Exercise 2

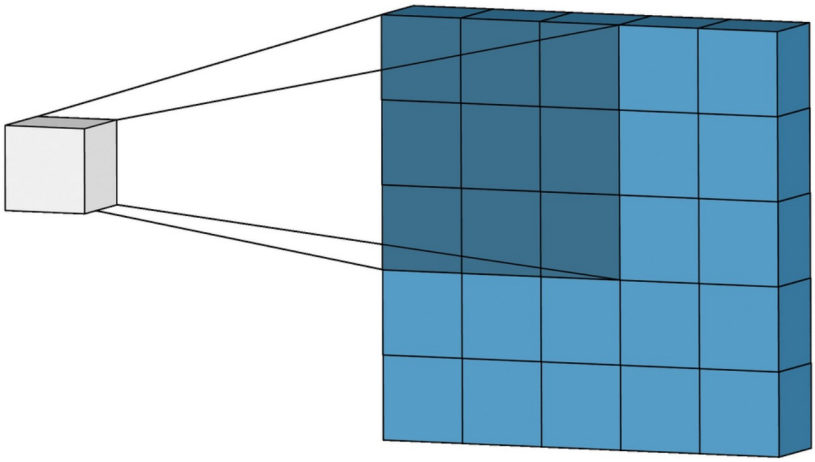
Computer Vision, Fall 2025

Name: Matias Paloranta

Instructions:

- Return the answer in PDF and Jupyter Notebook formats.
- Return latest on Sunday 21.9 at 23.50 via Moodle.

Ex 2.1 Edge detection and image filtering (3 points)



(a) Implement convolution function

- Implement convolution function to filter images with the user-defined kernel.
- Expect kernel sizes to be 3x3, 5x5, 7x7 or 9x9.
- In this exercise, we use only 1-channel grayscale images.
- Convolution function will be used in exercises 2.1b and 2.1c.

In [1]:
```
# Import Python packages needed in exercise 2.1
from matplotlib import pyplot as plt
import numpy as np
import cv2
```

In [2]:
```
def convolution(img, kernel):
    """
    Args:
```

```
        img: 1-channel grayscale image
        kernel: kernel to use in convolution.

    Returns:
        Convolved image
    """
    # below we will implement valid convolution only since it wasn't specified i
    # exercise if padding was required

    img_h, img_w = img.shape
    kernel_h, kernel_w = kernel.shape
    # for image of height/width x and kernel of height/width y
    # the kernel fits inside the image dimensions x - y + 1 times
    # this way we get the dimensions of the convolved (output) image
    output_h = img_h - kernel_h + 1
    output_w = img_w - kernel_w + 1

    output = np.zeros((output_h, output_w))

    for i in range(output_h):
        for j in range(output_w):
            # select the region in the image covered by the kernel
            image_region = img[i: i + kernel_h, j: j + kernel_w]
            output[i, j] = np.sum(np.multiply(image_region, kernel))

    return output
```

(b) Implement create_box_filter function and filter image

- Implement function to create different size box filters
- Read image "images/norway.jpeg" and convert it to grayscale
- Use your convolution function to filter the image with box filters
- Try different box filter sizes. How does the box size affect the result?

In [3]:
```
def create_box_filter(size):
    """
    Args:
        size: Integer that defines size of the box filter.

    Returns:
        box filter with shape size x size.
    """
    box_filter = np.ones((size, size))
    box_filter = box_filter/np.sum(box_filter)
    return box_filter
```

In [4]:
```
def show_image(img, title):
    """For less verbose plotting."""

    plt.imshow(img, cmap="gray")
    plt.title(title)
    plt.axis("off")
    plt.show()
```

In [5]:
```
img = cv2.imread("images/norway.jpg", cv2.IMREAD_GRAYSCALE)

box_filters = [create_box_filter(i) for i in [3, 5, 7, 9]]
```

```
box_filtered = [convolution(img, box_filter) for box_filter in box_filters]
imgs = [img] + box_filtered
titles = ["Original"] + [f"Box filter {size}" for size in "3x3 5x5 7x7 9x9".spli

for img, title in zip(imgs, titles):
    show_image(img, title)
```

Box filter 5x5

Box filter 7x7

The filtered image appears more blurred as the size of the box filter is increased.

(c) Implement create_gaussian_filter function and filter image

- Implement function to create a gaussian filters
- Read image images/norway.jpeg and convert it to grayscale
- Use your convolution function to filter the image with the different gaussian filters
- How does the change of sigma value and kernel size affect the results? Try with different values.

In [6]:
```python
def create_gaussian_filter(size, sigma):
    """
    Args:
        size: Integer that defines size of the gaussian filter.
        sigma: Standard deviation of Gaussian distribution

    Returns:
        gaussian filter with shape size x size and user defined sigma
    """

    gaussian_filter = np.zeros((size, size))
    # we want to center the filter around 0 such that e.g. index array [0 1 2 3
    c = size // 2
    for idx, _ in np.ndenumerate(gaussian_filter):
        d = (idx[0] - c)**2 + (idx[1] - c)**2
        gaussian_filter[idx] = np.exp(-d / (2 * sigma**2))

    # normalize such that the filter sums to 1
    gaussian_filter = gaussian_filter/np.sum(gaussian_filter)
    return gaussian_filter
```

In [7]:
```python
img = cv2.imread("images/norway.jpg", cv2.IMREAD_GRAYSCALE)
```

```
filters = [create_gaussian_filter(i, 1) for i in [3, 5, 7, 9]]
filtered = [convolution(img, gaussian_filter) for gaussian_filter in filters]
imgs = [img] + filtered
titles = ["Original"] + \
    [f"Gaussian filter {size}, " + r"$\sigma = 1$" for size in "3x3 5x5 7x7 9x9"

for img, title in zip(imgs, titles):
    show_image(img, title)
```

Gaussian filter 5x5, $\sigma = 1$

Gaussian filter 7x7, $\sigma = 1$

Changing the size of the Gaussian filter while keeping σ constant does not appear to change the convolved image much.

In [8]:
```
img = cv2.imread("images/norway.jpg", cv2.IMREAD_GRAYSCALE)
sigmas = np.linspace(1.0, 3.0, 4)
filters = [create_gaussian_filter(9, i) for i in np.linspace(1.0, 3.0, 4)]
filtered = [convolution(img, gaussian_filter) for gaussian_filter in filters]
imgs = [img] + filtered
titles = ["Original"] + \
    [f"Gaussian filter 9x9, " + r"$\sigma = $" + f"{sigma:.3f}" for sigma in sig

for img, title in zip(imgs, titles):
    show_image(img, title)
```

Original

Gaussian filter 9x9, $\sigma = 1.000$

Gaussian filter 9x9, $\sigma = 1.667$

Gaussian filter 9x9, $\sigma = 2.333$

Increasing σ makes the image more blurred, since the Gaussian distribution becomes wider thus making the values on the kernel edges contribute more.

(d) Edge detection

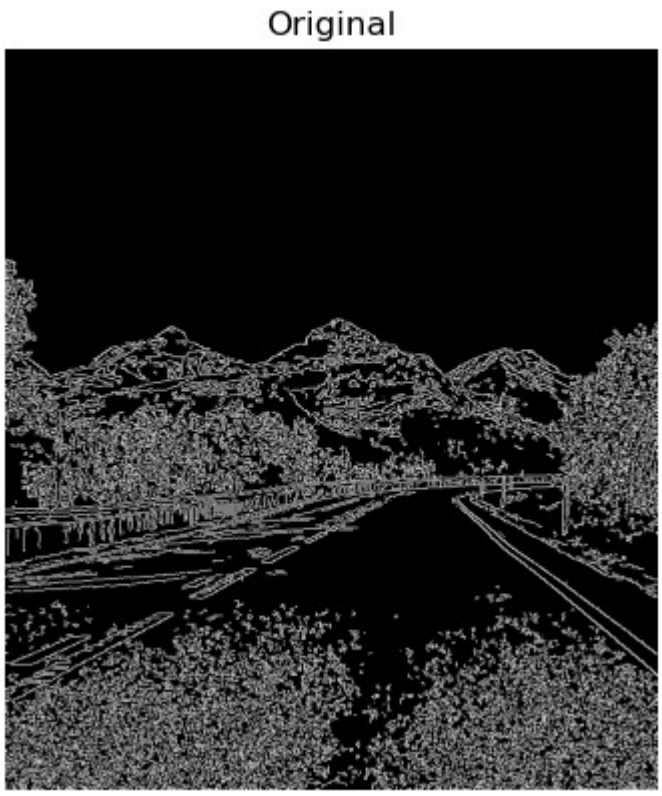
- Search edges from the original image and from all filtered images using OpenCV Canny edge detector. Visualize results.
- Which filter provides the best result? Why? Consider the best result to be one that could be used by the autonomous vehicle to stay in its lane (clear line markings and minimal amount of image noise).

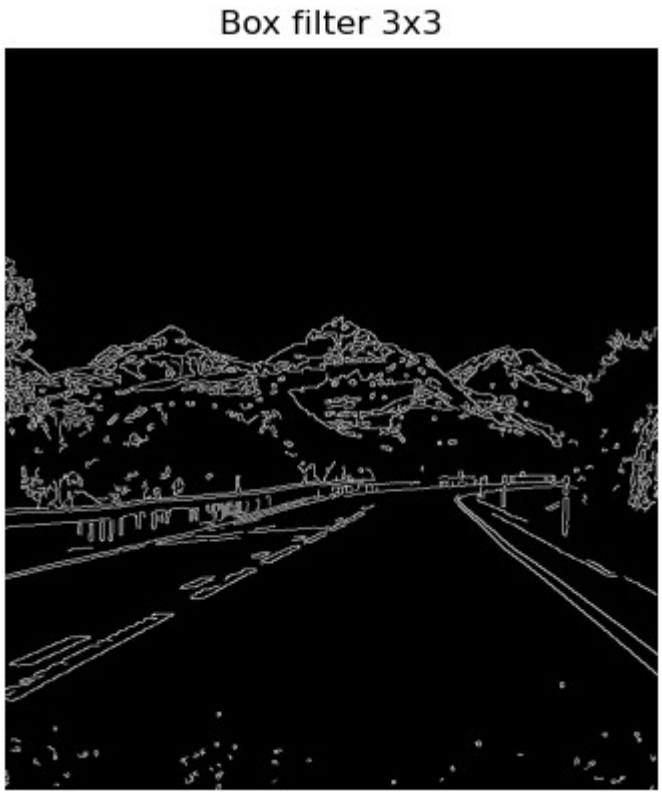
Hint:

- OpenCV Canny Edge Detection

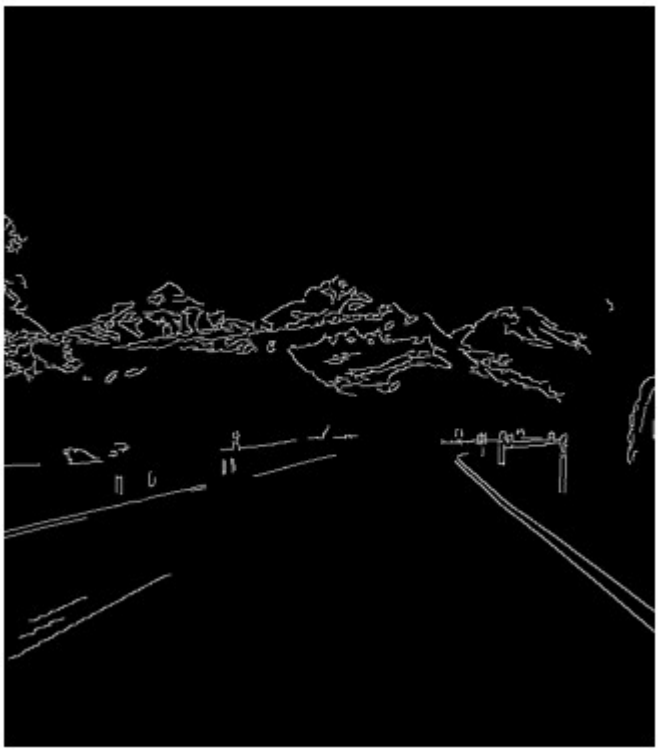
In [9]:
```
img = cv2.imread("images/norway.jpg", cv2.IMREAD_GRAYSCALE)
edges = cv2.Canny(img, 100, 200)
show_image(edges, title="Original")

sizes = [3, 5, 7, 9]

for size in sizes:
    kernel = create_box_filter(size)
    # throws error without casting to uint8
    filtered = convolution(img, kernel).astype(np.uint8)
    edges = cv2.Canny(filtered, 100, 200)
    show_image(edges, title=f"Box filter {size}x{size}")

#sigmas = np.linspace(1.0, 2.0, 4)
sigmas = [1.15]*4
for size, sigma in zip(sizes, sigmas):
    kernel = create_gaussian_filter(size, sigma)
```

```
    filtered = convolution(img, kernel).astype(np.uint8)
    edges = cv2.Canny(filtered, 100, 200)
    t = f"Gaussian filter {size}x{size}, " + r"$\sigma$ = " + f"{sigma:.3f}"
    show_image(edges, title=t)
```





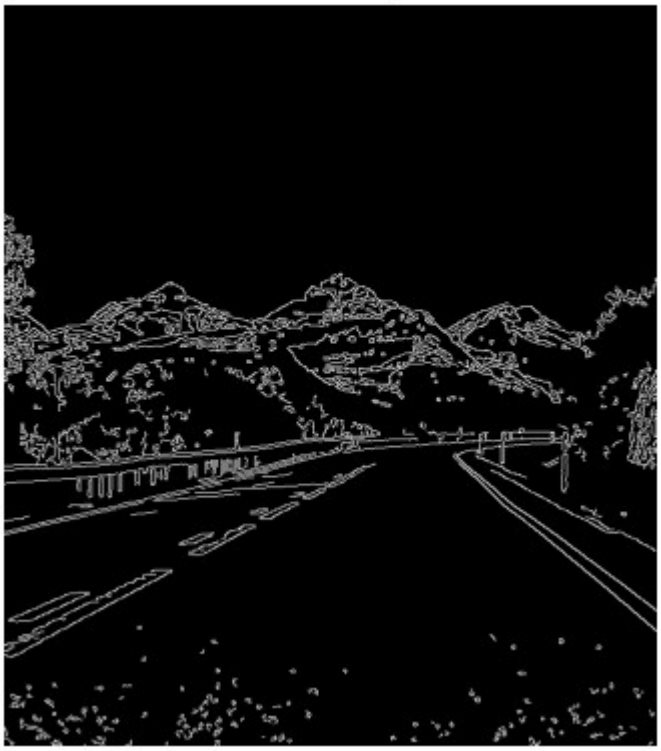
Box filter 5x5



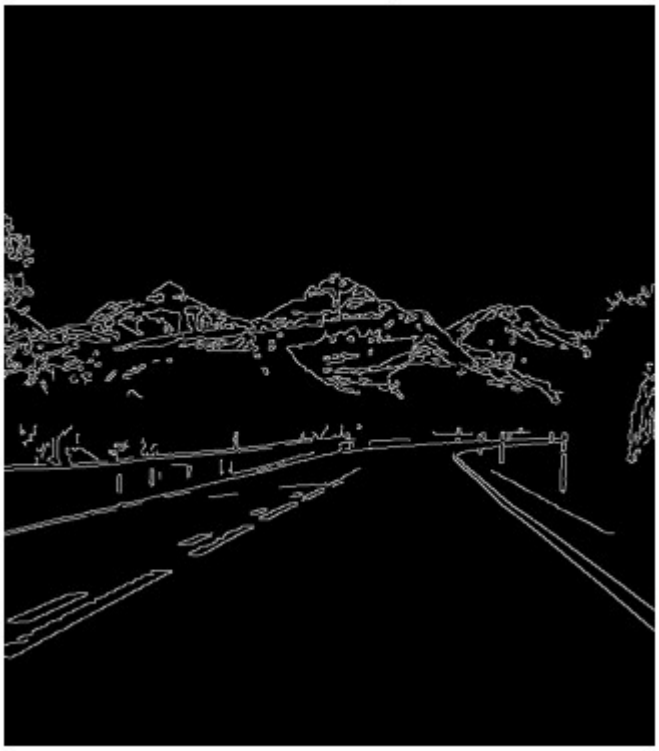
Box filter 7x7

Box filter 9x9

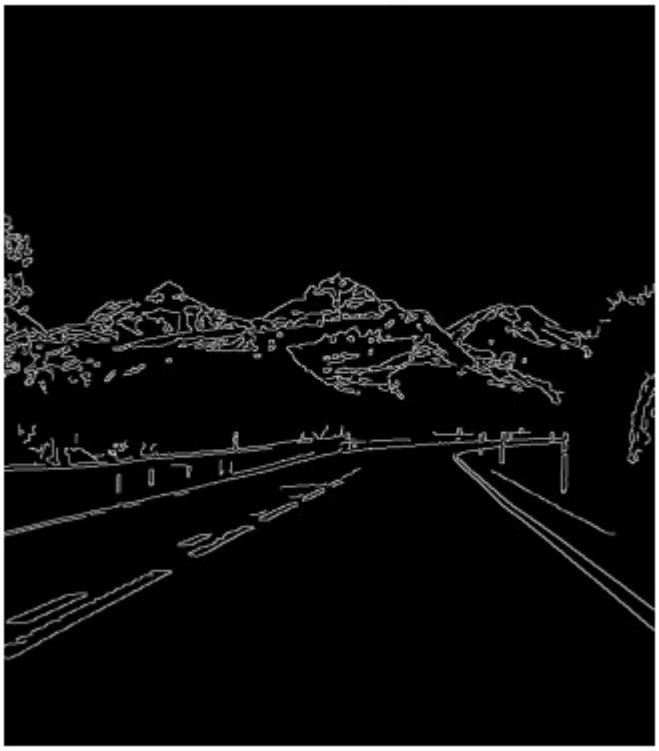
Gaussian filter 3x3, $\sigma = 1.150$

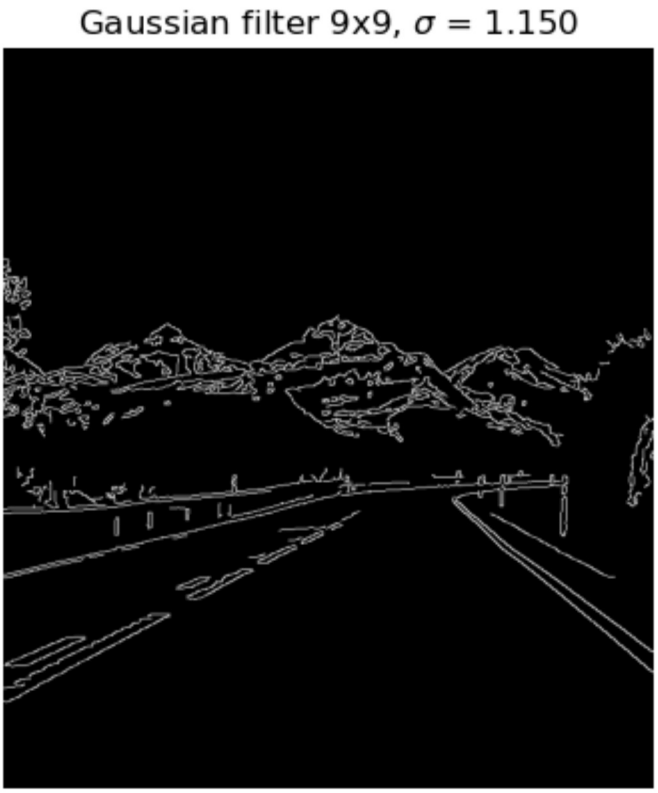


Gaussian filter 5x5, $\sigma = 1.150$



Gaussian filter 7x7, $\sigma = 1.150$





After messing around with σ values I would say that the best result is a Gaussian filter bigger than 3x3 and σ around 1.1-1.2. It preserves the lane markings quite well while removing a lot of the noise.

Ex 2.2 Detect and match features (3 p)

(a) Detect and visualize SIFT features

- Read images "images/GOPR1515_03850" and "images/GOPR1515_03852" and convert them to grayscale
- Run OpenCV SIFT feature detection
- Visualize found SIFT keypoints from both images

Hints:

- We provide you a function visualize_features which can be used to visualize your results.
- Depending on your OpenCV version, you may need to install OpenCV contrib package: Link to opencv-contrib-python
- OpenCV material that may be useful:
  - OpenCV cv::KeyPoint Class Reference
  - OpenCV cv::Feature2D Class Reference
  - Introduction to SIFT (Scale-Invariant Feature Transform in OpenCV)

In [10]:
```
def visualize_features(image, kp):
    """
    Visualize extracted features in the image
```

```
    Arguments:
       image: a grayscale image
       kp: list of the extracted keypoints

    Returns:
    """
    display = cv2.drawKeypoints(image, kp, None, color=(0,255,0), flags=0)
    plt.figure(figsize=(8, 6), dpi=100)
    plt.axis("off")
    plt.imshow(display)
```

In [11]:
```
img1 = cv2.imread("images/GOPR1515_03850.jpg", cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread("images/GOPR1515_03852.jpg", cv2.IMREAD_GRAYSCALE)

for i, img in enumerate([img1, img2]):
    sift = cv2.SIFT_create()
    kp = sift.detect(img, None)
    print(f"{len(kp)} keypoints in image {i + 1}")
    visualize_features(img, kp)
```

```
503 keypoints in image 1
455 keypoints in image 2
```

(b) Feature matching

- Implement feature matching algorithm (you can use OpenCV matching algorithm only to verify your results).
- How good is the result? How could you improve it?

Hint:

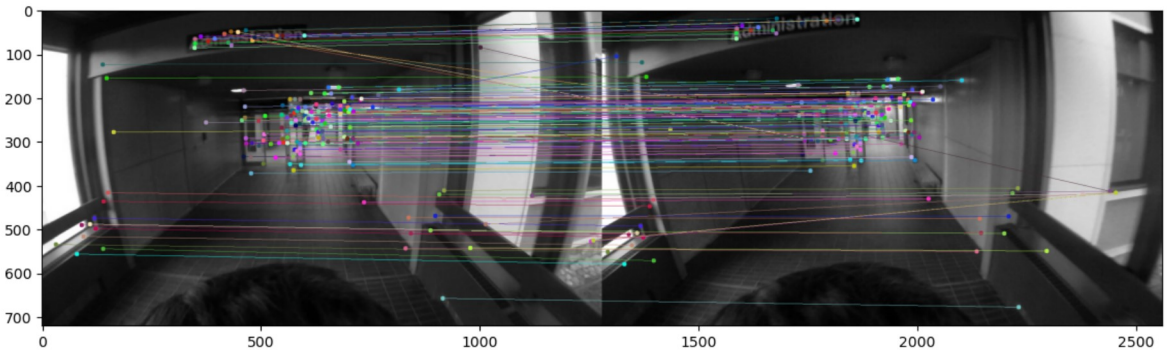
- We provide you a function visualize_matches which can be used to visualize your results.
- OpenCV Feature Matching tutorial. Do not use ready made matching algorithm in this exercise!

In [12]:
```
import random

def visualize_matches(image01, kp01, image02, kp02, matches):
    """
    Visualize corresponding matches in two images

    Arguments:
        image1: the first image in a matched image pair
        kp1: list of the keypoints in the first image
        image2: the second image in a matched image pair
        kp2: list of the keypoints in the second image
        matches: list of keypoint index pairs of matched features. Example forma

    Returns:
    """
    # Get width of the first image
    _ , width = image01.shape

    # Concatenate images
    img = cv2.hconcat([image01, image02])

    # Convert image from gray to RGB
    img = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
```

```
    # Add matches to image
    for match in matches:

        # Get random color
        r = random.randint(0,255)
        g = random.randint(0,255)
        b = random.randint(0,255)
        color = (r,g,b)

        # Get coordinates
        point_img01 = kp01[match[0]].pt
        point_img02 = kp02[match[1]].pt

        # Get x and y values from coordinates
        x_img01 = int(point_img01[0])
        y_img01 = int(point_img01[1])
        x_img02 = int(point_img02[0]) + width
        y_img02 = int(point_img02[1])

        # Draw small circle to matched keypoints
        cv2.circle(img, (x_img01, y_img01), radius=5, color=color, thickness=-1)
        cv2.circle(img, (x_img02, y_img02), radius=5, color=color, thickness=-1)

        # Draw line between matches
        cv2.line(img, (x_img01, y_img01), (x_img02, y_img02), color, thickness=1

    # Show image with matches
    plt.figure(figsize=(14, 14), dpi=100)
    plt.imshow(img)

# Dummy matches to test visualize_matches function.
# You can try it after you have read images and extracted keypoints in exercise
#matches = [(382, 422), (384, 424), (368, 407)]
#visualize_matches(img01, keypoints01, img02, keypoints02, matches)
```

In [13]:

```
kp1, desc1 = sift.detectAndCompute(img1, None)
kp2, desc2 = sift.detectAndCompute(img2, None)

def brute_force_feature_match(desc1, desc2, threshold=0.75):
    """Returns list of (idx_in_des_a, idx_in_des_b) with Lowe's ratio test."""
    matches = []

    for i, desc in enumerate(desc1):
        # Euclidean distance to all descriptors in desc2
        dists = np.sum((desc - desc2) ** 2, axis=1)
        # find the indices of the closest and second closest in terms of distanc
        idx_sorted = np.argsort(dists)
        best, second = idx_sorted[0], idx_sorted[1]
        ratio = np.sqrt(dists[best]) / (np.sqrt(dists[second]))
        if ratio < threshold:
            matches.append((i, best))

    return matches

matches = brute_force_feature_match(desc1, desc2)
# visualize every 10th of the matches
visualize_matches(img1, kp1, img2, kp2, matches)
```



The result is quite good, but there are some outliers. It could maybe be improved by e.g. using different ratio thresholds.

Ex 2.3: Fourier Transform (2 points)

Helpful resources:

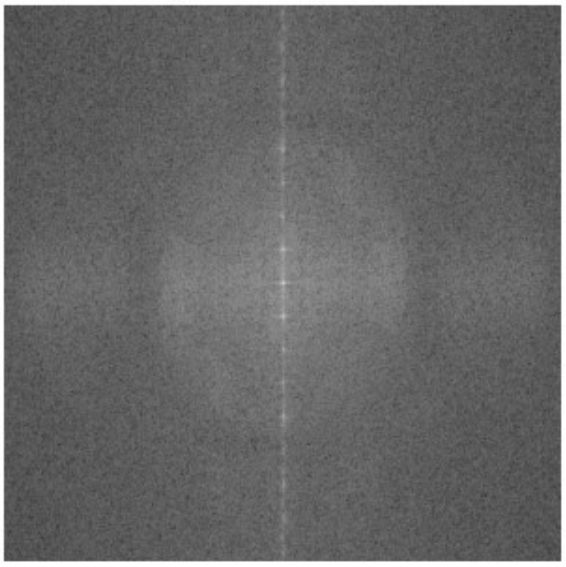
- You may utilize either OpenCV or NumPy for FFT and image rotation operations. You can use ready-made functions, don't have to implement FFT from scratch.
- Refer to this link for a detailed OpenCV Fourier Transform tutorial: OpenCV Fourier Transform Tutorial

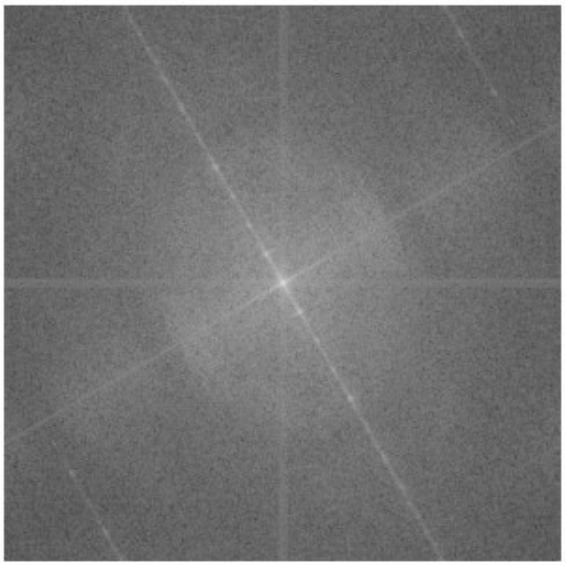
Task (a): Evaluating Image Orientation in Signal Space

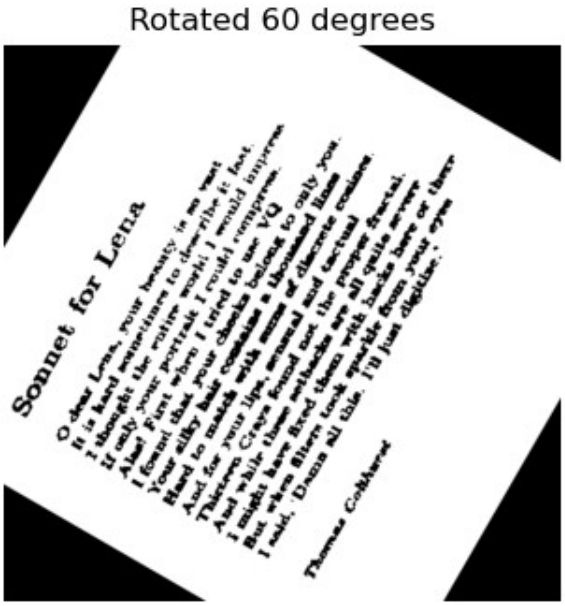
1. Load the image 'images/son3.png'.
2. Create three copies of the image: one in the original orientation, one rotated by 30 degrees, another by 60 degrees, and the last one by 90 degrees.
3. Perform the Fast Fourier Transform (FFT) followed by an FFT shift on all four images. Then visualize their magnitude spectrums.
4. Observe and describe the orientation of the text in the magnitude images.
5. Explain what causes the appearance of horizontal and vertical lines in the FFT magnitude spectrum. What is the x-axis and y-axis in the frequency domain?

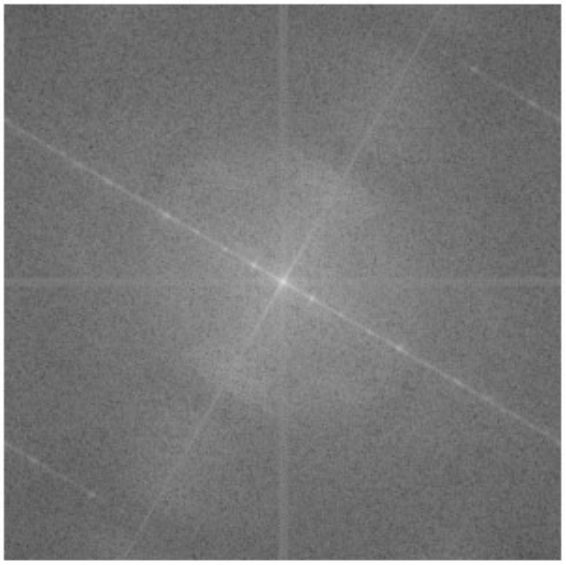
In [14]:
```
img = cv2.imread("images/son3.png", cv2.IMREAD_GRAYSCALE)

def rotate_image(img, angle):
    h, w = img.shape
    center = (w // 2, h // 2)
    rotation_matrix = cv2.getRotationMatrix2D(center, angle, scale=1.0)
    rotated_image = cv2.warpAffine(img, rotation_matrix, (w, h))
    return rotated_image

def visualize_magnitude_spectrum(img, title):
    f = np.fft.fft2(img)
    fshift = np.fft.fftshift(f)
    magnitude_spectrum = 20*np.log(np.abs(fshift))

    fig, ax = plt.subplots(1, 2)
    for i, title, im in zip(range(2), [title, "Magnitude spectrum"], [img, magni
        ax[i].imshow(im, cmap="gray")
        ax[i].set_title(title)
```

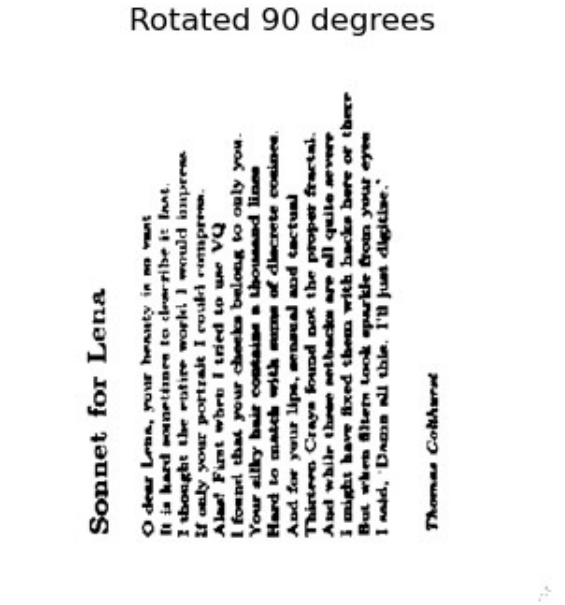
```
        ax[i].axis("off")
    plt.tight_layout()
    plt.show()

visualize_magnitude_spectrum(img, title="Original")

rot30 = rotate_image(img, 30)
visualize_magnitude_spectrum(rot30, title="Rotated 30 degrees")

rot60 = rotate_image(img, 60)
visualize_magnitude_spectrum(rot60, title="Rotated 60 degrees")

rot90 = rotate_image(img, 90)
visualize_magnitude_spectrum(rot90, title="Rotated 90 degrees")
```

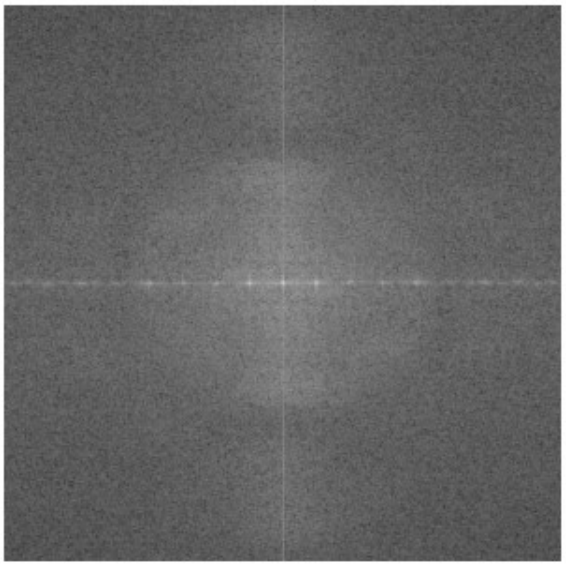












Answer:

4. The text is perpendicular to the main vertical line going through the center, i.e. horizontal in the original image. The line is rotated alongside the image and the text orientation is perpendicular to it.

5. The horizontal and vertical lines describe strong frequency components in those directions. The x axis represents frequencies along image columns and y axis along image rows.

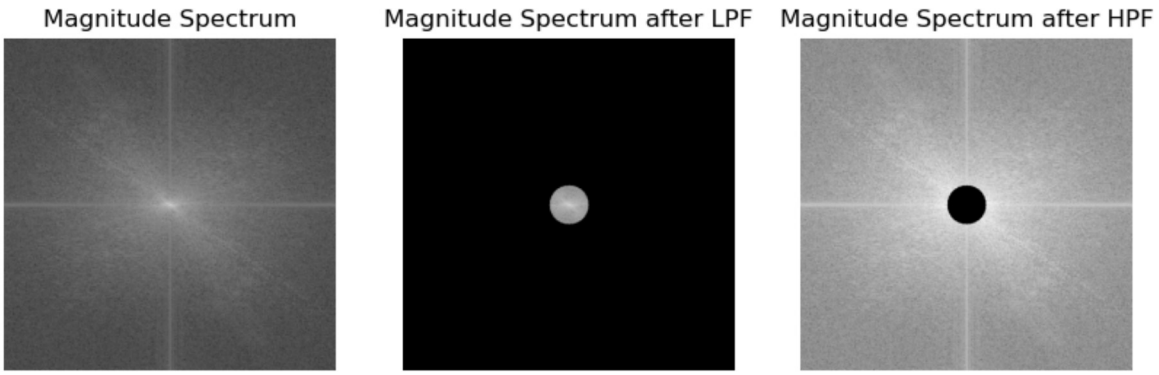
Task (b) High Pass Filter and Low Pass Filter

1. Open the image 'images/lenna.png' and convert it into grayscale.
2. Conduct a Fast Fourier Transform (FFT) on this image.
3. Next, create two copies of the Fourier magnitude spectrum. For one copy, apply a low pass filter (LPF) and for the other, apply a high pass filter (HPF). In both cases, utilize a circular mask with a 30-pixel radius for the filtering process. You can use the

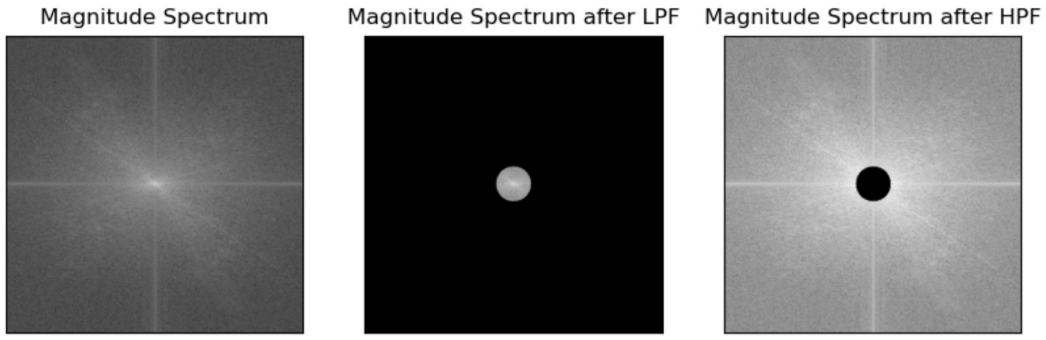
function create_circular_mask() below to create the mask for the filters.

In [15]:
```python
def create_circular_mask(img, radius):
    # Get image size
    h, w = img.shape

    # Get center coordinate
    center = (int(w/2), int(h/2))

    # Make open multi-dimensional "meshgrid"
    Y, X = np.ogrid[:h, :w]

    # Calculate distance from the image center to every pixel
    dist_from_center = np.sqrt((X - center[0])**2 + (Y-center[1])**2)

    # Threshold the result with the user defined radius
    mask = dist_from_center <= radius

    # Convert mask from boolean array to integer array
    return mask.astype(int)
```

In [16]:
```python
img = cv2.imread("images/lenna.png", cv2.IMREAD_GRAYSCALE)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
magnitude_spectrum = 20*np.log(np.abs(fshift))

radius = 30
mask = create_circular_mask(img, radius)

lpf = mask
hpf = 1 - mask

fshift_lpf = fshift * lpf
fshift_hpf = fshift * hpf
ms_lpf = 20*np.log(np.abs(fshift_lpf) + 1) # add 1 to avoid zero division
ms_hpf = 20*np.log(np.abs(fshift_hpf) + 1)

fig, ax = plt.subplots(1, 3, figsize=(10,10))
titles = ["Magnitude Spectrum", "Magnitude Spectrum after LPF", "Magnitude Spect
spectra = [magnitude_spectrum, ms_lpf, ms_hpf]
for i, s, title in zip(range(3), spectra, titles):
    ax[i].imshow(s, cmap="gray")
    ax[i].set_title(title)
    ax[i].axis("off")

plt.show()
```



4. Below are images of what the magnitude spectrum should look like before and after applying the the filters. Briefly explain what the filters are actually doing to the image. How do you expect it to change the image after the image is converted back from the frequency domain?



Answer:

4. The low pass filter allows only low frequencies to pass and removes high frequencies, and vice versa for the high pass filter. High frequencies correspond to fast changes (e.g. edges) so I would expect the reconstructed image to look smoothed/blurred after LPF and edges to look enhanced after HPF.

5. Perform an inverse Fourier transform on the spectra that have been filtered.
6. Finally, observe and discuss the impact that the LPF and HPF have had on the image based on the answer you gave in part 4. of the assignment. What would be a possible application for using the High Pass Filter or the Low Pass Filter?

In [17]:
```
f_lpf = np.fft.ifftshift(fshift_lpf)
img_lpf = np.fft.ifft2(f_lpf)
img_lpf = np.real(img_lpf)

f_hpf = np.fft.ifftshift(fshift_hpf)
img_hpf = np.fft.ifft2(f_hpf)
img_hpf = np.real(img_hpf)

imgs = [img] + [img_lpf, img_hpf]
titles = ["Original", "After LPF", "After HPF"]
fig, ax = plt.subplots(1, 3, figsize=(10, 10))
for i, im, title in zip(range(3), imgs, titles):
    ax[i].imshow(im, cmap="gray")
    ax[i].set_title(title)
    ax[i].axis("off")
plt.show()
```

Answer:

6. The low pass filter behaved pretty much as expected. However, the high pass filter resembles more edge detection than just sharpening the edges. And after a quick Google search confirmed that this is an actual use case for HPF. LPF can be used for e.g. denoising like the Gaussian filter.