Proposed solutions for LABexc4-ELE510-2023

October 6, 2023

1 ELE510 Image Processing with robot vision: LAB, Exercise 4, Spatial-domain filtering

Purpose: *To learn about Linear Filters and Local Image Features and its use in computer vision (image processing). Some basic experiments will be implemented using Pyhon, OpenCV and other packages.*

The theory for this exercise can be found in chapter 5 of the text book [1]. 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 first. 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

In this problem we want to get a better understanding of linear filtering using convolution.

The computations should be done first by hand on paper (attached a picture for your solution). Thereafter, check the results on the notebook with the pre-built functions.

Sobel and **Prewitt** masks are used to compute the two components of the gradient. They perform differentiation over a 3 pixel region in the horizontal (x) and vertical (y) direction respectively and smooth by a 3 pixel smoothing filter in the other direction. The masks represent separable 2D filters and can thereby be separated in a differentiation filter and a smoothing filter.

The **Sobel masks**:

$$\mathbf{h}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \mathbf{h}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \tag{1}$$

The **Prewitt masks**:

$$\mathbf{h}_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \qquad \mathbf{h}_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}. \tag{2}$$

a) Find the 1D **differentiation filter** and the 1D **smoothing filter** for the Sobel and Prewitt masks. The result will be similar for the x- and y-direction. It is therefore sufficient to find the result for one of the directions, e.g. the x-direction.

Consider the following image:

$$\mathbf{I}m = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{3}$$

Prewitt masks is obtained convolving a 1D Gaussian derivative kernel with a 1D box filter in the orthogonal direction:

$$Prewitt_x = \frac{1}{3} \underbrace{\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}}_{smoothing\,filter} \circledast \frac{1}{2} \underbrace{\begin{pmatrix} 1 & 0 & -1 \end{pmatrix}}_{derivative\,filter}$$

$$Prewitt_y = \frac{1}{3} \underbrace{\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}}_{smoothing\,filter} \circledast \frac{1}{2} \underbrace{\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}}_{derivative\,filter}$$

Sobel operator uses the Gaussian $\sigma^2 = 0.5$ for the smoothing kernel:

$$Sobel_x = \frac{1}{4} \underbrace{\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}}_{smoothing\,filter} \circledast \frac{1}{2} \underbrace{\begin{pmatrix} 1 & 0 & -1 \end{pmatrix}}_{derivative\,filter}$$

$$Sobel_y = \frac{1}{4} \underbrace{\begin{pmatrix} 1 & 2 & 1 \end{pmatrix}}_{smoothing\ filter} \circledast \frac{1}{2} \underbrace{\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}}_{derivative\ filter}$$

b) Filter this image using the **Prewitt** masks. Find the two output images, representing the differential along the horizontal and vertical directions.

$$G_x = Prewitt_x \circledast I(x, y)$$

With convolution, flipping the kernel in both axes is required:

https://stackoverflow.com/questions/45152473/why-is-the-convolutional-filter-flipped-in-convolutional-neural-networks http://195.134.76.37/applets/AppletConvol/Appl_Convol2.html

[1]:
```
from scipy import signal
import numpy as np
import matplotlib.pyplot as plt
import cv2

img = np.array(
    [[0,0,0,0,0,0],
     [0,0,1,1,0,0],
     [0,1,0,1,1,0],
     [0,1,1,0,1,0],
     [0,0,1,1,0,0],
     [0,0,0,0,0,0]], dtype=np.float64)

h_p_1x = np.array([1,0,-1])
h_p_2x = np.array([1,1,1])

print("\n Using OpenCV (we need to flip the 1D kernels) \n")
print("Convolve the Image with Prewitt x")
print(cv2.sepFilter2D(src=img, ddepth=-1, kernelX=h_p_1x[::-1], kernelY=h_p_2x[:
 ↪:-1], borderType=cv2.BORDER_REPLICATE))

print("Convolve the Image with Prewitt y")
print(cv2.sepFilter2D(src=img, ddepth=-1, kernelX=h_p_2x[::-1], kernelY=h_p_1x[:
 ↪:-1], borderType=cv2.BORDER_REPLICATE))

print("\n Using scipy (no need to flip the kernels) \n")
print("Convolve the Image with Prewitt x")
print(signal.sepfir2d(img, h_p_1x, h_p_2x))

print("Convolve the Image with Prewitt y")
print(signal.sepfir2d(img, h_p_2x, h_p_1x))
```

```
 Using OpenCV (we need to flip the 1D kernels)

Convolve the Image with Prewitt x
[[ 0.  1.  1. -1. -1.  0.]
 [ 1.  1.  1.  0. -2. -1.]
 [ 2.  2.  0.  0. -2. -2.]
 [ 2.  2.  0.  0. -2. -2.]
 [ 1.  2.  0. -1. -1. -1.]
 [ 0.  1.  1. -1. -1.  0.]]
Convolve the Image with Prewitt y
[[ 0.  1.  2.  2.  1.  0.]
 [ 1.  1.  2.  2.  2.  1.]
 [ 1.  1.  0.  0.  0.  1.]
 [-1.  0.  0.  0. -1. -1.]
 [-1. -2. -2. -2. -1. -1.]
 [ 0. -1. -2. -2. -1.  0.]]

 Using scipy (no need to flip the kernels)

Convolve the Image with Prewitt x
[[ 0.  1.  1. -1. -1.  0.]
 [ 1.  1.  1.  0. -2. -1.]
 [ 2.  2.  0.  0. -2. -2.]
 [ 2.  2.  0.  0. -2. -2.]
 [ 1.  2.  0. -1. -1. -1.]
 [ 0.  1.  1. -1. -1.  0.]]
Convolve the Image with Prewitt y
[[ 0.  1.  2.  2.  1.  0.]
 [ 1.  1.  2.  2.  2.  1.]
 [ 1.  1.  0.  0.  0.  1.]
 [-1.  0.  0.  0. -1. -1.]
 [-1. -2. -2. -2. -1. -1.]
 [ 0. -1. -2. -2. -1.  0.]]
```

c) Filter this image using the **Sobel** masks. Find the two output images, representing the differential along the horizontal and vertical directions.

[2]:
```
h_s_1x = np.array([1,0,-1])
h_s_2x = np.array([1,2,1])

print("\n Using OpenCV (we need to flip the 1D kernels) \n")
print("Convolve the Image with Sobel x")
print(cv2.sepFilter2D(src=img, ddepth=-1, kernelX=h_s_1x[::-1], kernelY=h_s_2x[:
  ↪:-1], borderType=cv2.BORDER_ISOLATED))

print("Convolve the Image with Sobel y")
```

```
print(cv2.sepFilter2D(src=img, ddepth=-1, kernelX=h_s_2x[::-1], kernelY=h_s_1x[:
  ↪:-1], borderType=cv2.BORDER_ISOLATED))

print("\n Using scipy (no need to flip the kernels) \n")
print("Convolve the Image with Sobel x")
print(signal.sepfir2d(img, h_s_1x, h_s_2x))

print("Convolve the Image with Sobel y")
print(signal.sepfir2d(img, h_s_2x, h_s_1x))
```

```
 Using OpenCV (we need to flip the 1D kernels)

Convolve the Image with Sobel x
[[ 0.  1.  1. -1. -1.  0.]
 [ 1.  2.  2. -1. -3. -1.]
 [ 3.  2.  0.  1. -3. -3.]
 [ 3.  3. -1.  0. -2. -3.]
 [ 1.  3.  1. -2. -2. -1.]
 [ 0.  1.  1. -1. -1.  0.]]
Convolve the Image with Sobel y
[[ 0.  1.  3.  3.  1.  0.]
 [ 1.  2.  2.  3.  3.  1.]
 [ 1.  2.  0. -1.  1.  1.]
 [-1. -1.  1.  0. -2. -1.]
 [-1. -3. -3. -2. -2. -1.]
 [ 0. -1. -3. -3. -1.  0.]]

 Using scipy (no need to flip the kernels)

Convolve the Image with Sobel x
[[ 0.  1.  1. -1. -1.  0.]
 [ 1.  2.  2. -1. -3. -1.]
 [ 3.  2.  0.  1. -3. -3.]
 [ 3.  3. -1.  0. -2. -3.]
 [ 1.  3.  1. -2. -2. -1.]
 [ 0.  1.  1. -1. -1.  0.]]
Convolve the Image with Sobel y
[[ 0.  1.  3.  3.  1.  0.]
 [ 1.  2.  2.  3.  3.  1.]
 [ 1.  2.  0. -1.  1.  1.]
 [-1. -1.  1.  0. -2. -1.]
 [-1. -3. -3. -2. -2. -1.]
 [ 0. -1. -3. -3. -1.  0.]]
```

d) Compute the gradient, $|\nabla I| = \|\nabla I\| = \sqrt{I_x^2(m,n) + I_y^2(m,n)}$, images based on the **Prewitt** and **Sobel** masks.

[3]:
```
def gradient(I_x, I_y):

    grad = np.round(np.hypot(I_x, I_y, dtype="float32"),2)

    return grad

Gp_x = signal.sepfir2d(img, h_p_1x, h_p_2x)
Gp_y = signal.sepfir2d(img, h_p_2x, h_p_1x)

Gs_x = signal.sepfir2d(img, h_s_1x, h_s_2x)
Gs_y = signal.sepfir2d(img, h_s_2x, h_s_1x)

print("Gradient based on Prewitt: \n", gradient(Gp_x, Gp_y))
print("\nGradient based on Sobel: \n", gradient(Gs_x, Gs_y))
```

```
Gradient based on Prewitt:
 [[0.   1.41 2.24 2.24 1.41 0.  ]
 [1.41 1.41 2.24 2.   2.83 1.41]
 [2.24 2.24 0.   0.   2.   2.24]
 [2.24 2.   0.   0.   2.24 2.24]
 [1.41 2.83 2.   2.24 1.41 1.41]
 [0.   1.41 2.24 2.24 1.41 0.  ]]

Gradient based on Sobel:
 [[0.   1.41 3.16 3.16 1.41 0.  ]
 [1.41 2.83 2.83 3.16 4.24 1.41]
 [3.16 2.83 0.   1.41 3.16 3.16]
 [3.16 3.16 1.41 0.   2.83 3.16]
 [1.41 4.24 3.16 2.83 2.83 1.41]
 [0.   1.41 3.16 3.16 1.41 0.  ]]
```

e) How will you interpret the results with respect to edges in the test image?

Edges correspond to a change of pixels' intensity. With gradients, we can analyze where those changes are generated and, thus, detect where possible edges can be.

1.2 Problem 2

Given a test image with black background (gray level 0), and a white rectangle (gray level value 1), of size 8×8 pixels in the center. Use the notebook to create a matrix representing this image.

Let the test image be of size 12×14.

Click here for a small hint

The test image can be created by the following numpy commands:

```
R = np.ones(shape=(8,8))
I = np.zeros(shape=(12,14))
I[2:10,3:11] = R
```

Use the notebook to do the necessary computations in the following questions.

Use the Prewitt masks:

$$\mathbf{h}_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \qquad \mathbf{h}_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}.$$

for the computation of the differentials, $\frac{\partial I}{\partial x} = I_x$ and $\frac{\partial I}{\partial y} = I_y$ respectively.

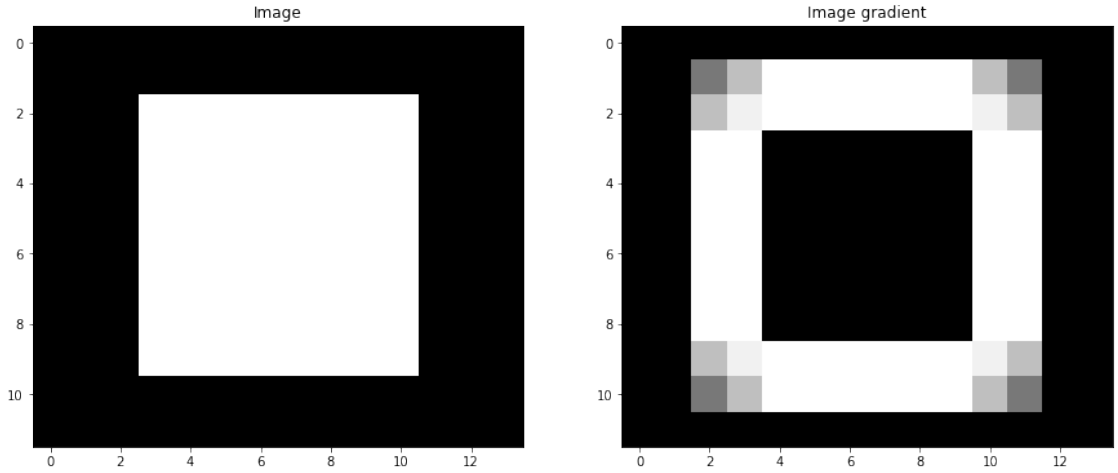
a) Compute and sketch the gradient of the test image using the 2-norm for the magnitude. Use $|\nabla I| = \|\nabla I\| = \sqrt{I_x^2(m,n) + I_y^2(m,n)}$. Show all relevant pixel values in the magnitude gradient image.

Hint: use `cv2.filter2D` function (Documentation) to perform a convolutional operation on an image using a specific mask.

[4]:
```
# Define the image
R = np.ones(shape=(8,8))
I = np.zeros(shape=(12,14))
I[2:10,3:11] = R

h_x = np.array([[1, 0, -1],
                [1, 0, -1],
                [1, 0, -1]
               ], dtype="float32")

h_y = np.array([[1, 1, 1],
                [0, 0, 0],
                [-1, -1, -1]
               ], dtype="float32")

G2_x = cv2.filter2D(I, -1, h_x)
G2_y = cv2.filter2D(I, -1, h_y)

grad = gradient(G2_x, G2_y)

print("The magnitude:")
print(grad)

plt.figure(figsize=(15,15))
plt.subplot(121), plt.imshow(I, cmap="gray"), plt.title("Image")
plt.subplot(122), plt.imshow(grad, cmap="gray"), plt.title("Image gradient")
plt.show()
```

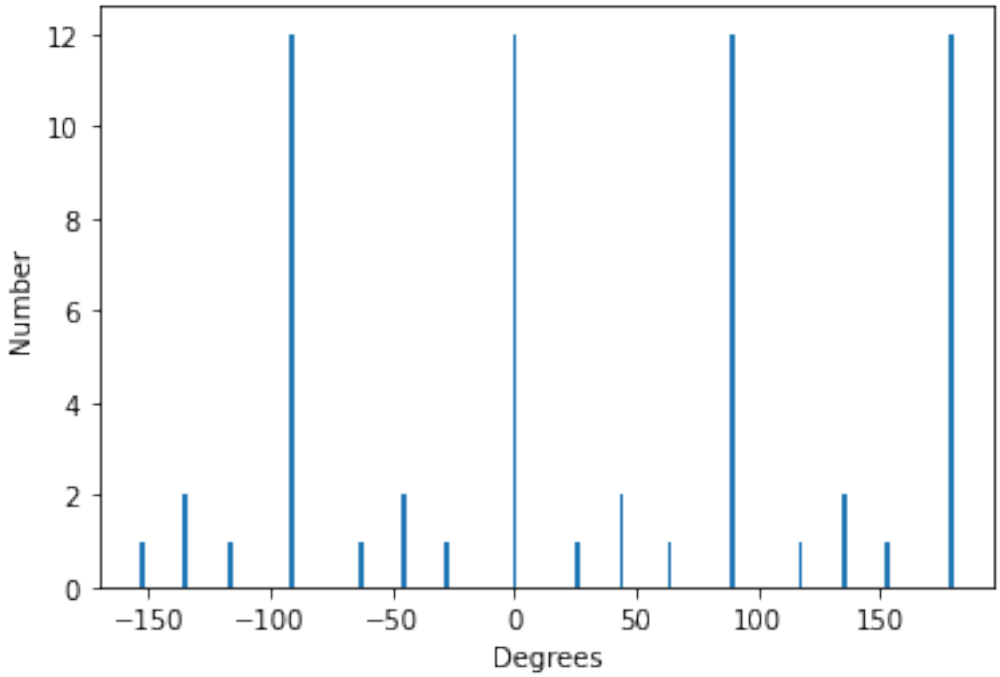
```
The magnitude:
[[0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   1.41 2.24 3.   3.   3.   3.   3.   3.   2.24 1.41 0.   0.  ]
 [0.   0.   2.24 2.83 3.   3.   3.   3.   3.   3.   2.83 2.24 0.   0.  ]
 [0.   0.   3.   3.   0.   0.   0.   0.   0.   0.   3.   3.   0.   0.  ]
 [0.   0.   3.   3.   0.   0.   0.   0.   0.   0.   3.   3.   0.   0.  ]
```

```
[0.   0.   3.   3.   0.   0.   0.   0.   0.   0.   3.   3.   0.   0.  ]
[0.   0.   3.   3.   0.   0.   0.   0.   0.   0.   3.   3.   0.   0.  ]
[0.   0.   3.   3.   0.   0.   0.   0.   0.   0.   3.   3.   0.   0.  ]
[0.   0.   3.   3.   0.   0.   0.   0.   0.   0.   3.   3.   0.   0.  ]
[0.   0.   2.24 2.83 3.   3.   3.   3.   3.   3.   2.83 2.24 0.   0.  ]
[0.   0.   1.41 2.24 3.   3.   3.   3.   3.   3.   2.24 1.41 0.   0.  ]
[0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  ]]
```



b) Sketch the histogram of gradient directions. Be precise in labeling the height of each bar in the histogram.

[5]:
```
phase = np.arctan2(G2_y, G2_x)

grad_dir = phase.flatten()*(180/np.pi)

plt.hist(grad_dir[grad.flatten()>0], bins=len(grad_dir))
plt.xlabel('Degrees')
plt.ylabel('Number')
plt.show()

print("""Here we can see the four highest stems representing the four sides of␣
 ↪the rectangle at -90, 0, 90 and 180 degrees.
The corners are found at -135, -45, 45 and 135 degrees, represented by two␣
 ↪pixels each.""")
```



Here we can see the four highest stems representing the four sides of the rectangle at -90, 0, 90 and 180 degrees.
The corners are found at -135, -45, 45 and 135 degrees, represented by two pixels each.

c) Sketch the Laplacian of the test image using the mask in previous equation. Show all relevant pixel values in the Laplacian image.

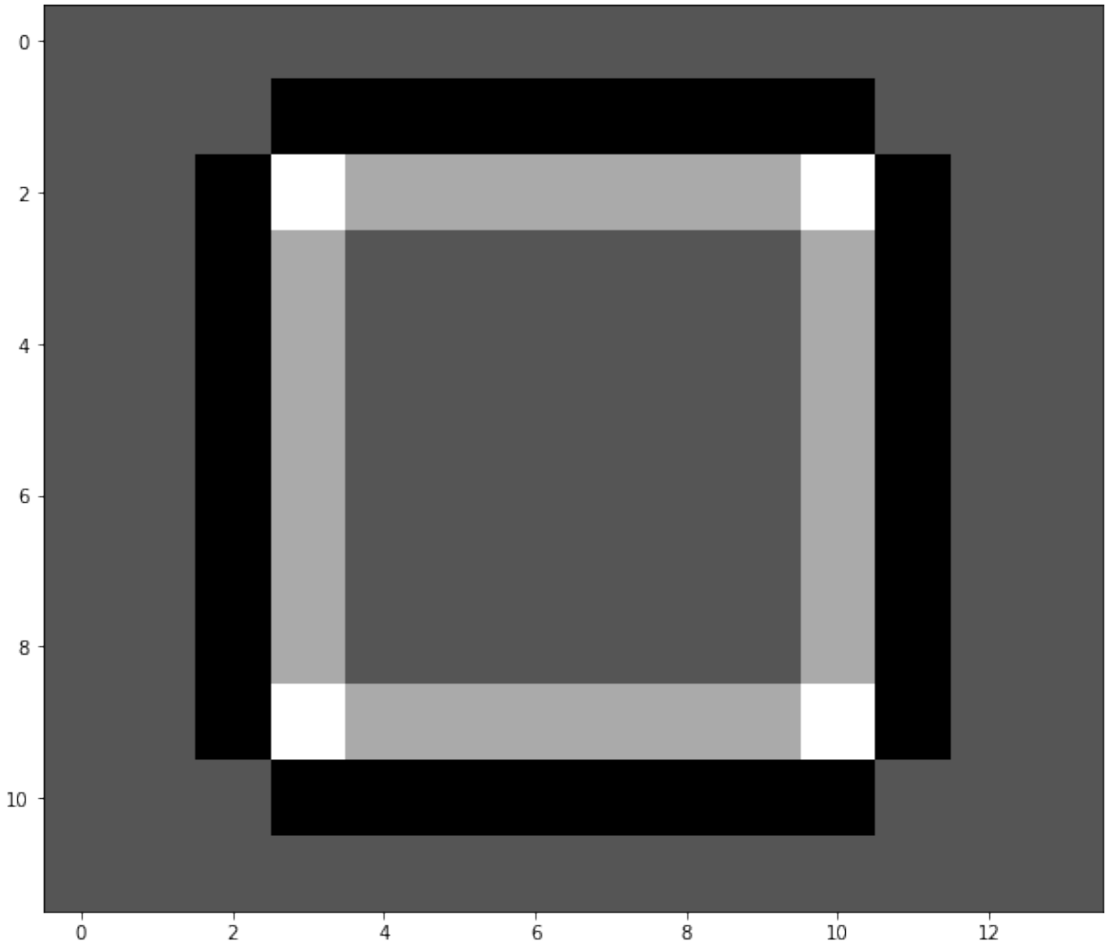
The Laplacian can be computed using the following mask:

$$\mathbf{h}_L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \tag{4}$$

[6]:
```
h_L = np.array([[0, -1, 0],
                [-1, 4, -1],
                [0, -1, 0]
               ], dtype="float32")

L = cv2.filter2D(I, -1, h_L)

print("Show the pixel values of the resulting image: ")
print(L)

print("Or if you want a visualization of the result: ")
plt.figure(figsize=(10,10))
```

```
plt.imshow(L, cmap="gray")
plt.show()
```

```
Show the pixel values of the resulting image:
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1.  0.  0.  0.]
 [ 0.  0. -1.  2.  1.  1.  1.  1.  1.  1.  2. -1.  0.  0.]
 [ 0.  0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.  0.]
 [ 0.  0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.  0.]
 [ 0.  0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.  0.]
 [ 0.  0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.  0.]
 [ 0.  0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.  0.]
 [ 0.  0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.  0.]
 [ 0.  0. -1.  2.  1.  1.  1.  1.  1.  1.  2. -1.  0.  0.]
 [ 0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
Or if you want a visualization of the result:
```



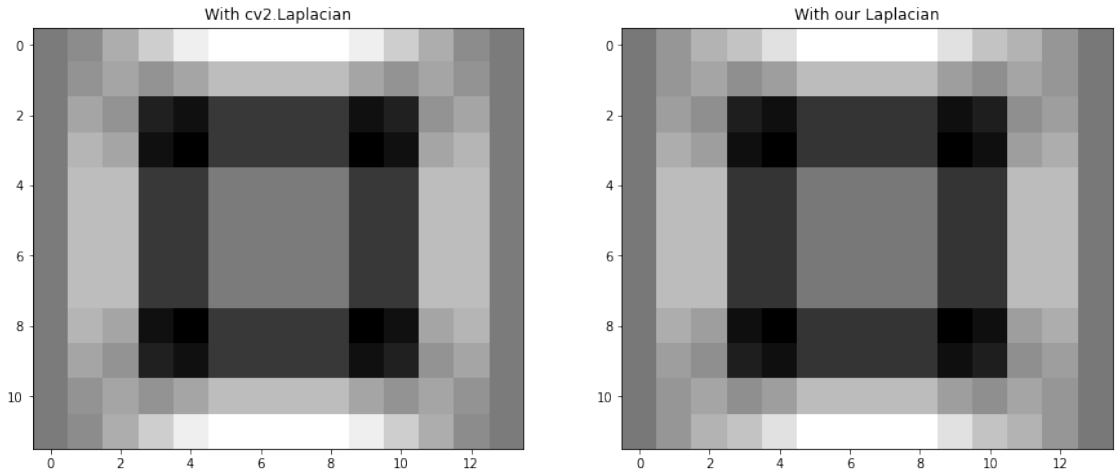
d) What is the resulting mask for computation of the Laplacian if the Prewitt masks are used for computation of the differentials?

Click here for optional hints

- Import the scipy package and use the convolve2d function (Documentation) for this task.
- If you want to check if the resulting mask is correct, use the built-in cv2 cv2.Laplacian(I, -1, ksize=5) function (Documentation) where I is the test image and ksize is the aperture size used to compute the second-derivative filters.

[7]:
```
h_xx = signal.convolve2d(h_x, h_x)
h_yy = signal.convolve2d(h_y, h_y)

h_LL = h_xx + h_yy

print(h_LL)

print("""The structure is the same as before, but the window is larger and the␣
 ↪lobe in the center is wider!""")

I1 = cv2.Laplacian(I, -1, ksize=5)
I2 = cv2.filter2D(I,-1,h_LL)

plt.figure(figsize=(15,15))
plt.subplot(121), plt.imshow(I1, cmap="gray"), plt.title("With cv2.Laplacian")
plt.subplot(122), plt.imshow(I2, cmap="gray"), plt.title("With our Laplacian")
plt.show()
print("""
We convolve the h_x and h_y masks with themselves to arrive at the second order␣
 ↪derivative kernels.
The HL laplacian mask can be used to filter our image in a similar way as the␣
 ↪second order Laplacian function in openCV (cv2.Laplacian).
""")
```

```
[[  2.   2.   1.   2.   2.]
 [  2.   0.  -4.   0.   2.]
 [  1.  -4. -12.  -4.   1.]
 [  2.   0.  -4.   0.   2.]
 [  2.   2.   1.   2.   2.]]
The structure is the same as before, but the window is larger and the lobe in
the center is wider!
```



We convolve the h_x and h_y masks with themselves to arrive at the second order derivative kernels.
The HL laplacian mask can be used to filter our image in a similar way as the second order Laplacian function in openCV (cv2.Laplacian).

1.3 Problem 3

One of the most common linear filters in computer vision applications is the Gaussian smoothing filter.

In this problem we want to study the use of Gaussian filters with different standard deviations, σ, and different sizes, $K \times K$, where K is odd ($K = 2k + 1$, k is integer). The filter kernel (mask) is found by using the OpenCV function `cv2.getGaussianKernel()` (Documentation). Start by finding filter masks as follows

a) h05: $\sigma = 0.5$, $K = 7$

b) h1: $\sigma = 1$, $K = 11$

c) h15: $\sigma = 1.5$, $K = 15$

Use the `plt.stem` function from Matplotlib and display each filter (sampled 1D Gaussian function).

If the size K is too small we will get a truncated Gaussian with a step at the tails.

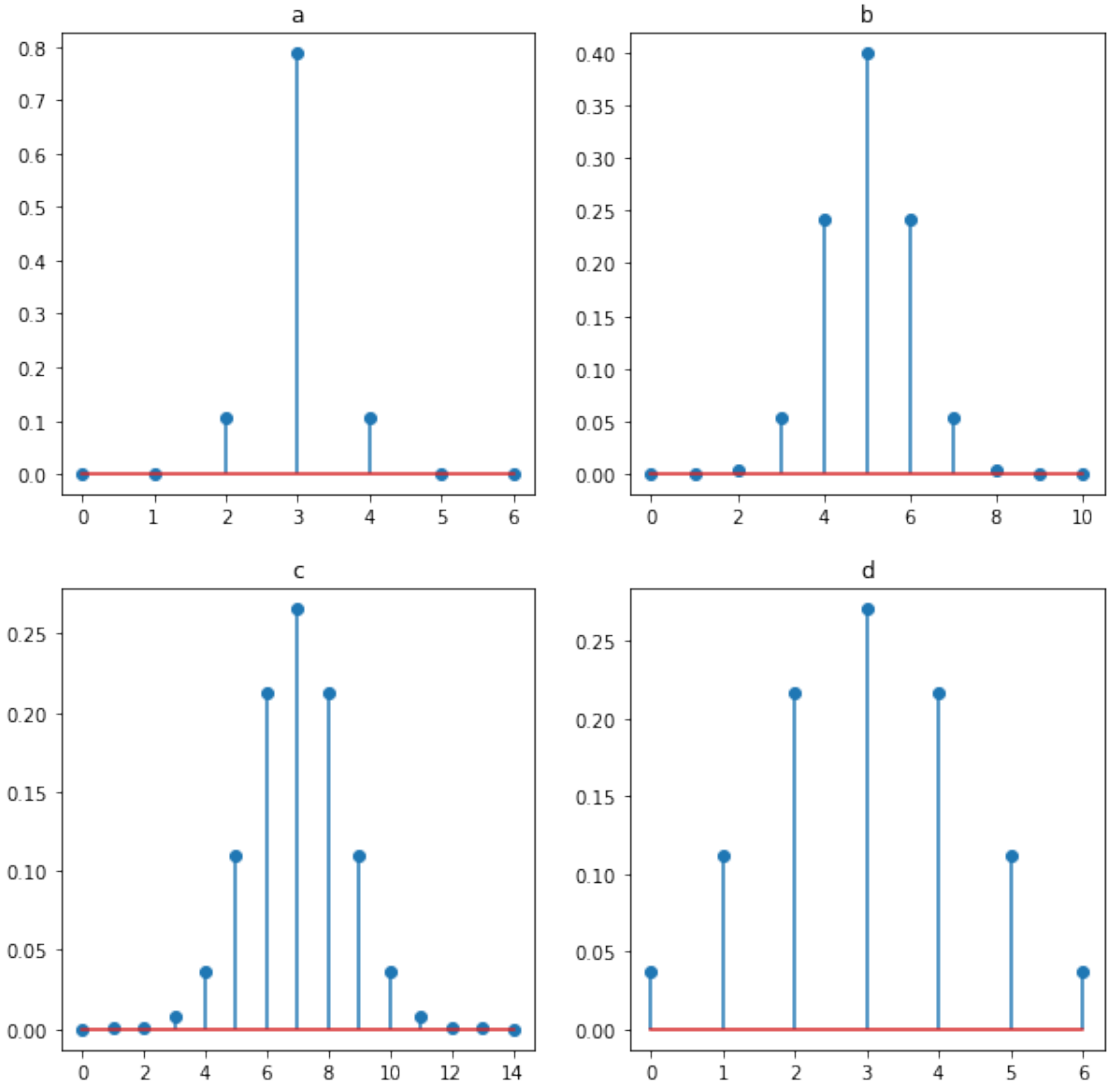
d) Show the result for c) above when $K = 7$.

If we want a proper Gaussian filter there is a connection between the value of σ and the size K. At three standard deviations, 3σ, the value of the Gaussian is 1% of its maximum value.

[8]:
```
h05 = cv2.getGaussianKernel(ksize=7, sigma=0.5)
h1 = cv2.getGaussianKernel(ksize=11, sigma=1)
h15 = cv2.getGaussianKernel(ksize=15, sigma=1.5)
```

```
h15_trunc = cv2.getGaussianKernel(ksize=7, sigma=1.5)

plt.figure(figsize=(10,10))

plt.subplot(2,2,1)
plt.stem(h05)
plt.title("a")

plt.subplot(2,2,2)
plt.stem(h1)
plt.title("b")

plt.subplot(2,2,3)
plt.stem(h15)
plt.title("c")

plt.subplot(2,2,4)
plt.stem(h15_trunc)
plt.title("d")
```

[8]: `Text(0.5, 1.0, 'd')`



1.4 Problem 4

In this exercise we want to study how two well-known filters perform on noise removal, namely the Gaussian and the median filter.

```
import cv2
from skimage.util import random_noise

Im = cv2.imread('./images/cameraman.jpg')
Im_gauss = random_noise(Im,  mode='gaussian', mean=0, var=0.01) # Gaussian white noise with va
Im_SP = random_noise(Im, 's&p', amount=0.05) # Salt and pepper noise on 5% of the pixels
```

a) Apply Gaussian smoothing to the original image, `Im`, using the defined filter kernels from problems 3a, 3b, and 3c. Explain the results.

[9]:
```
import cv2

Im = cv2.imread('images/cameraman.jpg')
Im = cv2.cvtColor(Im, cv2.COLOR_BGR2GRAY)

Im_h15 = cv2.filter2D(Im,-1,h15)

plt.figure(figsize=(30,30))
plt.subplot(2,2,(1,2)), plt.imshow(Im_h15, cmap='gray')
plt.title('Image with Gaussian filter: k=15, sigma=1.5')
plt.subplot(223), plt.imshow(cv2.sepFilter2D(Im,-1, kernelX=h15, kernelY=h15),␣
 ↪cmap='gray')
plt.title('Image with Gaussian filter: k=15, sigma=1.5')
plt.subplot(224), plt.imshow(cv2.filter2D(Im_h15,-1,h15.T), cmap='gray')
plt.title('Image, already filtered in one dimension, with Gaussian filter:␣
 ↪k=15, sigma=1.5 transposed (for the other dimension).')
plt.show()

print("""
Here it is presented an example of how to filter an image with a 2D filter.
- You can use cv2.sepFilter2D() using the kernelX and kernelY parameters.
- Or you can use the cv2.filter2D() using a 2D filter (obtained by multiplying␣
 ↪the 1D filter for its transpose)
- In alternative, you can perform 2 steps of cv2.filter2D() with the same 1D␣
 ↪filter but on the second step the filter must be transposed.

The last two filtered images show the same result achieved with with two of the␣
 ↪methods just briefly described.
""")
```

Here it is presented an example of how to filter an image with a 2D filter.
- You can use cv2.sepFilter2D() using the kernelX and kernelY parameters.
- Or you can use the cv2.filter2D() using a 2D filter (obtained by multiplying the 1D filter for its transpose)
- In alternative, you can perform 2 steps of cv2.filter2D() with the same 1D filter but on the second step the filter must be transposed.

The last two filtered images show the same result achieved with with two of the methods just briefly described.

[10]:
```
plt.figure(figsize=(30,30))
plt.subplot(221), plt.imshow(Im, cmap='gray')
plt.title('Original Image')
plt.subplot(222), plt.imshow(cv2.filter2D(Im,-1,h05*h05.T), cmap='gray')
plt.title('Image with Gaussian filter: k=7, sigma=0.5')
plt.subplot(223), plt.imshow(cv2.filter2D(Im,-1,h1*h1.T), cmap='gray')
plt.title('Image with Gaussian filter: k=11, sigma=1')
plt.subplot(224), plt.imshow(cv2.filter2D(Im,-1,h15*h15.T), cmap='gray')
plt.title('Image with Gaussian filter: k=15, sigma=1.5')
plt.show()

print("""The presented images are generated using a 2D gaussian filter.

A bigger Gaussian kernel results in more smoothing of the image.""")
```

The presented images are generated using a 2D gaussian filter.

A bigger Gaussian kernel results in more smoothing of the image.

Gaussian noise:

b) Apply the three Gaussian filters, described in problems 3a, 3b, and 3c, to the image `Im_gauss`. Explain the results.

c) Apply a median filter on the image `Im_gauss` using the command `scipy.ndimage.median_filter` (Documentation). How does this filter perform compared to the Gaussian filters?

[11]:
```
######## b) & c)
from skimage.util import random_noise
from scipy import ndimage

Im_gauss = random_noise(Im,  mode='gaussian', mean=0, var=0.01) # Gaussian␣
 ↪white noise with variance of 0.01

plt.figure(figsize=(20,30))
plt.subplot(3,2,(1,2)), plt.imshow(Im_gauss, cmap='gray')
plt.title("Original image with Gaussian white noise")
plt.subplot(323), plt.imshow(cv2.filter2D(Im_gauss,-1,h05*h05.T), cmap='gray')
plt.title('Image with random noise with Gaussian filter: k=7, sigma=0.5')
plt.subplot(324), plt.imshow(cv2.filter2D(Im_gauss,-1,h1*h1.T), cmap='gray')
plt.title('Image with random noise with Gaussian filter: k=11, sigma=1')
plt.subplot(325), plt.imshow(cv2.filter2D(Im_gauss,-1,h15*h15.T), cmap='gray')
plt.title('Image with random noise with Gaussian filter: k=15, sigma=1.5')
plt.subplot(326), plt.imshow(ndimage.median_filter(Im_gauss,size=5),␣
 ↪cmap='gray')
plt.title('Median filter')
plt.show()


print("""
The Gaussian filters reduce the amount of noise in the image, while the median␣
 ↪filter does not reduce the amount of Gaussian noise in the image.
""")
```


Original image with Gaussian white noise


Image with random noise with Gaussian filter: k=7, sigma=0.5


Image with random noise with Gaussian filter: k=11, sigma=1


Image with random noise with Gaussian filter: k=15, sigma=1.5


Median filter

`The Gaussian filters reduce the amount of noise in the image, while the median filter does not reduce the amount of Gaussian noise in the image.`

Salt & pepper noise:

d) Apply the three Gaussian filters, described in problems 3a, 3b, and 3c, to the image `Im_SP`. Explain the results.

e) Apply a median filter on the image `Im_SP` using the command `scipy.ndimage.median_filter` (Documentation). How does this filter perform compared to the Gaussian filters?

[12]:
```
######## d) & e)

Im_SP = random_noise(Im, 's&p', amount=0.05) # Salt and pepper noise on 5% of␣
 ↪the pixels

plt.figure(figsize=(20,30))
plt.subplot(321), plt.imshow(Im_SP, cmap='gray')
plt.title("Original image with S&P noise")
plt.subplot(322), plt.imshow(cv2.filter2D(Im_SP,-1,h05*h05.T), cmap='gray')
plt.title('Image with salt & pepper noise with Gaussian filter: k=7, sigma=0.5')
plt.subplot(323), plt.imshow(cv2.filter2D(Im_SP,-1,h1*h1.T), cmap='gray')
plt.title('Image with salt & pepper noise with Gaussian filter: k=11, sigma=1')
plt.subplot(324), plt.imshow(cv2.filter2D(Im_SP,-1,h15*h15.T), cmap='gray')
plt.title('Image with salt & pepper noise with Gaussian filter: k=15, sigma=1.
 ↪5')
plt.subplot(3,2,(5,6)), plt.imshow(ndimage.median_filter(Im_SP,size=3),␣
 ↪cmap='gray')
plt.title('Median filter')
plt.show()

print("""
When salt and pepper noise occurs in the image, a median filter can remove it␣
 ↪with minimal loss in image quality.
The Gaussian filter does not perform well.
""")
```


Original image with S&P noise


Image with salt & pepper noise with Gaussian filter: k=7, sigma=0.5


Image with salt & pepper noise with Gaussian filter: k=11, sigma=1

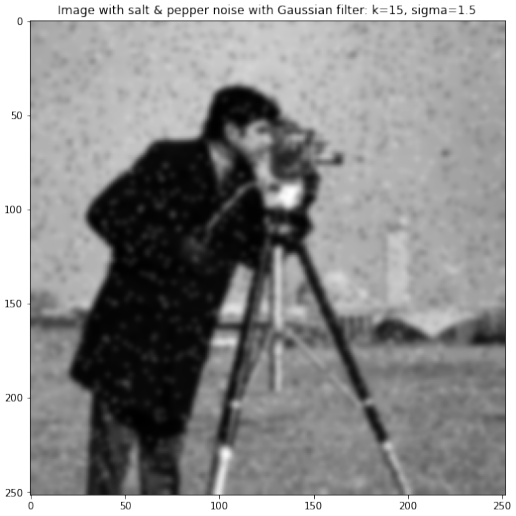


Image with salt & pepper noise with Gaussian filter: k=15, sigma=1.5


Median filter

```
When salt and pepper noise occurs in the image, a median filter can remove it
with minimal loss in image quality.
The Gaussian filter does not perform well.
```

We can see that, as Median filter replaces each pixel with the median of al the gray levels in a local neighborhood, it works better for the salt-and-pepper noise (in particular, with a small kernel size). On the otehr hand, the performance for white Gaussian noise is better in case of using a lowpass Gaussian filter.

1.4.1 Delivery (dead line) on CANVAS: 29.09.2023 at 23.59

1.5 Contact

1.5.1 Course teacher

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

1.5.2 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

1.6 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.