PROJECT 2 - REPORT .

A1. CONVOLUTION & DERIVATIVE FILTERS

INTRODUCTION:

A mathematical method for determining how closely two items are related is correlation. It is used to calculate the mask's reaction on an image, to use image processing terminology. On a matrix, a mask is applied from left to right. Every time, the mask moves one unit from left to right across the matrix. The mask is moved one unit down and begins to move from left to right once it reaches its farthest right. The core pixel receives the computed output, and neighboring pixels are also utilized in the process. Both 1-D and 2-D masks and matrices are possible. The mask's dimensions are typically taken to be odd so that the central pixel can be found.

- 1D $g(i) = \sum_k f(i+k)h(k)$
- 2D $g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l)$

$$g = f \otimes h$$

Another mathematical tool for combining two variables to get a result is a convolution. Convolution is a technique used in image processing where an input image is transformed by applying a kernel pixel-by-pixel. The convolution mask considers the pixel in question as well as its neighboring pixels while performing an operation on it, returning the result to that specific pixel. Thus, we deduce that the mask operator in image processing is convolution.

Cross Correlation:

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l)$$

$$g = f \otimes h$$

⊗: correlation operator

Convolution:

$$g(i,j) = \sum_{k,l} f(i-k, j-l)h(k,l) = \sum_{k,l} f(k,l)h(i-k, j-l)$$

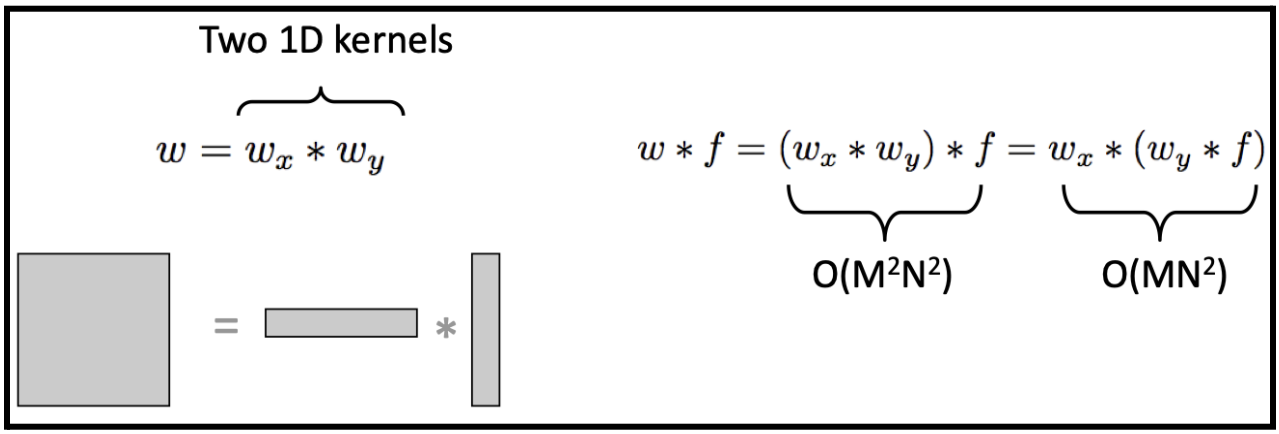
$$g = f * h$$

*: convolution operator

Convolution is the same as cross correlation with the kernel transposed about each axis. If the kernel is symmetric across the axes, then the two operations - correlation and convolution, are the same.

$$g = f \otimes h = f * \hat{h} \quad \hat{h}: \text{reflection of } h$$

In image processing, a separable filter can be expressed as the product of two simpler filters. Generally, two 1-dimensional filters are created from a 2-dimensional convolution procedure. This lowers the computational expenses from $O(M \cdot N \cdot m \cdot n)$ to $O(M \cdot N \cdot (m+n))$ on a N×M image with a m×n filter.



Derivative filters in signal and image processing approximate derivatives in discrete data by using weighted combinations of neighboring samples. They estimate the rate of change between discrete points by applying finite differences, which involve subtracting nearby samples over a small interval. Various types of derivative filters, such as Sobel or Prewitt operators, help identify edges or gradients within images by computing approximate derivatives in horizontal and vertical directions. These filters are crucial in analyzing changes in signals or images, aiding tasks like edge detection and feature extraction.

An edge can be defined as a rapid change in image intensity. A range of mathematical techniques known as edge detection are used to find edges, which are identified as curves in a digital image when there is an abrupt shift in brightness or, more formally, discontinuities. Step detection and change detection are two different terms for the same problem of identifying discontinuities in one-dimensional data and signals that change over time, respectively. In image processing, machine vision, and computer vision - especially in the domains of feature extraction and detection - edge detection is an essential tool.

The goal of spotting abrupt changes in picture brightness is to record significant occurrences and modifications to the world's characteristics. It may be demonstrated that discontinuities in image brightness are likely to correspond to the following under very general assumptions for an image generation model: discontinuities in material characteristics, differences in scene illumination, discontinuities in depth, and discontinuities in surface orientation.

METHODOLOGY AND DISCUSSION:

The goal of this assignment is to perform edge detection on the below 2 Images using convolution operation between Image and filters.

Input Images:

Pseudocode of Edge Detection:

- Apply Gaussian smoothing to reduce noise (separable 1D Gaussians).
  GaussImage(x,y) = [G1x]*([G1y]*Image(x,y))
- Calculate image gradient and gradient magnitude (1D x- and y-derivative filters).
  xderiv = [-1,0,1]*GaussImage(x,y) yderiv = [-1,0,1]T*GaussImage(x,y)
- Calculate Gradient Magnitude of Image.
  Gradmag(x,y) = sqrt(xderiv2+yderiv2)
- Threshold gradient magnitude image to select strongest edges.

As a base function we first create a code implementation for the convolution function which will take inputs of the filter, image, and filter dimensionality (1D or 2D) and return the convoluted image output. Refer code in appendix to find the same. Whilst convolving, keep track of the padding as well for the border cases of the image as shown in the code.

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\exp^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$= \left(\frac{1}{\sqrt{2\pi}\sigma}\exp^{-\frac{x^2}{2\sigma^2}}\right)\left(\frac{1}{\sqrt{2\pi}\sigma}\exp^{-\frac{y^2}{2\sigma^2}}\right)$$

The 1st Step is applying gaussian smoothing by using 1D separable gaussian filters. Below equation shows separability of 2D gaussian filter to give 2 1D gaussian filters. In order to build the Gaussian kernel using code, we have to pass three inputs: the kernel size, sigma value and the dimensionality of the gaussian kernel as demonstrated in the code in the Appendix.

We build the kernels and then go on to perform gaussian blurring on the image using the kernel. If we decide to use a 2D Gaussian kernel we can directly convolve the Image vs the filter. But if we decide to use 1D Gaussian filters we first multiply the image with 1 dimension then the other. Below are the blurring results in booth cases - Left side 2D kernel, right side 1D separable filters:

From the above images, we can conclude that the blurring has occurred to some extent. It would obviously depend on the kernel size and sigma values the user inputs for the gaussian parameters. The results of 2D and 1D cases visually match.

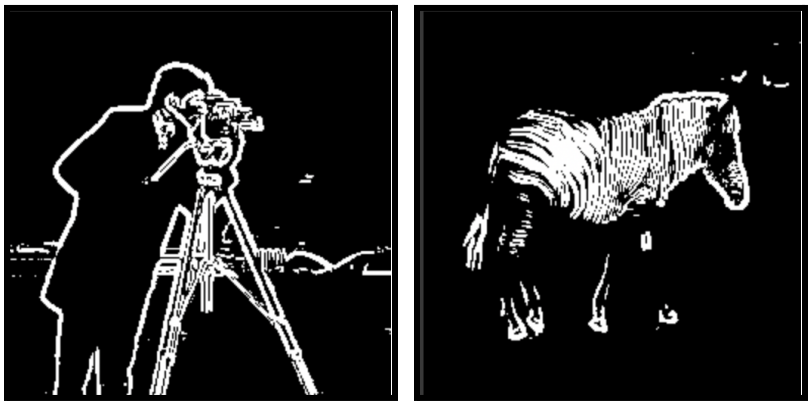
The next step involves using 1D derivative filters on the gaussian blurred images to find the x and y derivative images. The derivative filters can be hard coded as required by assignment to [-1,0,1] in horizontal and vertical directions. The derivative filters are convolved with the gaussian blurred images as per the above pseudocode to result in the below images - Left side is the x derivative and right side is the y derivative images:

From the above images, we can see that the x-derivative filter highlights image edges in the vertical direction whereas the y-derivative filter highlights the image edges in the horizontal direction.

The next step of edge detection involves creating the gradient magnitude image using the x and y derivative images shown above using the formula in the pseudocode. The code for the same can be found in the appendix. Below is the results obtained after applying the gradient magnitude function on the derivative images:

The last step of edge detection involves thresholding the gradient magnitude images to highlight edges. For finding the optimal threshold, we can use Otsu Thresholding as implemented in the code in the appendix. Using the Otsu Threshold eliminates the need to try out other threshold

values since Otsu finds the optimal threshold value for the given gradient image. Below are final results of the edge detection algorithm after thresholding.



A2. CROSS CORRELATION & TEMPLATE MATCHING

INTRODUCTION:

Cross-correlation is defined as the process of calculating the inner product of a template with the contents of an image window when the window is slid over all conceivable picture positions (r, c). Cross-correlation is one of several effective methods for: locating certain image material in a different image or determining the image's content's precise location. However, cross-correlation only functions under specific presumptions. The following should be the only modifications made between the source (template) and target images: Translation, Contrast, and Brightness.

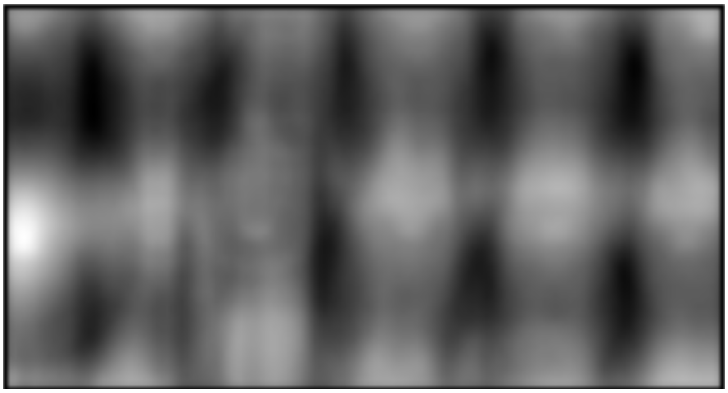
In digital image processing, template matching is a method for locating small portions of a picture that correspond to a template image. It can be applied to edge identification in pictures, mobile robot navigation, and industrial quality control. Detecting occlusion, which occurs when an object is partially hidden in an image, detecting non-rigid transformations, which occur when an object is distorted or imaged from different angles, sensitivity to illumination and background changes, background clutter, and scale changes are the main challenges in a template matching task. Finding a small template image's placement within a larger image is essentially what template matching entails. Typically, the template image's size is less than the search image's size. Normalized template matching requires us to subtract the mean intensity value of the template from each and every pixel in the template to get the zero-mean template.

METHODOLOGY AND DISCUSSION:

The goal of this assignment is to find the key template in the set of keys in the image. Given below are the input images and template on the left and right respectively.

We start with normalizing the template to get the zero mean template. We calculate the mean, min and max intensity values of the template image which are 167.42, 7, and 255 respectively. This can be done using the numpy library as demonstrated in the code in the appendix. Following is the zero mean template created from the above template after subtracting the template's mean value from each and every pixel in the template.

This zero mean template is now used for cross correlation. We go on to run the cross correlation function between the image and the zero mean template. The image is padded as per template size and then cross correlation is run between each and every overlapped region between the image and the template. The size of the correlation image is (image height - template height) X (image width - template width). The resultant image of the cross correlation is as follows:



The more closely the template matches a given overlapped region the higher is the intensity value in the above image. The peak value would thus indicate the best match for the template in the given image. We go on to do a pass through the correlation image created above to find this peak value and its respective index in the image as demonstrated in the code in the appendix.

The location found by the code matches my expectation since the template occurs in the exact region in the image and found by the peak location. Other peaks might have a close match to the template but the highest will always be the exact match or most similar overlapped region. Shown below is the correlation output with the highest peak location marked:

A3. COLOR EDGE DETECTION

INTRODUCTION:

In this section I have extended the edge detection algorithm shown in A1 to detect edges in different color spaces and their respective channels. A technique for representing an object's color using notation, like integers, is known as a color space.

One of the numerous distinct additive colorimetric color spaces based on the RGB color model is the RGB color space. While RGB color spaces are frequently used to describe the physical output of display devices like computer displays and television screens, some RGB color spaces are not directly displayable due to the usage of fictitious primaries.

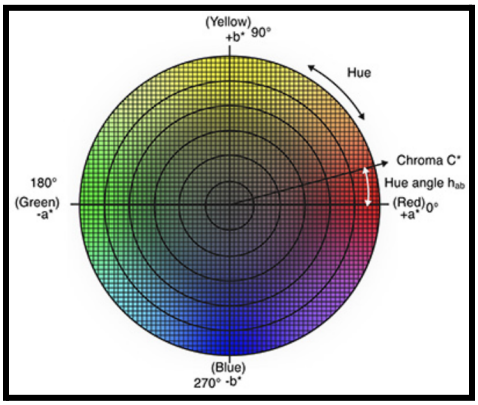
Three different kinds of color-sensitive cone cells are found in the normal human eye. Every cell reacts to light in the three wavelength categories—long, medium, and short—which are commonly referred to as red, green, and blue. The psychological impact of color vision is

processed from the combination of these cone cells' responses, which are collectively referred to as the Tristimulus values. What defines an RGB color space is:

- The additive primaries of red, green, and blue have different chromaticity coordinates.
- White point chromaticity, which is often measured using a standard illuminant.
- The transfer function that links chromaticity to tristimulus values is sometimes referred to as the tone response curve (TRC) or gamma.

RGB is OpenCV's default color space. In actuality, though, color is stored in the BGR format. With this additive color model, many color hues are produced by varying the intensities of Blue, Green, and Red.

For the purpose of communicating and expressing object color, the non-profit Commission Internationale de l'Eclairage (CIE), which is regarded as the authority on the science of light and color, has created color spaces, such as CIE XYZ, CIE L*a*b*, and CIE L*C*h. With the use of these tools, users may assess color characteristics, spot errors, and precisely communicate their conclusions to others in numerical terms. Some specialists in the industry prefer the L*C*h color space, which is similar to CIELAB and has a system that closely matches how the human eye perceives color. L* denotes brightness, C* denotes chroma, and h denotes hue angle in this color space. Chroma C* has a central value of 0 and represents the distance from the lightness axis (L*).



OpenCV does not have LCH color space thus in the code implementation, I have converted RGB to LAB and then to LCH using formulae as shown in the appendix.

EXPERIMENTATION AND DISCUSSION:

For the purpose of this assignment, I have used the following input image Lena.jpg.

The input image can be split into blue, green, and red channels using the split method of the cv2 library as demonstrated in the code in the appendix. Shown below is the step by step results of edge detection in individual channels:

Original Blue, Green, and Red components of Image:

The individual channel images above show the grayscale intensity of the respective color.

Gaussian Blurred Blue, Green, and Red components of Image:

X-derivative Filtering of Gaussian Blurred Image:

Y-derivative Filtering of Gaussian Blurred Image:

From the above x and y derivative filters of the individual color channels, it can be observed that the edges in the channels are identical but the strength of edges in each differs to some extent. This shows that edges of color images have components of all 3 color channels.

Gradient Magnitude of Derivative Filters:

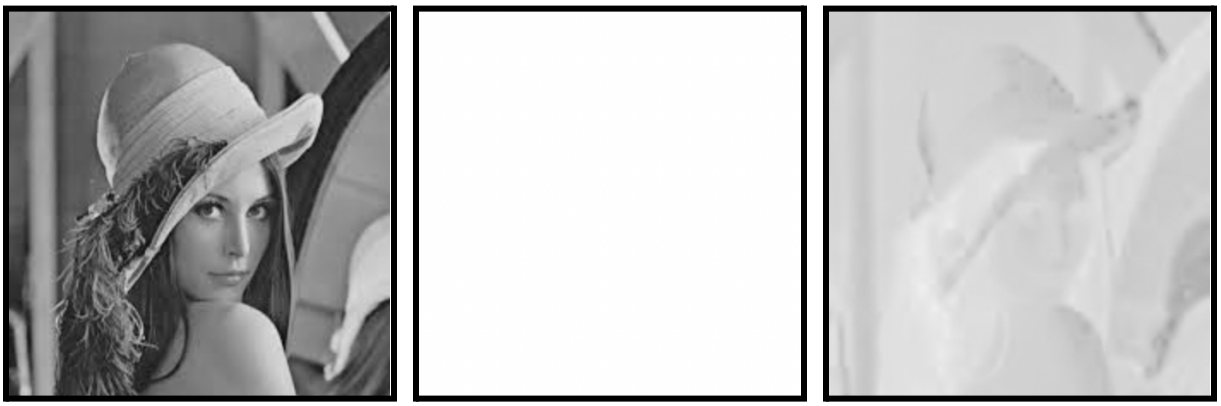
Otsu Thresholding of Gradient Magnitude Images:

The above is the solution for edge detection of individual color channels of the BGR colorspace. Using the merge function of the cv2 library, we can create the merged color edge image as shown below:

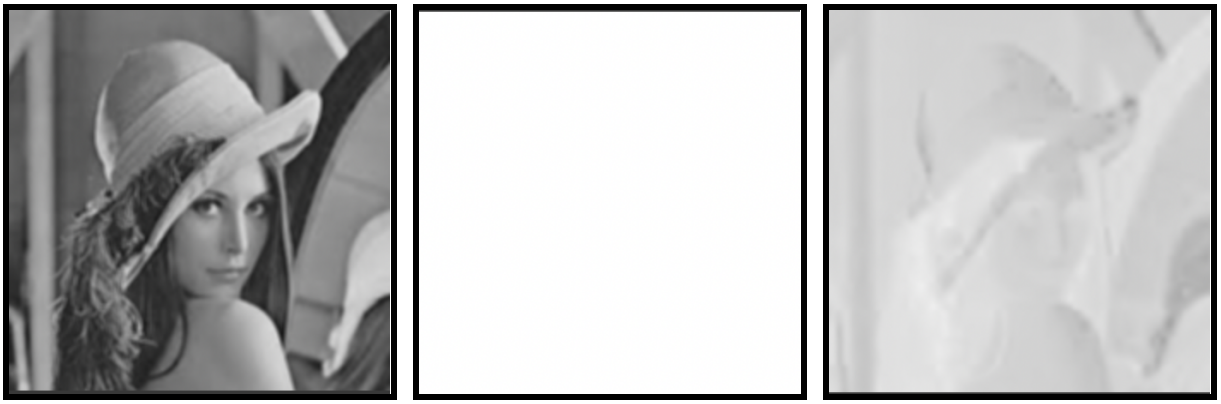
We can also perform the edge detection task on the grayscale image as shown below. The order of operations is the same as above.

If we compare the colored edge detection vs the grayscale edge detection, the colored version gives us the edge strengths in terms of the individual channels and the grayscale version gives us the edge strengths in the channels combined. Thus, the colored edge detection has the advantage of being a more accurate representation and being more visually appealing.

The RGB image is converted to LAB image which is subsequently converted to an LCH image as shown in the code in the appendix. Below images represent the L(Brightness), C(Chroma), and H(Hue) channels of the image respectively.



Gaussian Blurred L, C, H Images:

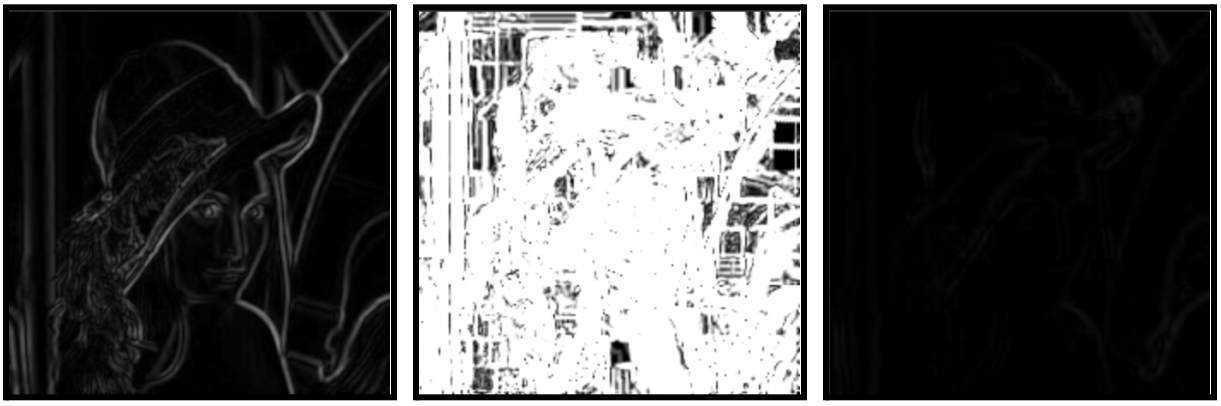


X-derivative filtering of Gaussian Blurred L, C, H Images:

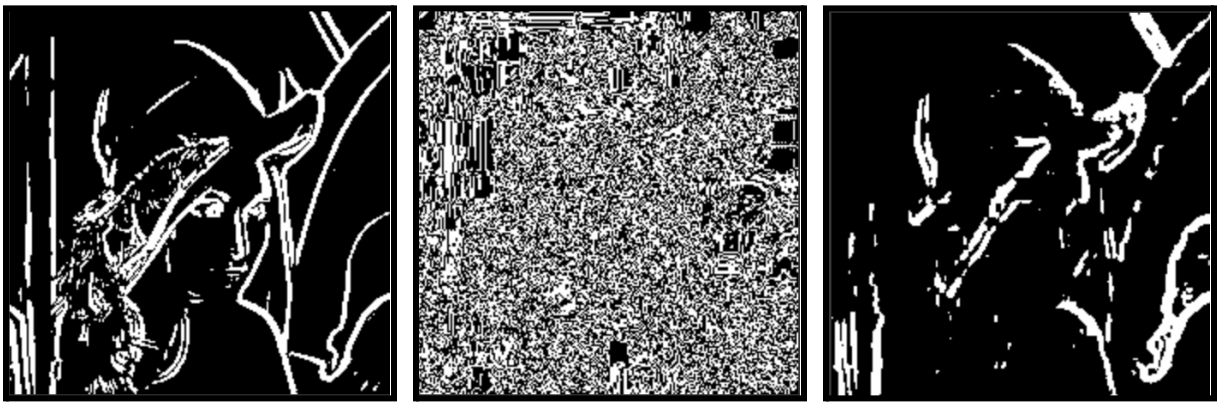
Y-derivative filtering of Gaussian Blurred L, C, H Images:

On observing the derivative filters, the chroma and hue components drastically change upon applying the derivative filters. Chroma component initially had values very close to white and the derivative filter exposed the chroma edges whereas the hue component had a very different representation earlier and the derivative filter has brought the intensities very close to black.

Gradient Magnitude of derivative filtered components:



Otsu Thresholding of Gradient Magnitude Images:



Here we can observe the contribution of each of the channels to the strength of the edges clearly.

APPENDIX: PROJECT CODE:

```
from PIL import Image
import numpy as np
from google.colab.patches import cv2_imshow
import cv2 as cv
import math
from scipy.signal import correlate2d
import matplotlib.pyplot as plt

def preprocess_img(im_in):
      original_image = Image.open(im_in)
      resized_image = original_image.resize((256, 256))
      im = np.array(resized_image.convert('L'))
```

```
    im_uint8 = ((im - np.min(im)) * (1/(np.max(im) - np.min(im)) *
    255)).astype(float)
    return im_uint8

def convolution(f, I, dim, ph, pw, typ):
    if typ == 'Mask':
    if dim == 2:
    im_conv = np.zeros((I.shape[0]-2*ph, I.shape[1]-2*pw))
    for y in range(ph, I.shape[0]-ph):
    for x in range(pw, I.shape[1]-pw):
    im_conv[y-ph][x-pw] = np.sum(np.multiply(I[y-ph:y+ph+1,
    x-pw:x+pw+1], f))
    elif dim == 1:
    fX = np.zeros((I.shape[0], I.shape[1] - pw + 1))
    for i, v in enumerate(f):
    fX += v * I[:, i : I.shape[1] - pw + i + 1]
    fY = np.zeros((fX.shape[0] - ph + 1, fX.shape[1]))
    for i, v in enumerate(f):
    fY += v * fX[i : I.shape[0] - ph + i + 1]
    im_conv = fY
    return im_conv
    elif typ == 'Derivative':
    fX = np.zeros((I.shape[0], I.shape[1] - pw + 1))
    for i, v in enumerate(f):
    fX += v * I[:, i : I.shape[1] - pw + i + 1]
    fY = np.zeros((I.shape[0] - ph + 1, I.shape[1]))
    for i, v in enumerate(f):
    fY += v * I[i : I.shape[0] - ph + i + 1, :]
    return fX, fY

def gaussianKernel(size, sigma, dim):
    padding = int((size-1)/2)
    if dim == 2:
    f = np.fromfunction(lambda x, y: (1/(2*math.pi*sigma**2)) * math.e
    ** ((-1*((x-(size-1)/2)**2+(y-(size-1)/2)**2))/(2*sigma**2)), (size,
    size))
    elif dim == 1:
    f = np.fromfunction(lambda x: math.e ** ((-1*(x-(size-1)/2)**2) /
    (2*sigma**2)), (size,))
    return f / np.sum(f), padding
```

```
def gaussianBlur(I, size, sigma, dim):
    if dim == 2:
    f, padding = gaussianKernel(size, sigma, dim)
    return convolution(f, I, dim, padding, padding, 'Mask')
    elif dim == 1:
    f, padding = gaussianKernel(size, sigma, dim)
    return convolution(f, I, dim, size, size, 'Mask')

def createBorder(img,TDLU):
    borderType = cv.BORDER_CONSTANT
    img = cv.copyMakeBorder(img, TDLU[0] , TDLU[1] , TDLU[2] , TDLU[3] ,
    borderType, None, 0)
    row, col = img.shape
    return img

def derivativeFilter(I):
    d = [-1, 0, 1]
    dX, dY = convolution(d, I, 1, len(d), len(d), 'Derivative')
    h, w = dX.shape
    dX = createBorder(dX, [0, 0, 0, h-w])
    h, w = dY.shape
    dY = createBorder(dY, [0,w-h,0,0])
    return dX, dY

def gradientMagnitude(I, dX, dY):
    for i in range(len(dX)):
    for j in range(len(dX[i])):
    I[i][j] = ((dX[i][j])**2 + (dY[i][j])**2)**0.5
    h, w = I.shape
    I = np.delete(I, [h-1, h-2], 0)
    I = np.delete(I, [w-1, w-2], 1)
    return I

def edgeDetection(im_in):
    print('Edge Det - Input')
    cv2_imshow(im_in)
    g = gaussianBlur(im_in, 3, 2, 2)
    print('2D Gaussian Blur')
    cv2_imshow(g)
    g1 = gaussianBlur(im_in, 3, 2, 1)
```

```
    print('1D Gaussian Blur')
    cv2_imshow(g1)
    dX, dY = derivativeFilter(g1)
    print('X Derivative')
    cv2_imshow(dX)
    print('Y Derivative')
    cv2_imshow(dY)
    gm = gradientMagnitude(im_in, dX, dY)
    print('Gradient Magnitude')
    cv2_imshow(gm)
    gm = (gm).astype(np.uint8)
    ret, thresh1 = cv.threshold(gm, 0, 255,
    cv.THRESH_BINARY+cv.THRESH_OTSU)
    print('Edge Detection Image')
    cv2_imshow(thresh1)
    return thresh1

im_in = preprocess_img('cameraman.png')
thresh1 = edgeDetection(im_in)
im_in = preprocess_img('zebra.png')
thresh1 = edgeDetection(im_in)

def template_matching(template, image):
    image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    template = cv.cvtColor(template, cv.COLOR_BGR2GRAY)

    mean = np.mean(template)
    max = np.amax(template)
    min = np.amin(template)

    print(mean, min, max)

    normalized_template = template - mean
    cv2_imshow(normalized_template)

    image = cv.convertScaleAbs(image)
    normalized_template = cv.convertScaleAbs(normalized_template)

    template_height, template_width = normalized_template.shape[:2]
    image_height, image_width = image.shape[:2]
```

```
    correlation_image = cv.matchTemplate(image, template,
    cv.TM_CCOEFF_NORMED)
    min_val, max_val, min_loc, max_loc = cv.minMaxLoc(correlation_image)

    top_left = max_loc
    bottom_right = (top_left[0] + template_width, top_left[1] +
    template_height)

    image_with_circle = image.copy()
    cv.circle(image_with_circle, max_loc, 10, (0, 0, 255), 2)

    plt.subplot(1, 3, 1), plt.imshow(correlation_image, cmap='gray')
    plt.title('Correlation Image'), plt.xticks([]), plt.yticks([])

    plt.subplot(1, 3, 2), plt.imshow(image, cmap='gray')
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.imshow(image, cmap='gray'), plt.axis('off')
    plt.gca().add_patch(plt.Rectangle(top_left, template_width,
    template_height, edgecolor='b', facecolor='none'))

    plt.subplot(1, 3, 3), plt.imshow(image_with_circle, cmap='gray')
    plt.title('Detected Location'), plt.xticks([]), plt.yticks([])

    plt.show()

image = cv.imread('multiplekeys.png')
template = cv.imread('Key1.jpeg')
cv2_imshow(template)
template_matching(template, image)

def rgb_to_lch(rgb):
    lab = cv.cvtColor(rgb, cv.COLOR_RGB2LAB)
    lab = lab.astype(np.float32)

    L = lab[:,:,0] / 255.0
    a = lab[:,:,1] - 128.0
    b = lab[:,:,2] - 128.0

    C = np.sqrt(np.square(a) + np.square(b))
```

```
    H = np.arctan2(b, a)
    H[H < 0] += 2 * np.pi
    H = np.degrees(H)

    L = (L * 255).astype(float)
    C = (C * 255).astype(float)
    H = (H * 255 / 360).astype(float)

    return L, C, H

def colorEdgeDetection(im_in):
    print('Original Image')
    cv2_imshow(im_in)
    b, g, r = cv.split(im_in)
    print('Blue')
    cv2_imshow(b)
    print('Green')
    cv2_imshow(g)
    print('Red')
    cv2_imshow(r)
    gray = cv.cvtColor(im_in, cv.COLOR_BGR2GRAY)
    print('Grayscale')
    cv2_imshow(gray)
    thresh1 = edgeDetection(b)
    thresh2 = edgeDetection(g)
    thresh3 = edgeDetection(r)
    thresh4 = edgeDetection(gray)
    merge = cv.merge([thresh3, thresh2, thresh1])
    print('Merge')
    cv2_imshow(merge)
    L, C, H = rgb_to_lch(im_in)
    print('L')
    cv2_imshow(L)
    print('C')
    cv2_imshow(C)
    print('H')
    cv2_imshow(H)
    thresh1 = edgeDetection(L)
    thresh2 = edgeDetection(C)
    thresh3 = edgeDetection(H)
```

```
im_in = cv.imread('lena.jpeg')
colorEdgeDetection(im_in)
```