Martina Carretta (1673930), Meritxell Carvajal (1671647)

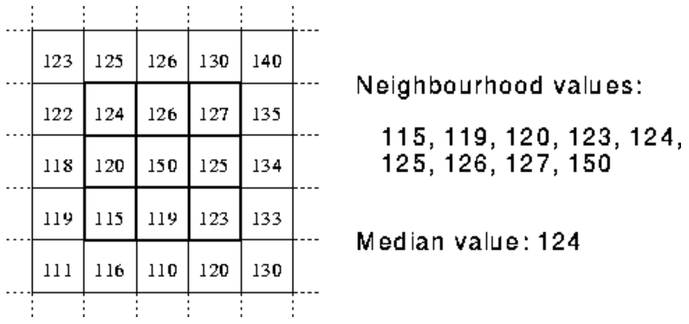
REPORT OF THE LAB 2

1. THEORETICAL UNDERSTANDING OF THE ALGORITHMS USED

SECTION 1: Median filters

The median filter is normally used to reduce noise in an image, somewhat like the mean filter. However, it often does a better job than the mean filter of preserving useful detail in the image.

Like the mean filter, the median filter considers each pixel in the image in turn and looks at its nearby neighbours to decide whether or not it is representative of its surroundings. Instead of simply replacing the pixel value with the mean of neighbouring pixel values, it replaces it with the median of those values. The median is calculated by first sorting all the pixel values from the surrounding neighbourhood into numerical order and then replacing the pixel being considered with the middle pixel value. (If the neighbourhood under consideration contains an even number of pixels, the average of the two middle pixel values is used.)



The choice between these two filters depends on the type of noise present in the image:

- Median filters are particularly effective at removing salt-and-pepper noise (random occurrences of black and white pixels). They preserve edges well because they pick the median value, which is less sensitive to extreme values (outliers) than the mean.
- Mean filters are good at reducing Gaussian noise (statistical noise having a probability density function equal to that of the normal distribution), but they can blur edges because they weigh all pixels in the kernel equally.

So, if you're dealing with salt-and-pepper noise or want to preserve edges, it's preferable to use a median filter. If you're dealing with Gaussian noise and edge preservation is not a priority, a mean filter might be more suitable.

A median filter is typically not implemented as a convolution operation because it involves finding the median value, which is not a linear operation. Convolution involves linear operations like addition and multiplication, making it suitable for filters like the mean filter, but not for the median filter.

In the spatial domain, a median filter operates by sliding a window (or kernel) over each pixel in the input image. The size of the window is typically an odd-sized square, e.g., 3x3, 5x5, etc. For each window position, the pixel intensities within the window are collected and sorted. The central pixel, as mentioned before, will be then assigned the median value from this sorted list of intensities. This process is repeated for every pixel in the image, resulting in a new image with the filtered values.

The window size of the median filter, often referred to as the kernel size, significantly affects the output image:

1. **Noise Reduction**: A larger window will be more effective at removing noise, especially 'salt-and-pepper' noise, because it considers a larger neighbourhood around each pixel. This makes it more likely that the median value will be a 'good' pixel value rather than a noisy one.

   However, besides noise removal, the median filter is advantageous in applications where preserving edge information is crucial. It can be used in image denoising, medical image processing, and edge-preserving smoothing.

2. **Edge Preservation**: Median filters are known for preserving edges while reducing noise because the median is less sensitive to extreme values than the mean. However, if the window size is too large, it may start to blur edges, especially if the edge only spans a few pixels.

3. **Detail Preservation**: A smaller window will preserve more of the fine details in an image. A larger window might start to lose these details because it's replacing each pixel with the median of a larger neighborhood.

4. **Computation Time**: A larger window size will increase computation time because for each pixel, more values need to be sorted to find the median.

   The computational complexity of median filtering is generally higher than that of Gaussian filtering. Median filtering requires sorting the pixel values within the window, which is an O(nlogn) operation for each window, where 'n' is the number of pixels in the window. Gaussian filtering, on the other hand, involves convolution, which is typically faster to compute using techniques like Fourier transforms. The complexity of Gaussian filtering depends on the size of the kernel but is often O(N^2) for a window of size N.

So, choosing the right window size is a trade-off between noise reduction, edge preservation, detail preservation, and computation time. The choice of window size depends on the specific application and the desired trade-off between noise reduction and detail preservation. A larger window captures a broader area of the image, which can lead to more smoothing and a reduction in fine details. While a smaller window preserves finer details but may be less effective in removing noise.

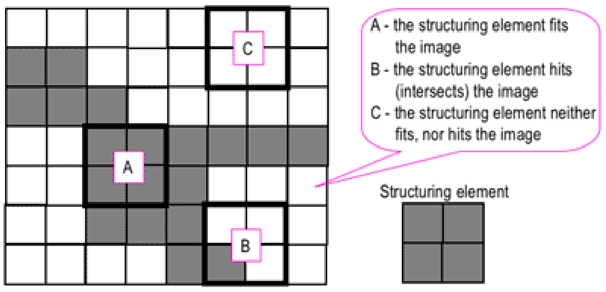
SECTION 2: Morphological operation

Image filters and thresholds enable us to detect structures of various shapes and sizes for different applications. Nevertheless, despite our best efforts, the binary images produced by our thresholds often still contain inaccurate or undesirable detected regions. They could benefit from some extra cleaning up.

At this stage, we are primarily working with shapes – morphology – so most of the techniques described are often called morphological operations.

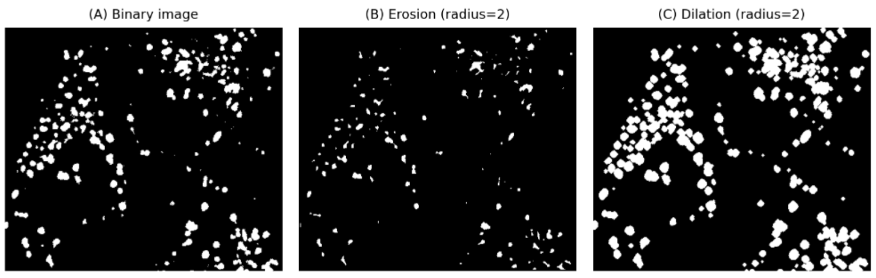
Morphological image processing is a collection of non-linear operations related to the shape or morphology of features in an image. According to Wikipedia, morphological operations rely only on the relative ordering of pixel values, not on their numerical values, and therefore are especially suited to the processing of binary images. Morphological operations can also be applied to greyscale images such that their light transfer functions are unknown and therefore their absolute pixel values are of no or minor interest.

Morphological techniques probe an image with a small shape or template called a structuring element. The structuring element is positioned at all possible locations in the image and it is compared with the corresponding neighbourhood of pixels. Some operations test whether the element "fits" within the neighbourhood, while others test whether it "hits" or intersects the neighbourhood:


C
A
B
A - the structuring element fits the image
B - the structuring element hits (intersects) the image
C - the structuring element neither fits, nor hits the image
Structuring element

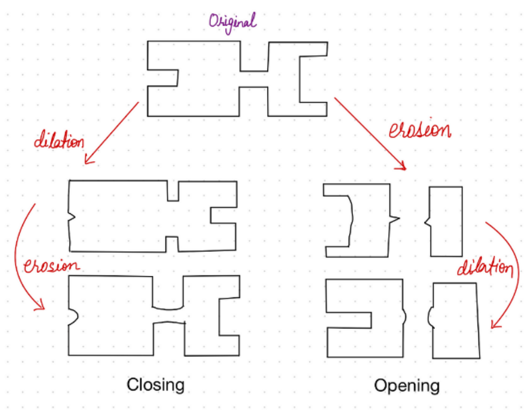
The basic morphological operations include:

- **Dilation:** Dilation is used to expand the boundaries of objects in an image. It is performed by convolving the image with a structuring element, where for each pixel, the maximum pixel value within the structuring element's neighborhood is taken as the new pixel value. Dilation can fill gaps, connect nearby objects, and make objects more prominent.
- **Erosion:** Erosion is used to shrink the boundaries of objects in an image; so, it will make objects in the binary image smaller. It is also performed by convolving the image with a structuring element, but this time the minimum pixel value within the structuring element's neighborhood is taken as the new pixel value. Erosion can remove small noise, detach connected objects, and make objects thinner.



Some common derived morphological operations include:

- **Opening:** An opening operation is an erosion followed by a dilation. Morphological opening is useful for removing small objects and thin lines from an image while preserving the shape and size of larger objects in the image.

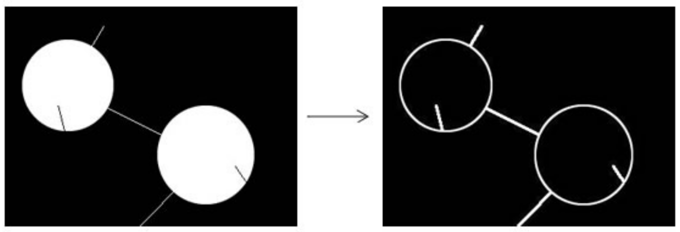


- **Closing:** A closing operation is the opposite of opening, i.e. a dilation followed by an erosion, and similarly changes the shapes of objects. The dilation can cause almost-connected objects to merge, and these often then remain merged after the erosion step. If you wish to count objects, but they are wrongly subdivided in the segmentation, closing may help make the counts more accurate. It can be used to close small gaps or small holes in objects.

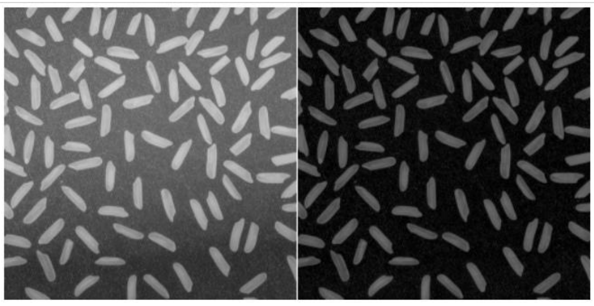
- **Skeletonization:** The process of skeletonization erodes all objects to centrelines without changing the essential structure of the objects, such as the existence of holes and branches. It is particularly useful with filamental or tube-like structures, such as axons or blood vessels.

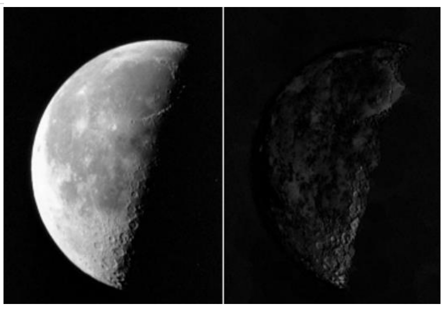
- **Morphological Gradient:** Morphological gradient is the difference between the dilation and the erosion of a given image. It is an image where each pixel value (typically non-negative) indicates the contrast intensity in the close neighbourhood of that pixel. It is useful for edge detection and segmentation applications.

  Morphological operations can be applied to multi-channel images, but they present challenges. When working with multi-channel images, you need to define structuring elements in each channel, and the operations may behave differently in each channel. Handling colour spaces, like RGB or HSV, can be complex because the operations must be applied to each channel independently. Moreover, ensuring consistency across channels and avoiding artifacts can be tricky.


Morphological Gradient

- **Top Hat:** The top-hat operation computes the difference between the original image and its opening. The top-hat transform can be used to enhance contrast in a grayscale image with nonuniform illumination. The transform can also isolate small bright objects in an image.
- **Bottom Hat:** The bottom-hat transform closes an image, then subtracts the original image from the closed image. The bottom-hat transform isolates pixels that are darker than other pixels in their neighbourhood. Therefore, the transform can be used to find intensity troughs in a grayscale image.


Top-hat


Bottom-hat

Another general binary morphological operation that can be used to look for particular patterns of foreground and background pixels in an image is the “hit-ot-miss” transform. It is actually the basic operation of binary morphology since almost all the other binary morphological operators can be derived from it.

The "hit-or-miss" transform is a morphological operation used for pattern matching and defect detection. It involves defining two structuring elements: a "hit" structuring element that represents the pattern to be matched and a "miss" structuring element that represents the background. The operation identifies locations where the "hit" structuring element matches the image, and the "miss" structuring element doesn't.

Mathematically, the hit-or-miss transform of an image A by a composite structuring element B = (C, D) is given by:

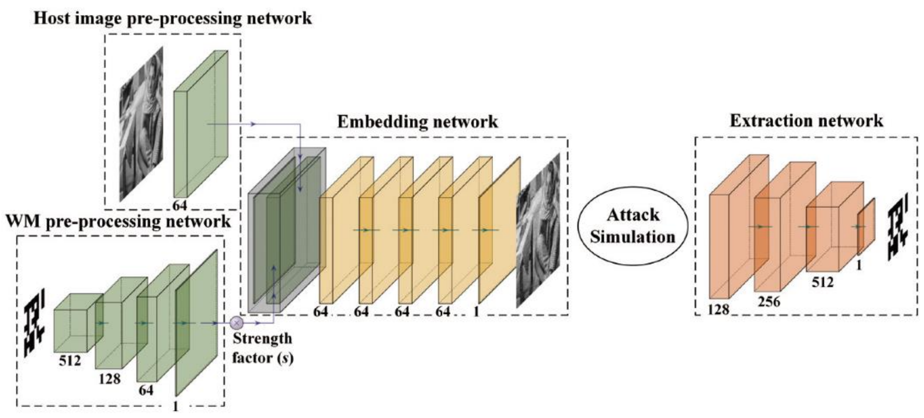
$$A \circ B = (A \ominus C) \cap (Ac \ominus D)$$

where $\ominus$ denotes the erosion, $\cap$ denotes the intersection, and Ac denotes the complement of A. The hit-or-miss transform can be useful in defect detection, as it can identify the locations where a certain pattern (characterized by B) occurs or deviates from the input image. For example, one can use the hit-or-miss transform to detect holes, cracks, or missing parts in an image of a manufactured product5.

SECTION 3: Watermarking in Image Processing

Watermarking is the process that embeds data called a watermark or digital signature or tag or label into a multimedia object such that watermark can be detected or extracted later to make an assertion about the object. The object may be an image or audio or video. In general, any watermarking scheme (algorithm) consists of three parts:

1. The watermark
2. The encoder (marking insertion algorithm)
3. The decoder and comparator (verification or extraction or detection algorithm)



Hence, this technique is used to protect digital content from unauthorized use and distribution. In image processing, watermarking is used to embed a digital signature or a logo into an image to indicate ownership or authenticity. It can also be used to track the usage of an image and prevent unauthorized copying or distribution. Digital watermarking is seen as a partial solution to the problem of securing copyright ownership.

There are different types of watermarks, but we will focus on visible and invisible ones.

A **visible watermark** is the one that overlays the photo and can take many forms, according to your individual needs and preferences. It can be a brand logo, a copyright text, a text with the company name or website, your personal signature and more.

The advantage of the visible watermark is that, being right there on the picture, it states without any ambiguity that the picture represents your intellectual property and it can be used by others only with your authorization. People can’t simply say they weren’t aware of the ownership of that photo, since the text or logo is right there.

On the other side, as you may imagine, an **invisible watermark** can not be seen or perceived by the human eye. This watermark can be regarded as a type of steganography and consists in embedding a series of personal information into the image data itself. The information can be anything you want, from your middle name to your personal email or company website.

Besides being unremovable, the main advantage of the invisible watermark is that it can detect if and where your original photos have been used without your permission. With the aid of specially developed software, you have the possibility to run a web search and discover the websites that are illegally using your visual content. You can rest assured you'll be able to identify the thief, easily prove ownership and take the required action.

Therefore, the main difference between visible and invisible watermarks is that visible watermarks can be clearly seen on the image, while invisible watermarks are hidden in the image and can only be revealed by special tools or methods. Visible watermarks are used to deter copying and promote branding, while invisible watermarks are used to prove ownership and detect tampering.

Dual watermark is the combination of visible and invisible watermark. An invisible watermark is used as a backup for the visible watermark. According to Working Domain, the watermarking techniques can be divided into two types:

a. Spatial Domain Watermarking Techniques
b. Frequency Domain Watermarking Techniques

In spatial domain techniques, the watermark embedding is done on image pixels while in frequency domain after marking techniques the embedding is done after taking image transforms.

Fourier Transform can be utilized for watermark embedding in the frequency domain. The process involves transforming the image and watermark into the frequency domain using techniques like the Discrete Fourier Transform (DFT) or the Fast Fourier Transform (FFT). Once the image and watermark are in the frequency domain, you can manipulate the frequency coefficients to embed the watermark. Thus, the energy of the watermark is distributed over the entire image after the transformation back to the spatial domain, which enables the implementation of stronger watermarks with less perceptual impact.

This approach can make the watermark less noticeable and more robust against common image processing operations like resizing or compression. However, it requires specialized algorithms to perform the embedding and extraction in the frequency domain.

There are several advanced techniques and methods that can make watermark extraction more efficient and accurate:

- Invisibility: The best way to evaluate invisibility is to conduct subject tests where both original and watermarked signals are presented to human subjects. However, due to the high volume of test images, subject tests are usually impractical. The most common evaluation method is to compute the peak signal-to-noise ratio (PSNR) between the host and watermarked signals.

- Effectiveness: Digital watermarking systems have a dependence on the input signal. Effectiveness refers to whether it is possible to detect a watermark immediately following the embedding process.

- Efficiency: Efficiency refers to the embedding capacity. For images, it is usually expressed in bits of information per pixel (bpp). A 512 x 512 image with 16 KB of embedded data has an embedding capacity of 0.5 bpp.

- Robust Watermarking Algorithms: These algorithms are designed to withstand various image processing operations, compression, and attacks, ensuring that the watermark remains intact and can be reliably extracted.

We will focus on Robust Watermarking Techniques: Robustness is one of the most commonly tested properties in digital watermarking systems. In many applications, it is unavoidable that the watermarked signal would be distorted before it reaches the detector. Robustness refers to the ability for the detector to detect the watermark after signal distortion, such as format conversion, introduction of transmission channel noise and distortion due to channel gains.

Robust watermarking techniques are designed to provide resilience against common image processing operations and attacks. Some notable methods include:

→ Spread Spectrum Watermarking: This technique spreads the watermark signal across the image's frequency components, making it robust against noise and compression.

→ DCT-based Watermarking: Using the Discrete Cosine Transform (DCT) for embedding watermarks can make the watermark more resilient to lossy compression.

→ Quantization Index Modulation (QIM): QIM methods embed watermarks by subtly altering quantization levels in the image, making the watermark more resistant to attacks.

→ Fragile Watermarking: Fragile watermarking is used for tamper detection and localization. It can detect even minor alterations to the image.

→ Key-Based Watermarking: Encryption and keys can be used to enhance the security of the watermark, ensuring only authorized parties can extract it.

The choice of technique depends on the specific requirements of the application.

2. PRACTICUM: RESULTS AND DISCUSSION

BLOCK 1: IMPORTING LIBRARIES

Imports: The code starts with importing necessary libraries, such as **cv2** for computer vision, **urllib3** for handling URLs (although it's not used in this code), **numpy** for numerical operations, **PIL** for image processing, **imutils** for various image processing utilities, and **matplotlib** for plotting images.

BLOCK 2: NON-LINEAR FILTERING (MEDIAN FILTER)

- **Median Filter Implementation:** def median_blur(image, kernel_size): This function implements a median filter operation. It takes an image and a kernel size as input, performs median filtering manually without using library implementations, and returns the filtered image.
- **Applying Median Filter:**
  img = cv2.imread("C:/Users/Momo/OneDrive/Escriptori/Computer vision/Lab 2/images/lenna.png")
  gray_image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  output = median_blur(gray_image, 3) output2 = median_blur(gray_image, 9)

    - Here, an image of Lenna is loaded, converted to grayscale (**gray_image**), and then the **median_blur** function is applied with different kernel sizes (3 and 9). The results are stored in **output** and **output2**.

- **Displaying Results:** fig, axes = plt.subplots(1, 3, figsize=(10, 6): This code section displays the original image, the result of median blur with a kernel size of 3, and the result with a kernel size of 9.
- **Adding Noise and Applying Median Blur:** Noise is added to the original image using the **add_noise** function, and then median blur is applied with different kernel sizes to the noisy images.
- **Comparing with OpenCV Implementation:** This section compares the hand-implemented median blur (**output**) with OpenCV's implementation (**blur**). An assertion checks if they are close; otherwise, an error message is displayed.

BLOCK 3: MORPHOLOGICAL OPERATORS IMPLEMENTATION

- **Application on Images:** We load a binary image, applies various morphological operations, and visualizes the results using Matplotlib.
- **Comparison with OpenCV:** We use OpenCV functions for morphological operations (erode, dilate, morphologyEx) to compare the results with the hand-implemented functions, ensuring correctness.
- **Additional Example:** Applying morphological operations on a hat image and compares the results with OpenCV functions.

- **Skeletonization Example:** Skeletonization on a grayscale image and comparisiom with a pre-existing skeleton image.

Bloc 4: Defect detection with Morphological operations

- **Load Images:** We loadgrayscale images and their corresponding ground truth masks.
- **Visualize Images and Masks:** Display the loaded images and masks in a 2x4 grid.
- **Define a Function to Identify Defects:** The **identify_defects** function takes an image and a threshold as parameters. It applies binary thresholding to identify defects in the image and returns an inverted binary mask.
- **Identify Defects for Each Image:** Defect masks are generated for each image using the **identify_defects** function with specific threshold values.

```
defect_mask1 = identify_defects(i0, 80) defect_mask2 = identify_defects(i2, 60)
defect_mask3 = identify_defects(i4, 50) defect_mask4 = identify_defects(i6, 32)
```

- **Define Kernels for Morphological Operations:** Different-sized kernels are defined for morphological operations (opening) to process each defect mask.

```
kernel1 = np.full((21, 21), 1)

kernel2 = np.full((3, 3), 1)

kernel3 = np.full((1, 1), 1)

kernel4 = np.full((5, 5), 1)
```

- **Apply Morphological Operations (Opening):** The **opening** operation is applied to each defect mask using the defined kernels.

```
defect1 = opening(defect_mask1, kernel1)

defect2 = opening(defect_mask2, kernel2)

defect3 = opening(defect_mask3, kernel3)

defect4 = opening(defect_mask4, kernel4)
```

- **Calculate Intersection over Union (IoU):** The **calculate_iou** function computes the Intersection over Union (IoU) metric between the predicted defect masks and the ground truth masks.

```
for i in range(len(defects)): iou = calculate_iou(defects[i], images[i + 4]) # images[i+4]
because the masks start at index 4 print(iou)
```

BLOCK 5: WATERMARKING PROCESS:

- **Applying Watermark:** The **apply_watermark** function takes the path of the original image and a watermark text as parameters. It opens the original image, creates a new image with a white background, pastes the original image onto the new image, and adds the watermark text. The watermarked image is then saved.
- **Displaying Original and Watermarked Images:** The code loads the original image, applies the watermark, and displays both the original and watermarked images side by side.

  og = cv2.imread("C:/Users/Momo/OneDrive/Escriptori/Computer vision/Lab 2/images/lenna.png") # Add the watermark and retrieve the result (and convert to grayscale) apply_watermark("C:/Users/Momo/OneDrive/Escriptori/Computer vision/Lab 2/images/lenna.png", "Watermark")

  img = cv2.imread("C:/Users/Momo/OneDrive/Escriptori/Computer vision/Lab 2/output_image.jpg")

Extracting Watermark:

1. **Easy Extraction:**
   - The **remove_watermark_easy** function takes two images (original and watermarked) and subtracts the grayscale versions of the two to obtain the watermark.
   - A binary threshold is applied to the obtained watermark to get a binary mask.
2. **Hard Extraction:**
   - The **remove_watermark_hard** function takes the watermarked image and performs thresholding to obtain a binary mask representing the watermark.
   - The binary mask is then applied to create an image with the watermark.
3. **Displaying Results:**
   - The code displays the original image, watermarked image, and extracted watermark using both the easy and hard extraction methods.

     result = remove_watermark_easy(og, img)

     watermark = remove_watermark_hard(img)

3. DIFFICULTIES / EXPLANATIONS

→ After, we had added noise of the type salt and pepper to the same image in different amounts to test if the blur function could mitigate or at least reduce that noise. We worked with two kernels: one of size 3 and the other of size 5; and seen that the median blur mitigates the noise even if it's pretty high when the correct kernel size is applied.

→ After that, we tried removing the noise with the gaussian and the box filters and seen they don't work as well.

<u>Morphological operators:</u>

We have done all functions except the convex hull which we tried and it didn't work as it needed to. We have uploaded a binary image of numbers and tried all the operators.

After plotting them, it was also checked if the functions were correct comparing them with the cv2 functions. And the assert did not raise any error.

The properties of each operation have been proved with simple matrices and simple structuring elements in order to ease the process to make it faster.

<u>Defect detection:</u>

We have loaded and plotted the given images. Then, we took the original images and applied a threshold according to each image to leave a mask where everything is black except for the defects. After that, we opened each mask.

The results between my masks and the one provided have been compared with the iou and they don't have high IoUs because the masks have not been precisely computed as it is difficult to completely separate the crack or the holes from the rest.

<u>Watermarking:</u>

First we have added text to the image as a watermark; then to be left with the watermark from the original and the watermarked, we subtracted one from the other and then applied a threshold. In order to do the same but the hard way (only with the watermarked image), we applied a threshold to separate the values below 233 and the ones above as the watermark is white.