Digital Image Processing - Lab Session 6

Sofia Noemi Crobeddu
student number: 032410554

International Master of Biometrics and Intelligent Vision
Université Paris-Est Cretéil (UPEC)

December 22, 2024

Contents

Introduction

In this project we analyze different types and methods to filter an image in the context of image processing. In particular, we deepen different versions of Sobel and Prewitt kernels, gradients of an image and spatial domain filters.

1 Operators

In the first part of the task, we applied two filters on T image: **Sobel** and **Prewitt**. The functions used were *sobel_v* and *prewitt_v* for vertical gradients, to highlight horizontal intensity changes in the image. They are two horizontal edge-enhancing operators, and they were applied using *skimage.filters* library. Moreover, after the application of the filters in their default horizontal edge-emphasizing configuration, we created **Sobel kernels** adapted to all four primary directions: North, South, West, and East, using the transpose operator and *rot90* function.
The code to implement the process is shown below, and the resulting images are in Figure 1. As we can see, Sobel filter is less sensitive to noise compared to Prewitt, since it includes Gaussian smoothing.

```
import numpy as np
from PIL import Image
from skimage import filters

#Load image
image_T = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\assignment 6 - IP\\IP6\\IP6\\T.png").convert('L')

#To array
image_array_T = np.array(image_T)

#Application of Sobel e Prewitt filters for horizontal labels
sobel_horiz_T = filters.sobel_v(image_array_T)
prewitt_horiz_T = filters.prewitt_v(image_array_T)

#Defining kernels for Sobel in the 4 directions
sobel_kernels = {
    "North": filters.sobel_v(image_array_T),
    "South": np.flip(filters.sobel_v(image_array_T)),
    "West": np.rot90(filters.sobel_h(image_array_T), k=1),
    "East": np.rot90(filters.sobel_h(image_array_T), k=-1)
}
```

```
import matplotlib.pyplot as plt
from matplotlib.colors import NoNorm
from matplotlib.gridspec import GridSpec

#Creating the grid for the graphs
fig = plt.figure(figsize=(15, 8))
gs = GridSpec(2, 4, figure=fig)

#First row with three columns
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[0, 1])
ax3 = fig.add_subplot(gs[0, 2])

#Second row with four columns
ax4 = fig.add_subplot(gs[1, 0])
ax5 = fig.add_subplot(gs[1, 1])
ax6 = fig.add_subplot(gs[1, 2])
ax7 = fig.add_subplot(gs[1, 3])

#Plot
ax1.imshow(image_T, cmap='gray', norm=NoNorm())
ax1.set_title("T image.")
ax1.axis('off')

ax2.imshow(sobel_horiz_T, cmap='gray')
ax2.set_title("Sobel horizontal for T image.")
ax2.axis('off')

ax3.imshow(prewitt_horiz_T, cmap='gray')
ax3.set_title("Prewitt horizontal for T image.")
ax3.axis('off')

#Labels in different directions
axes = [ax4, ax5, ax6, ax7]
```

```
for i, (direction, result) in enumerate(sobel_kernels.items()):
    axes[i].imshow(result, cmap=’gray’)
    axes[i].set_title(f"Sobel {direction} for T image.")
    axes[i].axis(’off’)

plt.tight_layout()
plt.show()
```

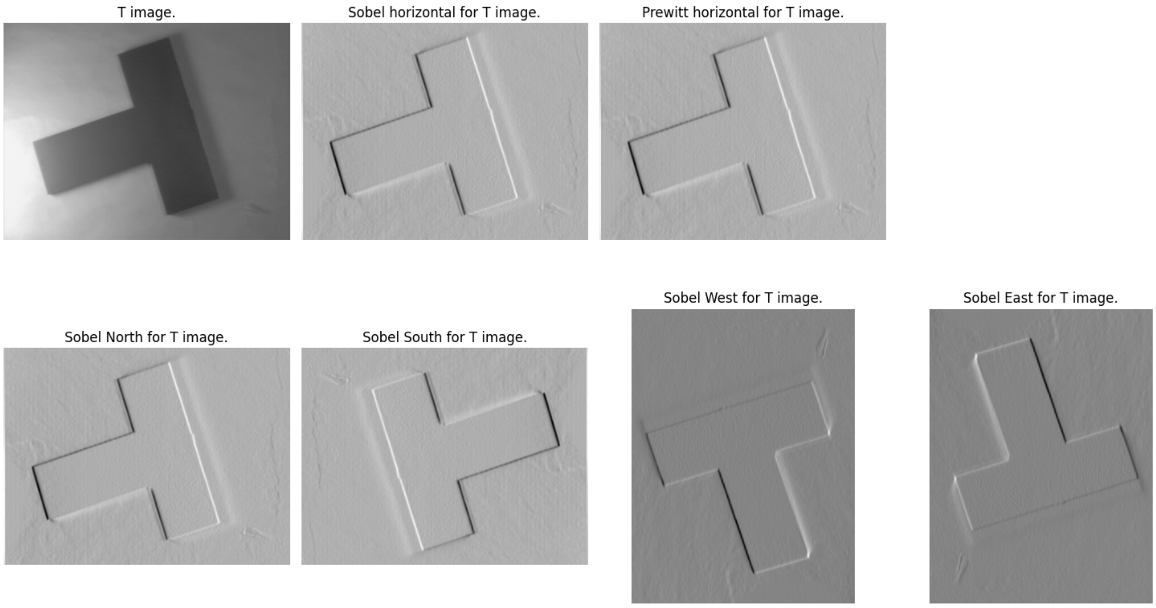

Fig. 1: Sobel with different directions and Prewitt filter on T image.

In the second part of the task T image was convolved with *Sobel kernels* in all the four directions, found before. This problem was addressed using the *mode='nearest'* option, that replicates edge values. The aim was to compute the maximum value at each pixel across all directions to produce the final output image. Moreover, due to memory constraints, T image was resized to half its original dimensions. The maximum value was calculated at each pixel across all directional convolution results using *reduce* function. The code is shown below and the resulting images are in Figure 2.

```
from scipy.ndimage import convolve
from skimage.transform import resize

#Resizing due to memory restructions
image_array_T = resize(image_array_T, (image_array_T.shape[0] // 2, image_array_T.shape[1] // 2))
#Application of the convolution for each direction
convolved_img_near = {}
for direction, kernel in sobel_kernels.items():
    convolved_img_near[direction] = convolve(image_array_T, kernel, mode=’nearest’)

#Combining results taking the maximum value on each pixel
output_img = np.maximum.reduce([img for img in convolved_img_near.values()])
```

```
#Plot
fig, axs = plt.subplots(1, 5, figsize=(20, 10))
#Convolution results
for i, (direction, result) in enumerate(convolved_img_near.items()):
    axs[i].imshow(result, cmap=’gray’)
    axs[i].set_title(f"Convolved image with {direction} direction.")
    axs[i].axis(’off’)

axs[4].imshow(output_img, cmap=’gray’)
axs[4].set_title("Output imgae - maximum of all directions.")
axs[4].axis(’off’)

plt.tight_layout()
plt.show()
```

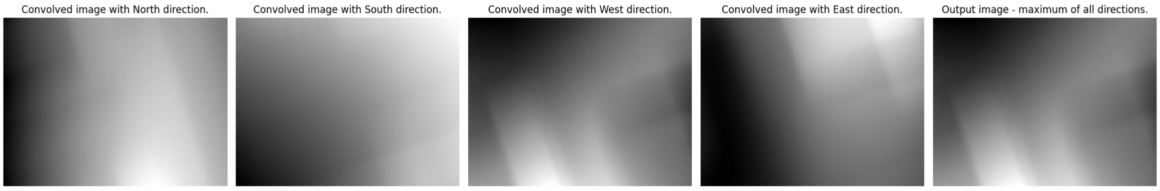

Fig. 2: Convolution of T image with different directions.

2 Gradient operator

In the first part of this task, we compute the gradient magnitude defined as:

$$magnitude = \sqrt{J_x^2 + J_y^2}$$

where J_x and J_y are the partial derivatives, representing gradients along the horizontal and vertical directions respectively.
After the calculation of the magnitude formula, the result is transformed into 8-bit integer type. The resulting magnitude of T image is shown in Figure 3, displayed using a grayscale colormap.

```
from skimage import img_as_float

#We need the image in float format in order to have a precise calculation of the gradients
image_float_T = img_as_float(image_array_T)

#Calculating the partial derivatives on x (horiz.) and y (vert.)
Jx, Jy = np.gradient(image_float_T)

#Magnitude of the gradient
magnitude_T = np.sqrt(Jx**2 + Jy**2)

#Normalizing and converting into uint8 type
magnitude_T_uint8 = np.uint8(np.clip(magnitude_T * 255 / np.max(magnitude_T), 0, 255))

#Plot
plt.figure(figsize=(6, 6))
plt.imshow(magnitude_T_uint8, cmap='gray')
plt.axis('off')
plt.show()
```

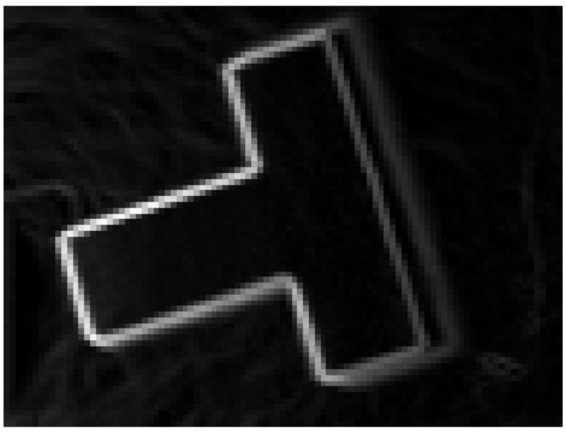

Fig. 3: Magnitude of the gradient of T image.

Continuing with the task, the objective here was to extract edges from T image. *edge* function is actually from MatLab, but we can find an equivalent one in Python from *cv2* library, i.e. *Canny* function. In order to apply it, we need to define two thresholds, a lower and a higher one, for edge detection: they were defined based on intensity percentiles. More specifically, the lower was set as the 10^{th} percentiles of the intensity values in the normalized T image, while the higher one as the 30^{th} percentiles. These values were taken from state-of-the-art ones, i.e. from literature settings. This algorithm first applies Gaussian smoothing to reduce noise, and after computes intensity gradients. Also, only the strongest edges are retained since there is the suppression of non-maximum points. The resulting edge map is displayed in grayscale in Figure 4.

```
import cv2

#Since edge() is a function of MatLab, here we use Canny.
#Calculation of the percentiles for the intensity values in normalized image
low_threshold = np.percentile(image_array_T, 10) #10%
high_threshold = np.percentile(image_array_T, 30) #30%

#Converting the image in uint8
image_T_uint8 = (image_array_T * 255).astype(np.uint8)

#Parameters for the function Canny
edges = cv2.Canny(image_T_uint8, threshold1=low_threshold, threshold2=high_threshold)

#Plot
plt.figure(figsize=(6, 6))
plt.imshow(edges, cmap='gray')
plt.axis('off')
plt.show()
```

Fig. 4: Edges detected in T image.

The last point of this task has the aim of visualizing the gradient vectors of T image, calculated before. For a better visualization, the step size of 10 written in the task was maintained, in order to subsample the gradient vectors and to reduce the number of arrows. Moreover, it was created a meshgrid in order to define the positions of the sampled points.
The gradient vectors at sampled points were plotted over the grayscale image through *quiver* function, where:

- the direction of each vector corresponds to the gradient direction;
- the magnitude of the vector represents the intensity of the gradient.

The final image with the legend of grayscale intensity is shown in Figure 5.

```
step=10 #Interval for the sampling (taken from the code of the task)

#Meshgrid for the points
x, y = np.meshgrid(np.arange(0, image_array_T.shape[1], step),
                   np.arange(0, image_array_T.shape[0], step))

#Subsampling of the gradients
Jx_sub = Jx[::step, ::step]
Jy_sub = Jy[::step, ::step]

```

```python
#Plot
plt.figure(figsize=(8, 8))
plt.imshow(image_array_T, cmap='gray', norm=NoNorm())
plt.colorbar(label="Intensity")
plt.axis('off')
#Adding the vectors of the gradients
plt.quiver(x, y, Jx_sub, -Jy_sub, color='red')
plt.show()
```

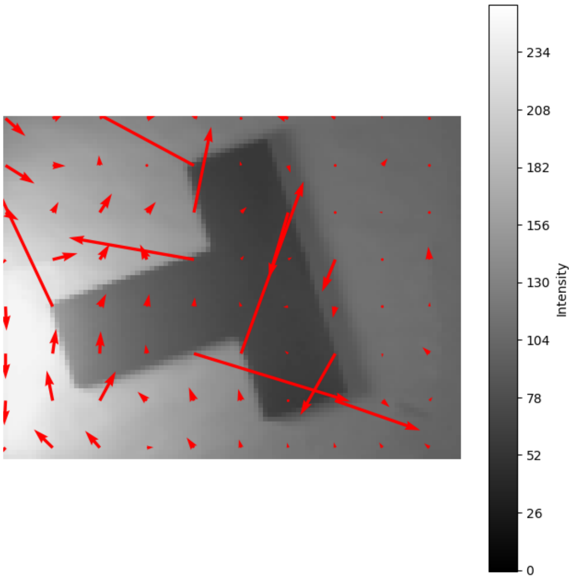

Fig. 5: Gradient vectors in T image.

3 Spatial domain filtering

In this task we filter T image using the convolution with three masks:

- **H1**, the **Averaging filter** for smoothing, defined as

$$H_1 = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

  This filter smooths the image by averaging the intensity values of neighboring pixels. Hence, the image will have reduced noise and softened edges.
- **H2**, the **Vertical gradient filter** for edge detection, defined as

$$H_2 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}.$$

  In this filter, the positive and negative coefficients detect edges along the vertical axis. Hence, the image will have highlighted intensity changes along the vertical axis.
- **H3**, the **Laplacian filter** for enhancing edges and contours, defined as

$$H_3 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$

In this filter, the matrix is designed in order to highlight all edges by emphasizing intensity changes in all directions.

For the convolution between the filter and the image, it was used *convolve2d* function. The resulting images are shown in Figure 6.

```
from scipy.signal import convolve2d

#Defining the three filters
#Averaging filter
H1 = (1/9)*np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 1]])
#Vertical gradient
H2 = np.array([[1, 0, -1],
               [1, 0, -1],
               [1, 0, -1]])
#Laplacian filter
H3 = np.array([[0, -1, 0],
               [-1, 4, -1],
               [0, -1, 0]])

#Applying filters using convolution
conv_H1 = convolve2d(image_array_T, H1, mode='same', boundary='symm')
conv_H2 = convolve2d(image_array_T, H2, mode='same', boundary='symm')
conv_H3 = convolve2d(image_array_T, H3, mode='same', boundary='symm')

#Plot
fig, axs = plt.subplots(1, 4, figsize=(20, 10))

axs[0].imshow(image_array_T, cmap='gray', norm=NoNorm())
axs[0].set_title("T image.")
axs[0].axis('off')

axs[1].imshow(conv_H1, cmap='gray')
axs[1].set_title("T image filtered with Averaging.")
axs[1].axis('off')

axs[2].imshow(conv_H2, cmap='gray')
axs[2].set_title("T image filtered with Vertical gradient.")
axs[2].axis('off')

axs[3].imshow(conv_H3, cmap='gray')
axs[3].set_title("T image filtered with Laplacian.")
axs[3].axis('off')

plt.tight_layout()
plt.show()
```

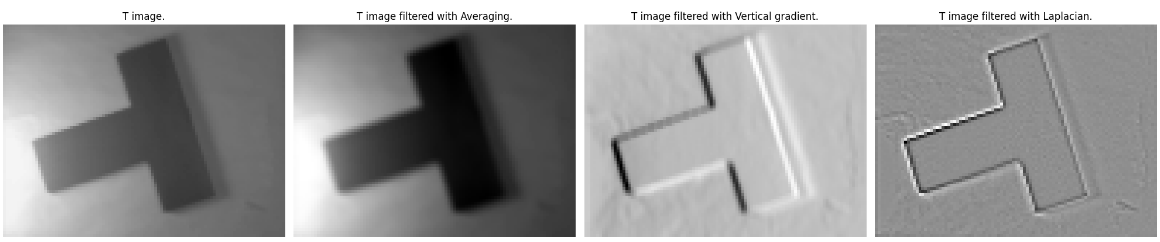

Fig. 6: Filtering in T image.

The second part of this task involves the application of a **median filter** to an image with **salt and pepper noise**. This noise was added using the function *random_noise* and with a probability of 0.2, simulating random black and white pixels. After this, the median filter was applied to the noisy image, in order to remove the noise while preserving edges. This technique replaces each pixel's value with the median of its neighbors within a defined kernel.
The resulting images are shown in Figure 7. As we can notice, the filter effectively removed noise while maintaining edge sharpness.

```
from skimage.util import random_noise
from scipy.ndimage import median_filter
```

```

#Adding 'salt & pepper' noise
noisy_image_T = random_noise(image_T_uint8, mode='s&p', amount=0.02)  #amount=0.02 to simulate the noise
noisy_image_T = (noisy_image_T * 255).astype(np.uint8)  #Conversion in uint8

#Applying the median filter
med_filt_image_T = median_filter(noisy_image_T, size=3)

#Plot
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.title("T image.")
plt.imshow(image_array_T, cmap='gray', norm=NoNorm())
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title("T image with salt & pepper noise.")
plt.imshow(noisy_image_T, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title("Noisy image filtered with median filter.")
plt.imshow(med_filt_image_T, cmap='gray')
plt.axis('off')

plt.tight_layout()
plt.show()
```

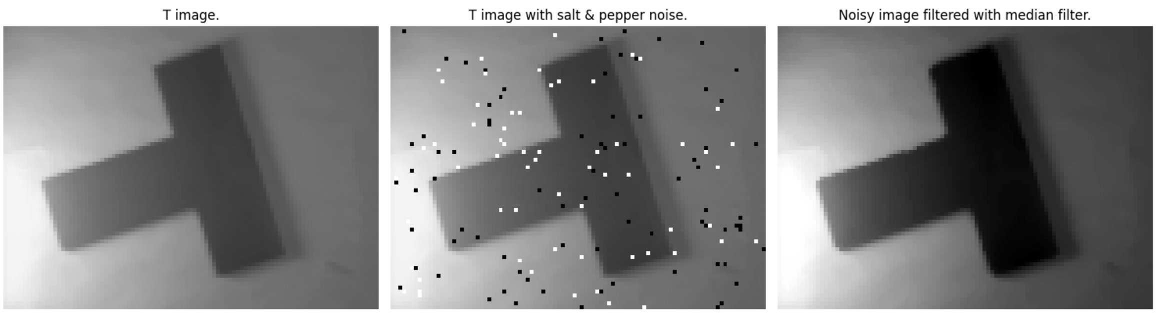

Fig. 7: Median filter on T image.

For this last part of the task, we analyze four grayscale images of the Lena dataset.

- In *lena1* image, a **Gaussian filter** was applied in order to reduce noise and smooth the image. The function used was *gaussian_filter*, with a sigma value of 1.2. This operation blurs the image, softening the transitions between the pixel values.
- In *lena2* image, we first apply a **median filter**, in order to reduce salt and pepper noise. The kernel size for this filter was set to 5. After this, a sharpeness kernel was applied in order to enhance the edges of the image. The sharpeness kernel used is a 3x3 kernel with negative values surrounding a central positive value, defined as:
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$
This kernel was applied using *filter2D* function, that enhances the edges of the image.
- In *lena3* image, first we apply the sharpeness kernel used also before, and after the **averaging filter H1** defined in the initial part of task 3. It helps in smoothing noise by averaging pixel values within a specified neighborhood. The function used was *convolve2d*.
- *lena4* image was denoised using the **Non Local Means (NLM) algorithm** through the function *fastNlMeansDenoising*. It works by averaging the pixels within a local neighborhood, while considering the similarity between the regions of the image. It is effective especially for removing noise

without eliminating details. The hyperparameter h of the function was set to 14 after some trials, in order to allow the use of more dissimilar pixels. A lower value is usually used for more similar ones.

The code to implement the filtering is shown below, and the final images both original and filtered, are shown in Figure 8.

```python
from scipy.ndimage import gaussian_filter

#Load images
image_lena1 = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 6 - IP\\IP6\\IP6\\lena1.tif").convert('L')
image_lena2 = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 6 - IP\\IP6\\IP6\\lena2.tif").convert('L')
image_lena3 = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 6 - IP\\IP6\\IP6\\lena3.tif").convert('L')
image_lena4 = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 6 - IP\\IP6\\IP6\\lena4.tif").convert('L')

#To array
image_array_lena1 = np.array(image_lena1)
image_array_lena2 = np.array(image_lena2)
image_array_lena3 = np.array(image_lena3)
image_array_lena4 = np.array(image_lena4)

#To lena1: gaussian filter
gauss_lena1 = gaussian_filter(image_array_lena1, sigma=1.2)
#To lena2: sharpeness kernel and median filter --> the image has salt & pepper noise.
med_lena2 = median_filter(image_array_lena2, 5) #median filter
kernel_sharpness = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]]) #kernel
med_lena2 = cv2.filter2D(med_lena2, -1, kernel) #sharpen the image
#To lena3: sharpeness kernel and averaging filter
sharpened_lena3 = cv2.filter2D(image_array_lena3, -1, kernel) #sharpen the image
conv_lena3 = convolve2d(sharpened_lena3, H1, mode='same', boundary='symm') #averaging filter
#To lena4: NLM algorithm
conv_H2_lena4 = cv2.fastNlMeansDenoising(image_array_lena4, h=14)

#Plot
fig, axs = plt.subplots(4,2, figsize=(12, 15))

axs[0,0].imshow(image_lena1, cmap='gray', norm=NoNorm())
axs[0,0].set_title("Lena1 image.")
axs[0,1].imshow(gauss_lena1, cmap='gray')
axs[0,1].set_title("Lena1 image with Gaussian filter.")

axs[1,0].imshow(image_lena2, cmap='gray', norm=NoNorm())
axs[1,0].set_title("Lena2 image.")
axs[1,1].imshow(med_lena2, cmap='gray')
axs[1,1].set_title("Lena2 image with sharpeness kernel and median filter.")

axs[2,0].imshow(image_lena3, cmap='gray', norm=NoNorm())
axs[2,0].set_title("Lena3 image.")
axs[2,1].imshow(conv_lena3, cmap='gray')
axs[2,1].set_title("Lena3 image with sharpeness kernel and Averaging filter.")

axs[3,0].imshow(image_lena4, cmap='gray', norm=NoNorm())
axs[3,0].set_title("Lena4 image.")
axs[3,1].imshow(conv_H2_lena4, cmap='gray')
axs[3,1].set_title("Lena4 image with Non-Local Means filter.")

plt.tight_layout()
plt.show()
```

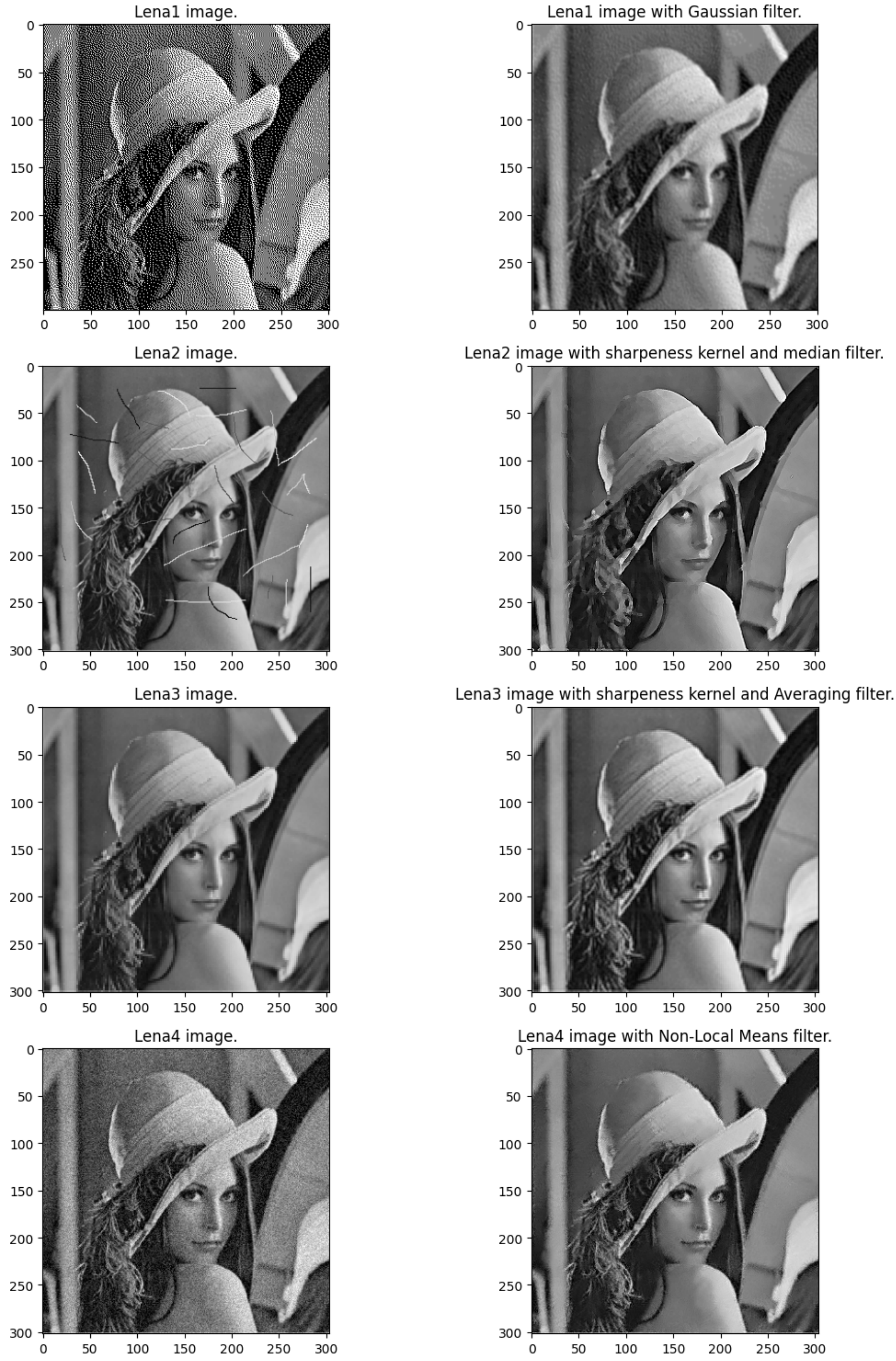

Fig. 8: Filtering on *lena* images.

References

[1] Geveo: Image Smoothing Algorithms. https://blog.geveo.com/Image-Smoothing-Algorithms (2024)

[2] Dilhani, S.: Digital Image Processing Filters. https://medium.com/@shashikadilhani97/digital-image-processing-filters-832ec6d18a73 (2024)

[3] YouTube: Image of YouTube Video. https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.youtube.com%2Fwatch%3Fv%3DB7FIuz8vAkE&psig=AOvVaw39hZ5T4yQmQlxTA4LAAaq8&ust=1734978308724000&source=images&cd=vfe&opi=89978449&ved=0CBcQjhxqFwoTCMDgoIKAvIoDFQAAAAAdAAAAABAE (2024)

[4] YouTube: B7FIuz8vAkE Video on YouTube. https://www.youtube.com/watch?v=B7FIuz8vAkE (2024)

[5] User, S.O.: How do I to implement Matlab's ‘edge‘ function in Python (OpenCV or skimage)? https://stackoverflow.com/questions/61272291/how-do-i-to-implement-matlabs-edge-function-in-python-opencv-or-skimage (2020)

[6] Majidzadeh, F.: Digital Image Processing and Pattern Recognition, (2023)

[7] GeeksforGeeks: Image Enhancement Techniques Using OpenCV Python. https://www.geeksforgeeks.org/image-enhancement-techniques-using-opencv-python/ (2024)

[8] Vidhya, A.: Sharpening an Image Using OpenCV Library in Python. https://www.analyticsvidhya.com/blog/2021/08/sharpening-an-image-using-opencv-library-in-python/ (2021)