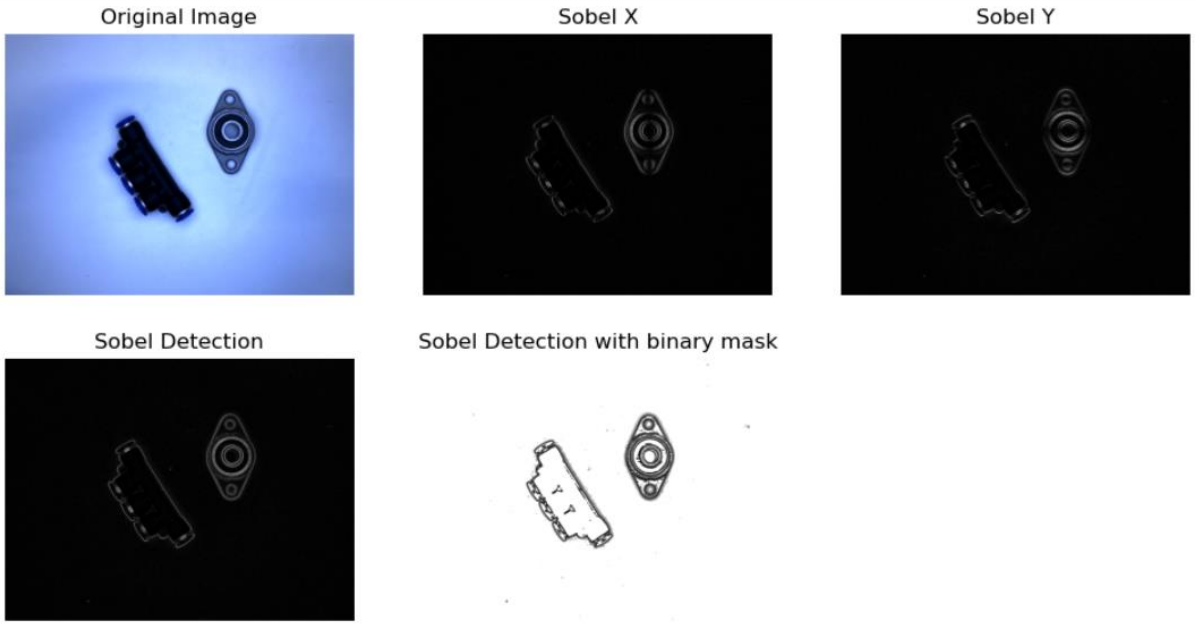
```python
# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Sobel X')
plt.imshow(Gx,cmap ='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Sobel Y')
plt.imshow(Gy,cmap ='gray')
plt.axis('off')

plt.figure(figsize=(12, 6))
plt.subplot(2, 3, 1)
plt.title('Sobel Detection')
plt.imshow(G,cmap ='gray')
plt.axis('off')

plt.subplot(2, 3, 2)
plt.title('Sobel Detection with binary mask')
plt.imshow(final_image,cmap ='gray')
plt.axis('off')

plt.show()
```

The threshold for the binary mask was adjusted differently for different images and detection methods.

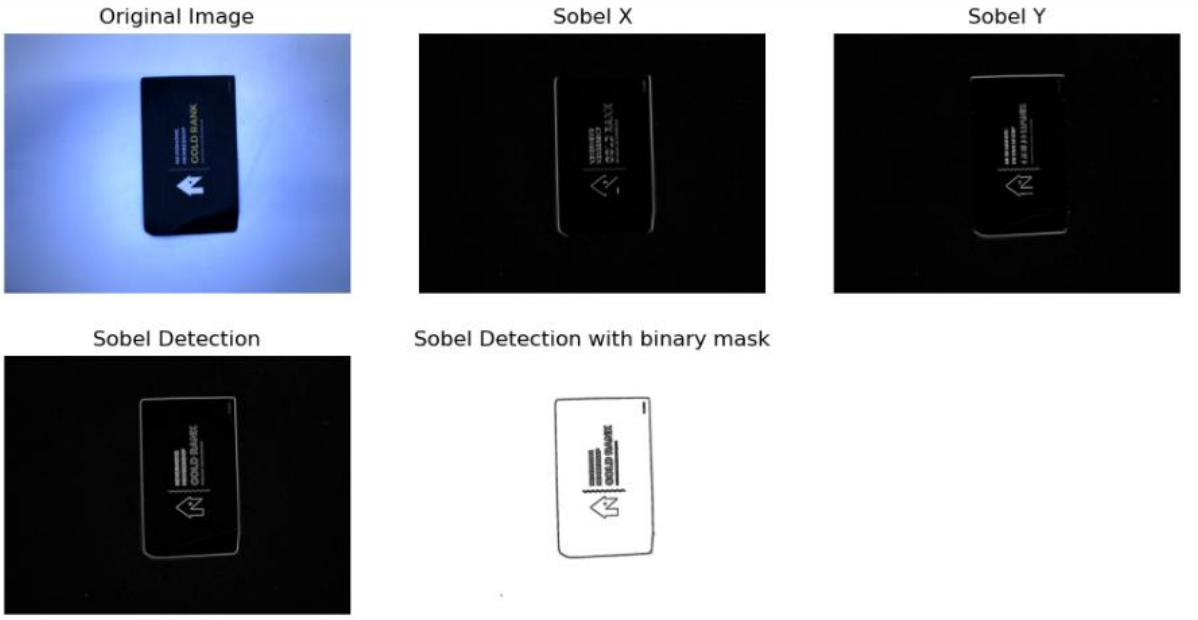
Picture 1



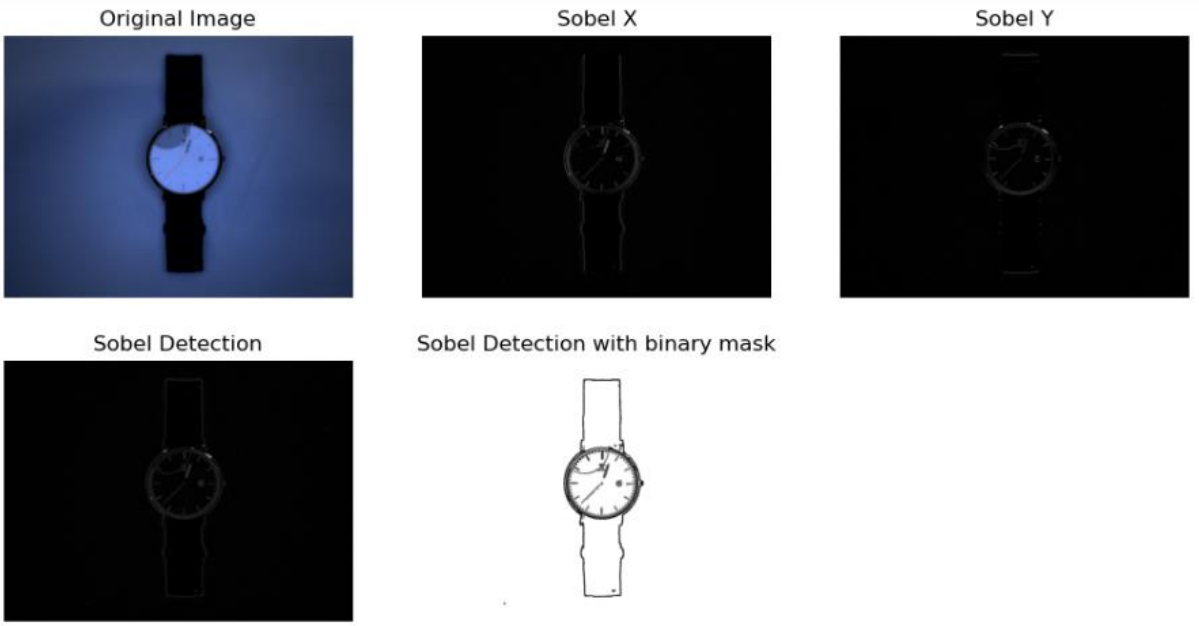
Picture 2



Picture 3



Picture 4



a2) Scharr Operator + global thresholding (Improvement of a1)

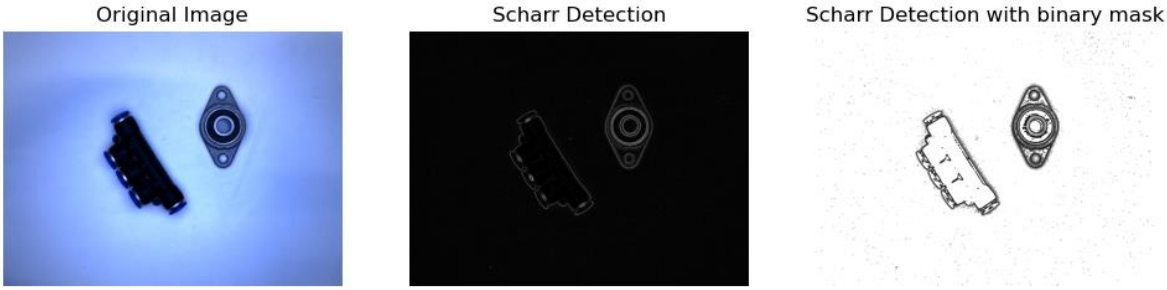
Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic2.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)

def scharr_edge_detection(image):
    # Convert to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    # Apply Scharr operator to find the x and y gradients
    Gx = cv2.Scharr(gray_image, cv2.CV_64F, 1, 0)
    Gy = cv2.Scharr(gray_image, cv2.CV_64F, 0, 1)

    # Compute the gradient magnitude
    gradient_magnitude = cv2.magnitude(Gx, Gy)

    return gray_image, gradient_magnitude

# Detect edges using Scharr operator
gray_image, edges = scharr_edge_detection(img)

# Threshold the magnitude image to get binary mask
_, mask = cv2.threshold(edges, 230, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Dilate the mask to ensure the entire object is covered
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)

# Invert the mask to get the background
mask_inv = cv2.bitwise_not(mask)

# Create a white background image
white_background = np.ones_like(gray_image) * 255

# Use the mask to extract the object from the original image
object_only = cv2.bitwise_and(gray_image, gray_image, mask=mask)
```

```
# Use the inverse mask to extract the background from the white background image
background_only = cv2.bitwise_and(white_background, white_background, mask=mask_inv)

# Combine the object and the white background
final_image = cv2.add(object_only, background_only)

# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Scharr Detection')
plt.imshow(edges,cmap ='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Scharr Detection with binary mask')
plt.imshow(final_image,cmap ='gray')
plt.axis('off')

plt.show()
```

Picture 1

Picture 2



Picture 3

Picture 4

a3) Robert Operator + global thresholding

Code

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic3.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

def roberts_cross_edge_detection(image):
    # Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    # Apply Roberts Cross kernels
    kernel_x = np.array([[1, 0],
                         [0, -1]])
    kernel_y = np.array([[0, 1],
                         [-1, 0]])

    # Convolve the image with the kernels
    horizontal_edges = cv2.filter2D(gray_image, -1, kernel_x)
    vertical_edges = cv2.filter2D(gray_image, -1, kernel_y)

    # Ensure both arrays have the same data type
    horizontal_edges = np.float32(horizontal_edges)
    vertical_edges = np.float32(vertical_edges)

    # Compute gradient magnitude
    gradient_magnitude = cv2.magnitude(horizontal_edges, vertical_edges)

    return gray_image, gradient_magnitude

# Apply Roberts Cross edge detection with thresholding
gray_image,gradient = roberts_cross_edge_detection(img)

# Threshold the magnitude image to get binary mask
_, mask = cv2.threshold(gradient, 15, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Dilate the mask to ensure the entire object is covered
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)
```

```python
# Invert the mask to get the background
mask_inv = cv2.bitwise_not(mask)

# Create a white background image
white_background = np.ones_like(gray_image) * 255

# Use the mask to extract the object from the original image
object_only = cv2.bitwise_and(gray_image, gray_image, mask=mask)

# Use the inverse mask to extract the background from the white background image
background_only = cv2.bitwise_and(white_background, white_background, mask=mask_inv)

# Combine the object and the white background
final_image = cv2.add(object_only, background_only)

# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Robert Detection')
plt.imshow(gradient,cmap ='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Robert Detection with binary mask')
plt.imshow(final_image,cmap ='gray')
plt.axis('off')

plt.show()
```

Picture 1

Picture 2

Picture 3

Picture 4

a4) Prewitt Operator + global thresholding

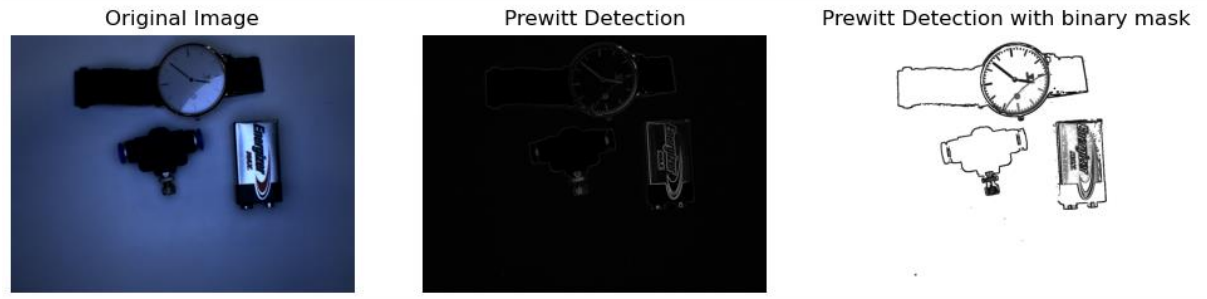
Code

```
import numpy as np
import cv2
from PIL import Image
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic1.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Apply gray scale
gray_img = np.round(0.299 * img[:, :, 0] +
                    0.587 * img[:, :, 1] +
                    0.114 * img[:, :, 2]).astype(np.uint8)

# Prewitt Operator
h, w = gray_img.shape
# define filters
horizontal = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])  # s2
vertical = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])  # s1

# define images with 0s
newgradientImage = np.zeros((h, w))

# offset by 1
for i in range(1, h - 1):
    for j in range(1, w - 1):
        horizontalGrad = (horizontal[0, 0] * gray_img[i - 1, j - 1]) + \
                         (horizontal[0, 1] * gray_img[i - 1, j]) + \
                         (horizontal[0, 2] * gray_img[i - 1, j + 1]) + \
                         (horizontal[1, 0] * gray_img[i, j - 1]) + \
                         (horizontal[1, 1] * gray_img[i, j]) + \
                         (horizontal[1, 2] * gray_img[i, j + 1]) + \
                         (horizontal[2, 0] * gray_img[i + 1, j - 1]) + \
                         (horizontal[2, 1] * gray_img[i + 1, j]) + \
                         (horizontal[2, 2] * gray_img[i + 1, j + 1])

        verticalGrad = (vertical[0, 0] * gray_img[i - 1, j - 1]) + \
                       (vertical[0, 1] * gray_img[i - 1, j]) + \
                       (vertical[0, 2] * gray_img[i - 1, j + 1]) + \
                       (vertical[1, 0] * gray_img[i, j - 1]) + \
                       (vertical[1, 1] * gray_img[i, j]) + \
                       (vertical[1, 2] * gray_img[i, j + 1]) + \
                       (vertical[2, 0] * gray_img[i + 1, j - 1]) + \
                       (vertical[2, 1] * gray_img[i + 1, j]) + \
                       (vertical[2, 2] * gray_img[i + 1, j + 1])
```

```
        # Edge Magnitude
        mag = np.sqrt(pow(horizontalGrad, 2.0) + pow(verticalGrad, 2.0))
        newgradientImage[i - 1, j - 1] = mag

# Threshold the magnitude image to get binary mask
_, mask = cv2.threshold(newgradientImage, 40, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Dilate the mask to ensure the entire object is covered
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)

# Invert the mask to get the background
mask_inv = cv2.bitwise_not(mask)

# Create a white background image
white_background = np.ones_like(gray_img) * 255

# Use the mask to extract the object from the original image
object_only = cv2.bitwise_and(gray_img, gray_img, mask=mask)

# Use the inverse mask to extract the background from the white background image
background_only = cv2.bitwise_and(white_background, white_background, mask=mask_inv)

# Combine the object and the white background
final_image = cv2.add(object_only, background_only)
```

```
# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Prewitt Detection')
plt.imshow(newgradientImage,cmap ='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Prewitt Detection with binary mask')
plt.imshow(final_image,cmap ='gray')
plt.axis('off')

plt.show()
```

Picture 1



Picture 2

Picture 3

Picture 4

3.2b) Gaussian-based method

b1) Canny Edge Detection + global thresholding

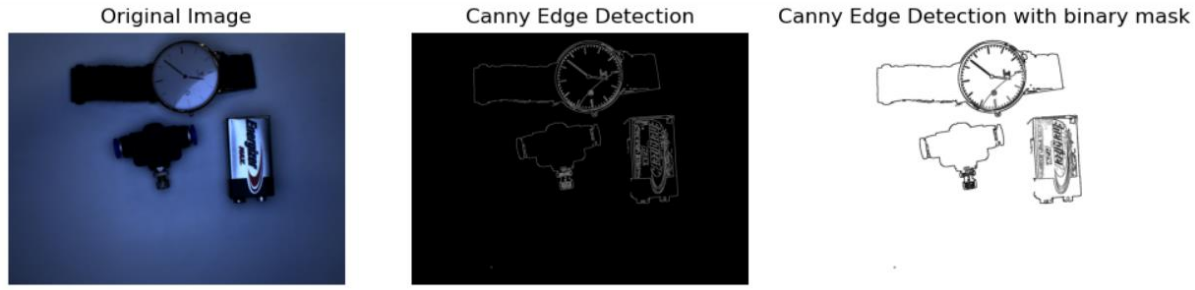
Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic1.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Apply Gaussian smoothing (optional)
blurred_image = cv2.GaussianBlur(gray_image, (3, 3), 0)

# Apply Canny edge detector
edges = cv2.Canny(blurred_image, 25, 65)

# Threshold the magnitude image to get binary mask
_, mask = cv2.threshold(edges, 1, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Dilate the mask to ensure the entire object is covered
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)

# Invert the mask to get the background
mask_inv = cv2.bitwise_not(mask)

# Create a white background image
white_background = np.ones_like(gray_image) * 255

# Use the mask to extract the object from the original image
object_only = cv2.bitwise_and(gray_image, gray_image, mask=mask)

# Use the inverse mask to extract the background from the white background image
background_only = cv2.bitwise_and(white_background, white_background, mask=mask_inv)

# Combine the object and the white background
final_image = cv2.add(object_only, background_only)
```

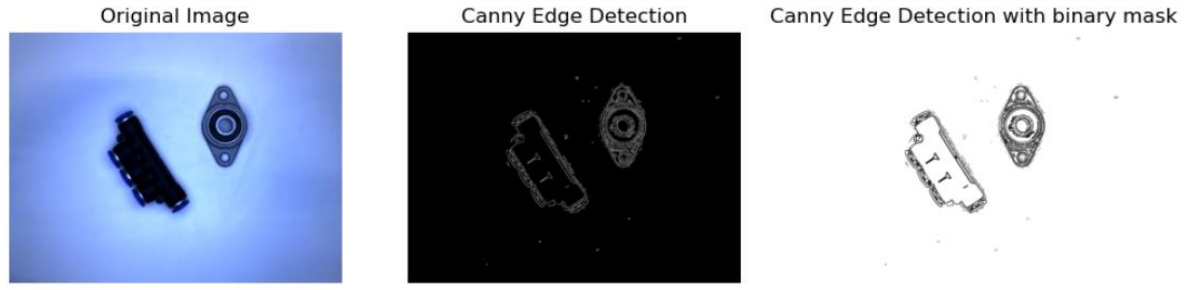
```
# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Canny Edge Detection')
plt.imshow(edges,cmap ='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Canny Edge Detection with binary mask')
plt.imshow(final_image,cmap ='gray')
plt.axis('off')

plt.show()
```

The threshold for weak edge and strong edge were defined at different values accordingly for Picture 1 up to Picture 4 to illustrate the best representation of the result.

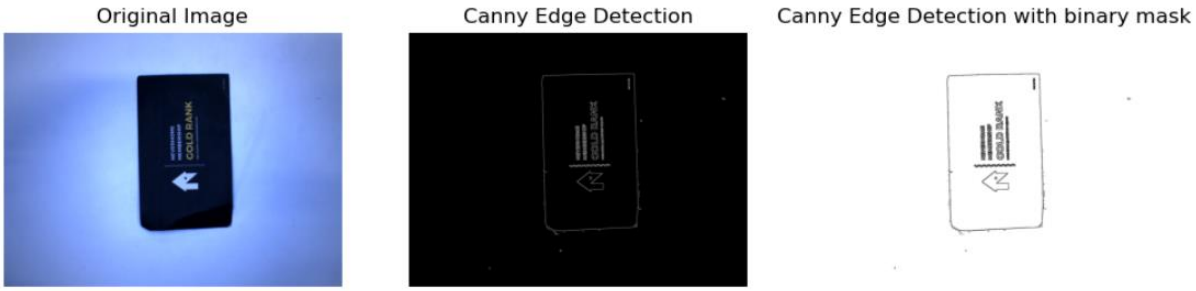
Picture 1



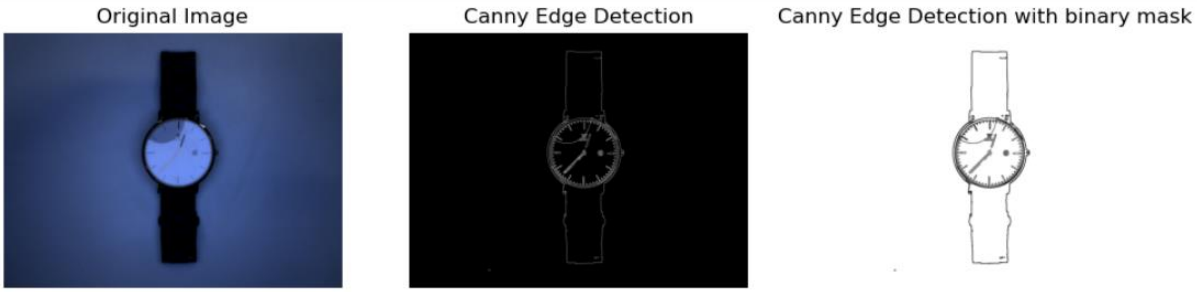
Picture 2



Picture 3



Picture 4



b2) Laplacian of Gaussian + global thresholding

Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic3.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Apply Gaussian smoothing (optional)
blurred_image = cv2.GaussianBlur(gray_image, (3, 3), 0)

# Apply the Laplacian operator
laplacian = cv2.Laplacian(blurred_image, cv2.CV_64F)

# Convert the result to 8-bit (0-255) range
laplacian_abs = cv2.convertScaleAbs(laplacian)

# Threshold the magnitude image to get binary mask
_, mask = cv2.threshold(laplacian_abs, 12, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Dilate the mask to ensure the entire object is covered
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)

# Invert the mask to get the background
mask_inv = cv2.bitwise_not(mask)

# Create a white background image
white_background = np.ones_like(gray_image) * 255

# Use the mask to extract the object from the original image
object_only = cv2.bitwise_and(gray_image, gray_image, mask=mask)

# Use the inverse mask to extract the background from the white background image
background_only = cv2.bitwise_and(white_background, white_background, mask=mask_inv)

# Combine the object and the white background
final_image = cv2.add(object_only, background_only)
```

```
# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Laplacian of Gaussian Detection')
plt.imshow(laplacian_abs,cmap ='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Laplacian of Gaussian with binary mask')
plt.imshow(final_image,cmap ='gray')
plt.axis('off')

plt.show()
```

Picture 1

Picture 2

Picture 3

Picture 4

3.2c) Global Thresholding

c1) Global Thresholding

Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic1.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Manual thresholding
threshold_value = 35
# Example manual threshold value
_, mask = cv2.threshold(gray_image, threshold_value, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Global thresholding')
plt.imshow(mask,cmap ='gray')
plt.axis('off')


plt.show()
```

Picture 1

Picture 2

Picture 3

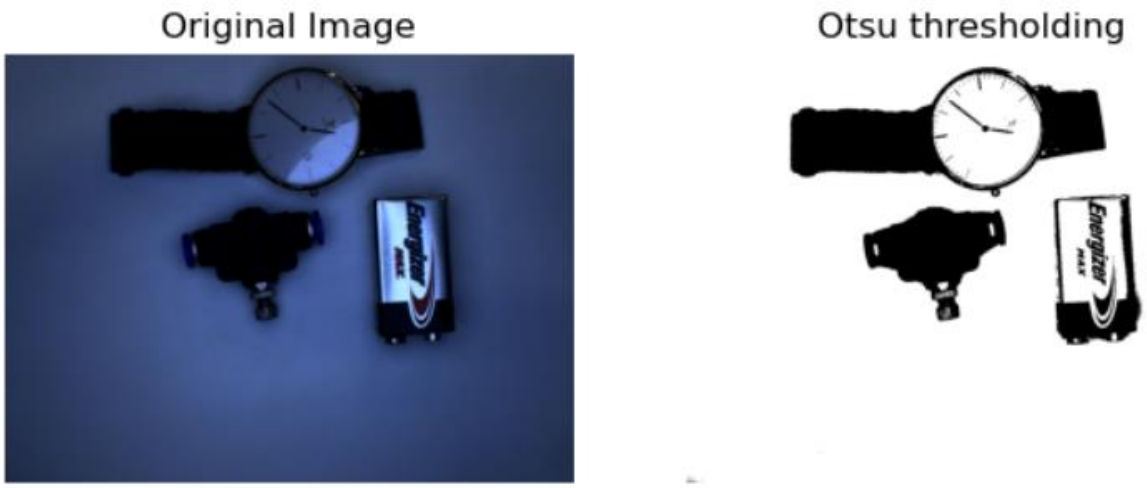
Picture 4

c2) Otsu’s Method

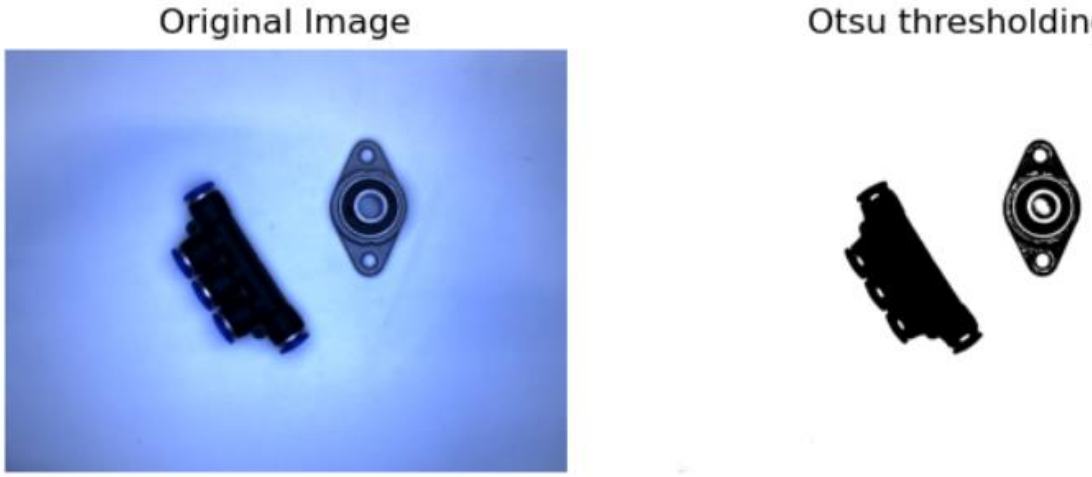
Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic1.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# applying Otsu thresholding
# as an extra flag in binary
# thresholding

'''
120,255 threshold values are ignored,because the otsu automatically
compute the optimal threshold value.
'''

ret, thresh1 = cv2.threshold(gray_image, 120, 255, cv2.THRESH_BINARY +
                                          cv2.THRESH_OTSU)


# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Otsu thresholding')
plt.imshow(thresh1,cmap ='gray')
plt.axis('off')

plt.show()
```

Picture 1

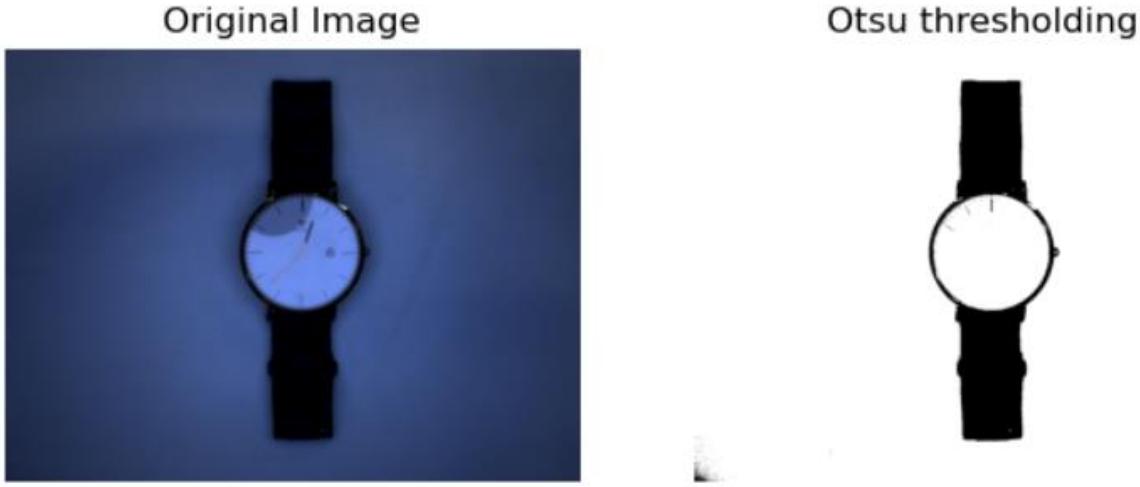


Picture 2



Picture 3

Picture 4



3.2d) Local Thresholding

d1) Adaptive Mean Thresholding

Code

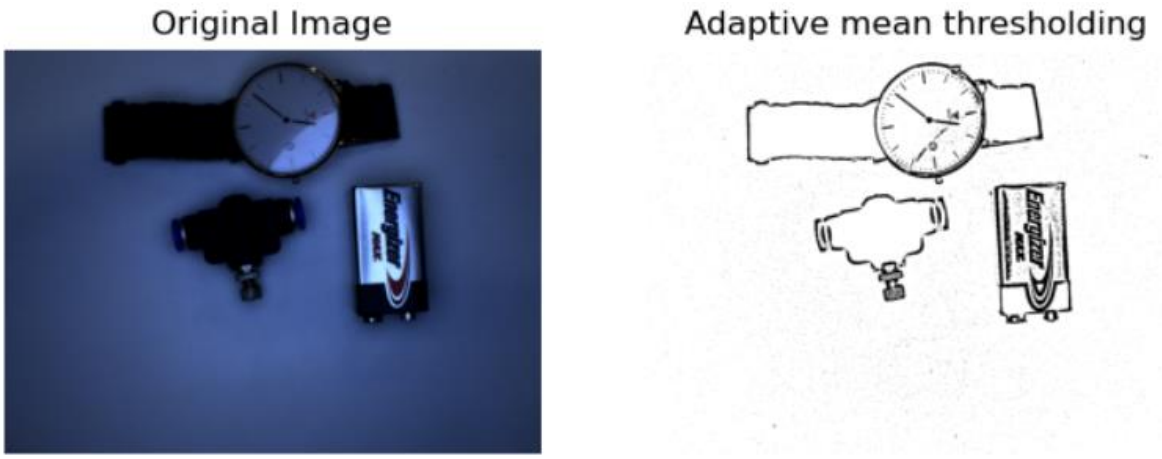
```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic4.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)


# Apply Adaptive Mean Thresholding
adaptive_mean = cv2.adaptiveThreshold(gray_image, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
                                      cv2.THRESH_BINARY, 19, 7)

# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Adaptive mean thresholding')
plt.imshow(adaptive_mean,cmap ='gray')
plt.axis('off')

plt.show()
```

Parameters:

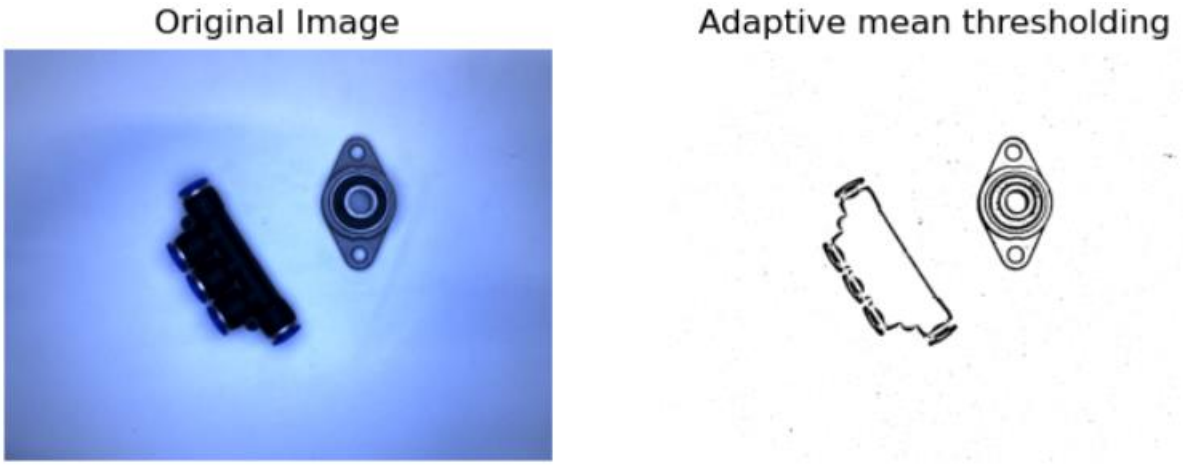
Size of neighbourhood pixels,

C (constant for subtracting the mean)

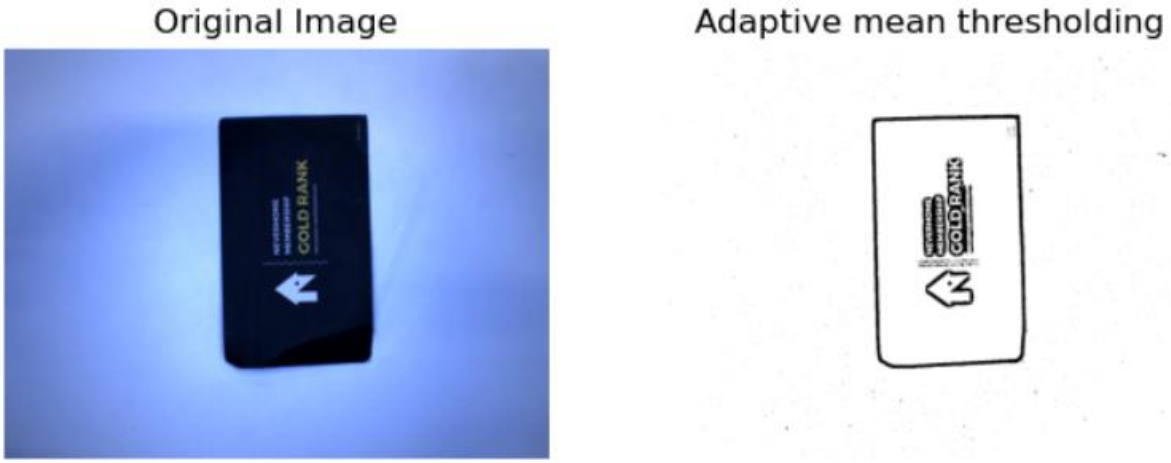
Picture 1



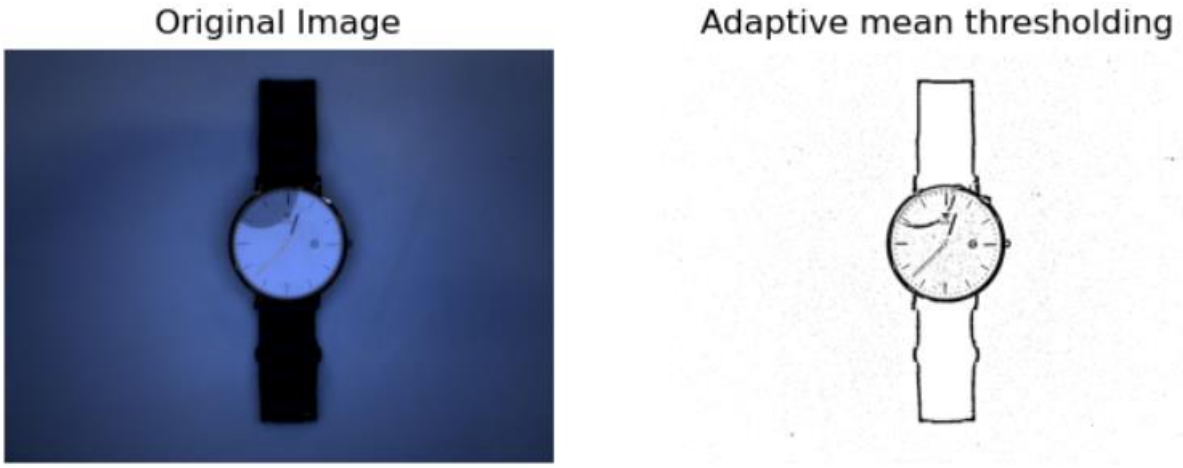
Picture 2



Picture 3



Picture 4

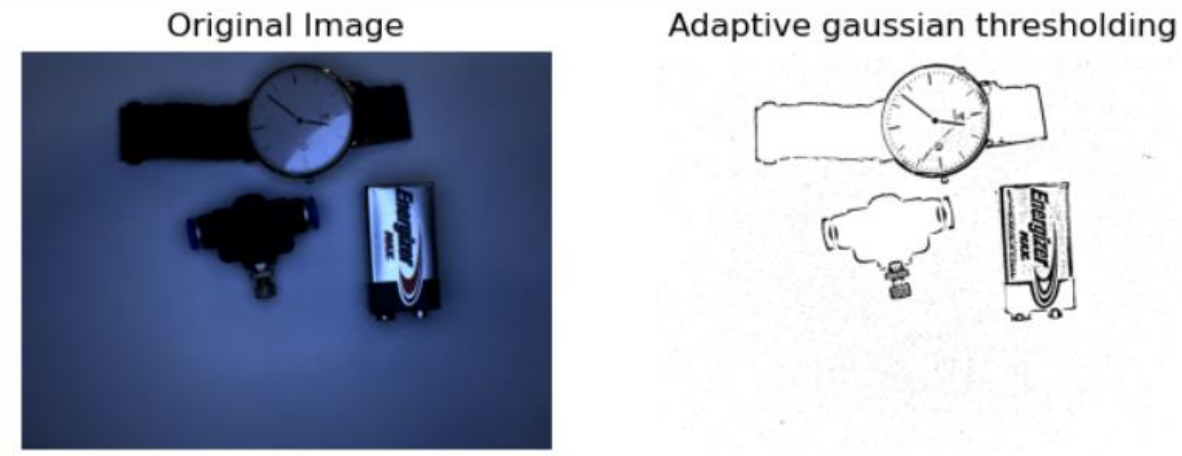


d2) Adaptive Gaussian Thresholding

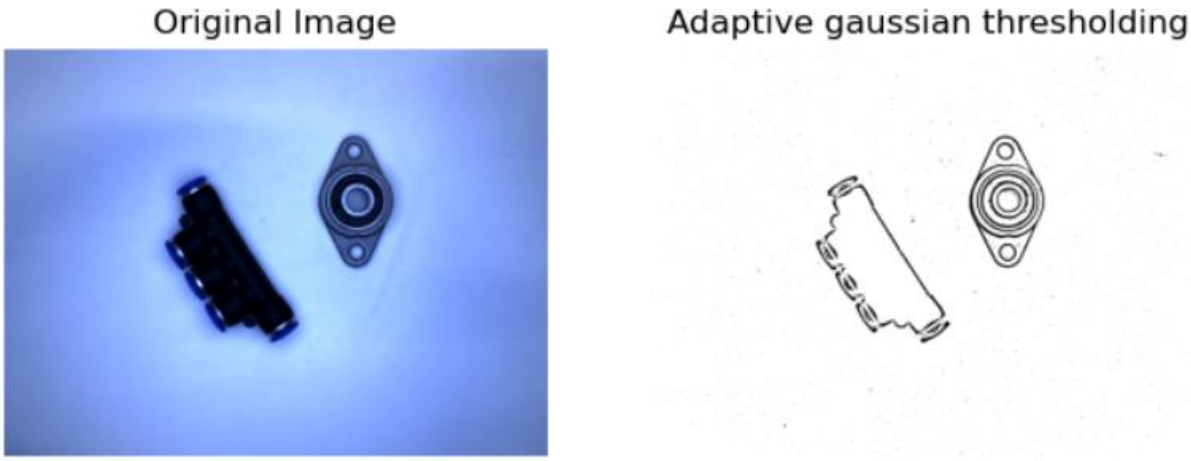
Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic1.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Apply Adaptive Gaussian Thresholding
adaptive_gaussian = cv2.adaptiveThreshold(gray_image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                          cv2.THRESH_BINARY, 25, 7)

# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Adaptive gaussian thresholding')
plt.imshow(adaptive_gaussian,cmap ='gray')
plt.axis('off')

plt.show()
```

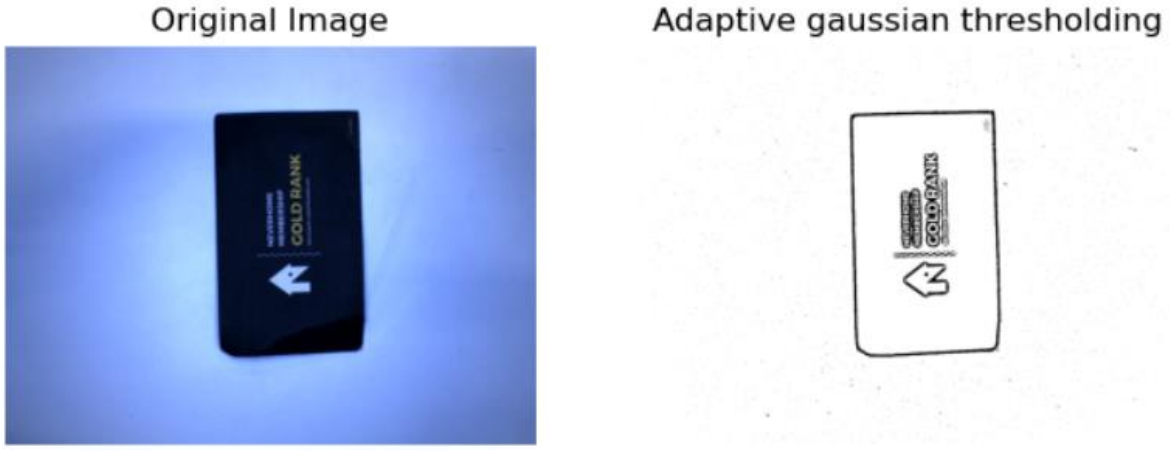
Picture 1



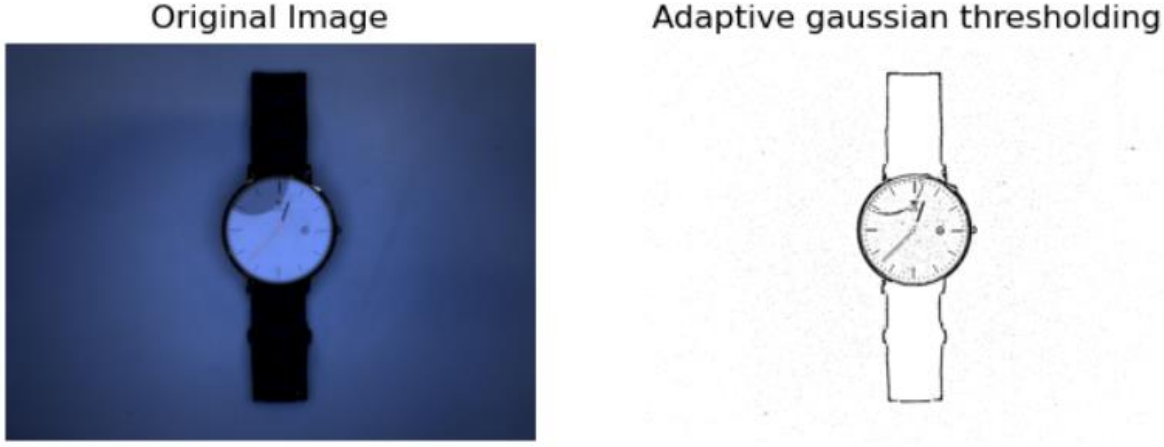
Picture 2



Picture 3



Picture 4



d3) Niblack’s method

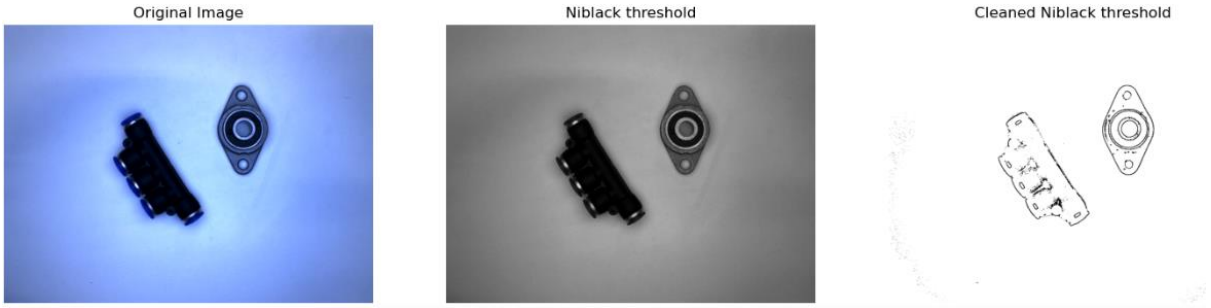
Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.filters import threshold_niblack
from scipy import ndimage

# Open the image
img = cv2.imread('pic1.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Apply Niblack thresholding
window_size = 3
thresh_niblack = threshold_niblack(gray_image, window_size=window_size, k=-2)

# Threshold the magnitude image to get binary mask
_, mask = cv2.threshold(thresh_niblack, 140, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Dilate the mask to ensure the entire object is covered
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)

# Invert the mask to get the background
mask_inv = cv2.bitwise_not(mask)

# Create a white background image
white_background = np.ones_like(gray_image) * 255

# Use the mask to extract the object from the original image
object_only = cv2.bitwise_and(gray_image, gray_image, mask=mask)

# Use the inverse mask to extract the background from the white background image
background_only = cv2.bitwise_and(white_background, white_background, mask=mask_inv)

# Combine the object and the white background
final_image = cv2.add(object_only, background_only)

final_image = ~final_image
```

```
# Plot the results
plt.figure(figsize=(18, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img)
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Niblack threshold')
plt.imshow(thresh_niblack, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Cleaned Niblack threshold')
plt.imshow(final_image<gray_image, cmap='gray')
plt.axis('off')


plt.show()
```

Picture 1

Picture 2



Picture 3

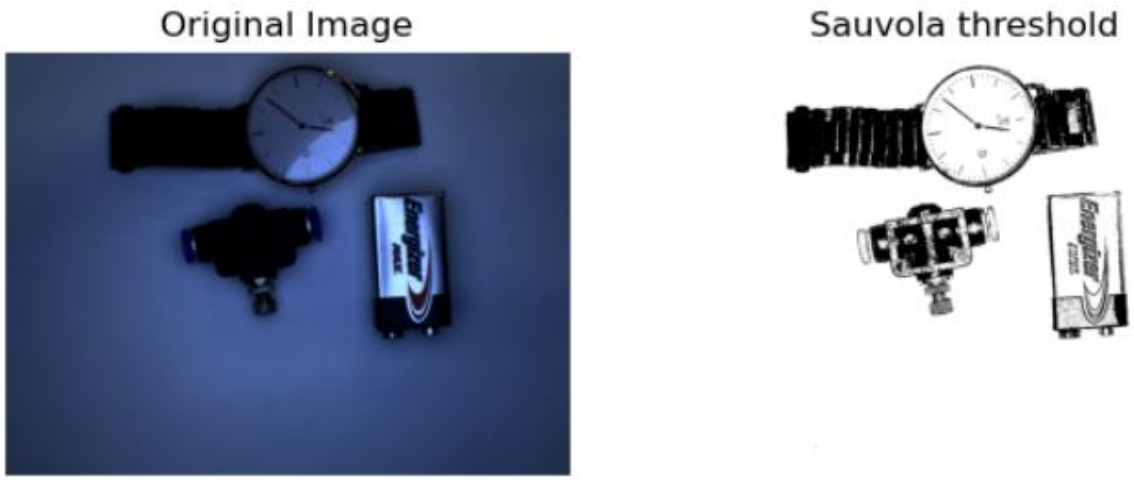
Picture 4

d4) Sauvola's method (Improvement of d3)

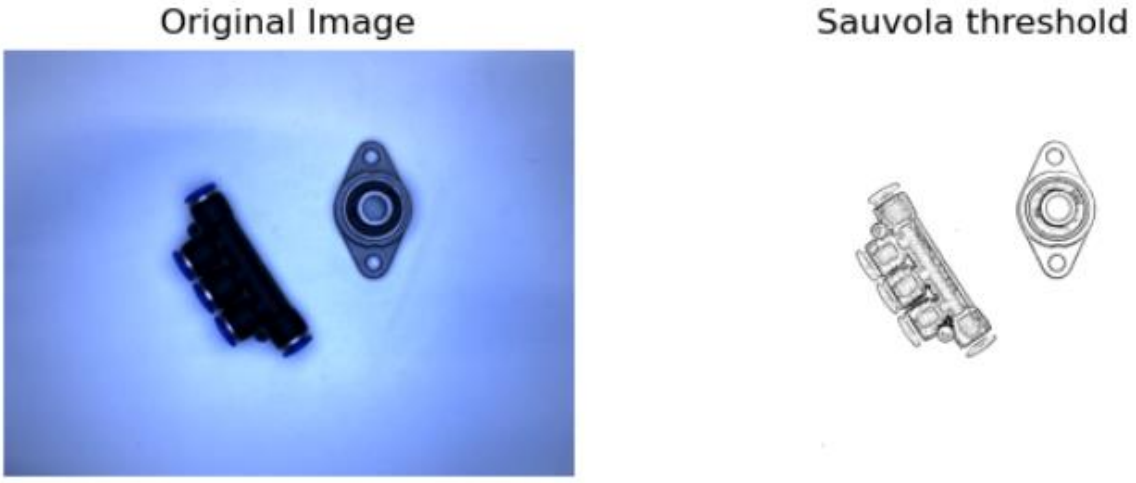
Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.filters import threshold_sauvola
from skimage import io, color

# Open the image
img = cv2.imread('pic1.png')
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Apply Sauvola's method using skimage
window_size = 5
thresh_sauvola = threshold_sauvola(gray_image, window_size=window_size)

binary_sauvola = gray_image > thresh_sauvola

# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Sauvola threshold')
plt.imshow(binary_sauvola,cmap ='gray')
plt.axis('off')

plt.show()
```

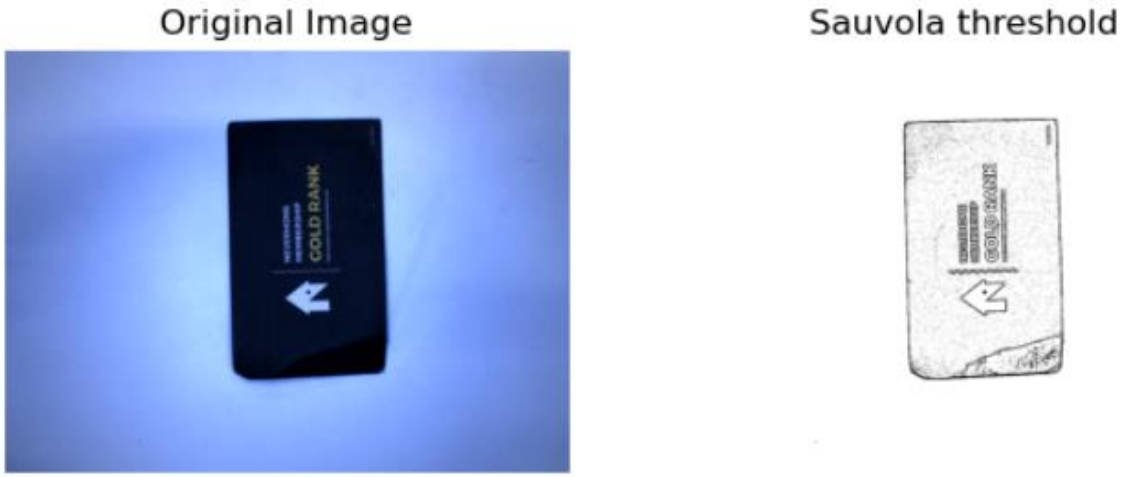
Picture 1



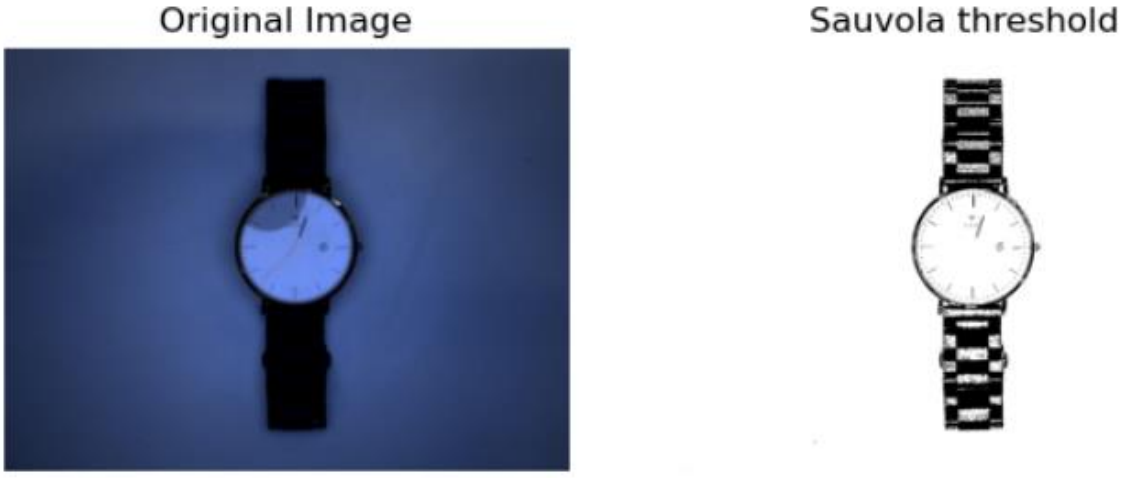
Picture 2



Picture 3



Picture 4



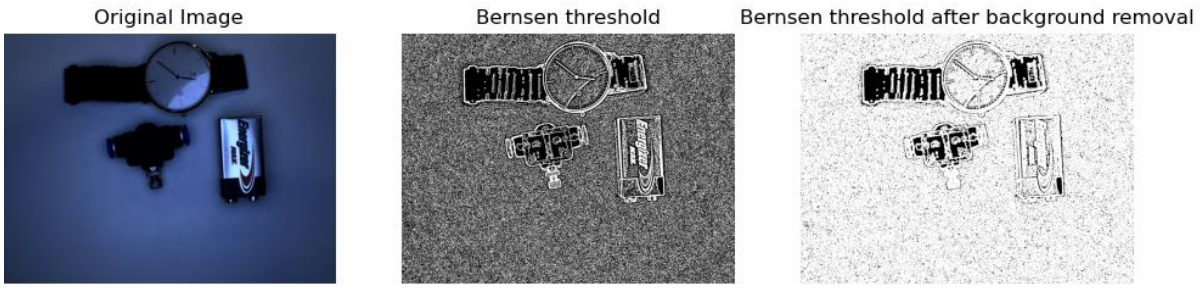
d5) Bernsen’s method

Code

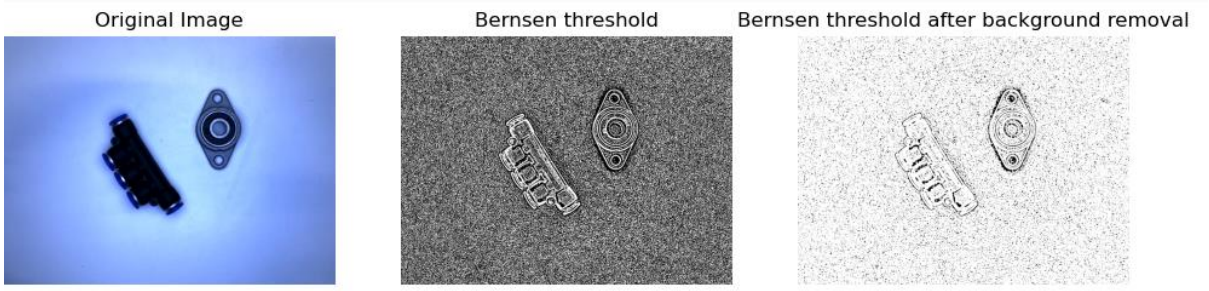
```
import mahotas
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Open the image
img = cv2.imread('pic2.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = np.array(img).astype(np.uint8)

# Convert to grayscale
gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Apply Gaussian smoothing (optional)
blurred_image = cv2.GaussianBlur(gray_image, (3, 3), 0)

# Apply Bernsen's method using Mahotas
window_size = 3
contrast_threshold = 1000

bernsen_result = mahotas.thresholding.bernsen(blurred_image, window_size, contrast_threshold)
bernsen_result = bernsen_result.astype(np.uint8)

# Threshold the magnitude image to get binary mask
_, mask = cv2.threshold(bernsen_result, 0, 255, cv2.THRESH_BINARY)

# Ensure mask is uint8
mask = mask.astype(np.uint8)

# Dilate the mask to ensure the entire object is covered
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)

# Invert the mask to get the background
mask_inv = cv2.bitwise_not(mask)

# Create a white background image
white_background = np.ones_like(gray_image) * 255

# Use the mask to extract the object from the original image
object_only = cv2.bitwise_and(gray_image, gray_image, mask=mask)

# Use the inverse mask to extract the background from the white background image
background_only = cv2.bitwise_and(white_background, white_background, mask=mask_inv)

# Combine the object and the white background
final_image = cv2.add(object_only, background_only)
final_image = ~final_image
```

```
# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(img)
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Bernsen threshold')
plt.imshow(bernsen_result, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Bernsen threshold after background removal')
plt.imshow(bernsen_result<final_image, cmap='gray')
plt.axis('off')

plt.show()
```

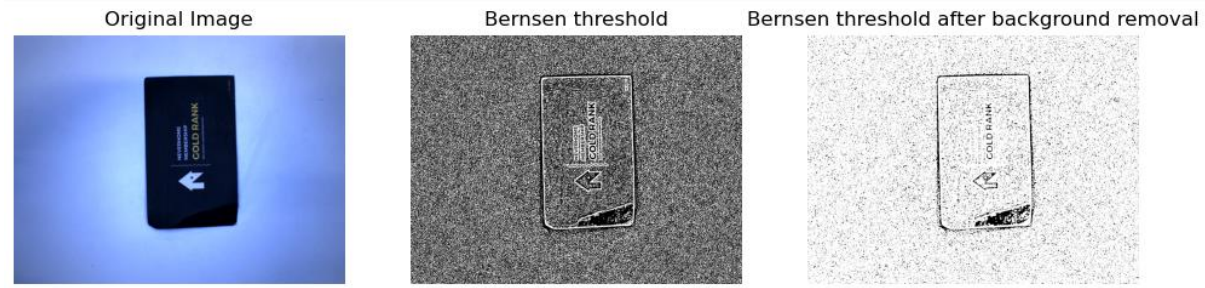
Picture 1



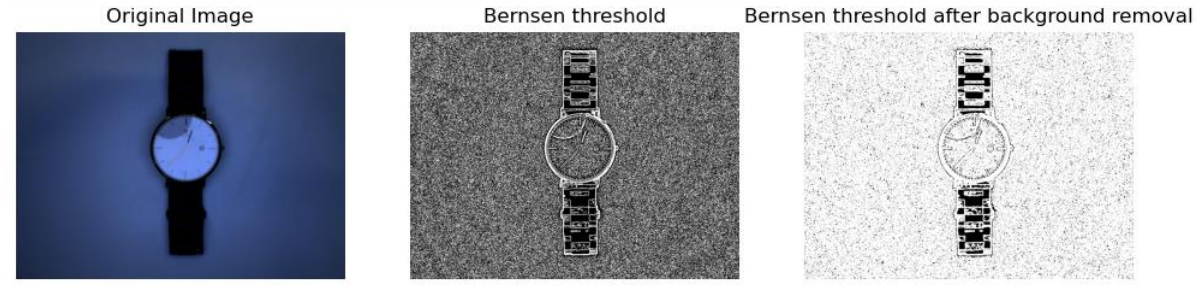
Picture 2



Picture 3



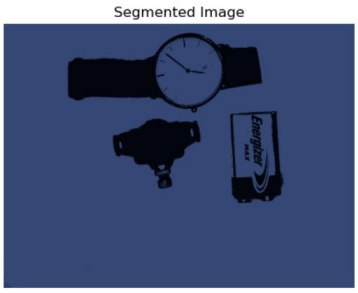
Picture 4



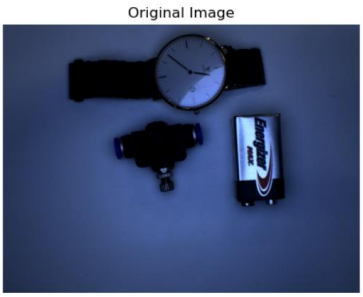
3.2e) K-mean clustering segmentation method

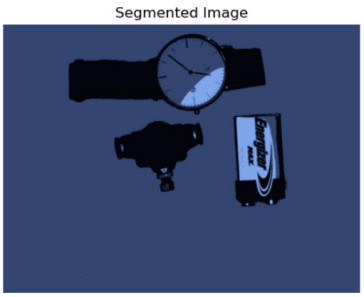
a) Picture 1

$k = 2$



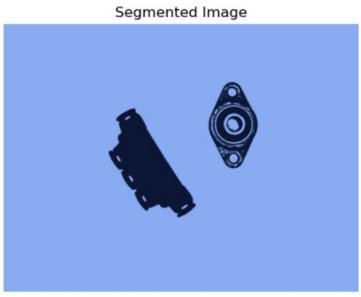
$k = 3$



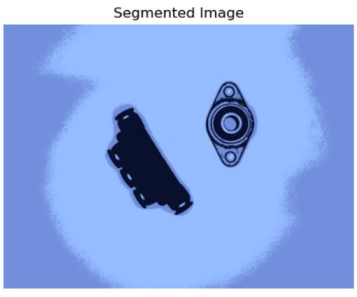


b) Picture 2

$k = 2$

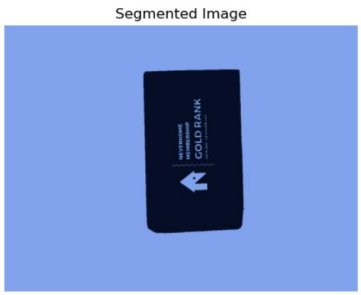


$k = 3$

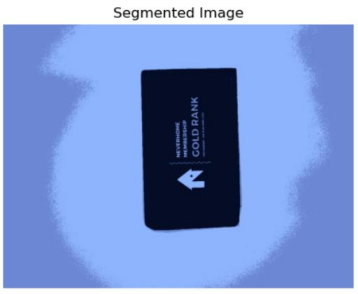


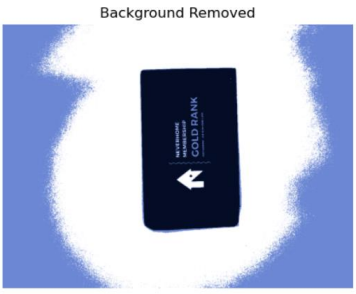
c) Picture 3

$k = 2$



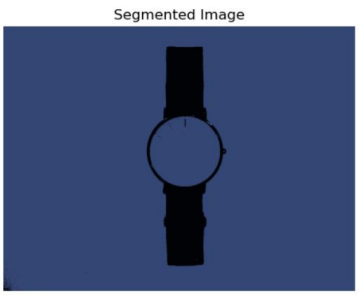
$k = 3$



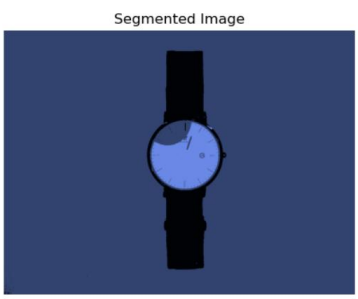


d) Picture 4

$k = 2$



$k = 3$



4.0 Discussion

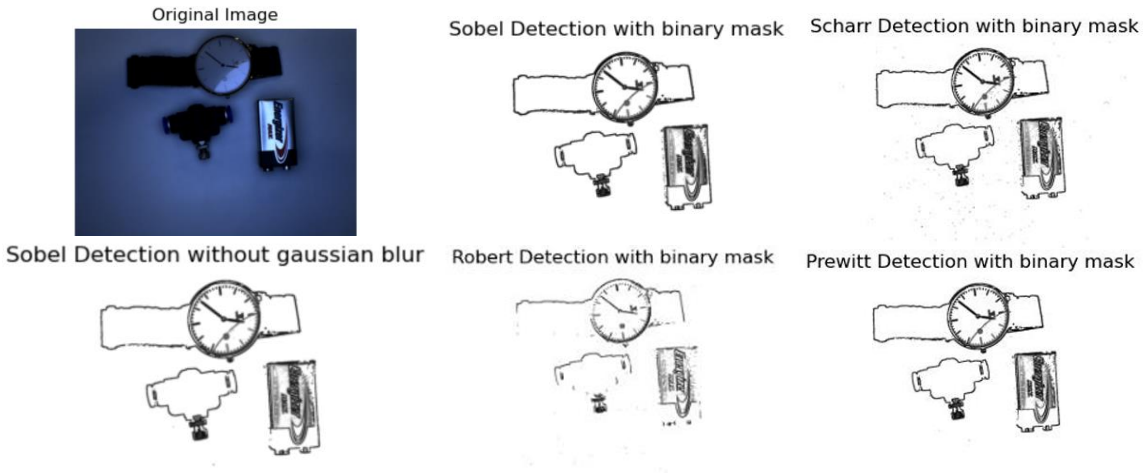


Figure 4.1: comparison the different type of gradient-based methods on Picture 1.

Among the four gradient-based methods, Sobel's method produces moderate results, clearly depicting and detecting more edges of the object with Gaussian blur. The combination effect effectively reduces noise in the background, although it may lose some finer details of the object. Prewitt and Scharr's method yield a similar effect on the output, offering a similar equilibrium between detecting edges and noise reduction. Additionally, Robert's method performs the worst, as it struggles to delineate the object's edge due to its simplistic and small kernel dimensions.

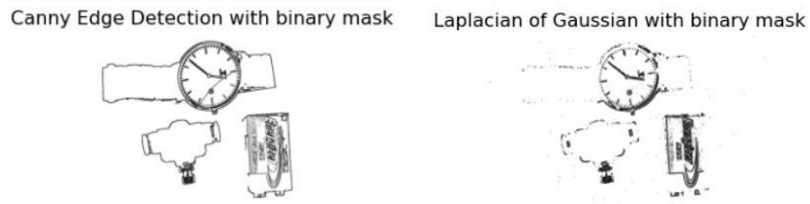


Figure 4.2: comparison the different type of gaussian-based methods on Picture 1.

Based on the observations, Canny edge detection tends to provide a more precise and defined edge of the object due to its multi-stage processes, such as non-maximum suppression, double thresholding and hysteresis, making it more robust on-edge detection compared to the Laplacian of Gaussian (LoG). The second derivative of the image intensity

approach nature in LoG might cause it to be more vulnerable to noise and less detect the defined edge.

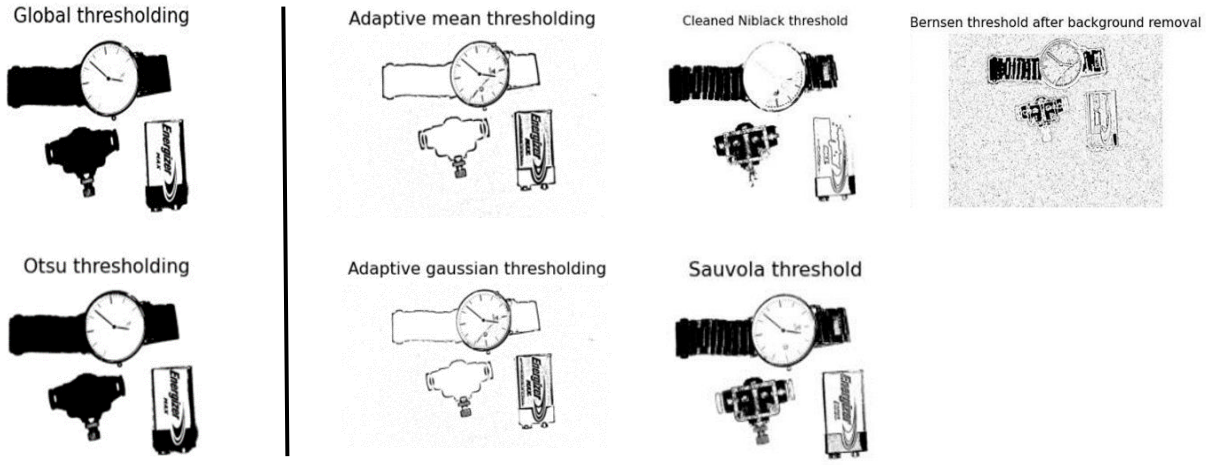


Figure 4.3: comparison the global and local thresholding methods on Picture 1.

One of the main differences between the global and local thresholds is that the global thresholding method has only one global threshold value for the entire image instead of calculating the threshold locally for each pixel based on the neighbourhood pixels. This will cause the global thresholding method to be unable to recognize the background when the foreground and background have similar intensity values. Some of the details on the watch screen were lost in the global threshold method due to varying light intensity, which split it into two parts with different light conditions. In addition, the only difference between the Otsu and global thresholding methods is that they calculate the optimal threshold automatically by minimizing the intra-class variance of the two classes of pixels or maximizing the inter-class variance but with the correct threshold setting in the general global thresholding method will generate similar effect as Otsu thresholding.

Furthermore, the adaptive mean thresholding method may present abrupt changes in light and dark areas, which can produce sharper edges, but it can be more sensitive to noise due to the direct use of the mean to calculate the threshold. The adaptive Gaussian thresholding method will have a smoother transition and lesser noise. Hence, it can produce a natural look representation of the image. The Sauvola method has an improvement effect over the Niblack method; its local threshold calculation includes the mean and standard deviation of the pixel intensities within the local neighbourhood and introduces dynamic

parameters to account for variables. Lastly, the Bernsen method obtains the mid-range of the maximum and minimum grey values in a local window, which may be effective for moderate lighting conditions and is not favourable for shallow contrast regions.

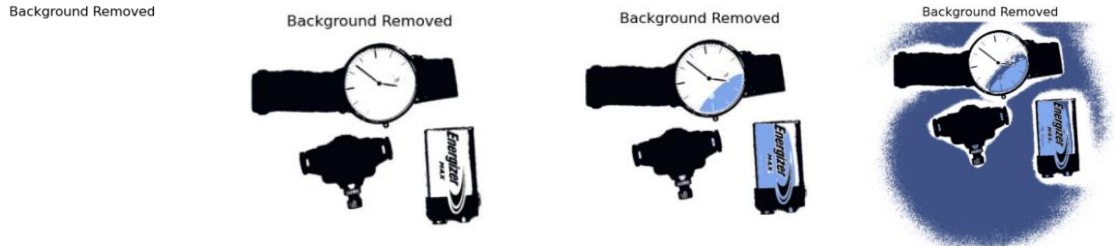


Figure 4.4: comparison of k = 1, k = 2, k = 3 and k = 4 for k-mean clustering method.

In the k-mean clustering method, when k = 1, it treats the entire image as a single cluster; it can only recognize one colour: the background colour. There is no distinction between foreground and background, making it unsuitable for object detection. In the k = 2 case, the object and the background colour can be identified as two clusters, and this is the optimal scenario for a clear distinction between the object and background. Progressive k values create additional cluster groups, further segmenting parts in the object and variation in the background, leading to over-segmentation, which may have some unwanted effect on background removal operation.

5.0 Conclusion

In conclusion, the experiment was conducted successfully and achieved the objectives outlined in the lab sheet. Various methodologies were applied in this lab, such as the gradient-based method (Sobel, Scharr), gaussian-based method (Canny, LoG), global (Otsu) and local thresholding (Niblack, Sauvola), and clustering method (k-mean). These techniques demonstrated clear and effective object detection and segmentation from the background.

As a result, the comprehensive coverage of image segmentation methods allows for a meaningful comparison of their output. This facilitates a discussion of their characteristics and differences, showing how they use different approaches and calculations to differentiate the object's pixels to separate it from the background.

References

1) GammaCode. (n.d.). *Comparing Edge Detection Methods*. [online] Available at: https://nikatsanka.github.io/comparing-edge-detection-methods.html. [Accessed at 3/8/2024]

2) Roboflow Blog. (2024). *What is Thresholding in Image Processing? A Guide.* [online] Available at: https://blog.roboflow.com/image-thresholding/. [Accessed at 4/8/2024]

3) GeeksforGeeks. (2019). *Python | Thresholding techniques using OpenCV | Set-3 (Otsu Thresholding)*. [online] Available at: https://www.geeksforgeeks.org/python-thresholding-techniques-using-opencv-set-3-otsu-thresholding/. [Accessed at 5/8/2024]

4) Reshma (2023). *Gradients in Image Processing*. [online] Scaler Topics. Available at: https://www.scaler.com/topics/gradients-in-image-processing/. [Accessed at 5/8/2024]

5) Roboflow Blog. (2024). *Edge Detection in Image Processing: An Introduction*. [online] Available at: https://blog.roboflow.com/edge-detection/. [Accessed at 5/8/2024]

6) Krishna, R. (2023). *Image Thresholding From Scratch*. [online] Geek Culture. Available at: https://medium.com/geekculture/image-thresholding-from-scratch-a66ae0fb6f09.

[Accessed at 5/8/2024]