Edge detection is a fundamental process in computer vision and image processing, serving as the foundation for various applications, including object recognition, image segmentation, and feature extraction. The purpose of this report is to discuss a Python implementation that demonstrates edge detection techniques and their application to an input image.

Before we dive into our implementation and results of edge detection, I want to introduce Derivative filters as they are used in image processing to approximate the derivatives of an image, which can provide information about how pixel intensities change from one location to another. These filters are discrete representations of derivatives and are applied to digital images to compute an approximation of the gradient of the image.

Here's how derivative filters are discrete approximations of derivatives:

In continuous mathematics, derivatives are defined for continuous functions, and they provide information about the rate of change of the function at a particular point. In image processing, we are dealing with discrete digital images, where pixel values are sampled at discrete locations.

A digital image can be thought of as a 2D grid of pixels. To compute derivatives in this discrete space, we use finite differences to approximate the change in pixel values. Instead of calculating the derivative of a continuous function, we calculate the difference in pixel values between neighboring pixels along the x and y axes.

Derivative filters, such as the Sobel, Prewitt, and Scharr filters, are designed to calculate these finite differences. For example, the 1D derivative filter [-1, 0, 1] is used to approximate the first derivative along the x-axis. It calculates the difference between the pixel value on the right and left sides of a central pixel. Similarly, the 1D filter [-1, 0, 1] calculates the first derivative along the y-axis.

To apply derivative filters to an image, we use convolution. We slide the derivative filter over the image and calculate the weighted sum of pixel values according to the filter coefficients. The result is an approximation of the gradient of the image, which represents how pixel values change along the x and y axes.

Methodology

To find edges in an image we follow a series of well-defined steps to process our image. Our steps include loading the image and verifying it is a grayscale image. Grayscale conversion simplifies the image, making it easier to detect edges. When performing convolution, a critical consideration is how to handle the boundary of the input image. It's common to extend the image boundaries by padding with zeros to ensure that the output has the same dimensions as the input. The amount of padding usually depends on the size of the filter and the desired behavior at the image boundaries. This approach is known as zero-padding.

To reduce noise in the image, a Gaussian filter is applied. This filter smooths the image while preserving important features, which leads to improving the quality of edge detection.

We used separable filtering, a technique that decomposes a 2D filter into two 1D filters, one among rows and one among columns. Separability approach significantly reducing computational complexity from O(n^2m^2) to O(n^2m) which is huge computational saving.

Our approach uses convolution with Sobel filters in both the horizontal (x) and vertical (y) directions to compute the image gradients [-1 0 1] and [-1 0 1]T . The gradient magnitude is calculated as the square root of the sum of the squared gradients in the x and y directions.

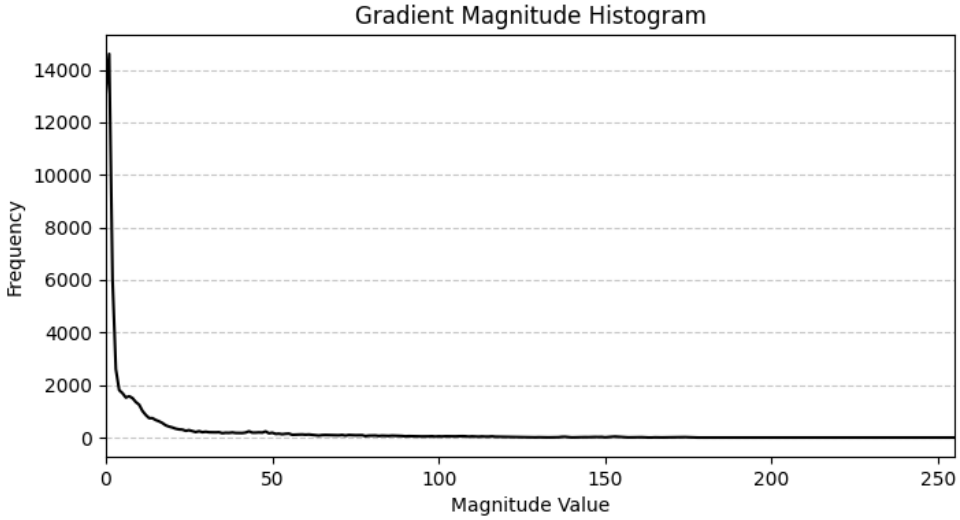
The gradient magnitude image provides information about both the presence and the location of edges in the image. The high-magnitude pixels pinpoint the exact locations of edges. The gradient magnitude can capture edges in various directions. It's not limited to horizontal or vertical edges. It can detect edges at any angle. The gradient magnitude is less sensitive to noise than simple intensity-based methods. Since it focuses on changes in intensity.

In the image below we can see our original image and the gradient magnitude image after applying gaussian and derivatives in both x, and y direction. Gradient image magnitude perfectly detected the edges of the image in all directions.

By applying a threshold to the gradient magnitude image, we can create a binary image where main edge pixels are highlighted . This binary image can be further processed for various applications like object recognition and image segmentation. To figure out the best Threshold we need to implement Histogram analysis for the magnitude gradient image.

Histogram Analysis: A histogram of the gradient magnitude image is generated. This histogram provides insights into the distribution of gradient magnitudes in the image and helps us pick up a threshold.

The histogram is typically visualized as a bar chart, with the x-axis representing the magnitude range (bin) and the y-axis representing the frequency count of how many pixels in the image have gradient magnitudes within the corresponding range.

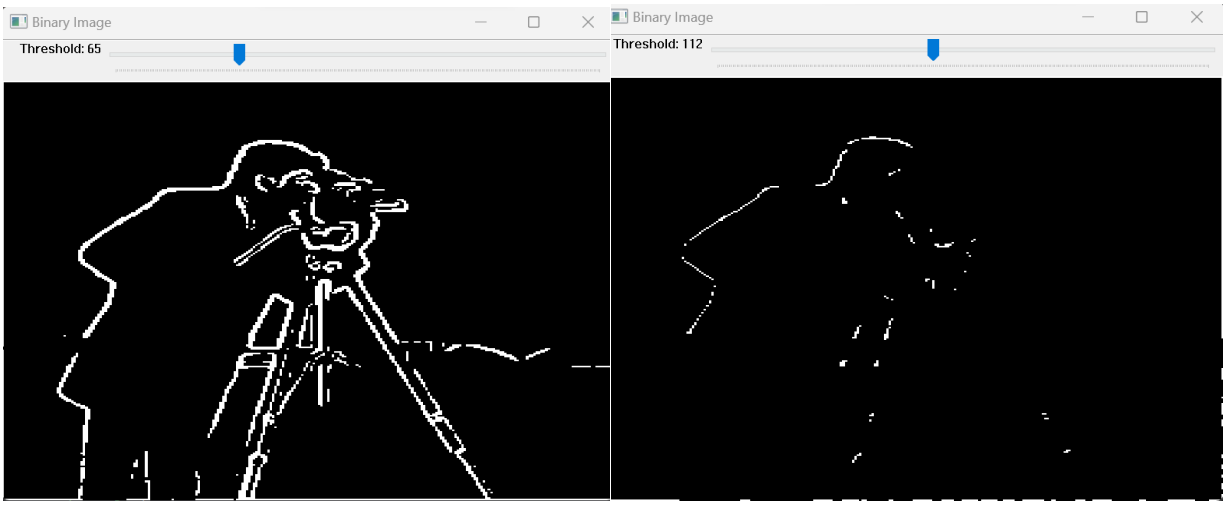


From the above Histogram we can see most of the pixel's intensities are below 40 as we see the histogram skewed and have huge peak on the left side then there is a uniform fixed number of pixels in the greyscale range from 50 to 255 greyscale.

Binary Edge Detection: A binary edge map is created by applying a threshold to the gradient magnitude image. Pixels with gradient magnitudes greater than the threshold are considered edges, while others are set to zero. This step highlights prominent edges in the image. I think threshold that lies between 30-to-40-pixel intensities has the clearest image showing edges in the gradient image as we can see below in the image.

We provide a graphical user interface (GUI) for users to interactively adjust the threshold value. A trackbar in the OpenCV window allows real-time threshold modification and updates of the binary image. When we apply different threshold values the binary image changes and the edges disappear and distorted which make sense as the magnitude gradient image most of its pixel are black and from the histogram they are the majority so if we decrease the threshold below 30 – 40 then some of the pixels becomes white and it get added to the edges and distorted our edges as below in the photo.

When you set a high threshold, many edges may fall below this threshold, causing them to be classified as non-edge regions (pixel value of 0) in the binary image. This results in the disappearance of these edges in the final visualization as we can see in the images below.



The threshold used in binary edge detection is a critical parameter. It determines which gradients are considered as edges.

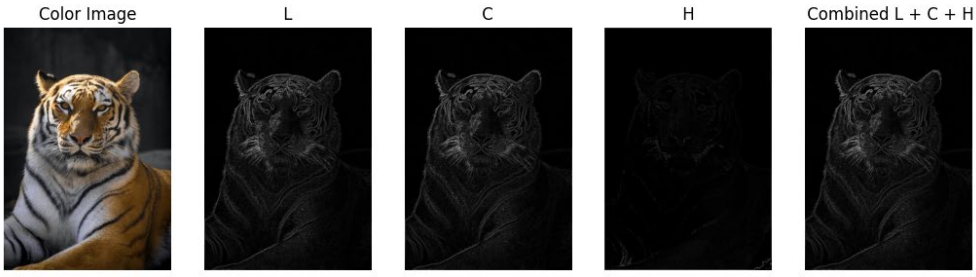
Color Edge Detection (Creative Part)

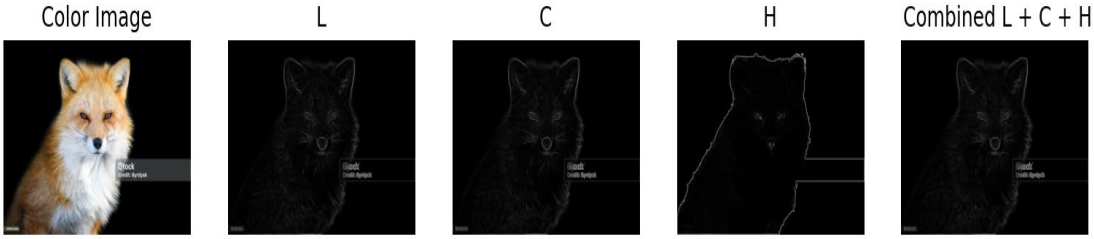
Here we are applying edge detection on color images and will compare our results with greyscale images. We need some processing for our color image to convert it from its RGB form to LCH color space, but we need to address why LCH and what is LCH color space.

LCH is a color representation that separates a color into these three components, making it easier to work with and understand the characteristics of a color. Lightness represents the perceived brightness of the color, Chroma represents the colorfulness, and Hue represents the dominant wavelength of the color.

LCH is used because it is perceptually uniform color space, and it is often used in color science and applications where accurate and intuitive color representation is important.

We extracted every separate channel (L, C, H) and applied gaussian filter, first derivative in the x, y direction and calculated the gradient magnitude for each channel separately. Also combined the result of each channel gradient magnitude divided by 3 to get the average and give equal weight for each channel in my result of calculating the combined gradient magnitude.





Based on our results above with color images we can observe that edge detection results for the L (Luminance) and C (Chrominance) channels are better than the H (Hue) channel in color images is often consistent with the characteristics of these channels and the nature of edge detection. Here's an explanation for the observed differences:

The L channel represents the brightness or intensity of the image. It contains most of the image's structural information. Edges in the L channel is related to changes in the intensity or brightness of the image. This makes it well-suited for capturing structural details and edges. Applying gradient-based edge detection to the L channel can effectively highlight edges in the image, making it a good choice for edge detection.

The C channel represents the color information in an image and is less sensitive to changes in intensity compared to the L channel. Edges in the C channel is related to color changes and can help differentiate objects with different colors. Applying gradient-based edge detection to the C channel can capture color transitions, which are also important for certain types of object boundaries.

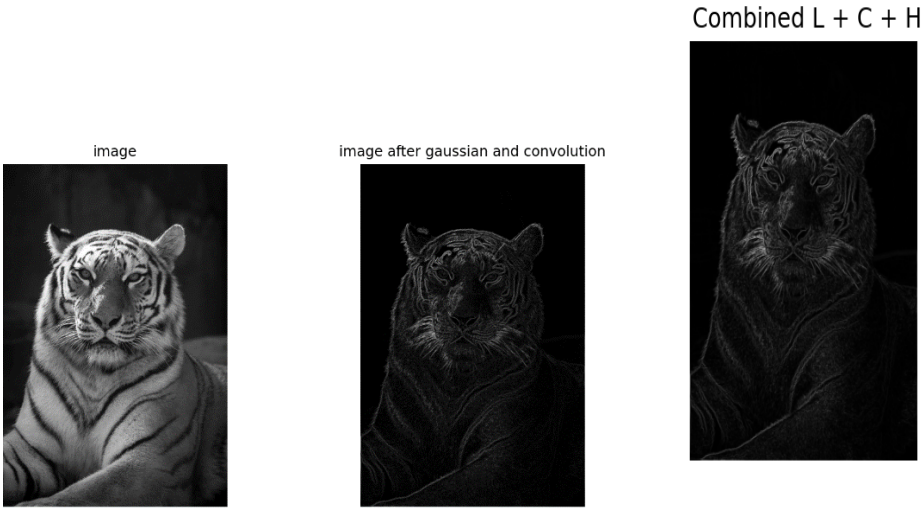
The H channel represents the dominant wavelength of the color, and it is less sensitive to intensity variations. Edges in the H channel may not be as pronounced as in the L and C channels because it primarily encodes color information, not intensity or brightness. Gradient-based edge detection may not be as effective in capturing edges in the H channel because it focuses on color transitions rather than intensity changes.

Combining the L, C, and H channels into a single image often provides a representation that includes both intensity and color information. Edge detection applied to this combined image takes advantage of the strengths of all three channels.

However, the effectiveness of edge detection in the combined image can vary depending on the specific image and the relative weights of the channels.

In summary, the L and C channels are more effective for edge detection in color images due to their inherent characteristics related to intensity and color changes. The H channel, focused primarily on color information, may not yield as prominent edge details when using gradient-based edge detection techniques. The combined LCH representation aims to capture both intensity and color information, but the results depend on the image and the nature of the edges present.

Also, I convert the color image into grey image and run the edge detection mechanism as part 1 of the assignment and the result for it is the left side of the below graph.



Looking at the result of the combined LCH gradient magnitude image vs the grey image gradient magnitude image, I think the result of edge detection highlighted in both images are good and so close to each other , that’s why in industry most of the time working with grey images.

Cross correlation

Template Selection and Preparation

We have an image that has multiple door keys and our task is to select our favorite key from an image and do cross correlation to find that key in the image.

First, we need to prepare our template image by calculating the minimum, maximum, and mean values of the template image to understand its characteristics.

Second, we normalize the template image by subtracting the mean value from all pixel values, resulting in a zero-mean template. There are several reasons for this:

Normalization makes the template invariant to global brightness changes in the original image. In other words, if the overall brightness of the image changes in lighting conditions the zero-mean template will still be able to detect the presence of the pattern.

Normalizing the template helps in focusing on the relative differences in intensity and texture rather than the absolute intensity values. This can improve the effectiveness of pattern matching.

Zero-mean templates often produce more pronounced peaks in the cross-correlation result. When you subtract the mean value from the template, it effectively centers the template around the zero value. This makes it easier to identify the presence of the template in the original image by locating the peaks, which represent regions where the pattern closely matches the template.

By subtracting the mean, you effectively remove the influence of constant noise or variations in illumination, making the pattern matching more robust to noise.

When you need to set a threshold for peak detection, having a zero-mean template simplifies the thresholding process. Peaks are more distinguishable from the background, making it easier to determine an appropriate threshold value.

The Cross correlation function computes the cross-correlation of an input image with a given kernel (template). The image and kernel are converted to float arrays, and padding is added to the input image to ensure that correlation can be computed near the image boundaries. The correlation is performed using nested loops. For each pixel in the output, it calculates the sum of element-wise products between the kernel and the corresponding region in the padded image.

The find peaks function is responsible for identifying peaks in the correlation result.

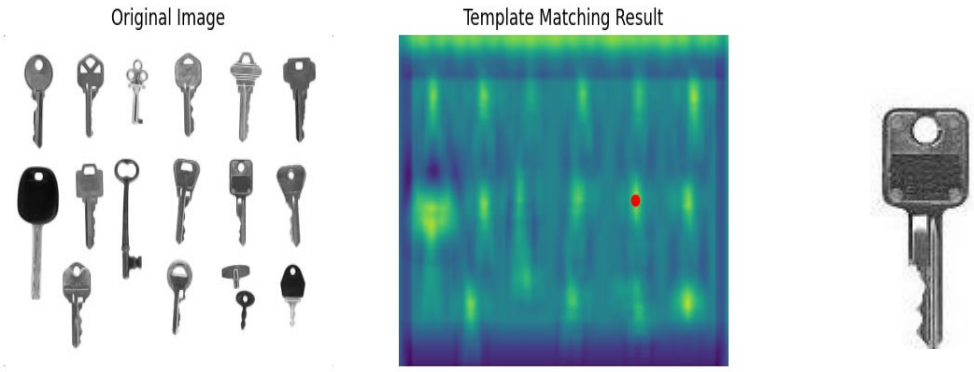
It takes the correlation result and a threshold as inputs. The threshold is a value that determines which correlation scores are considered peaks. I chooses threshold based on the maximum value in the correlation result (0.7 * max peak) just to see other local peaks in the image.

The function iterates over the correlation result image, skipping the image's border. For each pixel in the correlation result, it checks whether the correlation score at that pixel is greater than the specified threshold.

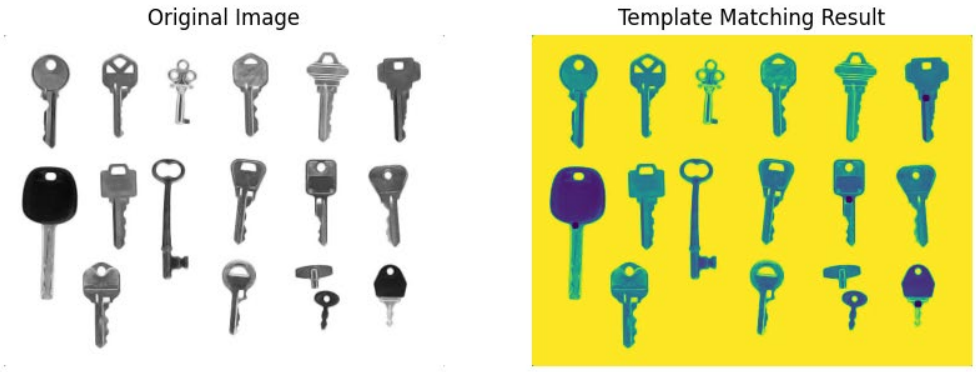
If a pixel's score is above the threshold, the function performs a local neighborhood check to see if it's a local maximum. A pixel is considered a peak if it's greater than all its eight neighboring pixels (top-left, top, top-right, left, right, bottom-left, bottom, and bottom-right).

If a peak is found, its (x, y) coordinates are added to the peak locations list.

The best match location (maximum peak) is marked in the correlation result image with a red circle.



As we choose our threshold of 0.7 of max peak, we can check what other local peaks in the image that our implementation picking up. Below we overlay the peak image with the original peak image so that one sees where major peaks were found.



Based on the threshold there are 3 other local peaks, the top and bottom of last column of keys in the image and second key in the first column of the right side of the image.

This process allows us to locate regions in the original image that closely match the template, and the identified peaks represent these matching locations.

Convolution – Edge detection for Grey Images code.

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

def Cross_correlation(image, kernel):
    # Convert the input image and kernel to float arrays
    image = image.astype(float)
    kernel = kernel.astype(float)

    # Get the dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Calculate the padding for boundary handling
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    # Create an output array to store the result
    output = np.zeros(image.shape, dtype=float)

    # Pad the image with zeros
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width,
pad_width)), mode='constant')

    # Perform 2D convolution
    for i in range(image_height):
        for j in range(image_width):
            output[i, j] = np.sum(padded_image[i:i+kernel_height,
j:j+kernel_width] * kernel)

    return output

def load_image(image_path):
    # Open the image using PIL (Python Imaging Library)
    image = Image.open(image_path)
    # Convert the image to a NumPy array (float32)
    image_data = np.array(image, dtype=np.float32)
    return image_data

def normalize_template(template):
    # Calculate the mean of the template
    template_mean = np.mean(template)

    # Subtract the mean from every pixel in the template
    normalized_template = template - template_mean

    return normalized_template

def find_peaks(result_correlation, threshold):
    # Create lists to store peak coordinates
    local_peak_locations = []
    max_peak_location = None

    # Loop through the correlation result
```

```
    for y in range(1, result_correlation.shape[0] - 1):
        for x in range(1, result_correlation.shape[1] - 1):
            if result_correlation[y, x] > threshold:
                # Check if it's a local maximum
                is_peak = (result_correlation[y, x] > result_correlation[y-1,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y-1,
x]) and \
                          (result_correlation[y, x] > result_correlation[y-1,
x+1]) and \
                          (result_correlation[y, x] > result_correlation[y,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y,
x+1]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x+1])

                if is_peak:
                    local_peak_locations.append((x, y))
                    if max_peak_location is None or result_correlation[y, x]
> result_correlation[max_peak_location[1], max_peak_location[0]]:
                        max_peak_location = (x, y)

    return local_peak_locations, max_peak_location


def overlay_peaks_on_image(image, peaks, max_peak):
    # Create a copy of the original image to overlay on
    result_image = image.copy()

    # Mark the max peak with a red circle
    if max_peak is not None:
        cv2.circle(result_image, (max_peak[0], max_peak[1]), 5, (255, 0, 0),
-1)  # Red circle

    # Mark local peaks with blue circles
    for x, y in peaks:
        cv2.circle(result_image, (x, y), 5, (0, 0, 255), -1)  # Blue circle


    return result_image


# Load image
template_image_path = "single-key-bw.jpg"
image_path = "multiplekeys.png"
template_image = load_image(template_image_path)
image = load_image(image_path)
normalized_template_image = normalize_template(template_image)

result_correlation = Cross_correlation(image,
kernel=normalized_template_image)
```

```
# Find peaks
threshold = 0.75 * np.max(result_correlation)
local_peaks, max_peak = find_peaks(result_correlation, threshold)

# Create an RGB image for visualization
result_image = np.dstack([result_correlation, result_correlation,
result_correlation])

# Mark local peaks with blue circles
for x, y in local_peaks:
    cv2.circle(result_image, (x, y), 5, (0, 0, 255), -1)  # Blue circle

# Mark the max peak with a red circle
if max_peak is not None:
    cv2.circle(result_image, (max_peak[0], max_peak[1]), 5, (255, 0, 0), -1)
# Red circle

# Visualize the result

# Overlay peaks on the original image
overlay_image = overlay_peaks_on_image(image, local_peaks, max_peak)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title("Template Matching Result")
plt.imshow(result_image, cmap='viridis')
plt.imshow(overlay_image)

plt.axis('off')

plt.show()

# Example usage:
print("Local Peak Locations:", local_peaks)
print("Max Peak Location:", max_peak)
```

Edge Detection for color images .

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

# color image edge detection

def convolution_2d(image, kernel):
    # Convert the input image and kernel to float arrays
    image = image.astype(float)
    kernel = kernel.astype(float)

    # Get the dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Calculate the padding for boundary handling
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    # Create an output array to store the result
    output = np.zeros(image.shape, dtype=float)

    # Pad the image with zeros
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width,
pad_width)), mode='constant')

    # Flip the kernel
    kernel = np.flipud(np.fliplr(kernel))

    # Perform 2D convolution
    for i in range(image_height):
        for j in range(image_width):
            output[i, j] = np.sum(padded_image[i:i + kernel_height, j:j +
kernel_width] * kernel)

    return output


# Gaussian filter for denoising
def gaussian_filter(size, sigma):
    kernel = np.fromfunction(lambda x, y: (1 / (2 * np.pi * sigma ** 2)) *
np.exp(
        -((x - (size - 1) / 2) ** 2 + (y - (size - 1) / 2) ** 2) / (2 * sigma
** 2)), (size, size))
    return kernel / np.sum(kernel)


def rgb_to_lch(rgb_image):
    # Convert an RGB image to LCH color space manually
    r, g, b = cv2.split(rgb_image)

    # Convert RGB to XYZ
    x = 0.4124564 * r + 0.3575761 * g + 0.1804375 * b
```

```
    y = 0.2126729 * r + 0.7151522 * g + 0.0721750 * b
    z = 0.0193339 * r + 0.1191920 * g + 0.9503041 * b

    # Convert XYZ to LCH
    l = y
    c = np.sqrt(x ** 2 + y ** 2 + z ** 2)
    h = np.arctan2(z, x)

    return cv2.merge([l, c, h])



def edge_detection_color_image(image):
    # Convert the color image to the LCH color space
    lch_image = rgb_to_lch(image)

    # Separate LCH channels
    l_channel, c_channel, h_channel = cv2.split(lch_image)

    # Apply Gaussian filtering to each channel
    l_channel_filtered = convolution_2d(l_channel, gaussian_filter(size=5,
sigma=1))
    c_channel_filtered = convolution_2d(c_channel, gaussian_filter(size=5,
sigma=1))
    h_channel_filtered = convolution_2d(h_channel, gaussian_filter(size=5,
sigma=1))


    # Compute derivatives for each channel
    dx_filter = np.array([[-1, 0, 1]], dtype=float)
    dy_filter = np.array([[-1], [0], [1]], dtype=float)

    dx_l = convolution_2d(l_channel_filtered, dx_filter)
    dy_l = convolution_2d(l_channel_filtered, dy_filter)
    gradient_magnitude_l = np.sqrt(dx_l ** 2 + dy_l ** 2)


    dx_c = convolution_2d(c_channel_filtered, dx_filter)
    dy_c = convolution_2d(c_channel_filtered, dy_filter)
    gradient_magnitude_c = np.sqrt(dx_c ** 2 + dy_c ** 2)

    dx_h = convolution_2d(h_channel_filtered, dx_filter)
    dy_h = convolution_2d(h_channel_filtered, dy_filter)
    gradient_magnitude_h = np.sqrt(dx_h ** 2 + dy_h ** 2)

    # Sum the gradient magnitude images from the L, C, and H channels as i
want to give equal weight for each channel
    # in my calc to gradient magnitude

    gradient_magnitude_combined = (gradient_magnitude_l +
gradient_magnitude_c + gradient_magnitude_h) / 3.0

    return gradient_magnitude_l,gradient_magnitude_c, gradient_magnitude_h,
gradient_magnitude_combined
```

```
# Load a color image
image_path = "tiger.jpeg"
#image_path ="foox.jpg"
color_image = cv2.imread(image_path)
print(color_image.shape)
# Perform edge detection on each channel and the combined LCH

gradient_magnitude_l,gradient_magnitude_c, gradient_magnitude_h,
gradient_magnitude_combined = edge_detection_color_image(color_image)




# Create a figure with 1 row and 5 columns
fig, axes = plt.subplots(1, 5, figsize=(15, 5))

# Plot the color image in the first subplot
axes[0].set_title("Color Image")
axes[0].imshow(cv2.cvtColor(color_image, cv2.COLOR_BGR2RGB))
axes[0].axis('off')

# Plot the edge detection results in the other subplots
edge_results = [gradient_magnitude_l, gradient_magnitude_c,
gradient_magnitude_h, gradient_magnitude_combined]
titles = [" L ", " C ", " H ", " Combined L + C + H"]

for i in range(4):
    axes[i + 1].set_title(titles[i])
    axes[i + 1].imshow(edge_results[i], cmap='gray')
    axes[i + 1].axis('off')

plt.show()
```

Cross Correlation .

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import cv2
def Cross_correlation(image, kernel):
    # Convert the input image and kernel to float arrays
    image = image.astype(float)
    kernel = kernel.astype(float)

    # Get the dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Calculate the padding for boundary handling
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    # Create an output array to store the result
    output = np.zeros(image.shape, dtype=float)

    # Pad the image with zeros
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width,
pad_width)), mode='constant')


    # Perform 2D convolution
    for i in range(image_height):
        for j in range(image_width):
            output[i, j] = np.sum(padded_image[i:i+kernel_height,
j:j+kernel_width] * kernel)

    return output

def load_image(image_path):
    # Open the image using PIL (Python Imaging Library)
    image = Image.open(image_path)
    # Convert the image to a NumPy array (float32)
    image_data = np.array(image, dtype=np.float32)
    return image_data


def normalize_template(template):
    # Calculate the mean of the template
    template_mean = np.mean(template)

    # Subtract the mean from every pixel in the template
    normalized_template = template - template_mean

    return normalized_template
```

```
def find_peaks(result_correlation, threshold):
    # Create an empty list to store peak coordinates
    peak_locations = []

    # Loop through the correlation result
    for y in range(1, result_correlation.shape[0] - 1):
        for x in range(1, result_correlation.shape[1] - 1):
            if result_correlation[y, x] > threshold:
                # Check if it's a local maximum
                is_peak = (result_correlation[y, x] > result_correlation[y-1,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y-1,
x]) and \
                          (result_correlation[y, x] > result_correlation[y-1,
x+1]) and \
                          (result_correlation[y, x] > result_correlation[y,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y,
x+1]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x+1])

                if is_peak:
                    peak_locations.append((x, y))

    return peak_locations




# Load image
template_image_path = "single-key-bw.jpg"
image_path = "multiplekeys.png"
template_image = load_image(template_image_path)
image = load_image(image_path)
normalized_template_image = normalize_template(template_image)

result_correlation = Cross_correlation(image, kernel=
normalized_template_image)

print("correlationResult", result_correlation)

# Find the location of the maximum value (best match)
best_match = np.unravel_index(np.argmax(result_correlation),
result_correlation.shape)

print(best_match)

# Visualize the result
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image, cmap='gray')
```

```
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title("Template Matching Result")
plt.imshow(result_correlation, cmap='viridis')
plt.plot(best_match[1], best_match[0], 'ro')  # Mark the best match
plt.axis('off')

plt.show()


# Example usage:
threshold = 0.7 * np.max(result_correlation)
peaks = find_peaks(result_correlation, threshold)
print("Peak Locations:", peaks)

# Example usage:
```

overlay the peak image with the original peak image

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

def Cross_correlation(image, kernel):
    # Convert the input image and kernel to float arrays
    image = image.astype(float)
    kernel = kernel.astype(float)

    # Get the dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Calculate the padding for boundary handling
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    # Create an output array to store the result
    output = np.zeros(image.shape, dtype=float)

    # Pad the image with zeros
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width,
pad_width)), mode='constant')

    # Perform 2D convolution
    for i in range(image_height):
        for j in range(image_width):
            output[i, j] = np.sum(padded_image[i:i+kernel_height,
j:j+kernel_width] * kernel)

    return output

def load_image(image_path):
    # Open the image using PIL (Python Imaging Library)
    image = Image.open(image_path)
    # Convert the image to a NumPy array (float32)
    image_data = np.array(image, dtype=np.float32)
    return image_data

def normalize_template(template):
    # Calculate the mean of the template
    template_mean = np.mean(template)

    # Subtract the mean from every pixel in the template
    normalized_template = template - template_mean

    return normalized_template

def find_peaks(result_correlation, threshold):
    # Create lists to store peak coordinates
    local_peak_locations = []
```

```
    max_peak_location = None

    # Loop through the correlation result
    for y in range(1, result_correlation.shape[0] - 1):
        for x in range(1, result_correlation.shape[1] - 1):
            if result_correlation[y, x] > threshold:
                # Check if it's a local maximum
                is_peak = (result_correlation[y, x] > result_correlation[y-1,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y-1,
x]) and \
                          (result_correlation[y, x] > result_correlation[y-1,
x+1]) and \
                          (result_correlation[y, x] > result_correlation[y,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y,
x+1]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x-1]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x]) and \
                          (result_correlation[y, x] > result_correlation[y+1,
x+1])

                if is_peak:
                    local_peak_locations.append((x, y))
                    if max_peak_location is None or result_correlation[y, x]
> result_correlation[max_peak_location[1], max_peak_location[0]]:
                        max_peak_location = (x, y)

    return local_peak_locations, max_peak_location


def overlay_peaks_on_image(image, peaks, max_peak):
    # Create a copy of the original image to overlay on
    result_image = image.copy()

    # Mark the max peak with a red circle
    if max_peak is not None:
        cv2.circle(result_image, (max_peak[0], max_peak[1]), 5, (255, 0, 0),
-1)  # Red circle

    # Mark local peaks with blue circles
    for x, y in peaks:
        cv2.circle(result_image, (x, y), 5, (0, 0, 255), -1)  # Blue circle


    return result_image


# Load image
template_image_path = "single-key-bw.jpg"
image_path = "multiplekeys.png"
template_image = load_image(template_image_path)
image = load_image(image_path)
normalized_template_image = normalize_template(template_image)
```

```
result_correlation = Cross_correlation(image,
kernel=normalized_template_image)

# Find peaks
threshold = 0.75 * np.max(result_correlation)
local_peaks, max_peak = find_peaks(result_correlation, threshold)

# Create an RGB image for visualization
result_image = np.dstack([result_correlation, result_correlation,
result_correlation])

# Mark local peaks with blue circles
for x, y in local_peaks:
    cv2.circle(result_image, (x, y), 5, (0, 0, 255), -1)  # Blue circle

# Mark the max peak with a red circle
if max_peak is not None:
    cv2.circle(result_image, (max_peak[0], max_peak[1]), 5, (255, 0, 0), -1)
# Red circle

# Visualize the result

# Overlay peaks on the original image
overlay_image = overlay_peaks_on_image(image, local_peaks, max_peak)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title("Template Matching Result")
plt.imshow(result_image, cmap='viridis')
plt.imshow(overlay_image)

plt.axis('off')

plt.show()

# Example usage:
print("Local Peak Locations:", local_peaks)
print("Max Peak Location:", max_peak)
```