ASSIGNMENT 1

COMP338 – COMPUTER VISION

HARSIMRAN SONDHI

201469795

Table of Contents

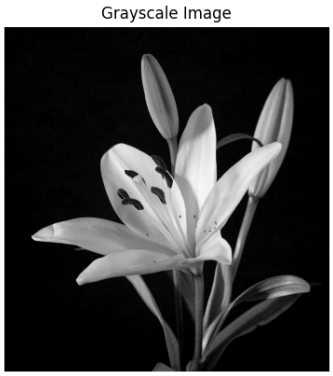
TASK 1 – Canny Edge Detection

Canny Edge Detection

Canny Edge Detection is an image processing algorithm that uses noise reduction, gradient calculation, non-maximum suppression, double threshold processing, and edge tracking, to extract edges from images. It is robust to noise, produces thin and well-defined edges, and offers good localisation, for application in object detection, image segmentation and feature extraction.

1.1. Reading, converting and displaying the greyscale image

```
#Reads image
image_path = '/Users/SimiSondhi/Downloads/grayscale-image.jpg'
image = cv2.imread(image_path)

#Convert to greyscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

#Display the greyscale image
plt.imshow(gray_image, cmap='gray')
plt.title('Grayscale Image')
plt.axis('off')
plt.show()
```



1.2. Noise Reduction

Noise reduction through **Gaussian Blurring** is used to smooth out random variations whilst observing important image details, like edges and textures. This involves using a gaussian kernel, a function that averages the pixel values within a local neighbourhood in the image. The size and deviation of the gaussian kernel determines the degree of smoothing applied to the image, where larger kernels and higher standard deviations result in greater smoothing.

Gaussian Kernel is defined as:

$$f\left(x \mid \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where, *x* and *y* are the pixel coordinates relative to the centre of the kernel and σ is the standard deviation of the Gaussian Distribution, so the smaller the value of sigma, then the steeper the shape, and the larger the value of sigma, then the flatter the shape.

```
#Applying Gaussian Blur
img = image.copy()  #Original image for display
blur = cv2.GaussianBlur(img,(3,3),0)    #Gaussian blur with 3x3 kernal

#Dispalying original and blurred lines
plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(blur),plt.title('Gaussian Blur')
plt.xticks([]), plt.yticks([])
plt.show()
```

Where in *cv2.GaussianBlur(img, (3,3), 0)*:

- *(3,3)*: is the Kernel used to control the area for smoothing.
- *0*: is the standard deviation of the kernel and the value of *0* lets OpenCV automatically calculate based on the kernel size.

1.3. Computing gradients using Laplacian and Sobel Filters

The Laplacian filter detects edges by calculating the second derivative of the image, by identifying regions where the intensity changes rapidly.[1]

The Sobel filter finds the derivative of the image and the filter combines gaussian smoothing and differentiation, so the result is resistant to the noise. [2]

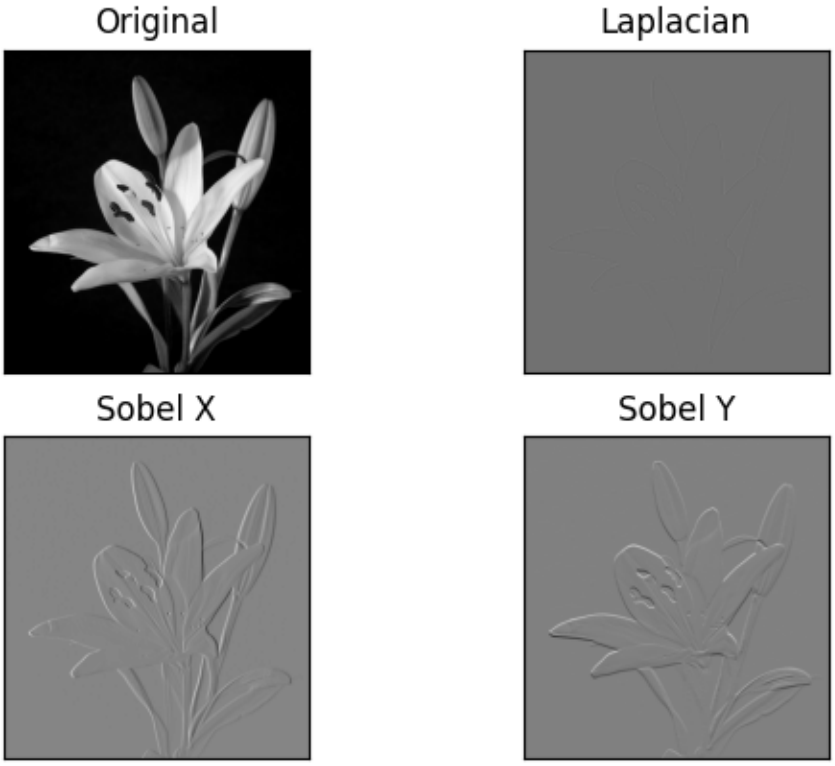
Laplacian filter only gives gradient direction, whereas Sobel Filter gives gradient magnitude and direction hence why I have used the Sobel filter within my code.

Where, *cv2.CV_64F* ensures the output image can store negative values and higher precision.

```
#Computing gradients using lapalacian and Sobel
laplacian = cv2.Laplacian(img,cv2.CV_64F)   #Detect edges by calculating the second derivatives of the image
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)  #Detects edges in the horizontal direction by computing the gradient along the x-axis
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)  #Detects edges in the vertical direction by computing the gradient along the y-axis.

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')    #Displays the original grayscale image
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')   #Displays the Laplacian edge-detected image
plt.title('Laplacian'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray') #Displays the horizontal edge-detected image (Sobel X)
plt.title('Sobel X'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray') #Displays the vertical edge-detected image (Sobel Y)
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])
plt.show()

# Compute gradient magnitude and direction
gradient_magnitude = np.sqrt(sobelx**2 + sobely**2)
gradient_direction = np.arctan2(sobely, sobelx) * (180 / np.pi)  # Converts to degrees
gradient_direction = (gradient_direction + 180) % 180  # ensure graient direction is within range [0, 180] degrees
```

*gradient_magnitude = np.sqrt(sobelx**2 + sobely**2)* calculates the gradient magnitude at each pixel, and the gradient magnitude represents the strength of the edges at each pixel.

*gradient_direction = np.arctan2(sobely, sobelx) * (180 / np.pi)* calculates the gradient direction/angle at each pixel where *np.arctan2(sobely, sobelx)* calculates the angle in radians



1.4 Non-Maximum Suppression

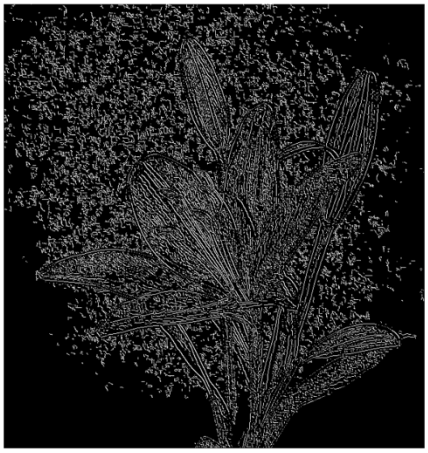
Non-Maximum Suppression refines the edges detected by gradient computation, keeping the most significant edges.[3] This produces thin edges and reduces noise.

```
# Non-Maximum Suppression function
def non_maximum_suppression(magnitude, direction):
    X, Y = magnitude.shape  #where x and y are dimensions of the image
    suppressed = np.zeros((X, Y), dtype=np.float32) #matrix to store output after NMS

    for i in range(1, X-1): #loop to go through each pixel in image
        for j in range(1, Y-1):
            angle = direction[i, j]%100 #gradient direction is normalised to range [0, 180] and % ensures consistency even if angle exceeds 180 degrees

            # Determine neighbors based on gradient direction
            if (0 <= angle < 22.5) or (157.5 <= angle <= 180):  #horizontal direction
                pixels = [magnitude[i, j+1], magnitude[i, j-1]]
            elif 22.5 <= angle < 67.5:  #diagnoal direction
                pixels = [magnitude[i+1, j-1], magnitude[i-1, j+1]]
            elif 67.5 <= angle < 112.5: #vertical direction
                pixels = [magnitude[i+1, j], magnitude[i-1, j]]
            elif 112.5 <= angle < 157.5:    #other direction
                pixels = [magnitude[i-1, j-1], magnitude[i+1, j+1]]

            # suppression: if magnitude is greater or equal to its neighbouring pixels in the gradient direction, it is a local maximum and kept
            # else, pixel is set to 0
            if magnitude[i, j] >= max(pixels):
                suppressed[i, j] = magnitude[i, j]

    return suppressed

# Apply Non-Maximum Suppression
nms_result = non_maximum_suppression(gradient_magnitude, gradient_direction)

# Display the NMS result
plt.imshow(nms_result, cmap='gray')
plt.title('Non-Maximum Suppression')
plt.axis('off')
plt.show()
```

1.5 Double Thresholding

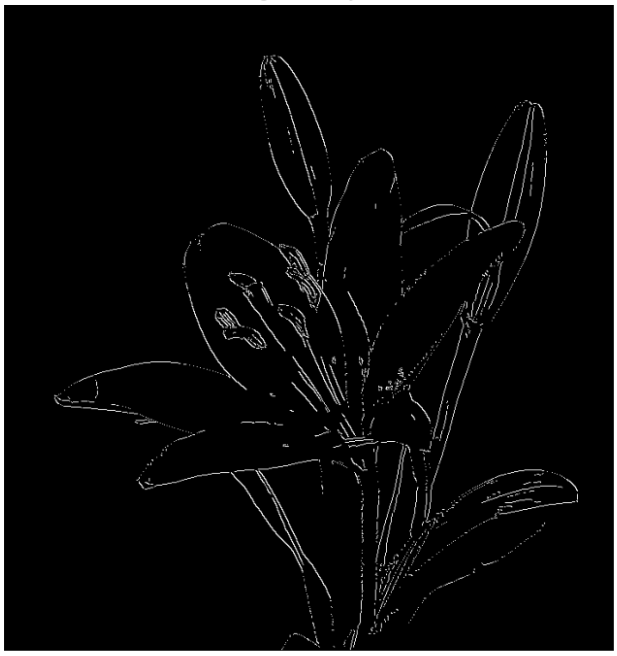
Double thresholding sorts the edges based on their gradient magnitudes (weak or strong) which ensures that only significant edges are kept maintaining edge structure.

Initially the threshold values I started with were Low Threshold = 40 and High Threshold = 100, as they were what I used within the OpenCV's in-built canny implementation. However, I noticed that the output contained significant noise, and numerous weak edges were detected, cluttering the image.



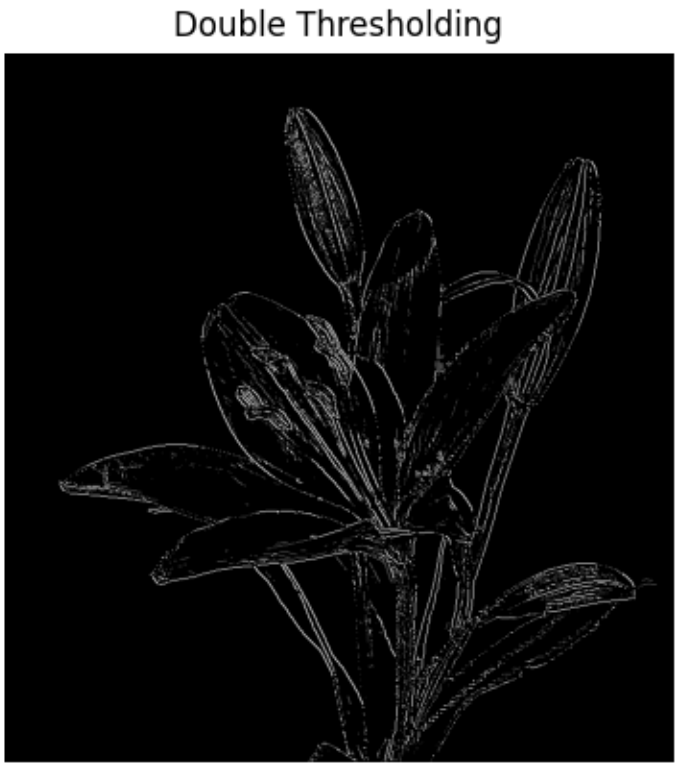
To address the noise problem, I gradually increased both the low and high threshold values to make the algorithm stricter in classifying edges.

I incremented the threshold values too much, to Low Threshold = 1000 and High Threshold = 1500, which resulted in a loss of important edges, over-simplifying my image.



This lead me to find the optimal threshold values of 550 and 1000. (image below)

```
def double_threshold(image, low_threshold, high_threshold):
    strong = 255  # Value for strong edges
    weak = 100    # Value for weak edges

    strong_edges = (image >= high_threshold)    #boolean array where True corresponds to pixels in the image with values greater or equal to strong edges
    weak_edges = (image >= low_threshold) & (image < high_threshold)    #boolean array where True corresponds with values between low and high threshold (weak edges)

    # Create an output image
    thresholded = np.zeros_like(image, dtype=np.uint8)  #creates image with the same shape as input and data type of unit8
    thresholded[strong_edges] = strong #sets pixels in threshold to 255 (strong edge value)
    thresholded[weak_edges] = weak #sets pixels in threshold to 100 (strong edge value)

    #All other pixels remain at 0, for the non-edges
    return thresholded, weak, strong

# Apply Double Thresholding
thresholded_result, weak, strong = double_threshold(nms_result, low_threshold, high_threshold)

# Display the result of Double Thresholding
plt.imshow(thresholded_result, cmap='gray')
plt.title('Double Thresholding')
plt.axis('off')
plt.show()
```



1.6 Edge Tracking by Hysteresis

Edge tracking by hysteresis ensures only the valid edges are preserved by linking the strong and weak edges and subsequently eliminates noise and improves clarity of the edges that are mapped.

```
def edge_tracking_by_hysteresis(thresholded, weak, strong):
    rows, cols = thresholded.shape  #extracts the number of rows and columns in the threshold image
    for i in range(1, rows - 1):    #loop that goes through each pixel in image
        for j in range(1, cols - 1):
            if thresholded[i, j] == weak:   #to focus on pixels with weak edges in threshold image
                # Check 8-connected neighbouring pixels for strong edges
                if ((thresholded[i+1, j-1] == strong) or (thresholded[i+1, j] == strong) or
                    (thresholded[i+1, j+1] == strong) or (thresholded[i, j-1] == strong) or
                    (thresholded[i, j+1] == strong) or (thresholded[i-1, j-1] == strong) or
                    (thresholded[i-1, j] == strong) or (thresholded[i-1, j+1] == strong)):
                    thresholded[i, j] = strong  #if a weak edge is connected to a strong edge, becomes strong edge
                else:
                    thresholded[i, j] = 0  #if weak edge pixel has no strong edge neighbouring pixels, its suppressed so marked as non-edge
    return thresholded

# Apply Edge Tracking by Hysteresis
final_edges = edge_tracking_by_hysteresis(thresholded_result, weak, strong)

# Display the final edge-detected image
plt.imshow(final_edges, cmap='gray')
plt.title('Final Edges after Hysteresis')
plt.axis('off')
plt.show()
```

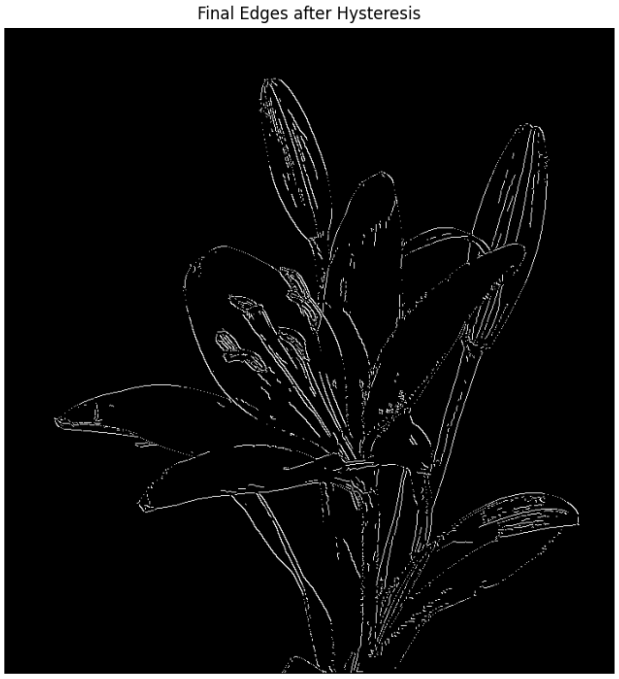


Image output using OpenCV's in-built Canny Function

Implementing the Canny edge detection algorithm without using OpenCV's in-built functions deepened my understanding of each step within the process. During the process of implementing my own edge detection algorithm, I compared my results with those produced by OpenCV's in-built canny function. Despite my implementation of the canny function producing similar results, I know it does not exactly match the output of OpenCV's. The built-in function produced more defined edges that I struggled to r manually replicate.

Ideally, I would like my implementation to produce more defined edges, similar to the output in the OpenCV function. In order to achieve this, initially I applied a stronger Gaussian blur to smooth out small details and noise that could result in minor edges, in order to reduce noise and prevent weak edges from being picked up during the edge detection process. So, I used a larger kernel size for the Gaussian blur, (9, 9), which effectively smoothed out the image, reducing noise significantly. However, I found that I was over-blurring, where it also blurred the edges, leading to less defined edges in the final output, so it was then harder to distinguish finer details. As a result, I then used a smaller kernel size, (3, 3) , because this kept more edge detail while still providing noise reduction.

In addition, to detect broader gradients and reduce sensitivity to small changes, I first used a larger kernel size (ksize=7) in the cv2.Sobel function. However, I found that using a smaller kernel size (ksize=3) in the Sobel operator was more optimal in order to detect the subtle edges that were being missed when using a larger kernel.

The running time for my implementation (7.3071 seconds) is longer than OpenCV's (3.2851 seconds) as, unlike the built-in function, my code displays each step in the edge detection process, and OpenCV's in-built Canny function is highly optimised to perform all steps without the extra overhead, making it more efficient.

Task 2.1

Scale-Invariant Feature Transform (SIFT)

SIFT is an algorithm for detecting and describing local features in images, and is noted for its invariance to scale, rotation, and partial invariance to illumination and affine transformations. SIFT works by detecting key points through a Difference of Gaussians function, in order to identify stable points across different scales in an image, and these key points are then localised, refined, and assigned an orientation based on local image gradients. Then, by analysing the gradients around it, a dimensional descriptor is created for each key point. SIFT highly effective for object recognition and matching tasks however, its computational cost is high which limits how applicable it is in real-time systems.

[5][4]

Speeded-Up Robust Features (SURF)

SURF was designed as a faster alternative to SIFT while maintaining similar robustness, by adding features in order to improve the speed. SURF uses integral images for efficient computation and approximates the Hessian matrix with box filters to quickly detect key points, and for feature description it employs responses to capture local information around each key point, to generate a 64-dimensional or 128-dimensional descriptor. [8] SURF is good at handling images with blurring and rotation, but not good at handling viewpoint change and illumination change. [7] So, as SURF achieves a good balance between speed and performance, it is suitable for real-time applications.

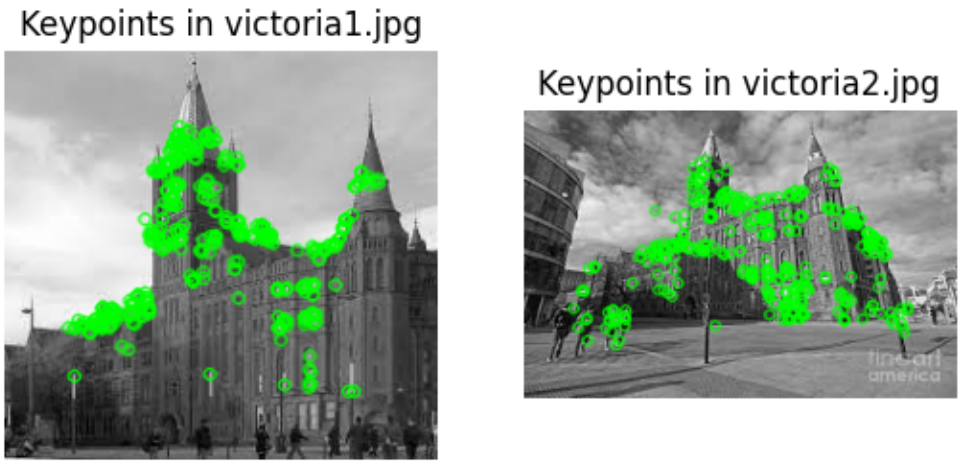
Oriented FAST and Rotated BRIEF (ORB)

ORB is a computationally efficient and open-source alternative to both SIFT and SURF. ORB combines the FAST key point detector to identify strong key points quickly. [9] It then uses the BRIEF descriptor, which is a binary descriptor that provides efficient and compact feature representation. [4] ORB orients the BRIEF descriptor based on the intensity-weighted centroid of the key point's neighbourhood, which ensures there is rotation invariance. ORB is suitable for real-time applications on devices with limited resources, as it has low computational cost and high speed.

Task 2.2

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt

img1 = cv.imread('/Users/SimiSondhi/Downloads/Images for task2 (1) 2/victoria1.jpg', cv.IMREAD_GRAYSCALE)
img2 = cv.imread('/Users/SimiSondhi/Downloads/Images for task2 (1) 2/victoria2.jpg', cv.IMREAD_GRAYSCALE)

# Initiate ORB detector
orb = cv.ORB_create()

# find the keypoints with ORB
kp1 = orb.detect(img1,None)
kp2 = orb.detect(img2,None)

# compute the descriptors with ORB
kp1, des1 = orb.compute(img1, kp1)
kp2, des2 = orb.compute(img2, kp2)

# draw only keypoints location,not size and orientation
img1_kp = cv.drawKeypoints(img1, kp1, None, color=(0,255,0), flags=0)
img2_kp = cv.drawKeypoints(img2, kp2, None, color=(0,255,0), flags=0)

plt.subplot(1, 2, 1)
plt.title('Keypoints in victoria1.jpg')
plt.imshow(img1_kp, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('Keypoints in victoria2.jpg')
plt.imshow(img2_kp, cmap='gray')
plt.axis('off')

plt.show()
```

[Utilised code provided by OpenCV tutorial – 11]

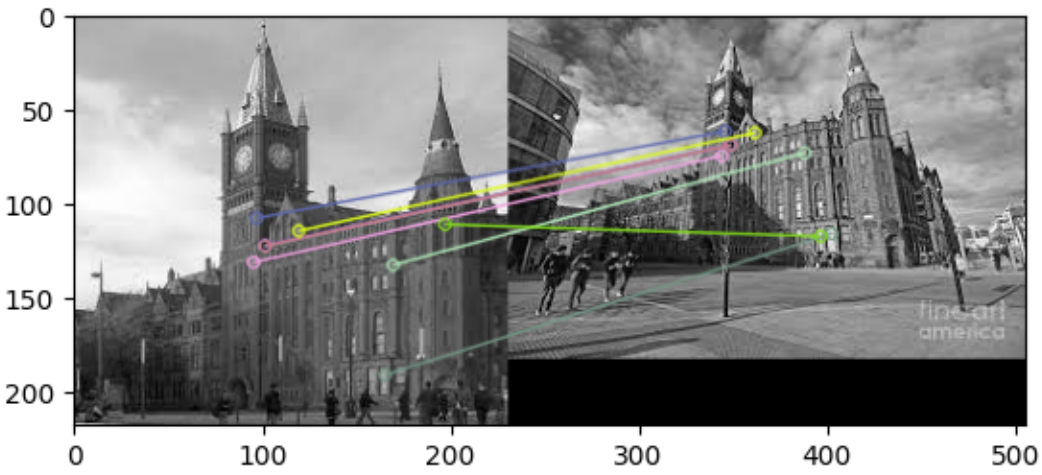


Task 2.3

Using SIFT

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

# Load images
img1 = cv.imread('/Users/SimiSondhi/Downloads/Images for task2 (1) 2/victoria1.jpg', cv.IMREAD_GRAYSCALE)
img2 = cv.imread('/Users/SimiSondhi/Downloads/Images for task2 (1) 2/victoria2.jpg', cv.IMREAD_GRAYSCALE)


# Initiate SIFT detector
sift = cv.SIFT_create()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# BFMatcher with default params
bf = cv.BFMatcher()
matches = bf.knnMatch(des1,des2,k=2)

# Apply ratio test
good = []
for m,n in matches:
    if m.distance < 0.79*n.distance:
        good.append([m])

# cv.drawMatchesKnn expects list of lists as matches.
img3 = cv.drawMatchesKnn(img1,kp1,img2,kp2,good,None,flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

plt.imshow(img3),plt.show()
```

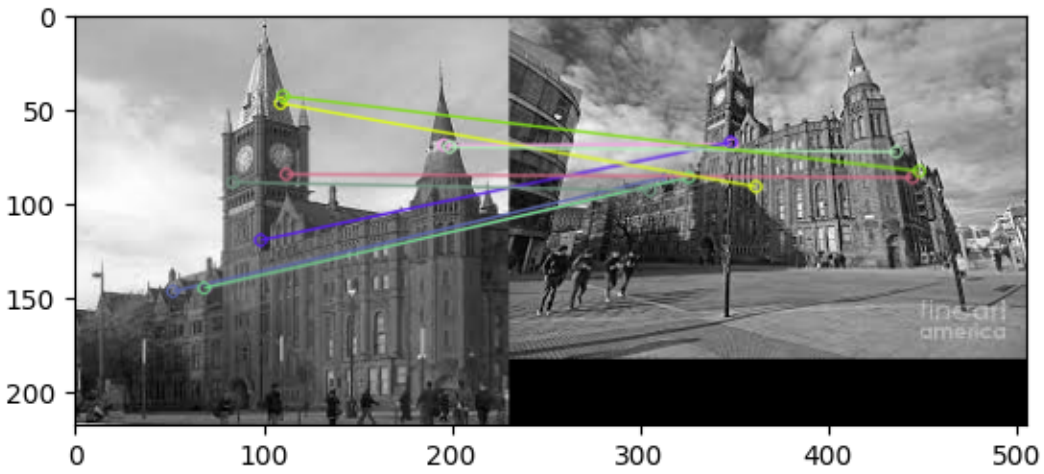
[Utilised code provided by OpenCV tutorial – 10]



Using ORB

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

# Load images
img1 = cv.imread('/Users/SimiSondhi/Downloads/Images for task2 (1) 2/victoria1.jpg', cv.IMREAD_GRAYSCALE)
img2 = cv.imread('/Users/SimiSondhi/Downloads/Images for task2 (1) 2/victoria2.jpg', cv.IMREAD_GRAYSCALE)


# Initiate ORB detector
orb = cv.ORB_create()

# find the keypoints and descriptors with ORB
kp1, des1 = orb.detectAndCompute(img1,None)
kp2, des2 = orb.detectAndCompute(img2,None)
# create BFMatcher object
bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)

# Match descriptors.
matches = bf.match(des1,des2)

# Sort them in the order of their distance.
matches = sorted(matches, key = lambda x:x.distance)

# Draw first 10 matches.
img3 = cv.drawMatches(img1,kp1,img2,kp2,matches[:10],None,flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

plt.imshow(img3),plt.show()
```

[Utilised code provided by OpenCV tutorial – 10]



SIFT vs. ORB

Both SIFT and ORB extracted 7 descriptors each however, SIFT only had 1 mismatch, whilst ORB had 4 mismatches. This indicates that SIFT provided more accurate key point matches, which is most likely because of its robustness against scale, rotation, and illumination changes. In addition, ORB, despite supposedly being faster and more computationally efficient, struggled with accurate feature matching in this case, resulting in a higher rate of mismatches. [4] This aligns with the broader understanding in computer vision literature that SIFT generally excels in environments with complex transformations or lighting variations, while ORB prioritises speed over precision [4][5]. Therefore, SIFT is the better-performing method for the two images due to its higher accuracy in key point matching.

References

[1] docs.opencv.org. (n.d.). *OpenCV: Laplace Operator*. [online] Available at: https://docs.opencv.org/3.4/d5/db5/tutorial_laplace_operator.html.

[2] Opencv.org. (2024). *OpenCV: Image Filtering*. [online] Available at: https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#gacea54f142e81b6758cb6f375ce782c8d [Accessed 15 Nov. 2024].

[3] OpenCV (n.d.). *OpenCV: Canny Edge Detection*. [online] docs.opencv.org. Available at: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html.

[4] Rublee, E., Rabaud, V., Konolige, K. and Bradski, G. (2011). *ORB: An efficient alternative to SIFT or SURF*. [online] IEEE Xplore. doi:https://doi.org/10.1109/ICCV.2011.6126544.

[5] Lowe, D.G. (2004). Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2), pp.91–110. doi:https://doi.org/10.1023/b:visi.0000029664.99615.94.

[6] Bay, H., Tuytelaars, T. and Luc Van Gool (2006). SURF: Speeded Up Robust Features. In: *Computer Vision – ECCV 2006*. [online] pp.404–417. doi:https://doi.org/10.1007/11744023_32.

[7] docs.opencv.org. (n.d.). *OpenCV: Introduction to SURF (Speeded-Up Robust Features)*. [online] Available at: https://docs.opencv.org/4.x/df/dd2/tutorial_py_surf_intro.html.

[8] Bay, H., Tuytelaars, T. and Luc Van Gool (2006). SURF: Speeded Up Robust Features. In: *Computer Vision – ECCV 2006*. [online] pp.404–417. doi:https://doi.org/10.1007/11744023_32.

[9] OpenCV (n.d.). *OpenCV: ORB (Oriented FAST and Rotated BRIEF)*. [online] docs.opencv.org. Available at: https://docs.opencv.org/4.x/d1/d89/tutorial_py_orb.html.

[10] Opencv.org. (2024). *OpenCV: Feature Matching*. [online] Available at: https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html#autotoc_md1240 [Accessed 13 Nov. 2024].

[11] Opencv.org. (2024). *OpenCV: ORB (Oriented FAST and Rotated BRIEF)*. [online] Available at: https://docs.opencv.org/4.x/d1/d89/tutorial_py_orb.html#autotoc_md1244 [Accessed 13 Nov. 2024].