Proposed solutions for LABexc6-ELE510-2023

October 18, 2023

1 ELE510 Image Processing with robot vision: LAB, Exercise 6, Image features detection.

Purpose: *To learn about the edges and corners features detection, and their descriptors.*

The theory for this exercise can be found in chapter 7 of the text book [1] and in appendix C in the compendium [2]. See also the following documentations for help: - OpenCV - numpy - matplotlib - scipy

IMPORTANT: Read the text carefully before starting the work. In many cases it is necessary to do some preparations before you start the work on the computer. Read necessary theory and answer the theoretical part frst. The theoretical and experimental part should be solved individually. The notebook must be approved by the lecturer or his assistant.

Approval:

The current notebook should be submitted on CANVAS as a single pdf file.

`To export the notebook in a pdf format, goes to File -> Download as -> PDF via LaTeX (.pdf).`

Note regarding the notebook: The theoretical questions can be answered directly on the notebook using a *Markdown* cell and LaTex commands (if relevant). In alternative, you can attach a scan (or an image) of the answer directly in the cell.

Possible ways to insert an image in the markdown cell:

```
![image name]("image_path")
```

```
<img src="image_path" alt="Alt text" title="Title text" />
```

Under you will find parts of the solution that is already programmed.

```
<p>You have to fill out code everywhere it is indicated with `...`</p>
<p>The code section under `######## a)` is answering subproblem a) etc.</p>
```

1.1 Problem 1

Intensity edges are pixels in the image where the intensity (or graylevel) function changes rapidly.

The **Canny edge detector** is a classic algorithm for detecting intensity edges in a grayscale image that relies on the gradient magnitude. The algorithm was developed by John F. Canny in 1986. It is a multi-stage algorithm that provides good and reliable detection.

a) Create the **Canny algorithm**, described at pag. 336 (alg. 7.1). For the last step (`EDGELINKING`) you can either use the algorithm 7.3 at page 338 or the `HYSTERESIS THRESHOLD` algorithm 10.3 described at page 451. All the following images are taken from the text book [1].

Remember:

- Sigma (second parameter in the Canny algorithm) is not necessary for the calculation since the Sobel operator (in opencv) combines the Gaussian smoothing and differentiation, so the results is nore or less resistant to the noise.
- We are defining the low and high thresholds manually in order to have a better comparison with the predefined opencv function. It is possible to extract the low and high thresholds automatically from the image but it is not required in this problem.

b) Test your algorithm with a image of your choice and compare your results with the predefined function in opencv:

`cv2.Canny(img, t_low, t_high, L2gradient=True)`

Documentation.

1.1.1 P.S. :

The goal of this problem it is not to create a **perfect** replication of the algorithm in opencv, but to understand the various steps involved and to be able to extract the edges from an ima ge using these steps.

[1]:
```
def computeImageGradient(Im):
    # Sobel operator  to find the first derivate in the horizontal and vertical␣
 ↪directions
    g_x = cv2.Sobel(Im, ddepth=cv2.CV_32F, dx=1, dy=0, ksize=3)
    g_y = cv2.Sobel(Im, ddepth=cv2.CV_32F, dx=0, dy=1, ksize=3)

    ##############################
    # Calculate the magnitude and the gradient direction like it is performed␣
 ↪during the assignment 4 (problem 2a)
    G_mag = np.hypot(g_x,g_y)
    G_phase = np.arctan2(g_y, g_x)

    return G_mag, G_phase
```

[2]:
```
# NonMaxSuppression algorithm
def nonMaxSuppression(G_mag, G_phase):
    G_localmax = np.zeros((G_mag.shape))

    # For each pixel, adjust the phase to ensure that -pi/8 <= theta < 7*pi/8

    # Here x represents the columns and y represent the rows!
    for y in range(1,G_mag.shape[0]-1):
        for x in range(1,G_mag.shape[1]-1):
            theta = G_phase[y][x]
```

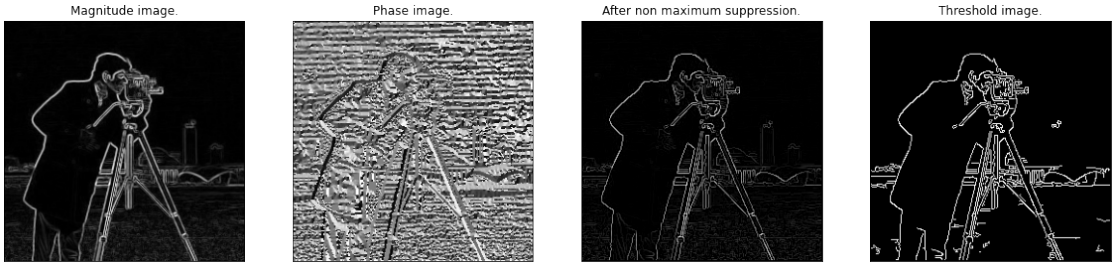
```
            neigh_1 = -1; neigh_2 = -1
            if theta >= 7*np.pi/8: theta -= np.pi
            if theta < -np.pi/8: theta += np.pi

            if -np.pi/8 <= theta < np.pi/8: neigh_1 = G_mag[y][x-1]; neigh_2 =␣
↪G_mag[y][x+1]
            elif np.pi/8 <= theta < 3*np.pi/8: neigh_1 = G_mag[y-1][x-1];␣
↪neigh_2 = G_mag[y+1][x+1]
            elif 3*np.pi/8 <= theta < 5*np.pi/8: neigh_1 = G_mag[y-1][x];␣
↪neigh_2 = G_mag[y+1][x]
            elif 5*np.pi/8 <= theta < 7*np.pi/8: neigh_1 = G_mag[y+1][x-1];␣
↪neigh_2 = G_mag[y-1][x+1]

            # If the pixel isx a local maximum in the direction of the␣
↪gradient, then retain the value, otherwise set it to 0
            if G_mag[y][x] >= neigh_1 and G_mag[y][x] >= neigh_2:␣
↪G_localmax[y][x] = G_mag[y][x]
            else: G_localmax[y][x] = 0

    return G_localmax
```

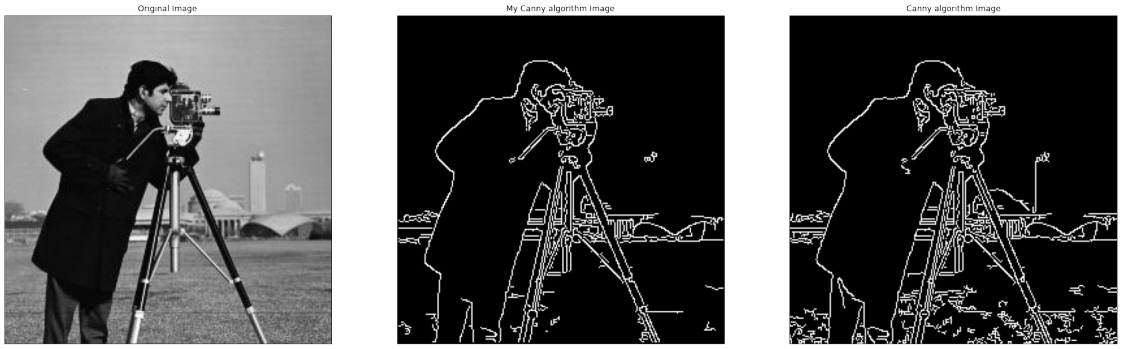
[3]:
```
# Edge linking step with low and high thresholds
def edgeLinking(G_localmax, t_low, t_high):
    I_edges = np.zeros((G_localmax.shape))
    frontier = []

    for x in range(1,G_localmax.shape[0]-1):
        for y in range(1,G_localmax.shape[1]-1):
            if G_localmax[x,y]>t_high:
                frontier.append((x,y))
                I_edges[x,y] = 1

    while len(frontier)>0:
        p = frontier.pop()
        p_x, p_y = p[0], p[1]
        neighbors = [(p_x+1,p_y-1),(p_x+1,p_y),(p_x+1,p_y+1),
                     ␣
↪(p_x,p_y-1),(p_x,p_y+1),(p_x-1,p_y-1),(p_x-1,p_y),(p_x-1,p_y+1)]

        for (i,j) in neighbors:
            if G_localmax[i,j]>t_low and I_edges[i,j]!=1:
                frontier.append((i,j))
                I_edges[i,j] = 1

    return I_edges
```

[6]:
```
"""
Function that performs the Canny algorithm
Input:
    - Im: image in grayscale
    - t_low: first threshold for the hysteresis procedure (edge linking)
    - t_high: second threshold for the hysteresis procedure (edge linking)
"""
def my_cannyAlgorithm(Im, t_low, t_high):
    ## Compute the image gradient
    G_mag, G_phase = computeImageGradient(Im)

    ## NonMaxSuppression algorithm
    G_localmax = nonMaxSuppression(G_mag, G_phase)

    ## Edge linking
    if t_low>t_high: t_low, t_high = t_high, t_low
    I_edges = edgeLinking(G_localmax, t_low, t_high)

    plt.figure(figsize=(20,20))
    plt.subplot(141), plt.imshow(G_mag, cmap='gray')
    plt.title('Magnitude image.'), plt.xticks([]), plt.yticks([])
    plt.subplot(142), plt.imshow(G_phase, cmap='gray')
    plt.title('Phase image.'), plt.xticks([]), plt.yticks([])
    plt.subplot(143), plt.imshow(G_localmax, cmap='gray')
    plt.title('After non maximum suppression.'), plt.xticks([]), plt.yticks([])
    plt.subplot(144), plt.imshow(I_edges, cmap='gray')
    plt.title('Threshold image.'), plt.xticks([]), plt.yticks([])
    plt.show()

    return I_edges
```

[7]:
```
import cv2, os
import numpy as np
import matplotlib.pyplot as plt

Im = cv2.imread(os.path.join('./images/cameraman.jpg'), cv2.IMREAD_GRAYSCALE)

t_low = 100
t_high = 250
I_edges = my_cannyAlgorithm(Im, t_low, t_high)

plt.figure(figsize=(30,10))
plt.subplot(131), plt.imshow(Im, cmap='gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(132), plt.imshow(I_edges, cmap='gray')
plt.title('My Canny algorithm Image'), plt.xticks([]), plt.yticks([])
```

```
plt.subplot(133), plt.imshow(cv2.Canny(Im,t_low, t_high, L2gradient=False),␣
 ↪cmap='gray')
plt.title('Canny algorithm Image'), plt.xticks([]), plt.yticks([])
plt.show()
```





2 Problem 2

One of the most popular approaches to feature detection is the **Harris corner detector**, after a work of Chris Harris and Mike Stephens from 1988.

a) Use the function in opencv `cv2.cornerHarris(...)` (Documentation) with `blockSize=3, ksize=3, k=0.04` with the **./images/chessboard.png** image to detect the corners (you can find the image on CANVAS).

b) Plot the image with the detected corners found.

Hint: Use the function `cv2.drawMarker(...)` (Documentation) to show the corners in the image.

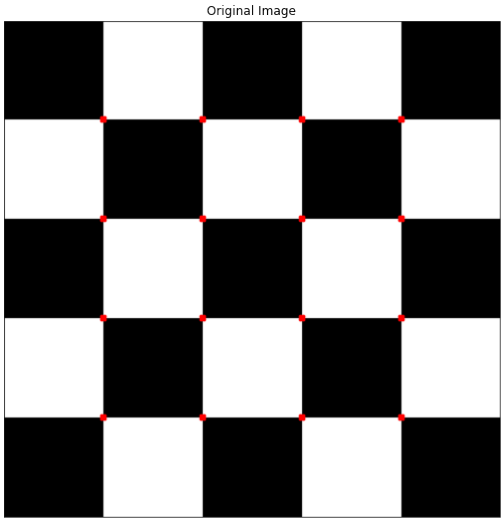
c) Detect the corners using the images **./images/arrow_1.jpg**, **./images/arrow_2.jpg** and **./images/arrow_3.jpg**; describe and compare the results in the three images.

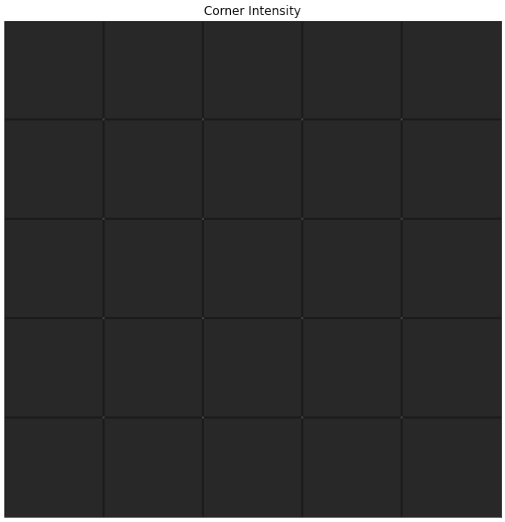
d) What happen if you change (increase/decrease) the `k` constant for the "corner points"?

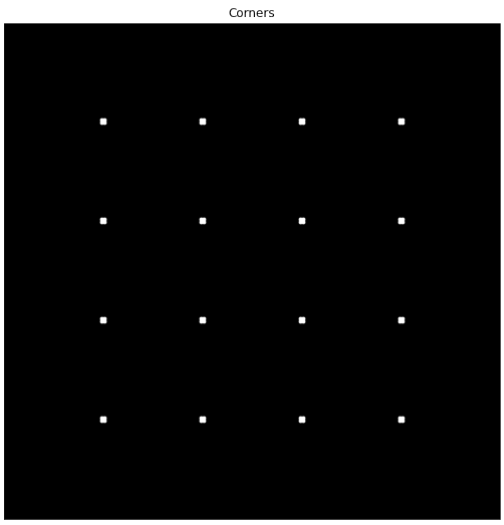
[8]:
```
def getCornersFromHarris(Im, blockSize, ksize, k, markerSize, thickness):
    # Convert the image from color to grayscale
```

```
    if len(Im.shape)>2: gray_img = cv2.cvtColor(Im, cv2.COLOR_BGR2GRAY)
    else: gray_img = Im
    dst = cv2.cornerHarris(gray_img, dst=cv2.CV_32F, blockSize=blockSize,␣
 ↪ksize=ksize, k=k)

    # Normalize the image
    dst_norm = np.zeros(dst.shape, dtype=np.float32)
    cv2.normalize(dst, dst_norm, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX)

    dst = cv2.dilate(dst,None)
    thres = 0.05*dst.max()

    #Drawing a cross around corners
    corners = np.zeros(gray_img.shape, dtype=np.float32)

    for x in range(dst_norm.shape[0]):
        for y in range(dst_norm.shape[1]):
            if dst[x,y] > thres:
                cv2.drawMarker(Im, (y,x), color=[255,0,0], markerType=cv2.
 ↪MARKER_TILTED_CROSS, markerSize=markerSize, thickness=thickness)
                cv2.drawMarker(corners, (y,x), color=255, markerType=cv2.
 ↪MARKER_TILTED_CROSS, markerSize=markerSize, thickness=thickness)

    return dst_norm, corners
```

[9]:
```
################
# Answer for 2a and 2b

Im = cv2.imread(os.path.join('./images/chessboard.png'))

dst_norm, corners = getCornersFromHarris(Im, blockSize=3, ksize=3, k=0.04,␣
 ↪markerSize=10, thickness=10)

plt.figure(figsize=(20,20))
plt.subplot(221), plt.imshow(Im)
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(222), plt.imshow(dst_norm,cmap="gray", vmin=0, vmax=255)
plt.title('Corner Intensity'), plt.xticks([]), plt.yticks([])
plt.subplot(223), plt.imshow(corners,cmap="gray", vmin=0, vmax=255)
plt.title('Corners'), plt.xticks([]), plt.yticks([])
plt.show()
```

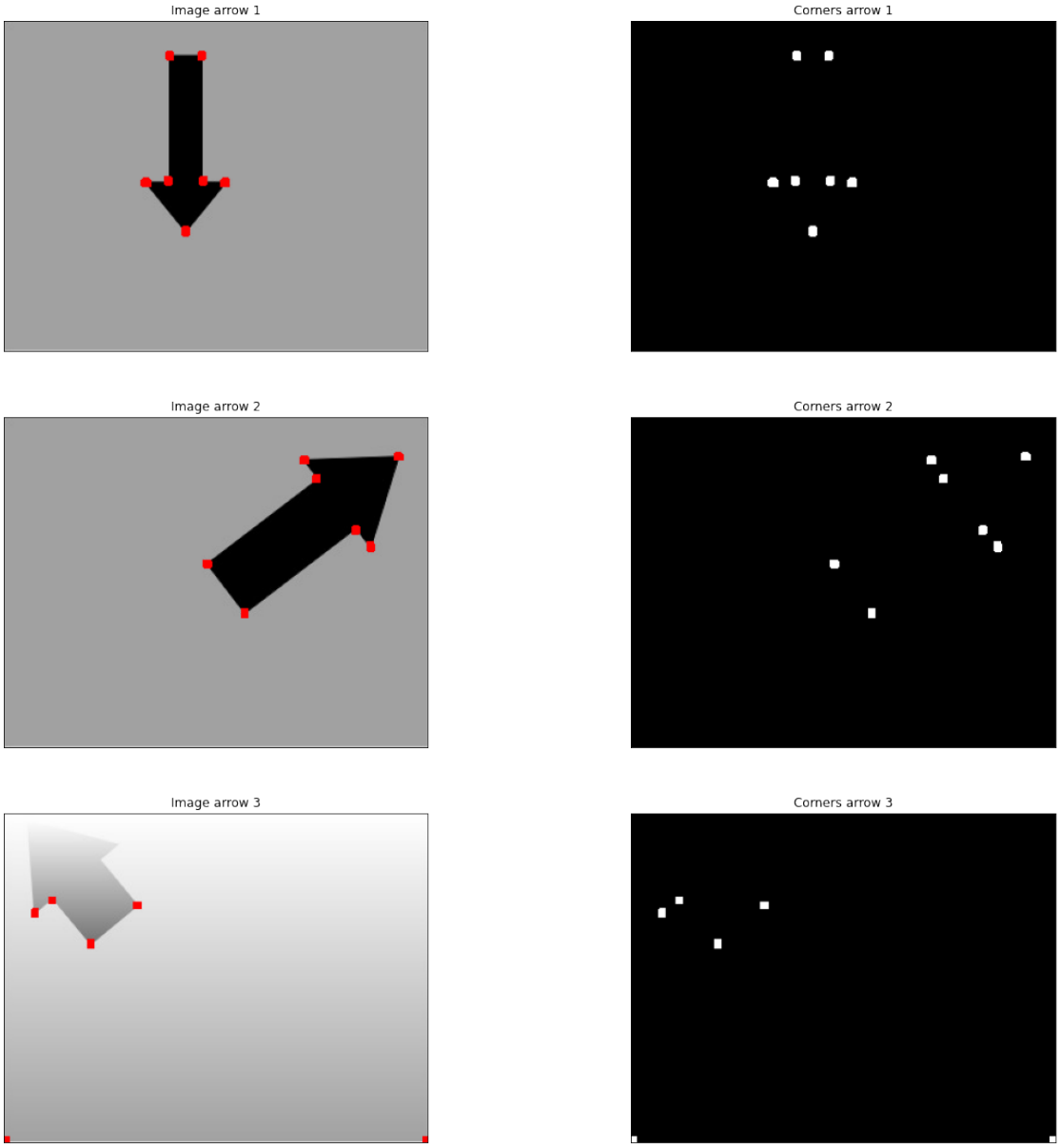






[10]:
```
Im_1 = cv2.imread(os.path.join('./images/arrow_1.jpg'))
Im_2 = cv2.imread(os.path.join('./images/arrow_2.jpg'))
Im_3 = cv2.imread(os.path.join('./images/arrow_3.jpg'))

dst_norm_1, corners_1 = getCornersFromHarris(Im_1, blockSize=3, ksize=3, k=0.
 ↪04, markerSize=2, thickness=1)
dst_norm_2, corners_2 = getCornersFromHarris(Im_2, blockSize=3, ksize=3, k=0.
 ↪04, markerSize=2, thickness=1)
dst_norm_3, corners_3 = getCornersFromHarris(Im_3, blockSize=3, ksize=3, k=0.
 ↪04, markerSize=2, thickness=1)

plt.figure(figsize=(20,20))
```

```
plt.subplot(321), plt.imshow(Im_1,cmap="gray")
plt.title('Image arrow 1'), plt.xticks([]), plt.yticks([])
plt.subplot(322), plt.imshow(corners_1,cmap="gray")
plt.title('Corners arrow 1'), plt.xticks([]), plt.yticks([])
plt.subplot(323), plt.imshow(Im_2,cmap="gray")
plt.title('Image arrow 2'), plt.xticks([]), plt.yticks([])
plt.subplot(324), plt.imshow(corners_2,cmap="gray")
plt.title('Corners arrow 2'), plt.xticks([]), plt.yticks([])
plt.subplot(325), plt.imshow(Im_3,cmap="gray")
plt.title('Image arrow 3'), plt.xticks([]), plt.yticks([])
plt.subplot(326), plt.imshow(corners_3,cmap="gray")
plt.title('Corners arrow 3'), plt.xticks([]), plt.yticks([])
plt.show()
```



3 Problem 3

a) What is the SIFT approach? Describe the steps involved.

The SIFT (Scale Invariant Feature Transform) is a feature detector algorithm, which consists of ... steps: - the first step is to fine the gaussian scale space. Thereafter this is used to build a Laplacian pyramid of the image with 4 images pr octace; - then, for each pixel, for two scales pr octave, The algorithm search for local maxima among its 26 neighborsof the laplacian pyramid(8 in the same image, 9 at the next smallest scale, and 9 at the next largest scale); Two scales pr octave is used. This gives the candidate key points.
- a final step discards candidate points in untextured areas and along intensity edges. This is done in a way similar to the Harris corner deterction, but only on the candidate points are considered, not on all the pixels of the image.

b) Why this approach is more popular than the Harris detector?

The popularity of the SIFT feature detector is due to its leveraging of scale space to find features regardless of their scale in the image. The Harris detector is NOT scale invariant, and a corner might look like edge if the resolution is increased. However, since SIFT utelize the scale space and look for corner points over many scales, it is invariant to the scale.

c) Explain the difference between a feature detector and a feature descriptor.

A feature detector returns locations in the image where there is sufficient texture in all directions for the location to be of potential interest. In other words it gives corner/feature/ points also called key points.

A feature descriptor is a vector associated with a corner/feature key point. First a feature detector is used, thereafter a descriptor is used to describe that area around the feature point in the image so that this can be used for matching, calibration, stitching etc.

3.1 Contact

3.1.1 Course teacher

Professor Kjersti Engan, room E-431, E-mail: kjersti.engan@uis.no

3.1.2 Teaching assistant

Saul Fuster Navarro, room E-401 E-mail: saul.fusternavarro@uis.no

Jorge Garcia Torres Fernandez, room E-401 E-mail: jorge.garcia-torres@uis.no

3.2 References

[1] S. Birchfeld, Image Processing and Analysis. Cengage Learning, 2016.

[2] I. Austvoll, "Machine/robot vision part I," University of Stavanger, 2018. Compendium, CANVAS.