Proposed solutions for LABexc4-ELE510-2021

September 21, 2021

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 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

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

The computations should be done by hand on paper until you are confident that you know how to do it.

Thereafter you can use the notebook to complete and check the results.

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:

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

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

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

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.

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

Proposed answers: a)

For the Sobel mask we get:

$h_{1x} = [1 \quad 0 \quad -1]$
and
$h_{2x} = [1 \quad 2 \quad 1]^T$
With the result
$h_x = h_{1x} h_{2x}$

And for the Prewitt mask:

$$h_{1x} = [1 \quad 0 \quad -1]$$
and
$$h_{2x} = [1 \quad 1 \quad 1]^T$$
With the result
$$h_x = h_{1x} h_{2x}$$

Note that the resulting 2D mask is a matrix found by the outer product of the two 1D masks, where the differentiation for the x-direction is written as a row vector and the smoothing filter is given as a column vector.

b)

Prewitt

$$Im_x = Im * h_x = \begin{bmatrix} 1 & 0 & -1 & 1 & 0 & -1 \\ 1 & 1 & 0 & 0 & -1 & -1 \\ 1 & 2 & 1 & -1 & -2 & -1 \\ 1 & 2 & 1 & -1 & -2 & -1 \\ 1 & 1 & 0 & 0 & -1 & -1 \\ 1 & 0 & -1 & 1 & 0 & -1 \end{bmatrix}$$

$$Im_y = Im * h_y = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ -1 & 0 & 1 & 1 & 0 & -1 \\ 1 & 0 & -1 & -1 & 0 & 1 \\ 0 & -1 & -2 & -2 & -1 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

c)

Sobel

$$Im_x = Im * h_x = \begin{bmatrix} 1 & 0 & -1 & 1 & 0 & -1 \\ 2 & 1 & -1 & 1 & -1 & -2 \\ 1 & 3 & 2 & -2 & -3 & -1 \\ 1 & 3 & 2 & -2 & -3 & -1 \\ 2 & 1 & -1 & 1 & -1 & -2 \\ 1 & 0 & -1 & 1 & 0 & -1 \end{bmatrix}$$

$$Im_y = Im * h_y = \begin{bmatrix} 1 & 2 & 1 & 1 & 2 & 1 \\ 0 & 1 & 3 & 3 & 1 & 0 \\ -1 & -1 & 2 & 2 & -1 & -1 \\ 1 & 1 & -2 & -2 & 1 & 1 \\ 0 & -1 & -3 & -3 & -1 & 0 \\ -1 & -2 & -1 & -1 & -2 & -1 \end{bmatrix}$$

d)

Image gradients

For the Prewitt filter the gradient is:

$$|\nabla Im| = \begin{bmatrix} 1.4142 & 1.0000 & 1.4142 & 1.4142 & 1.0000 & 1.4142 \\ 1.0000 & 1.4142 & 2.0000 & 2.0000 & 1.4142 & 1.0000 \\ 1.4142 & 2.0000 & 1.4142 & 1.4142 & 2.0000 & 1.4142 \\ 1.4142 & 2.0000 & 1.4142 & 1.4142 & 2.0000 & 1.4142 \\ 1.0000 & 1.4142 & 2.0000 & 2.0000 & 1.4142 & 1.0000 \\ 1.4142 & 1.0000 & 1.4142 & 1.4142 & 1.0000 & 1.4142 \end{bmatrix}$$

For the Sobel mask the gradient is:

$$|\nabla Im| = \begin{bmatrix} 1.4142 & 2.0000 & 1.4142 & 1.4142 & 2.0000 & 1.4142 \\ 2.0000 & 1.4142 & 3.1623 & 3.1623 & 1.4142 & 2.0000 \\ 1.4142 & 3.1623 & 2.8284 & 2.8284 & 3.1623 & 1.4142 \\ 1.4142 & 3.1623 & 2.8284 & 2.8284 & 3.1623 & 1.4142 \\ 2.0000 & 1.4142 & 3.1623 & 3.1623 & 1.4142 & 2.0000 \\ 1.4142 & 2.0000 & 1.4142 & 1.4142 & 2.0000 & 1.4142 \end{bmatrix}$$

1.2 Problem 2

Given a test image with black background (gray level 0), and a white rectangle (gray level value 1), of size 6×8 pixels in the center. Use the notebook to create a matrix representing this image.

Let the test image be of size 10×12.

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

Use the Prewitt masks:

$$\mathbf{h}_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \qquad \mathbf{h}_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}. \tag{4}$$

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.

Click here for an optional hint

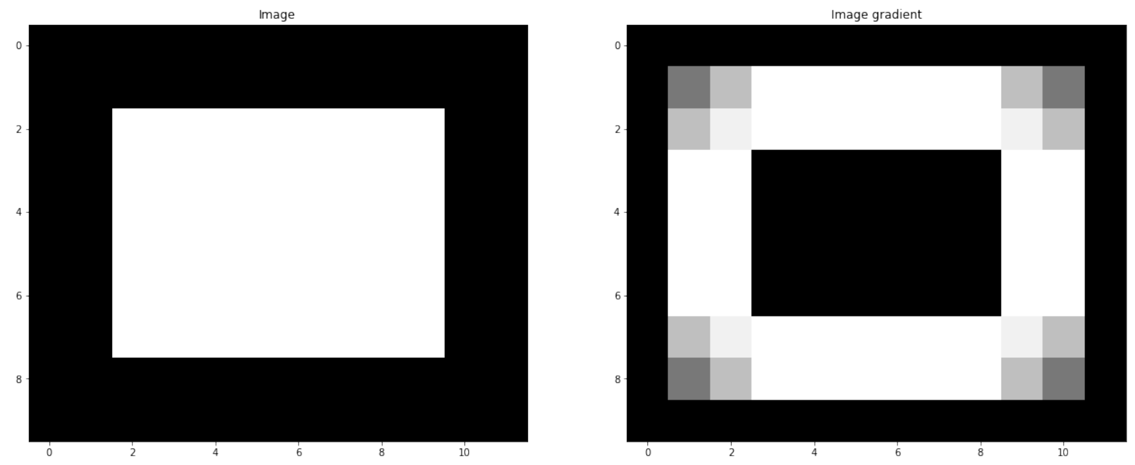
Use cv2.filter2D function (Documentation) to perform a convolutional operation on an image using a specific mask.

[1]:
```python
# Import useful packages
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

[2]:
```python
######## a)
# Define the image
```

```
R = np.ones(shape=(6,8))
I = np.zeros(shape=(10,12))
I[2:8,2:10] = R

# Define h_x and h_y
h_x = np.zeros(shape=(3,3))
h_y = np.zeros(shape=(3,3))
h_x[:,0] = 1
h_x[:,-1] = -1
h_y[0,:] = 1
h_y[-1,:] = -1

# Convolution operation with the masks
I_x = cv2.filter2D(I, -1, h_x)
I_y = cv2.filter2D(I, -1, h_y)

# Calculate the gradient of I
gradient_I = np.around(np.hypot(I_x, I_y), 2) # equal to: np.around(np.
 ↪sqrt(I_x**2 + I_y**2),2)
print("The magnitude:")
print(gradient_I)

plt.figure(figsize=(20,20))
plt.subplot(121), plt.imshow(I, cmap="gray"), plt.title("Image")
plt.subplot(122), plt.imshow(gradient_I, cmap="gray"), plt.title("Image␣
 ↪gradient")
plt.show()
```

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

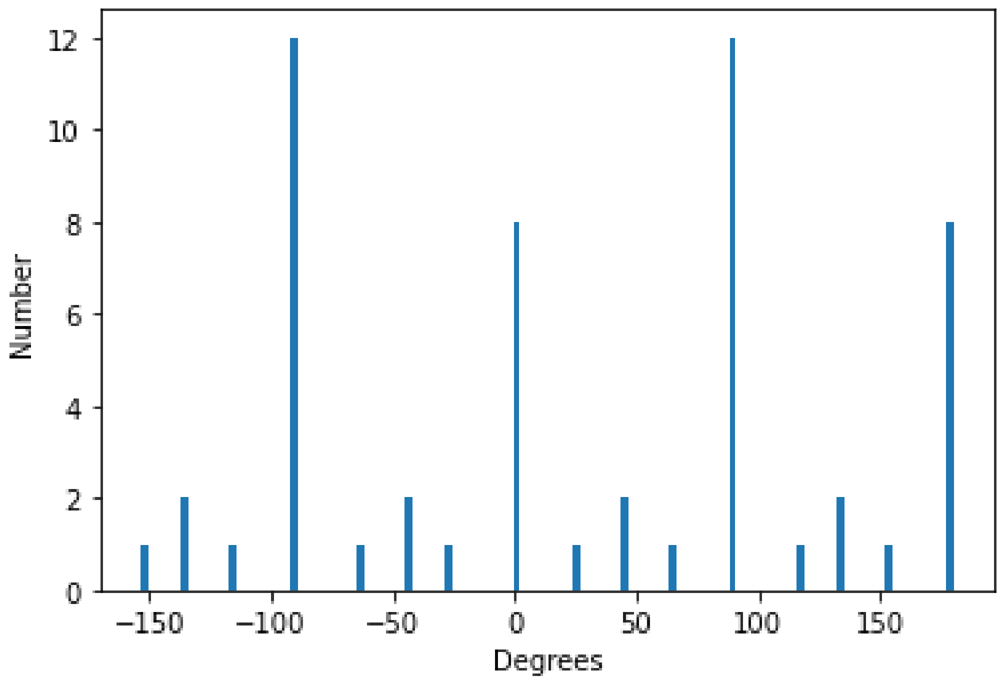


b) Sketch the histogram of gradient directions (**in degrees**).

Click here for an optional hint

The numpy.arctan2 function (Documentation) might be useful for this task to convert the x-y coordinates into radiants.

[3]:
```
######## b)

# flat the masks
flatI_x = I_x.flatten()
flatI_y = I_y.flatten()
# Calculate the gradient direction in degrees
gradient_directions = (180/np.pi) * np.arctan2(flatI_y,flatI_x)

plt.hist(gradient_directions[gradient_I.flatten()>0],␣
 ↪bins=len(gradient_directions))
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.
```

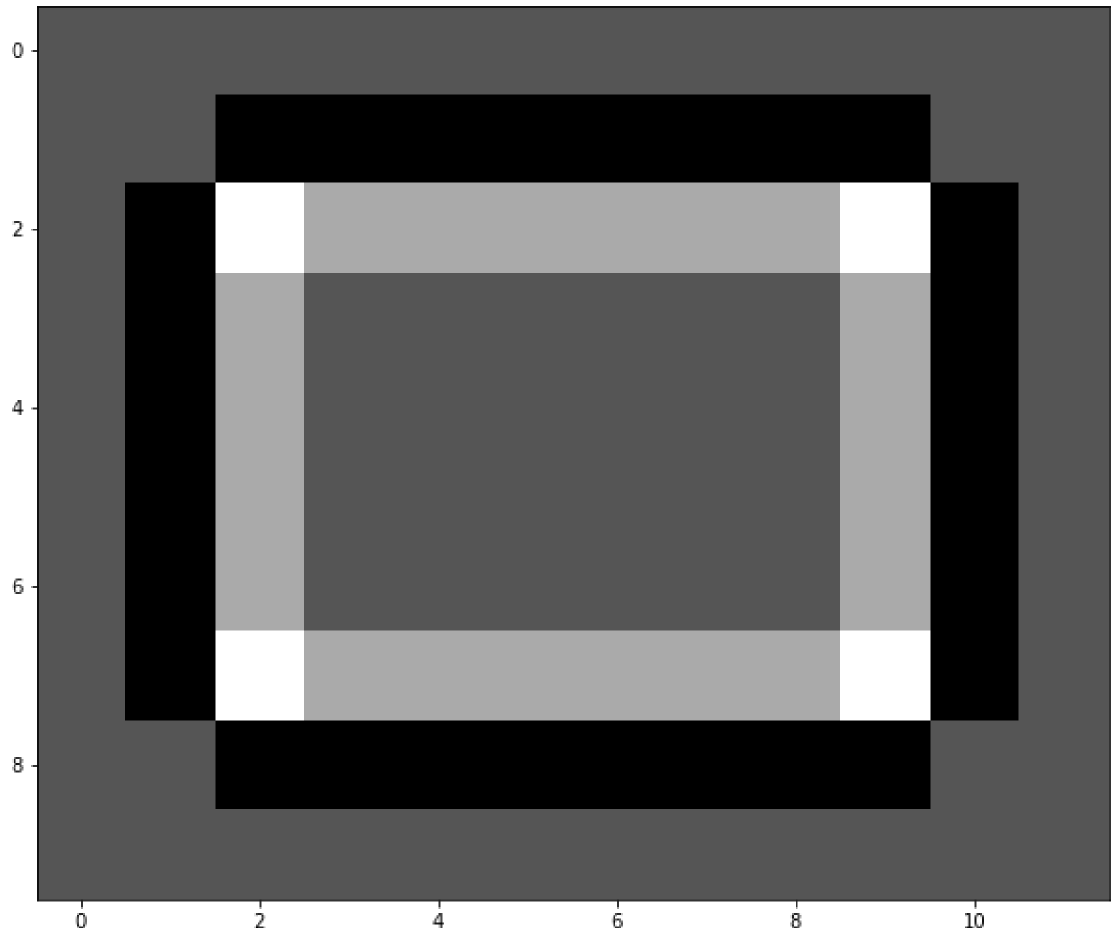
The Laplacian can be computed using the following mask:

$$\mathbf{h}_L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \tag{5}$$

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

[4]:
```
######## 3.
h_L = np.array([[0,-1,0],[-1,4,-1],[0,-1,0]]) # Laplacian mask
IL = cv2.filter2D(I, -1, h_L) # Convolve the mask with the test image I

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

print("Or if you want a visualization of the result: ")
plt.figure(figsize=(10,10))
plt.imshow(IL, 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. -1. -1. -1. -1. -1. -1. -1. -1.  0.  0.]
 [ 0. -1.  2.  1.  1.  1.  1.  1.  1.  2. -1.  0.]
 [ 0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.]
 [ 0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.]
 [ 0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.]
 [ 0. -1.  1.  0.  0.  0.  0.  0.  0.  1. -1.  0.]
 [ 0. -1.  2.  1.  1.  1.  1.  1.  1.  2. -1.  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.]]
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?

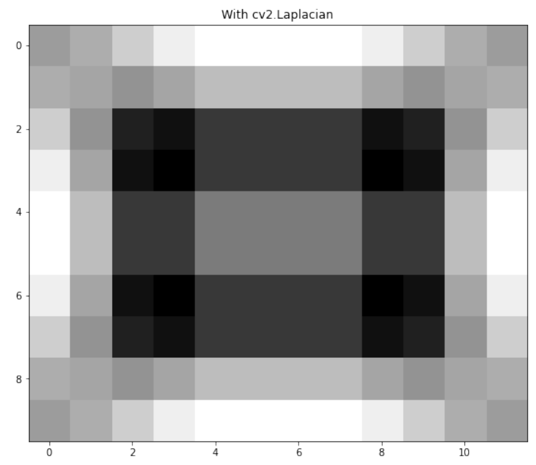
Click here for optional hints

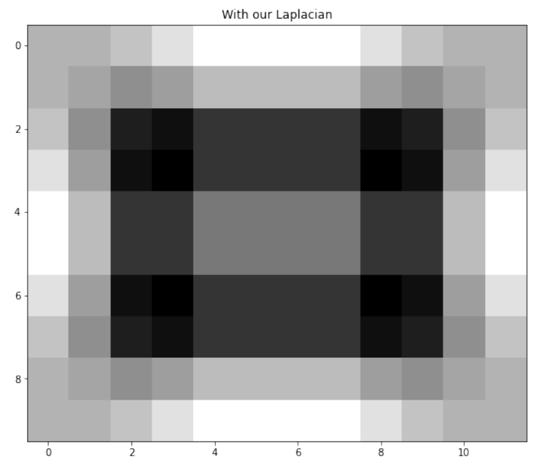
- Import the scipy package and use the convolve2d function (Documentation) for this task.

- 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.

```
[5]: ######## 4.
from scipy import signal # import scipy and use the convolve2d function

h_xx = signal.convolve2d(h_x, h_x)
h_yy = signal.convolve2d(h_y, h_y)
HL = h_xx + h_yy
print(HL)

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,HL)

plt.figure(figsize=(20,20))
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) h1: $\sigma = 1$, $K = 9$

b) h15: $\sigma = 1.5$, $K = 11$

c) h2: $\sigma = 2$, $K = 15$

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

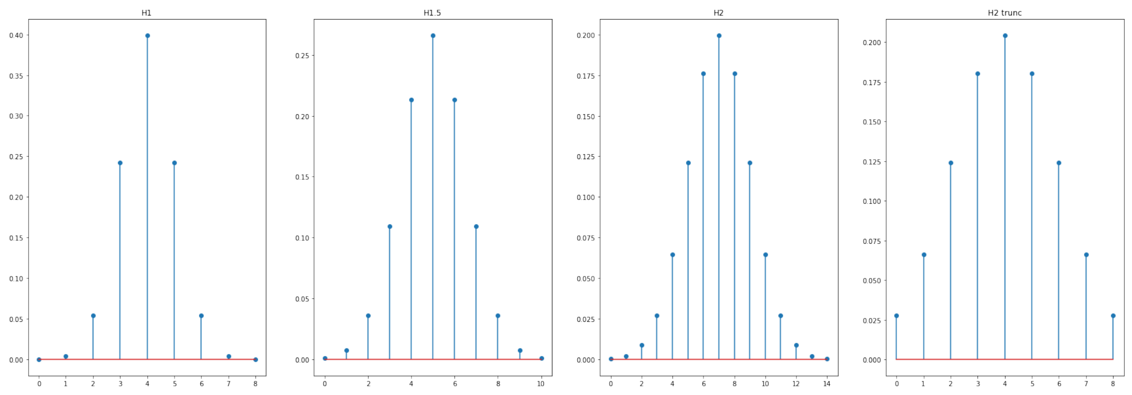
If the size K is too small we will get a truncated Gaussian with a step at the tails.

d) Show the result for c) above when $K = 9$.

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.

[6]:
```
h1 = cv2.getGaussianKernel(9,1)

h15 = cv2.getGaussianKernel(11,1.5)
```

```
h2 = cv2.getGaussianKernel(15,2)

plt.figure(figsize=(30,10))
######### a)
plt.subplot(141), plt.stem(h1,use_line_collection=True)
plt.title("H1")

######### b)
plt.subplot(142), plt.stem(h15,use_line_collection=True)
plt.title("H1.5")

######### c)
plt.subplot(143), plt.stem(h2,use_line_collection=True)
plt.title("H2")

######### d)
h2trunc = cv2.getGaussianKernel(9,2)

plt.subplot(144), plt.stem(h2trunc,use_line_collection=True)
plt.title("H2 trunc")
plt.show()

print("From the last result we see the abrupt transition at the borders of the␣
 ↪filter mask when the length is too short.")
```



```
From the last result we see the abrupt transition at the borders of the filter
mask when the length is too short.
```

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=0.05, var=0.01) # Gaussian white noise with
Im_SP = random_noise(Im, 's&p', amount=0.05) # Salt and pepper noise on 5% of the pixels
```

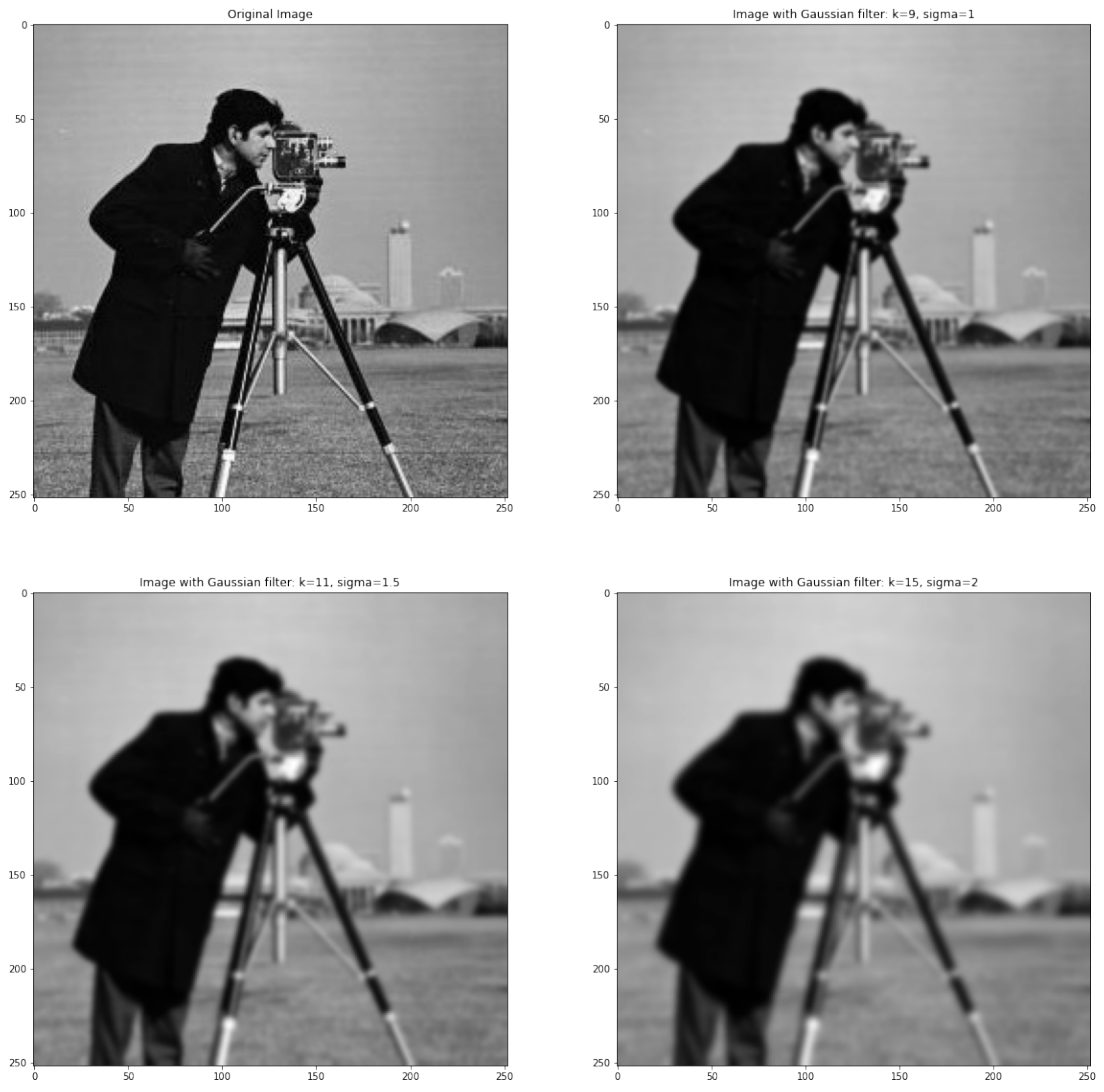
a) Apply Gaussian smoothing to the original image, `Im`, using the defined filter kernels from problem 3. Explain the results.

```
[7]: ######## a)
     import cv2

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

     Im_h2 = cv2.filter2D(Im,-1,h2)

     plt.figure(figsize=(30,30))
     plt.subplot(2,2,(1,2)), plt.imshow(Im_h2, cmap='gray')
     plt.title('Image with Gaussian filter: k=15, sigma=2')
     plt.subplot(223), plt.imshow(cv2.sepFilter2D(Im,-1, kernelX=h2, kernelY=h2),␣
      ↪cmap='gray')
     plt.title('Image with Gaussian filter: k=15, sigma=2')
     plt.subplot(224), plt.imshow(cv2.filter2D(Im_h2,-1,h2.T), cmap='gray')
     plt.title('Image, already filtered in one dimension, with Gaussian filter:␣
      ↪k=15, sigma=2 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.
     """)
```

Image, already filtered in one dimension, with Gaussian filter: k=15, sigma=2 transposed (for the other dimension).

```
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.
```

[8]:
```python
# Continuation of a)

plt.figure(figsize=(20,20))
plt.subplot(221), plt.imshow(Im, cmap='gray')
plt.title('Original Image')
plt.subplot(222), plt.imshow(cv2.filter2D(Im,-1,h1*h1.T), cmap='gray')
plt.title('Image with Gaussian filter: k=9, sigma=1')
plt.subplot(223), plt.imshow(cv2.filter2D(Im,-1,h15*h15.T), cmap='gray')
plt.title('Image with Gaussian filter: k=11, sigma=1.5')
plt.subplot(224), plt.imshow(cv2.filter2D(Im,-1,h2*h2.T), cmap='gray')
plt.title('Image with Gaussian filter: k=15, sigma=2')
plt.show()

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

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


Original Image
Image with Gaussian filter: k=9, sigma=1
Image with Gaussian filter: k=11, sigma=1.5
Image with Gaussian filter: k=15, sigma=2
0
50
100
150
200
250

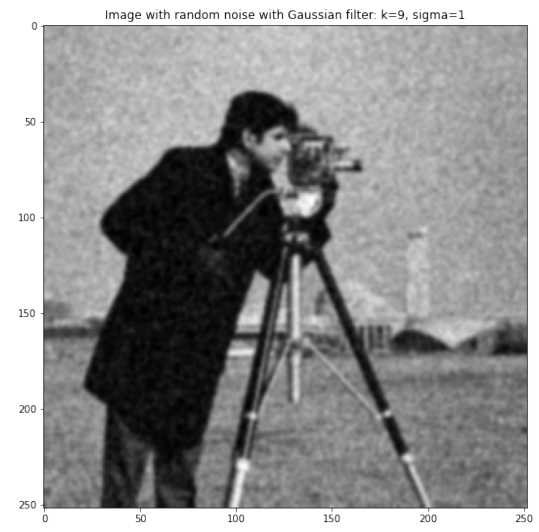
```
The presented images are generated using a 2D gaussian filter.

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

Gaussian noise:

b) Apply the three Gaussian filters, described in problem 3, 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?

[9]:
```
######## 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=(20,20))
plt.subplot(221), plt.imshow(cv2.filter2D(Im_gauss,-1,h1*h1.T), cmap='gray')
plt.title('Image with random noise with Gaussian filter: k=9, sigma=1')
plt.subplot(222), plt.imshow(cv2.filter2D(Im_gauss,-1,h15*h15.T), cmap='gray')
plt.title('Image with random noise with Gaussian filter: k=11, sigma=1.5')
plt.subplot(223), plt.imshow(cv2.filter2D(Im_gauss,-1,h2*h2.T), cmap='gray')
plt.title('Image with random noise with Gaussian filter: k=15, sigma=2')
plt.subplot(224), 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.
""")
```



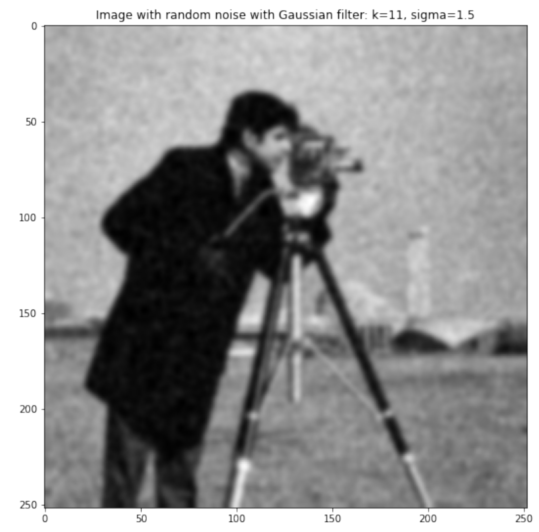
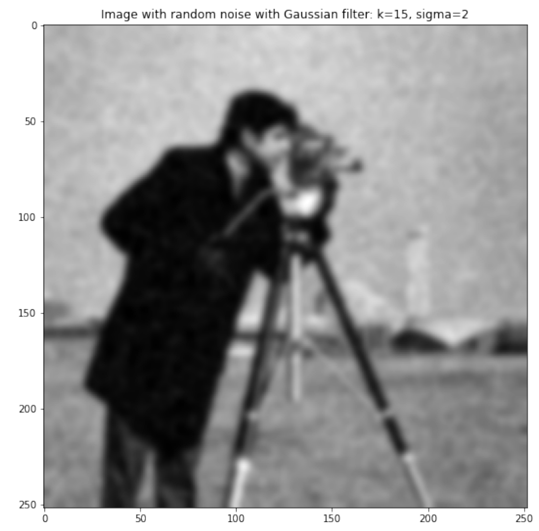
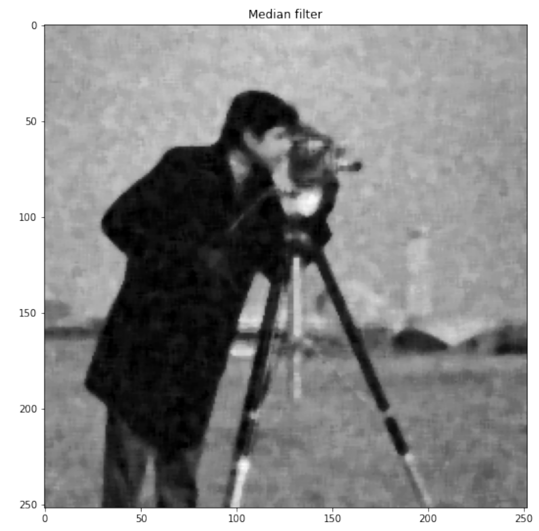


Image with random noise with Gaussian filter: k=11, sigma=1.5





```
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.
```

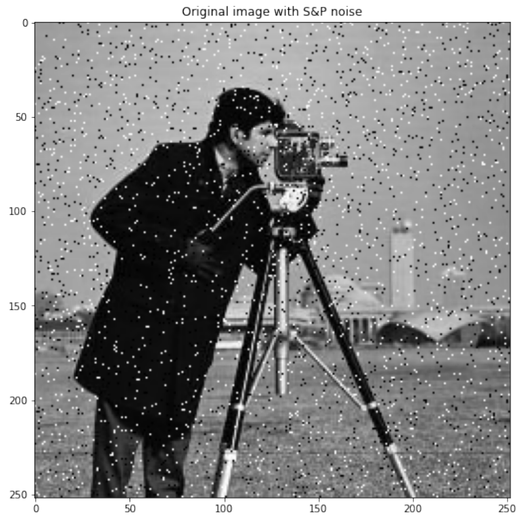
Salt & pepper noise:

d) Apply the three Gaussian filters, described in problem 3, 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?

[10]:
```
######## d) & e)
```

```python
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,h1*h1.T), cmap='gray')
plt.title('Image with salt & pepper noise with Gaussian filter: k=9, sigma=1')
plt.subplot(323), plt.imshow(cv2.filter2D(Im_SP,-1,h15*h15.T), cmap='gray')
plt.title('Image with salt & pepper noise with Gaussian filter: k=11, sigma=1.
↪5')
plt.subplot(324), plt.imshow(cv2.filter2D(Im_SP,-1,h2*h2.T), cmap='gray')
plt.title('Image with salt & pepper noise with Gaussian filter: k=15, sigma=2')
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
0
50
100
150
200
250

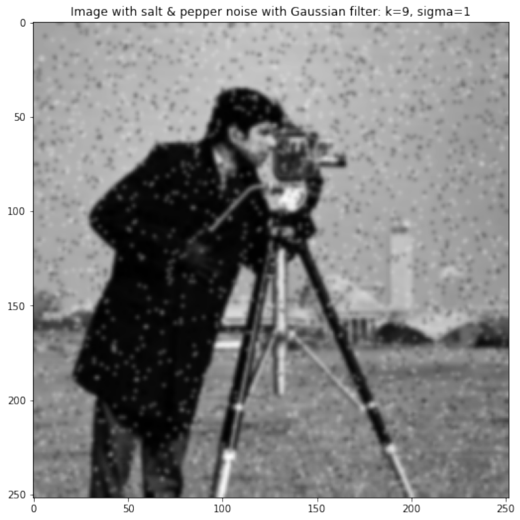


Image with salt & pepper noise with Gaussian filter: k=9, sigma=1
0
50
100
150
200
250

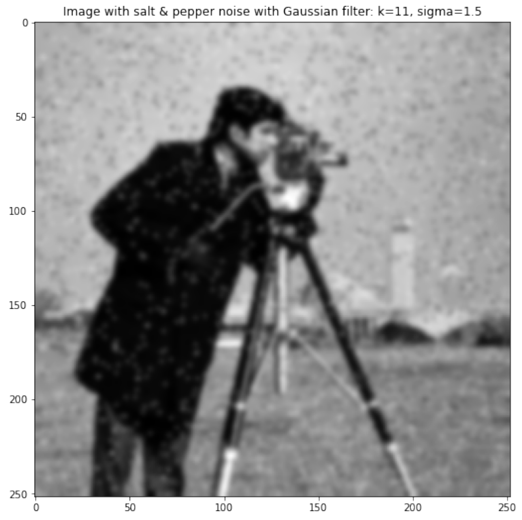


Image with salt & pepper noise with Gaussian filter: k=11, sigma=1.5
0
50
100
150
200
250

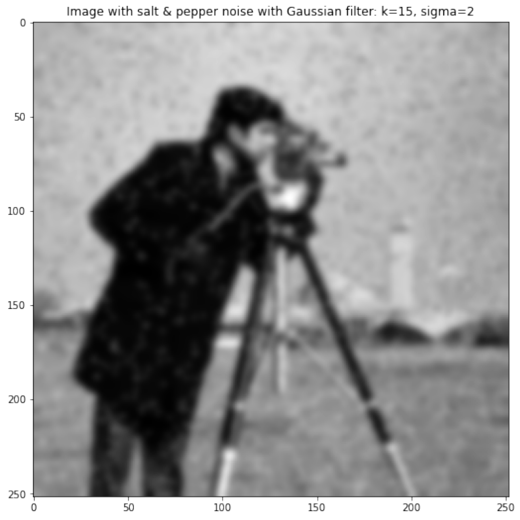
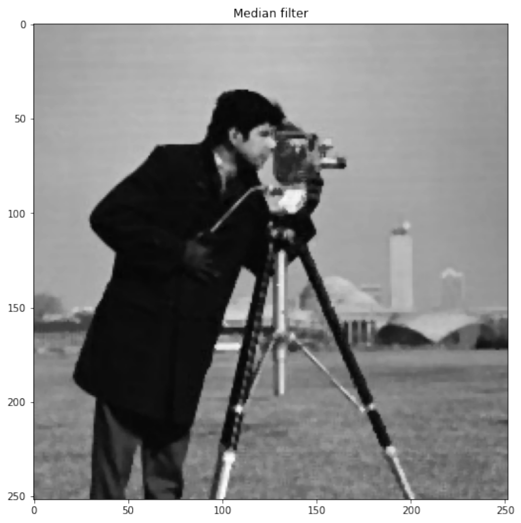


Image with salt & pepper noise with Gaussian filter: k=15, sigma=2
0
50
100
150
200
250


Median filter
0
50
100
150
200
250

```
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.
```

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

Tomasetti Luca, room E-401 E-mail: luca.tomasetti@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.