COMPUTER VISION
PROJECT-2

Meet oswal
mo2532@nyu.edu

INDEX

Convolution and Derivative Filters:

In computer vision, convolution operations play a crucial role in analyzing and transforming images effectively. These operations involve applying various filters, also known as kernels, to extract important details such as edges, textures, and patterns from images. This area of study delves into the core concepts of convolution and derivative filters, particularly focusing on their significance in edge detection. The primary aim is to develop a 2D convolution function from scratch, without relying on existing libraries, to deepen our understanding of how these operations impact image processing.

Through this process, we aim to showcase the versatility of convolution in modifying images for edge detection, a critical aspect in numerous computer vision applications. Prior to diving into implementation, it's crucial to grasp the fundamental principles of correlation and convolution, as they serve as foundational mathematical techniques essential for comprehending image processing and various computer vision tasks.

Introduction to Correlation:

- Correlation serves as a metric to gauge the similarity between two sets of data or signals. In image processing, this measure is pivotal for tasks like template matching and object detection, enabling the recognition of patterns or attributes within extensive images. This involves traversing a smaller image, known as a template, across a larger one and computing the correlation at each position to identify areas where the larger image closely aligns with the template. Such a procedure significantly aids in object detection and pattern recognition.

- Correlation is a means of assessing the likeness between two signals or images. In image processing, it holds significance particularly in template matching tasks. This method entails superimposing a smaller image (the template) onto a larger one to identify regions of resemblance. Its application extends to object detection within images, facilitating the identification of specific features or patterns.

Introduction to Convolution:

- Convolution represents a mathematical procedure that blends two functions to generate a third, illustrating how one function influences the shape of the other. In the context of image processing, convolution employs a filter (or kernel) that traverses the image. At each position, the convolution computes the sum of the element-wise product between the kernel and the section of the image it covers, thereby assigning a new value to the pixel situated at the center of the kernel. This operation plays a pivotal role in various image processing tasks, including blurring, sharpening, and notably, edge detection. Its linear and shift-invariant characteristics

render convolution highly suitable for these tasks, offering a robust framework for filtering and enhancing images.

- Convolution, as a mathematical operation, serves to apply a filter to an image. It entails flipping the filter both horizontally and vertically and then sliding it across the image, multiplying corresponding elements and summing them to generate a new image. Its linearity and shift-invariance make it well-suited for tasks such as filtering, edge detection, and blurring in the realm of image processing.

Difference between Correlation and Convolution:

- The key disparity between correlation and convolution lies in the initial **handling of the kernel or filter**. Despite being subtle, this contrast holds significant implications across a spectrum of fields including signal processing, image processing, and deep learning.

- In **correlation**, the kernel (or filter**) directly interacts with the signal** (or image) without any modification. This involves moving the kernel across the signal or image and calculating the sum of products at each position, all without changing the orientation of the kernel.

- On the contrary, **convolution requires an initial step of flipping the kernel** both horizontally and vertically. This flipping alters the kernel's orientation before its application. Through this process, the input signal merges with the kernel, blending them together, which is a crucial aspect of how convolution alters the signal or image compared to correlation.

- Despite their operational disparities, both correlation and convolution are indispensable in the field of computer vision. Convolution is primarily utilized for applying diverse filters for tasks such as **edge detection, blurring, and feature extraction** from images. Conversely, correlation finds utility in **template matching and object detection**, where the objective is to identify similar patterns or objects within images.

- Regarding **symmetry** in convolution, when dealing with symmetric kernels like those used for Gaussian blur, **convolution and correlation produce identical results**. This occurs because flipping a symmetric kernel does not change its orientation, resulting in the operation's effect being the same regardless of whether it's applied through convolution or correlation. This convergence of techniques, particularly within machine learning, can blur the distinction between them in certain scenarios.

- In essence, the technical divergence between correlation and convolution originates from the flipping requirement of the kernel in convolution. This difference not only influences their application scope but also affects their interpretative use across various disciplines. While correlation seeks to assess the similarity or match between shapes or signals, convolution intricately combines two signals, generating a new signal that illustrates how one is shaped or influenced by the other.

Separable Filters:

- A separable filter distinguishes itself by its ability to break down into the product of two simpler, one-dimensional filters. This property proves advantageous for executing advanced filtering operations on images with improved computational efficiency.
- Imagine you aim to apply a 2D filter (or kernel) to an image using convolution. This procedure involves sliding the filter over the image, performing element-wise multiplications between the filter and the corresponding image section it overlaps, and then aggregating these multiplications to produce a new image. While effective, this approach can be computationally demanding, particularly for large filters or high-resolution images.
- The benefit of a separable filter becomes evident by enabling the convolution to occur in two consecutive steps of one-dimensional filtering: first along one dimension (e.g., horizontally), and then along the other dimension (e.g., vertically). This approach significantly reduces the computational workload, offering a more efficient alternative for applying intricate filters to images.
- Directly applying a 2D filter to an image demands substantial computational resources. For each pixel in the image, it's necessary to conduct m x n multiplications, where m and n represent the dimensions of the filter, followed by summing these results. Executing this operation for every pixel in an N x M image results in a total computational complexity of O(M . N . m . n).
- However, the computational burden can be notably alleviated if the 2D filter is separable, implying it can be expressed as the product of two one-dimensional filters. One of these filters operates horizontally, while the other operates vertically. By employing this approach, you first apply the horizontal 1D filter to each row of the image, then apply the vertical 1D filter to each column (or vice versa).
- This technique significantly improves efficiency by reducing the number of required multiplications. Instead of performing m x n multiplications for each pixel, you only need to conduct m + n multiplications per pixel. As a result, the overall computational complexity is markedly reduced to O(M . N . (m + n)), rendering it a much more efficient method for applying complex filters to images.
- Example: (h(x) * g(x)) * f(x) => (h(x) * f(x)) * g(x) where h(x) and g(x) together make a 2D filter which is applied on f(x).

Advantage of Separable Filter – Computational Efficiency :

The enhancement in computational efficiency stems from the reduction in required operations:

- When applying the 1D horizontal filter to each pixel in a row, it necessitates m multiplications, covering all N×M pixels, resulting in a complexity of O(M·N·m).
- Similarly, applying the 1D vertical filter involves n multiplications for each pixel in a column, leading to a complexity of O(M·N·n).

As a result, the overall computational complexity is effectively reduced to O(M·N·(m+n)), representing a significant decrease compared to the original O(M·N·m·n) (particularly noticeable for larger images or filters).

Gaussian Filter:

- The project aims to utilize a Gaussian filter for denoising images, taking advantage of its ability to effectively reduce noise while simultaneously smoothing out image details. This quality makes Gaussian filters particularly suitable for initial processing stages in tasks like edge detection, where clear and noise-free images are essential for accurately identifying edges.

- A Gaussian filter operates based on the Gaussian function, which applies a smooth, bell-shaped curve to give more emphasis to central pixels and less to those farther away. This weighted averaging effect facilitates effective noise reduction and gentle image smoothing, making Gaussian filters a preferred choice for image preparation before further analysis.
- While applying a 2D Gaussian filter can be beneficial for image processing, it may require significant computational resources due to the need for multiple operations on each pixel throughout the entire image. However, the separability of the Gaussian filter into two 1D Gaussian filters—one operating horizontally and the other vertically—offers a significant optimization.
- By breaking down the 2D Gaussian filter into its one-dimensional counterparts, the filtering process can be performed with substantially reduced computational effort. Initially, the horizontal 1D Gaussian filter is applied across each row of the image, followed by the application of the vertical 1D Gaussian filter down each column. This two-step approach maintains the effectiveness of Gaussian filtering in noise reduction and smoothing while significantly reducing the computational complexity involved.

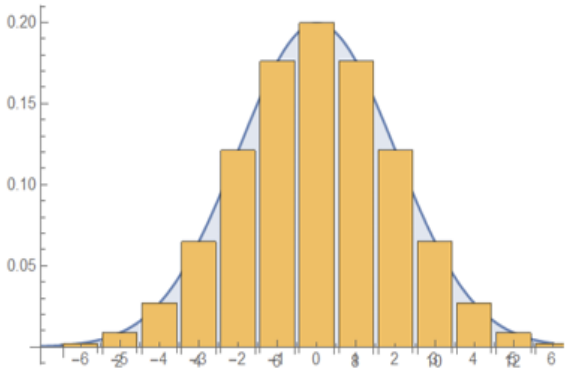
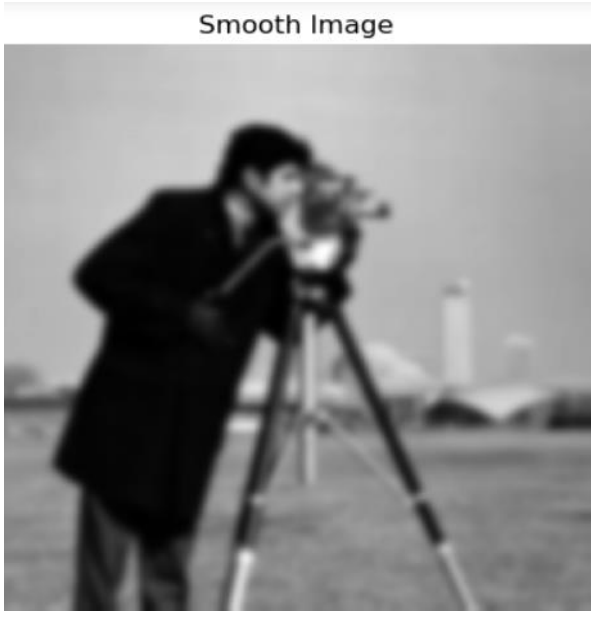


Fig 4: Gaussian Filter

Implementation of Convolution:

- Convolution operations and filters were utilized for edge detection, starting with preprocessing the image to prepare it for subsequent operations.
- Following preprocessing, a Gaussian filter was applied to the image due to its effectiveness in noise reduction, preparing it for gradient calculation and thresholding to identify significant edges.
- To ensure the Gaussian filter accurately captures the essential characteristics of the Gaussian distribution, two sigma (σ) values were considered: 2.0 and 3.0.
    - With σ=2.0, the filter is narrower, resulting in less blurring and better retention of details, making it suitable for images with minimal noise or where detail sharpness is prioritized.

    - With σ=3.0, a broader filter is created, introducing more blurring but providing enhanced smoothing, which is preferable for images with significant noise.
- The filter size is determined using the formula 6σ+1, ensuring coverage of 3σ on each side from the center to adequately represent the Gaussian distribution's key attributes.
- Given the σ value of 2.0, the filter size would be 6×2+1=13.
- The choice of σ=2.0 was made to avoid excessive blurring of finer details or leaving too much noise, striking a balance between noise reduction and detail preservation based on observed smoothing levels in the image.
- Consequently, Gaussian smoothing with separable 1D Gaussian filters was applied to the image, effectively reducing noise while retaining important details. Utilizing a σ value of 2.0 and a filter size of 13 achieved a favorable balance between noise reduction and detail preservation.
- The final denoised image, following the application of Gaussian blur, is presented below.

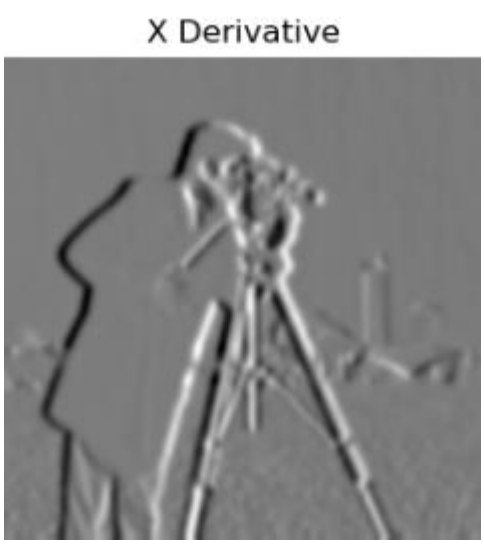


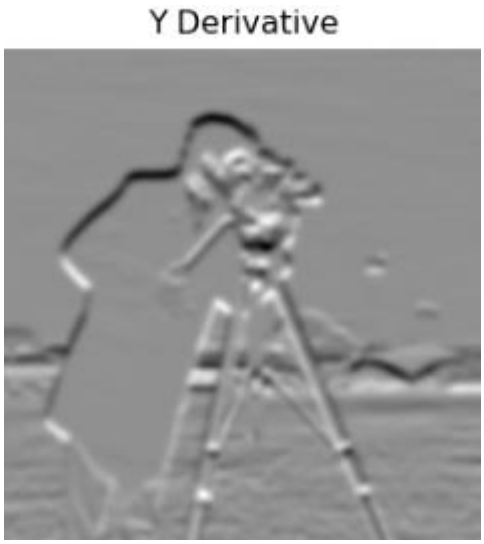
a) Implementation of First Derivative:

- For edge detection, we utilize first derivative filters, namely Gx and Gy, on the denoised image of the cameraman. These filters help calculate the gradient magnitude, effectively highlighting edges by detecting changes in intensity. This method is crucial for identifying borders that separate different areas within the images.
- The application of the Gx derivative filter, represented by [-1 0 1], on the denoised image highlights vertical edges. Conversely, the Gy derivative filter, denoted as [-1 0 1] ^T, emphasizes horizontal edges.
- To synthesize the gradient image from these derivative images, we employ the formula for gradient magnitude: gradient magnitude = square root of (derivative image Gx ^ 2 + derivative

image Gy ^2). This equation combines the squared magnitudes of the Gx and Gy images, providing a comprehensive visualization of edge strength across the image.

- The gradient magnitude images accurately depict edges, which are essential for subsequent image analysis tasks. By applying a threshold to these gradient magnitude images to enhance edges, we effectively distinguish edge pixels from non-edge pixels, improving the accuracy of our edge detection results.
- The subsequent image demonstrates the outcomes of the edge detection algorithm after implementing thresholding, showcasing a more refined identification of edges within the image.





b) **Implementation of Laplacian or Second Derivative**:

- The Laplacian filter is employed to emphasize regions in an image where there's a sudden change in intensity, often indicating the presence of an edge. By computing the second derivative of the image's intensity, this filter highlights areas where the intensity gradient—or the rate of intensity change—is particularly steep. Unlike directional edge-detection filters, the

Laplacian filter is isotropic, meaning it identifies edges in any orientation, offering comprehensive edge detection.

- The 3x3 matrix used for the Laplacian filter provides a discrete approximation of this operator, assessing a pixel's intensity relative to its neighbors. When applied to our pre-processed image, regions with uniform intensity yield values close to zero, while edges produce significantly higher values, allowing precise edge localization. This step follows Gaussian smoothing, employing a predefined convolution function to seamlessly apply the filter.

- The specific 3x3 Laplacian filter utilized for this purpose is as follows:
    - [[0,-1,0],[-1,4,-1],[0,-1,0]]
- Following the application of the Laplacian filter, edges are identified at the zero-crossings of the second derivative, where the Laplacian's sign in the image changes. This alteration indicates a transition from light to dark areas or vice versa. We analyze the Laplacian-filtered image to detect pixels with positive values adjacent to at least one negative value, as this signifies edge locations.

- While the identified edges are visible in the images, some may not be particularly prominent. The detected edges correspond to areas of significant intensity shifts in the original image.
- To further refine the results, we aim to filter out only strong edges, those with significant intensity changes. This step ensures that only truly relevant edges are highlighted. To achieve this, we implement a delta threshold to filter edges based on their "strength" or the distinctness of their intensity transitions, focusing on preserving only the most significant edges.

- The final image effectively depicts critical edge positions within the original image, delineating transitions between different areas, objects, or attributes.

Cross-Correlation and Template Matching:

Introduction to Template Matching:

- Object recognition stands as a fundamental task in computer vision, focusing on identifying instances of specific objects within an image. Template matching emerges as a straightforward yet effective technique for achieving this, particularly beneficial when the object's appearance remains consistent across instances. This method entails sliding a template image over the target image and computing a similarity score at each position, ultimately enabling accurate localization of the object within the image.

- In image processing, cross-correlation serves as a vital tool for assessing the similarity between two images or signals at varying positions. This technique plays a pivotal role in template matching procedures, where a predefined template (or kernel) is incrementally shifted across a larger target image. At each step, the template undergoes a comprehensive comparison with specific regions of the target image, leveraging cross-correlation to gauge their similarity.

- This methodology is applied in a project aiming to locate multiple instances of a meerkat within an original image. By utilizing a template image as a comparative filter-mask and conducting cross-correlation against the target image—modified to achieve zero-mean normalization—a correlation map is generated, facilitating the identification process.

- Global peaks within the correlation map signify significant matches between the template and the larger image, serving as the desired outcome by precisely indicating the presence and whereabouts of the targeted object or pattern. Meanwhile, other peaks on the map indicate local similarities or patterns resembling the template's features but with lesser intensity compared to the global peak. These occurrences often arise from noise, texture variations, or distinct elements in the image sharing a partial resemblance to the template.

Implementation and Results of Template Matching:

- Both the original and template images underwent initial transformations into grayscale and subsequent conversion into floating-point types, facilitating more manageable numerical computations. The template image was further normalized to achieve a zero mean, prioritizing shape attributes over brightness fluctuations. This normalization step is crucial for ensuring consistent matching across different lighting conditions.

- Normalizing a template image to zero mean is a critical preprocessing step in template matching, particularly in techniques utilizing cross-correlation. This adjustment involves subtracting the average pixel value of the template from each pixel, resulting in a template with a mean value of zero. Such a procedure significantly enhances the effectiveness of the matching process by emphasizing the structural characteristics or patterns of the template rather than its absolute pixel values. These values may vary due to factors like lighting disparities, shadows, or other image variations. As a result, the process becomes more focused

on the template's shape and edges, enabling more accurate identification of these elements in the target image, regardless of changes in brightness or contrast.

- The refined filtered image serves as a pivotal outcome of our image processing workflow, integrating normalization, cross-correlation, and contrast enhancement to pinpoint regions in the original image that correspond to a given template. Initially, the template image is standardized to zero mean to prevent any bias from its brightness or contrast during the matching process. This zero-mean template is then subjected to cross-correlation with the original image, generating a convolution map that illustrates the similarity between the template and each image region.

- To improve the contrast of the convolution map, its intensity values are adjusted by scaling, ensuring the lowest value becomes 0 and the highest 255. This adjustment enhances the visibility of regions with high similarity, aiding in the identification of matches. Subsequently, a threshold is applied to the adjusted convolution image to isolate the most significant matches, resulting in a binary map where only the strongest matching regions are highlighted.

- Thus, the refined filtered image offers a clear depiction of where the template aligns with the original image, simplifying the identification and analysis of these areas. This approach amalgamates technical accuracy with practical visualization strategies, providing an accessible means to comprehend and interpret the outcomes of the template matching process.

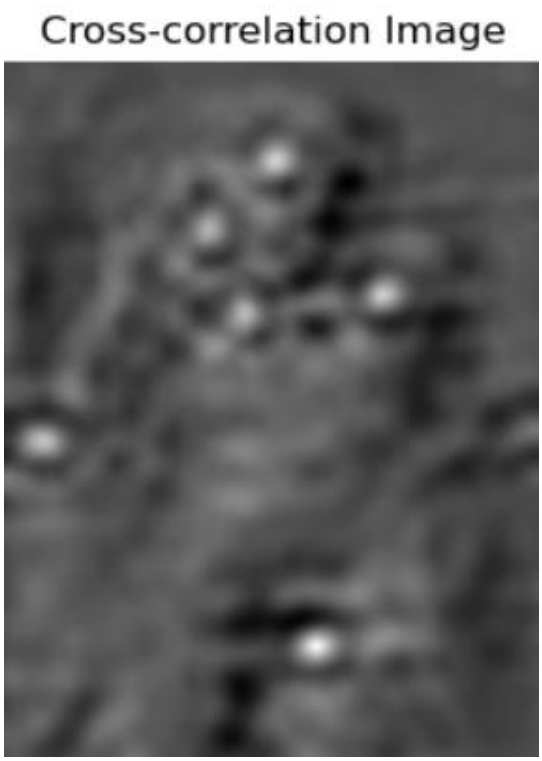
- In investigating the impact of thresholding on the enhanced correlation map, a systematic methodology was employed to comprehend how varying threshold values influence object detection within an image. Initially, the threshold was established at 60% of the maximum intensity value 160, informed by preliminary evaluations. At this threshold, the detection algorithm demonstrated optimal effectiveness in identifying all target objects, including each member of the animal family, while minimizing false positives. The resultant binary image exhibited a well-balanced sensitivity and specificity in detection, accurately highlighting all meerkats. Thus, this threshold value represented an optimal point where most relevant detections were captured, while irrelevant areas were minimized.

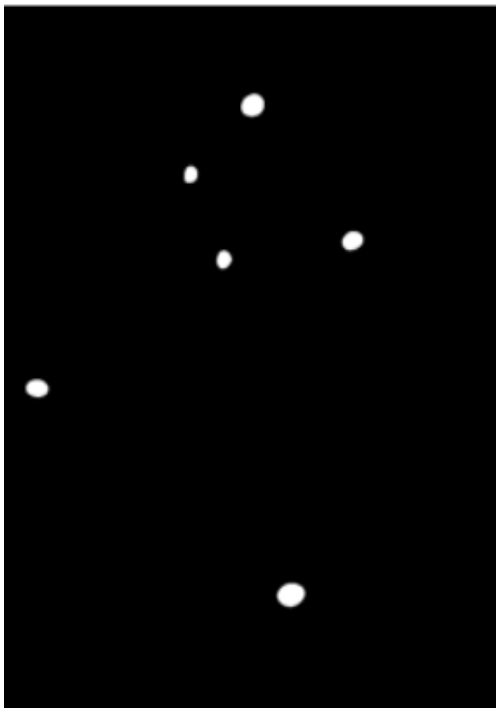
- Subsequent experiments involved incrementally increasing the threshold values beyond 60%, specifically to 170 and 180 of the maximum intensity. Higher thresholds led to more distinct peaks in the binary images; however, this came at the expense of missing some objects. For instance, at a threshold of 180, a discernible reduction in detection completeness was observed, as the algorithm failed to identify a meerkat positioned at the bottom left of the image. This underscored a trade-off between achieving clearer peak distinctions and maintaining detection completeness.

- Original Image:

- Template Image:

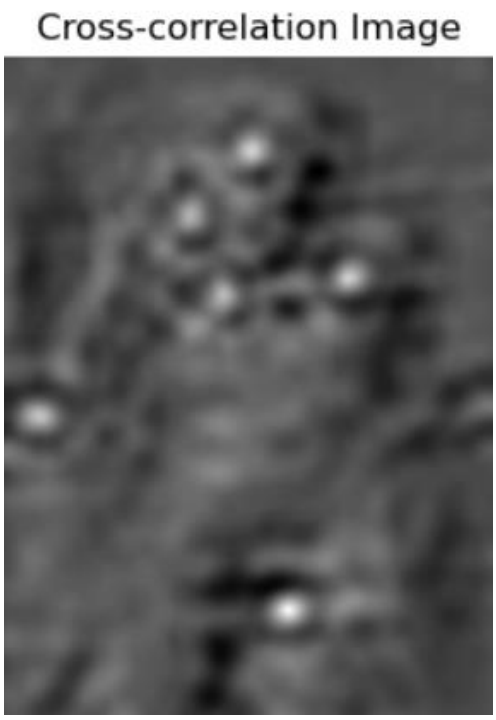
- Results with threshold value of 160:

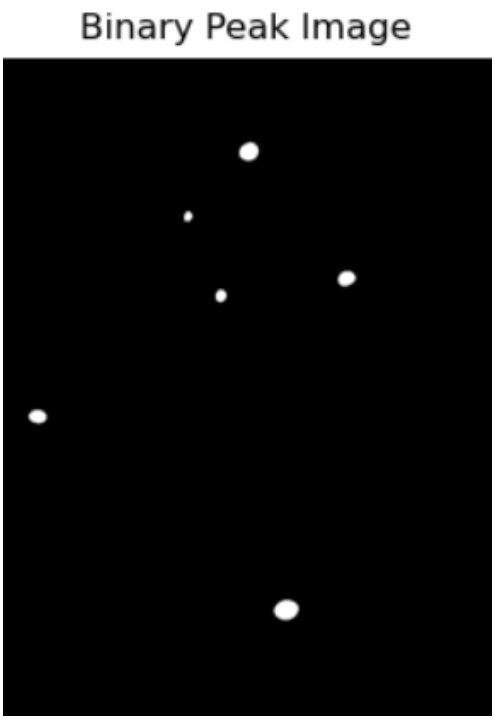




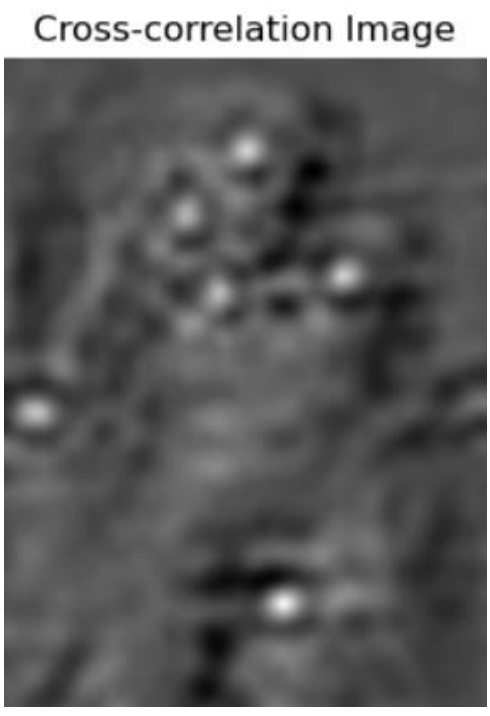
- Number of Match Found = 6

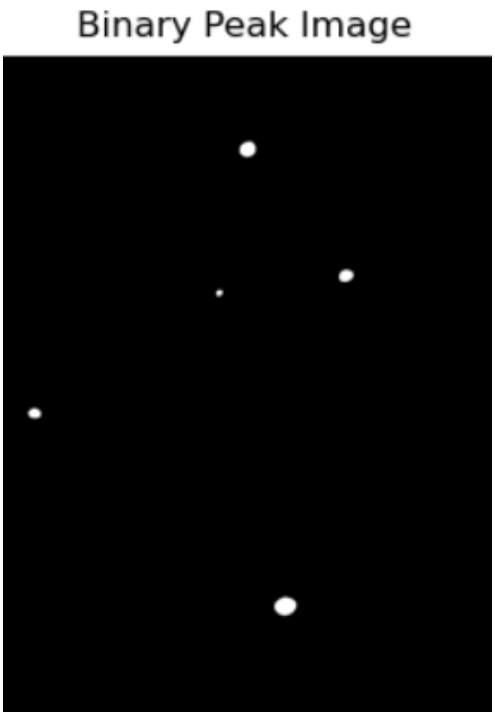
- Results with threshold values of 170:
- Number of Match Found = 6





- Results with threshold values of 180
- Number of Match Found = 5





- The thresholding experiments highlighted the crucial significance of choosing a suitable threshold value for object detection using cross-correlation and template matching. An optimal threshold strikes a balance between the requirement for specificity in detecting the desired objects and the necessity to minimize the inclusion of irrelevant detections. Our results illustrate that while zero-mean normalization of the template image improves the resilience of the detection process, the overall effectiveness of the method greatly depends on meticulous threshold adjustment. This adjustment must be customized to meet the specific requirements of the application and the distinctive attributes of the images under analysis.

Creative Part

Introduction:

In the creative part we have done 2 experiments of character matching and object detection using: Cross-correlation Template Matching. Character matching is done using an image of a group of characters of a video game and the template for this image is a single character. For object detection we have an image of a landscape in a video game and the object(template) is the backside of the character.

Implementation and Results:

- Image Preparation and Preprocessing: Utilizing a suitable high-resolution grayscale image of the parking lot and a template image depicting the target car type, both images were converted to floating-point format. The template image underwent normalization to highlight structural details over brightness variations.

- Template Matching: Employed a custom convolution function to slide the normalized template across the parking lot image, computing similarity measures to identify potential matches.

- Intensity Adjustment and Thresholding: Standardized the intensity of the filtered image to a 0-255 scale and applied multiple thresholds to effectively isolate matches. Detected areas on the original image were visualized using red dots for further analysis.

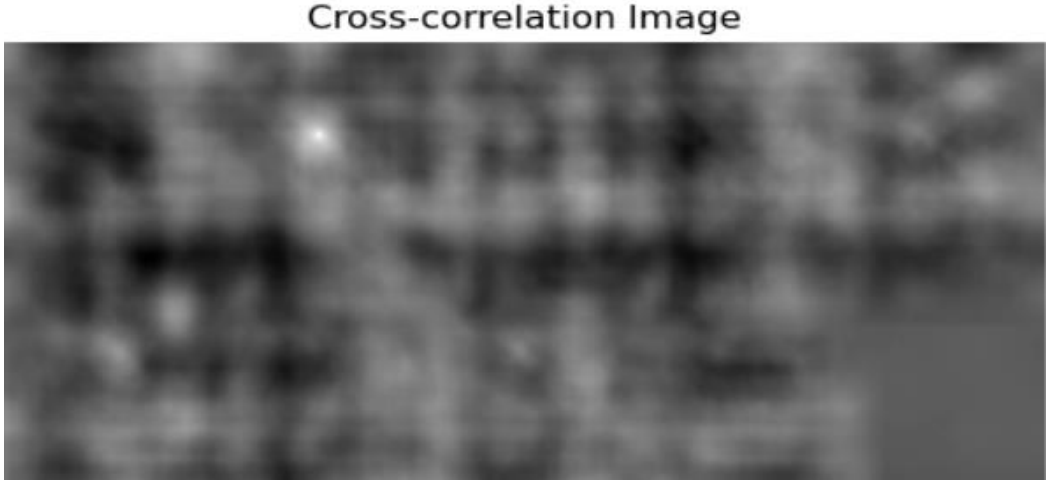
- Detection Overlay: Illustrated the processing stages, encompassing the original image, adjusted convolution output, binarized peak image, and an overlay of detected peaks superimposed on the original image. The overlay employed red dots (represented in grayscale due to the figure's color map) to denote detected car locations, facilitating visual evaluation of detection accuracy.

Result of Character Matching:

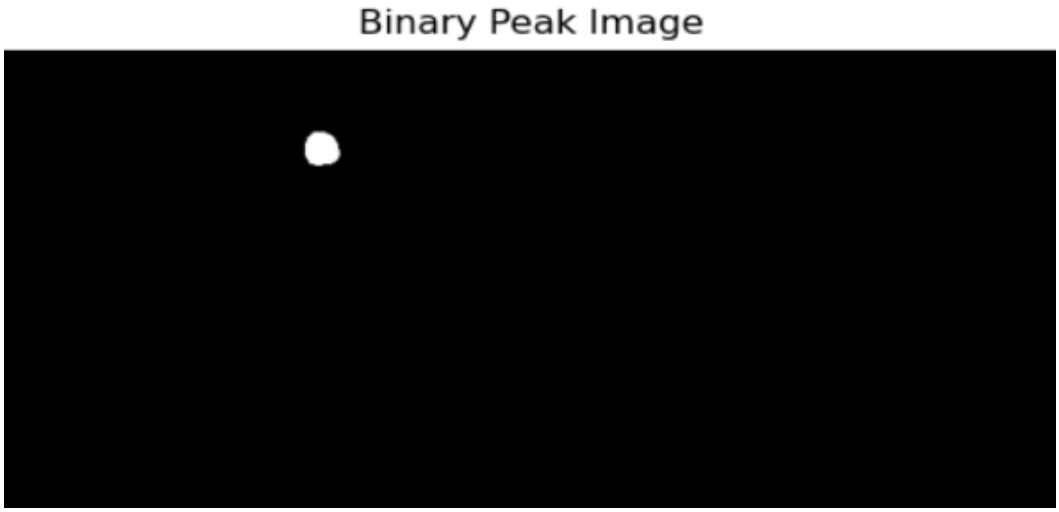
- Image:

- Template:

- Cross-Correlation:



- Binary Image:



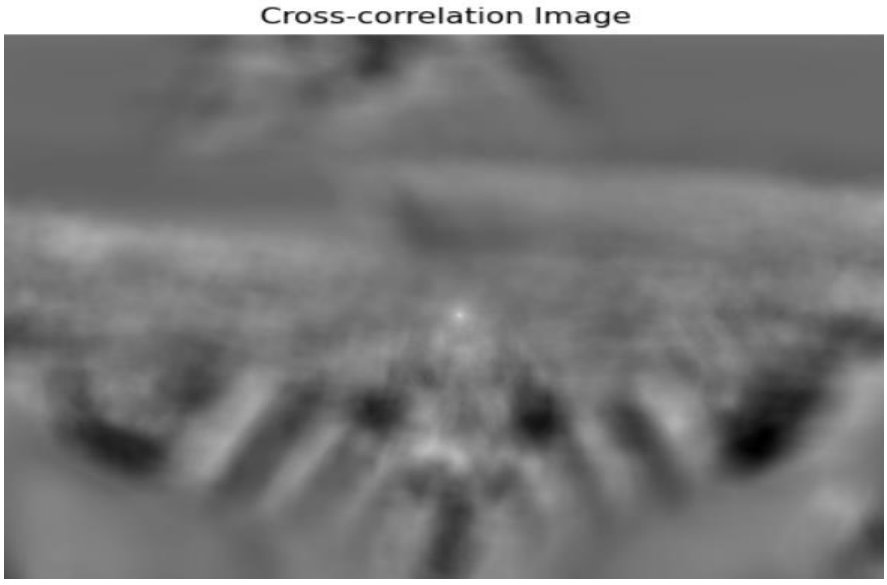
- Overlay Image:

Results of Object Detection:

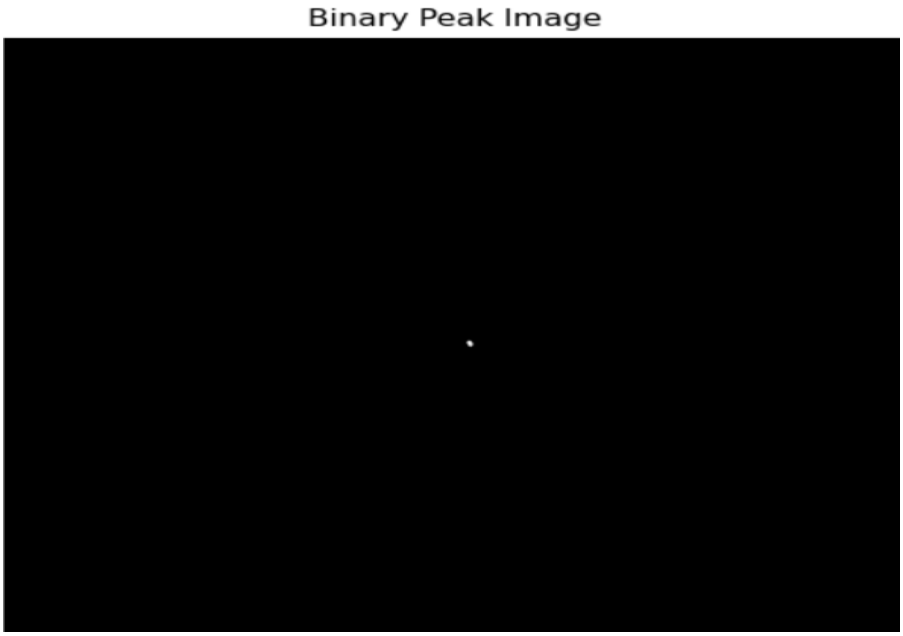
- Image:

- Template:

- Cross-correlation:



- Binary Image:



- Overlay Image:

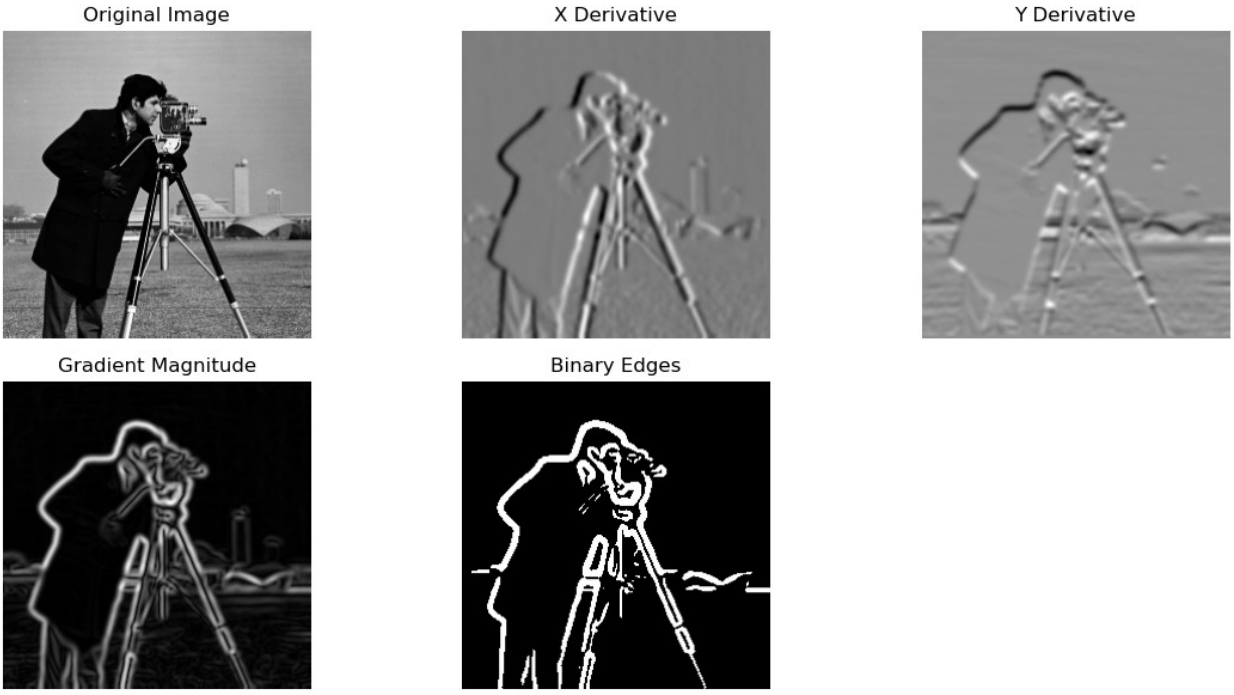
Q1

```
#Using 1 multipourpose convolve function

from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
# Define the convolution function
def convolution(f, I):
    # Calculate padding dimensions
    row_padding = f.shape[0] // 2
    column_padding = f.shape[1] // 2
    pad_width = ((row_padding, row_padding), (column_padding,
column_padding))
    I = np.pad(I, pad_width, mode='reflect')  # Apply padding to the
input image
    new_matrix = [] # Initialize an empty matrix to store the
convolved image
    for i in range(I.shape[0]- (f.shape[0] - 1)): # Iterate over the
image to perform convolution
        new_row = []
        for j in range(I.shape[1] - (f.shape[1] - 1)):
            # Perform element-wise multiplication and summation
            z = np.sum(I[i:i + f.shape[0] , j:j + f.shape[1]] * f)

            new_row.append(z)
        new_matrix.append(new_row)
    return np.array(new_matrix)
# Define the Gaussian filter function
def gaussian_filter(sigma):
    size = int(6 * sigma + 1)

    center = (size // 2) + 1

    # Create 1D Gaussian filter
    filter_1d = np.zeros(size, dtype = float)
    for i in range(size):
        x = i - center
        filter_1d[i] = np.exp(-x**2 / (2 * sigma**2))

    # Normalize the filter
    filter_1d /= np.sum(filter_1d)

    return filter_1d

# Load the image
image_pil = Image.open('cameraman.png')
image_np = np.array(image_pil)
img_array_float = image_np.astype(float)
```

```
# Define the derivative filters
x_filter = np.array([[-1, 0, 1]])
y_filter = np.array([[-1],[0], [1]])

# Generate Gaussian filter for smoothing
Gx = gaussian_filter(2)
Gx = Gx.reshape(1, Gx.shape[0])
Gy = Gx.T

# Apply Gaussian smoothing
I = convolution(Gy, img_array_float)
I = convolution(Gx, I)

# Display the smoothed image
plt.imshow(I, cmap='gray')
plt.title('Smooth Image')
plt.axis('off')

plt.show()
```

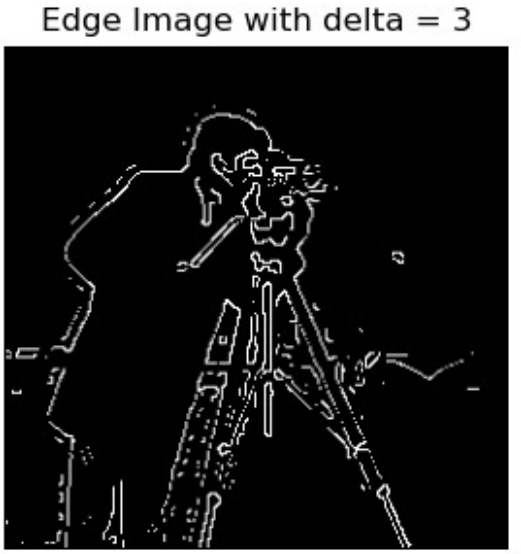

```
#First derivative
dx = convolution(x_filter, I)
dy = convolution(y_filter, I)
# Compute gradient magnitude
gradient_magnitude = np.sqrt(dx**2 + dy**2)
```

```
# Define threshold for binary edges
threshold = 25
binary_edges = np.where(gradient_magnitude > threshold, 255,
0).astype(np.uint8)
# print(binary_edges)
# Plot the images
plt.figure(figsize=(12, 6))

plt.subplot(2, 3, 1)
plt.imshow(image_np, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(2, 3, 2)
plt.imshow(dx, cmap='gray')
plt.title('X Derivative')
plt.axis('off')

plt.subplot(2, 3, 3)
plt.imshow(dy, cmap='gray')
plt.title('Y Derivative')
plt.axis('off')

plt.subplot(2, 3, 4)
plt.imshow(gradient_magnitude, cmap='gray')
plt.title('Gradient Magnitude')
plt.axis('off')

plt.subplot(2, 3, 5)
plt.imshow(binary_edges, cmap='gray')
plt.title('Binary Edges')
plt.axis('off')

plt.tight_layout()
plt.show()
```




```
#second derivative or laplacian
# Define the Laplacian filter
laplacian_filter = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])
# Apply Laplacian filter to the smoothed image
laplacian_image = convolution(laplacian_filter, I)
# Initialize edge image with zeros
height, width = laplacian_image.shape
edge_image = np.zeros_like(laplacian_image)
# Define delta threshold
delta = 3

# Iterate through each pixel in the Laplacian filtered image
for i in range(1, height - 1):
        for j in range(1, width - 1):
             # Check if the pixel value is positive
            if laplacian_image[i, j] > 0:
                # Extract neighboring pixels
                neighbors = laplacian_image[i-1:i+2, j-1:j+2]
                # Check if any neighbor has a negative value
                if np.any(neighbors < 0):
                    # Check if the maximum neighbor value minus the
current pixel value exceeds the delta threshold
                    if np.max(neighbors) - laplacian_image[i, j] >=
delta:
                        # Set the corresponding pixel in the edge
image to 255
                        edge_image[i, j] = 255
```

```
# Plot the Laplacian filter image and the edge image
plt.figure(figsize=(6, 3))  # Adjust the figure size as needed
plt.subplot(1, 2, 1)  # First subplot
plt.imshow(laplacian_image, cmap='gray')
plt.title('Laplacian Filter Image')
plt.axis('off')

plt.subplot(1, 2, 2)  # Second subplot
plt.imshow(edge_image, cmap='gray')
plt.title('Edge Image with delta = 3')
plt.axis('off')

plt.tight_layout()  # Adjust the spacing between subplots for better
readability
plt.show()
```



Q2

```
from PIL import Image

# Load the original image
original_image = np.array(Image.open('animal-family-
25.jpg').convert('L'), dtype=float)

# Load the template image
template_image = np.array(Image.open('animal-family-25-
template.jpg').convert('L'), dtype=float)

# Calculate min, max, and mean value of the template image
mean_val = np.mean(template_image)
print("mean: ", mean_val)
# Normalize the template image
```

```
normalized_template = template_image - mean_val

# Now, the mean of the normalized template should be 0.0
mean_normalized_template = np.mean(normalized_template)
print("Mean of the normalized template:", mean_normalized_template)
# Get correlation-image
correlation_image =  convolution(normalized_template, original_image)
# Adjust correlation image to prevent saturation
adjusted_correlation_image = (correlation_image -
np.min(correlation_image)) / (np.max(correlation_image) -
np.min(correlation_image)) * 255
```

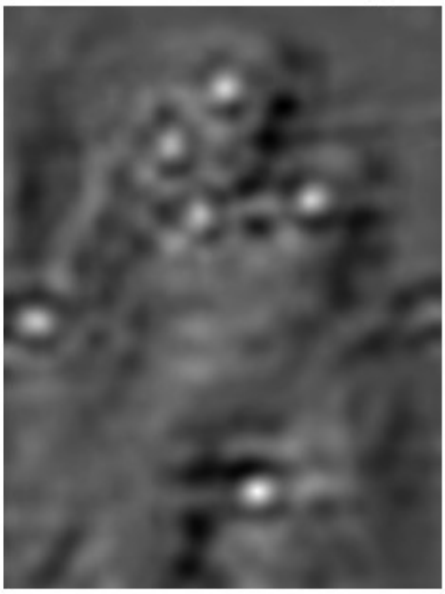
```
mean:  135.27836163836164
Mean of the normalized template: 5.5242035831962075e-15
```

```
# Fuction to calcuate connected white patches
def connected_components(image):
    visited = np.zeros_like(image)
    labels = np.zeros_like(image)
    label_count = 1

    def dfs(row, col, label):
        if row < 0 or col < 0 or row >= image.shape[0] or col >=
image.shape[1]:
            return
        if visited[row, col] or image[row, col] == 0:
            return
        visited[row, col] = 1
        labels[row, col] = label
        for dr, dc in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            dfs(row + dr, col + dc, label)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if not visited[i, j] and image[i, j] == 255:
                dfs(i, j, label_count)
                label_count += 1

    return labels

from scipy.ndimage import label

for i in range(3):
    threshold = 160 + (i * 10)
    binary_peak_image = (adjusted_correlation_image >=
threshold).astype(np.uint8) * 255
    overlay_image = original_image + binary_peak_image
    print("Threshold: ", threshold)
#     labeled_image = connected_components(binary_peak_image)
    labeled_array, num_features = label(binary_peak_image)
```

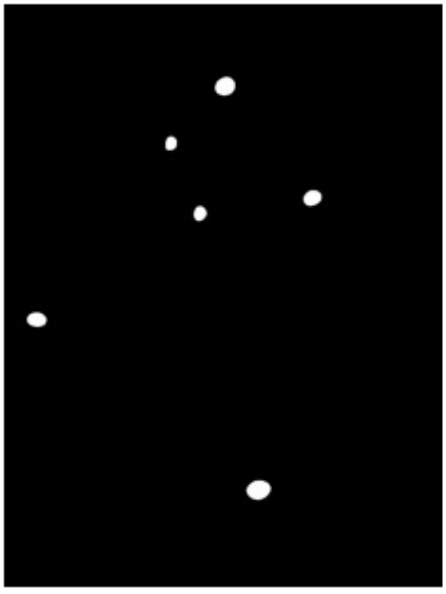
```
#     num_white_patches = np.max(labeled_image)
    num_white_patches = num_features
    print("Number of Matches (connected components):",
num_white_patches)
    # Plotting
    fig, axes = plt.subplots(2, 2, figsize=(10, 7))

    # Plot original image
    axes[0, 0].imshow(original_image, cmap='gray')
    axes[0, 0].set_title('Original Image')

    # Plot cross-correlation image
    axes[0, 1].imshow(adjusted_correlation_image, cmap='gray')
    axes[0, 1].set_title('Cross-correlation Image')

    # Plot binary peak image
    axes[1, 0].imshow(binary_peak_image, cmap='gray')
    axes[1, 0].set_title('Binary Peak Image')

    # Plot overlay image
    axes[1, 1].imshow(overlay_image, cmap='gray')
    axes[1, 1].set_title('Overlay Image')

    # Hide axes
    for ax in axes.flatten():
        ax.axis('off')

    plt.tight_layout()
    plt.show()

Threshold:  160
Number of Matches (connected components): 6
```

Original Image

Cross-correlation Image



Binary Peak Image

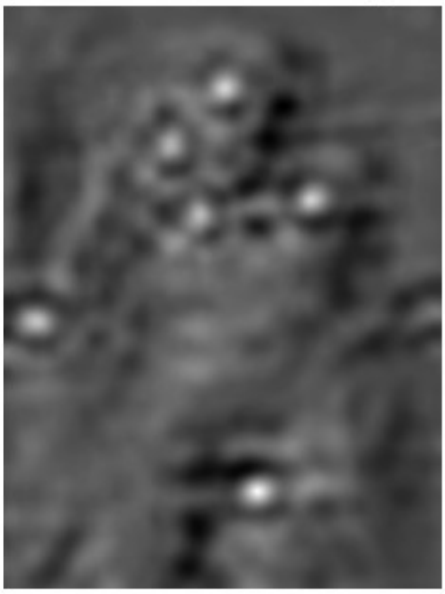


Overlay Image

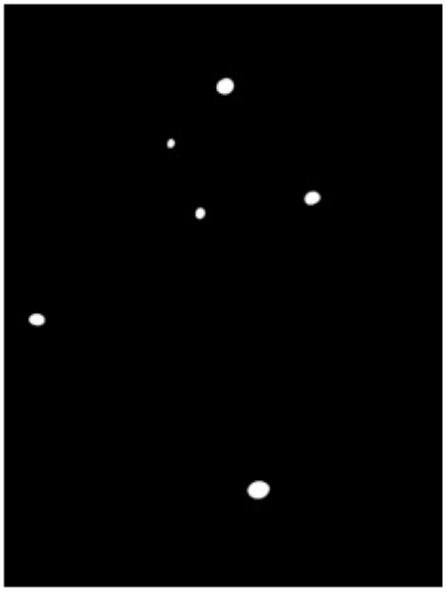
```
Threshold:  170
Number of Matches (connected components): 6
```

Original Image

Cross-correlation Image



Binary Peak Image

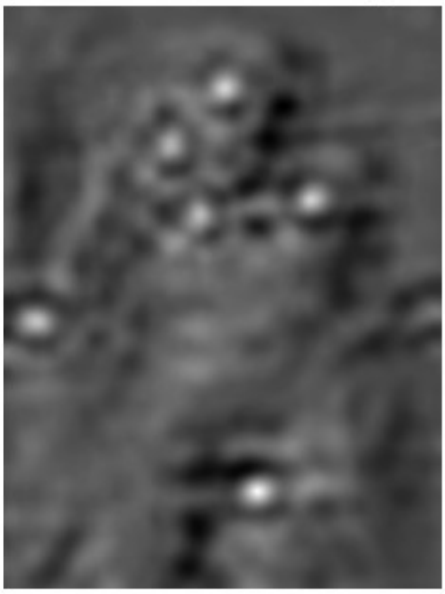


Overlay Image

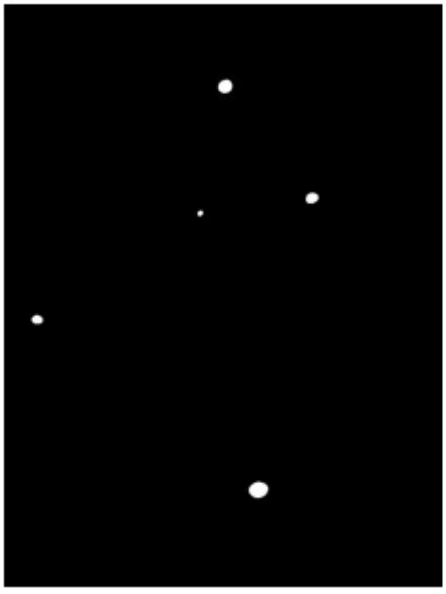
```
Threshold:  180
Number of Matches (connected components): 5
```

Original Image

Cross-correlation Image



Binary Peak Image



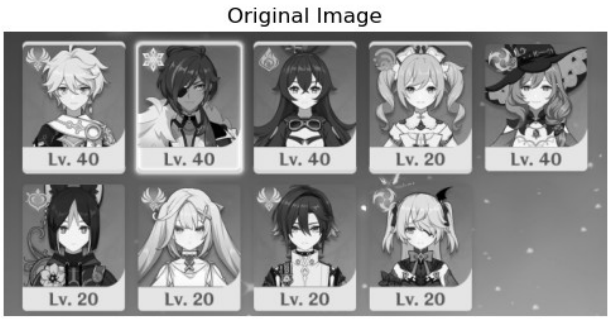
Overlay Image

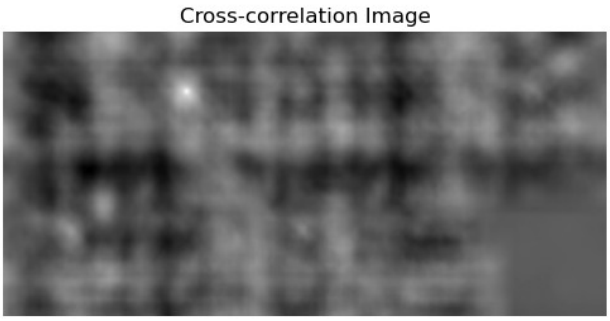
Q3

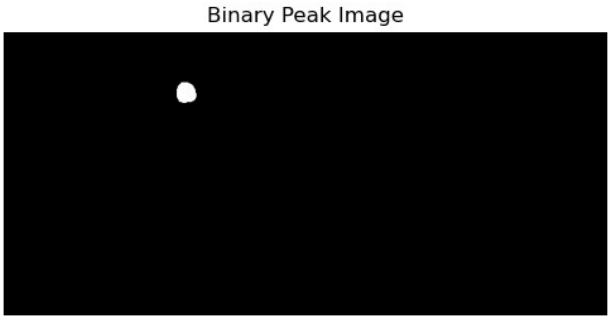
```
# All Steps in Q2 together:
def find_in_image(original_image, template_image, limit):
    mean_val = np.mean(template_image)
    normalized_template = template_image - mean_val
    mean_normalized_template = np.mean(normalized_template)
    correlation_image =  convolution(normalized_template,
original_image)
    print(correlation_image.shape)
```

```
    adjusted_correlation_image = (correlation_image -
np.min(correlation_image)) / (np.max(correlation_image) -
np.min(correlation_image)) * 255
    threshold = limit
    binary_peak_image = (adjusted_correlation_image >=
threshold).astype(np.uint8) * 255
    overlay_image = original_image + binary_peak_image
    # labeled_image = connected_components(binary_peak_image)
    labeled_array, num_features = label(binary_peak_image)
    # num_white_patches = np.max(labeled_image)
    num_white_patches = num_features
    print("Number of Matches (connected components):",
num_white_patches)
    # Plotting
    fig, axes = plt.subplots(2, 2, figsize=(10, 10))

    # Plot original image
    axes[0, 0].imshow(original_image, cmap='gray')
    axes[0, 0].set_title('Original Image')

    # Plot cross-correlation image
    axes[0, 1].imshow(adjusted_correlation_image, cmap='gray')
    axes[0, 1].set_title('Cross-correlation Image')

    # Plot binary peak image
    axes[1, 0].imshow(binary_peak_image, cmap='gray')
    axes[1, 0].set_title('Binary Peak Image')

    # Plot overlay image
    axes[1, 1].imshow(overlay_image, cmap='gray')
    axes[1, 1].set_title('Overlay Image')

    # Hide axes
    for ax in axes.flatten():
        ax.axis('off')

    plt.tight_layout()
    plt.show()

original_image = np.array(Image.open('image_3.png').convert('L'),
dtype=float)
template_image = np.array(Image.open('image_2.png').convert('L'),
dtype=float)
find_in_image(original_image, template_image, 170)
```

```
Number of Matches (connected components): 1
```





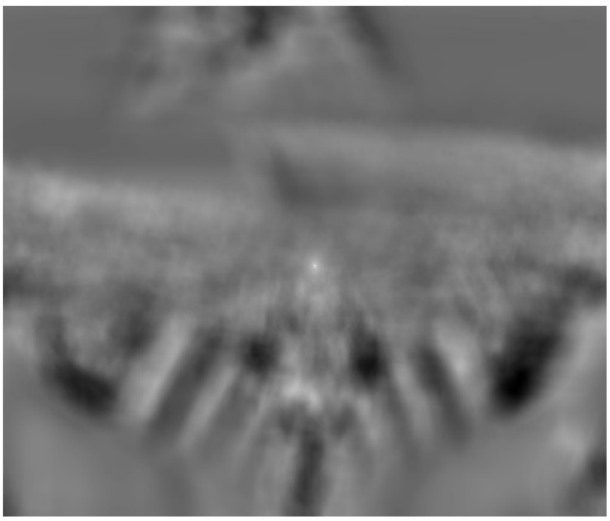



```
original_image = np.array(Image.open('image_4.png').convert('L'),
dtype=float)
template_image = np.array(Image.open('image_1.png').convert('L'),
dtype=float)
find_in_image(original_image, template_image, 190)
```

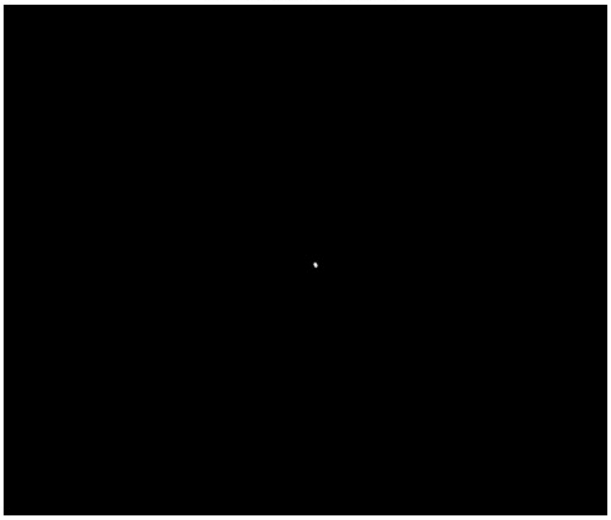
```
(912, 1077)
Number of Matches (connected components): 1
```

Original Image

Cross-correlation Image



Binary Peak Image



Overlay Image