Coursework 1: Image filtering

In this coursework you will practice techniques for image filtering. The coursework includes coding questions and written questions. Please read both the text and the code in this notebook to get an idea what you are expected to implement.

What to do?

- Complete and run the code using `jupyter-lab` or `jupyter-notebook` to get the results.
- Export (File | Save and Export Notebook As...) the notebook as a PDF file, which contains your code, results and answers, and upload the PDF file onto Scientia.
- Instead of clicking the Export button, you can also run the following command instead: `jupyter nbconvert coursework_01_solution.ipynb --to pdf`
- If Jupyter complains about some problems in exporting, it is likely that pandoc (https://pandoc.org/installing.html) or latex is not installed, or their paths have not been included. You can install the relevant libraries and retry. Alternatively, use the Print function of your browser to export the PDF file.
- If Jupyter-lab does not work for you at the end (we hope not), you can use Google Colab to write the code and export the PDF file.

Dependencies:

You need to install Jupyter-Lab (https://jupyterlab.readthedocs.io/en/stable/getting_started/installation.html) and other libraries used in this coursework, such as by running the command: `pip3 install [package_name]`

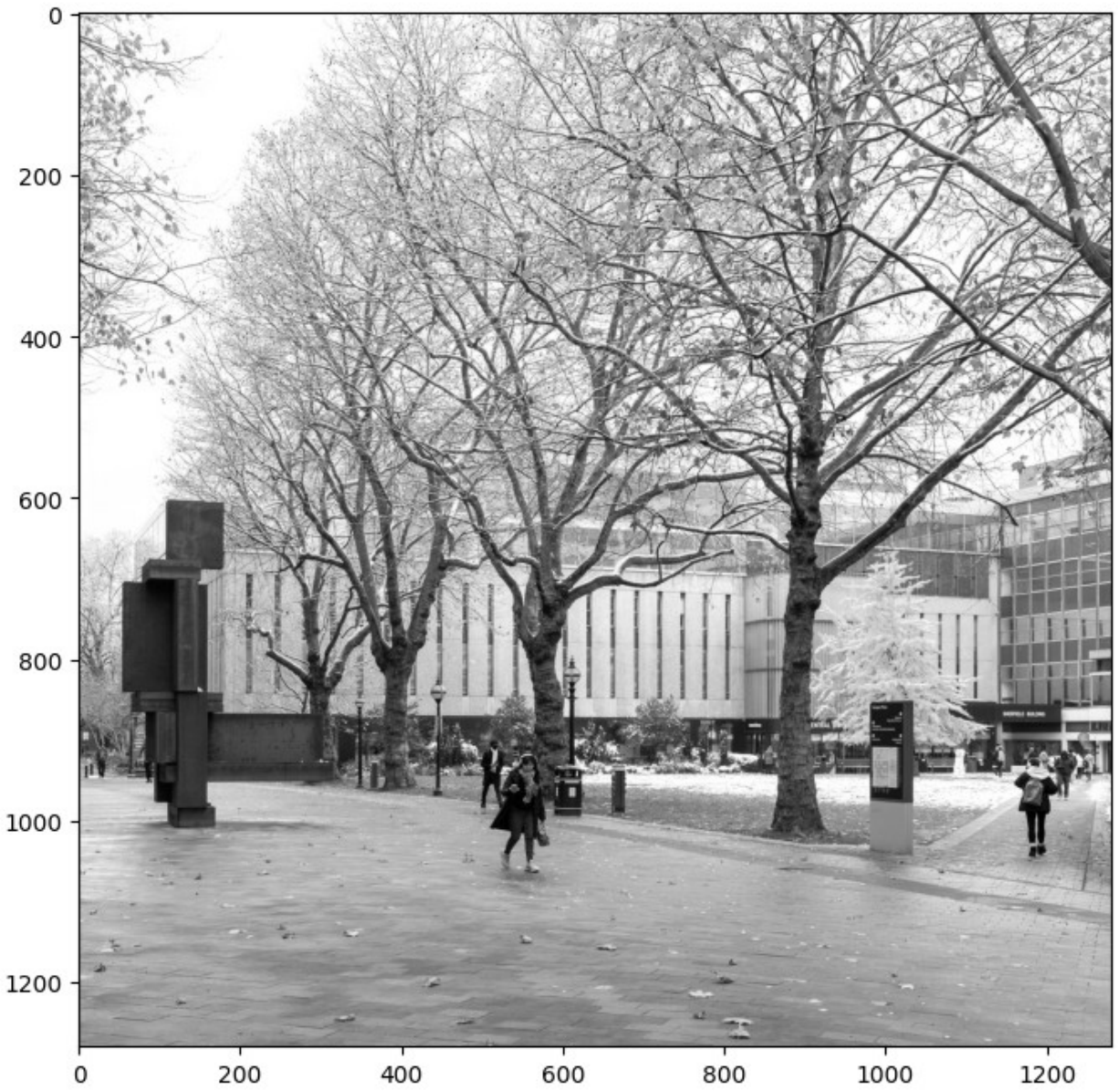
```
# Import libaries (provided)
import imageio.v3 as imageio
import numpy as np
import matplotlib.pyplot as plt
import noise
import scipy
import scipy.signal
import math
import time
```

1. Moving average filter (20 points).

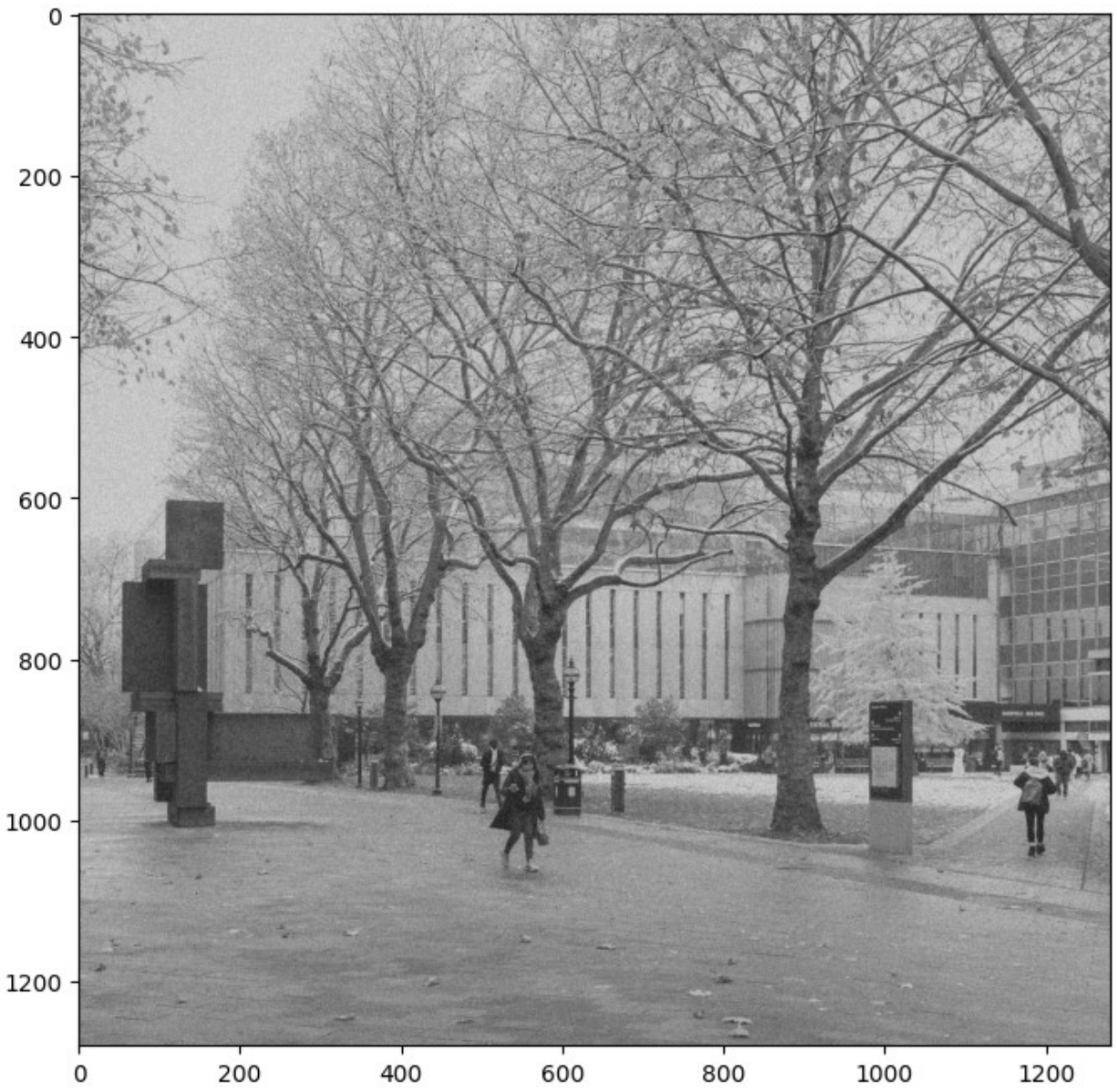
Read the provided input image, add noise to the image and design a moving average filter for denoising.

You are expected to design the kernel of the filter and then perform 2D image filtering using the function `scipy.signal.convolve2d()`.

```
# Read the image (provided)
image = imageio.imread('campus_snow.jpg')
plt.imshow(image, cmap='gray')
plt.gcf().set_size_inches(8, 8)
```




```
# Corrupt the image with Gaussian noise (provided)
image_noisy = noise.add_noise(image, 'gaussian')
plt.imshow(image_noisy, cmap='gray')
plt.gcf().set_size_inches(8, 8)
```



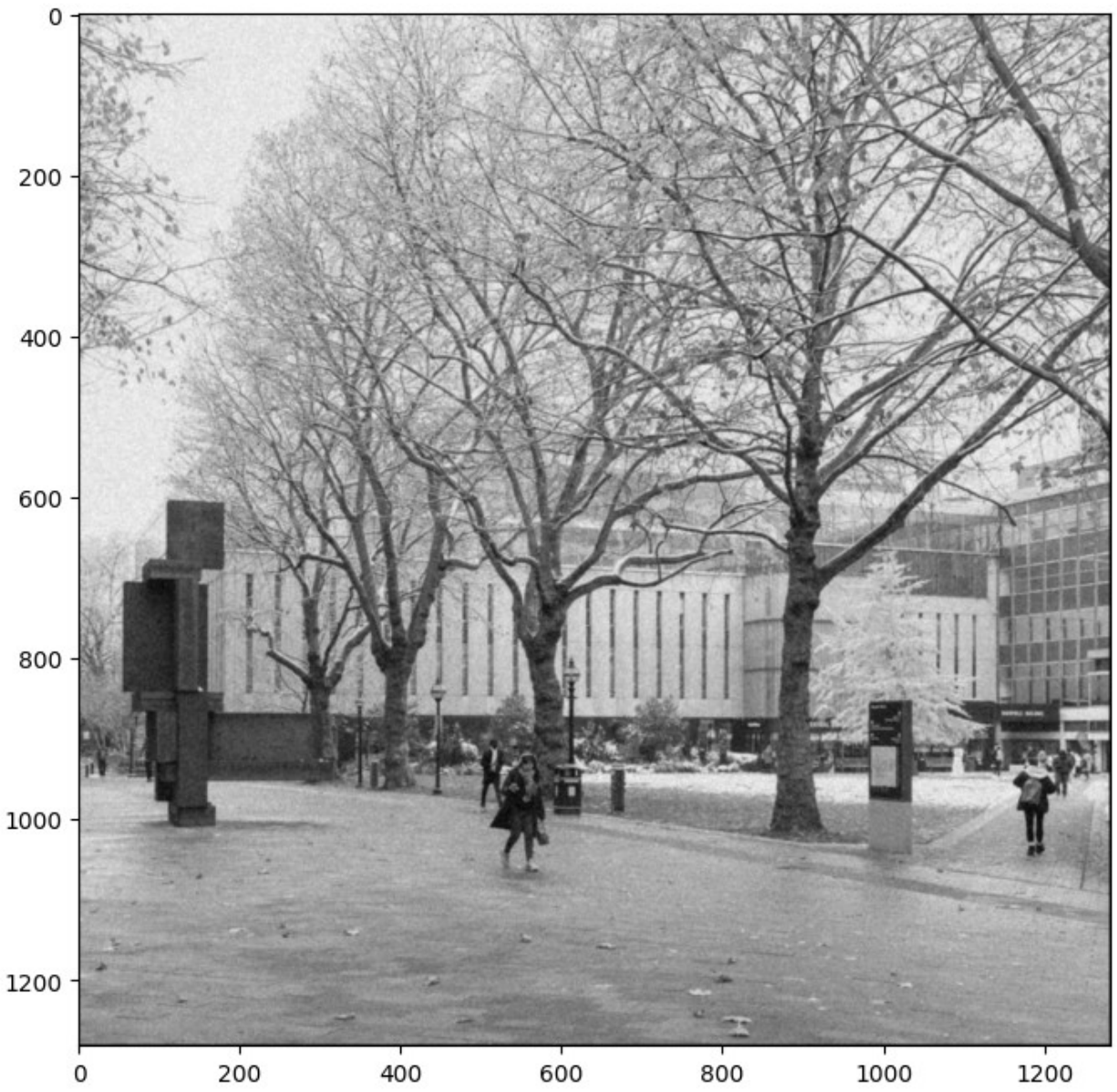
Note: from now on, please use the noisy image as the input for the filters.

1.1 Filter the noisy image with a 3x3 moving average filter. Show the filtering results.

```
# Design the filter h
### Insert your code ###
kernel_size = 3
h = np.ones((kernel_size, kernel_size)) / kernel_size**2

# Convolve the corrupted image with h using scipy.signal.convolve2d
function
```

```
### Insert your code ###
image_filtered_3x3 = scipy.signal.convolve2d(image_noisy, h)

# Print the filter (provided)
print('Filter h:')
print(h)

# Display the filtering result (provided)
plt.imshow(image_filtered_3x3, cmap='gray')
plt.gcf().set_size_inches(8, 8)
```

```
Filter h:
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]
```

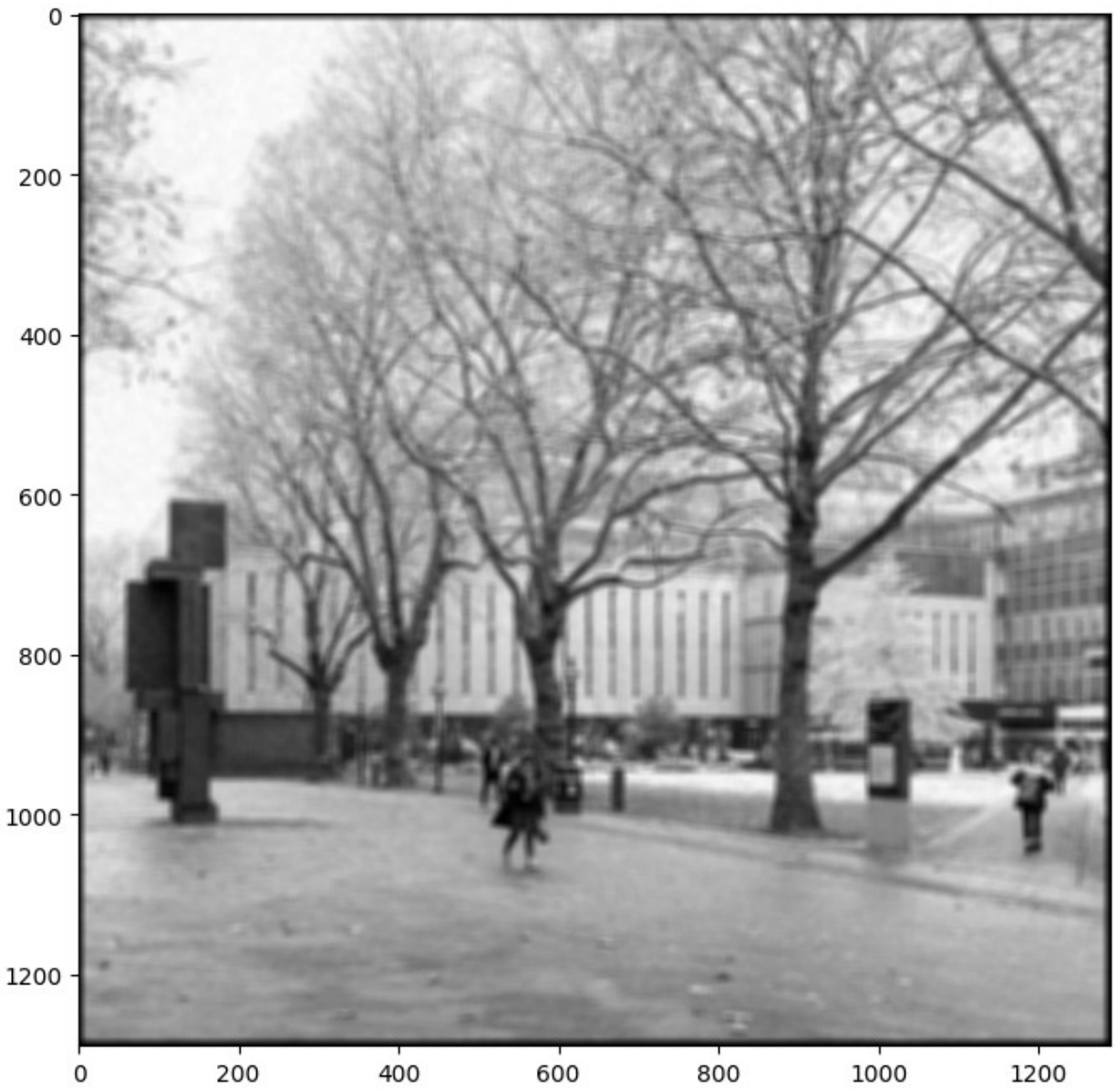


1.2 Filter the noisy image with a 11x11 moving average filter.

```
# Design the filter h
### Insert your code ###
kernel_size = 11
h = np.ones((kernel_size, kernel_size)) / kernel_size**2

# Convolve the corrupted image with h using scipy.signal.convolve2d
function
### Insert your code ###
image_filtered_11x11 = scipy.signal.convolve2d(image_noisy, h)

# Print the filter (provided)
print('Filter h:')
```

```python
print(h)

# Display the filtering result (provided)
plt.imshow(image_filtered_11x11, cmap='gray')
plt.gcf().set_size_inches(8, 8)
```

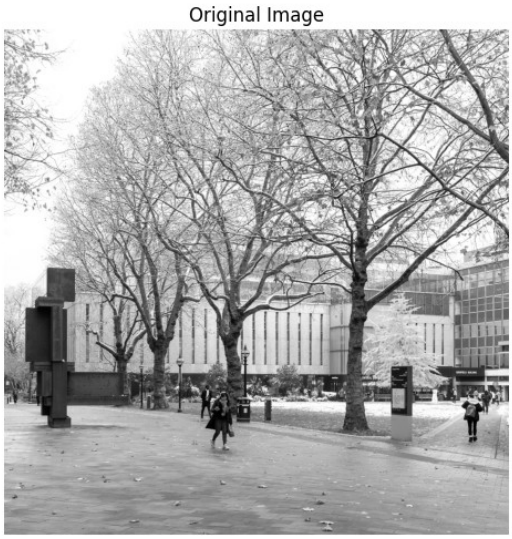
```
Filter h:
[[0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]
 [0.00826446 0.00826446 0.00826446 0.00826446 0.00826446 0.00826446
  0.00826446 0.00826446 0.00826446 0.00826446 0.00826446]]
```

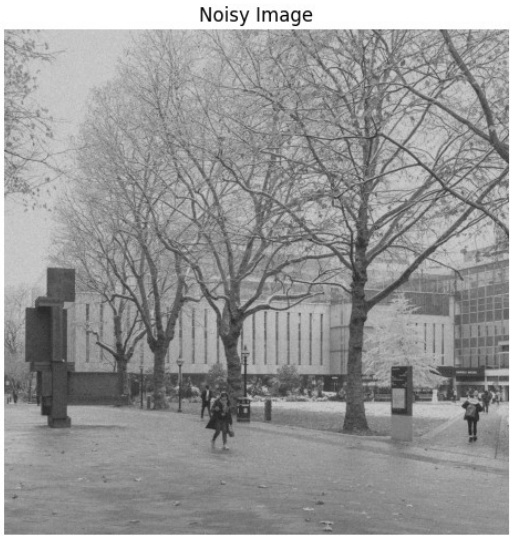


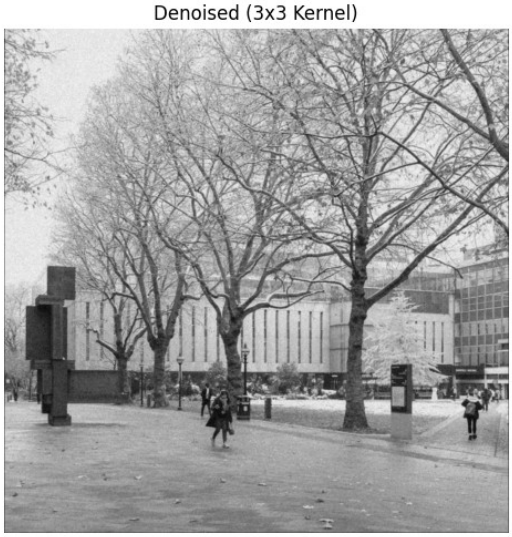
Visualisation of all 4 images together

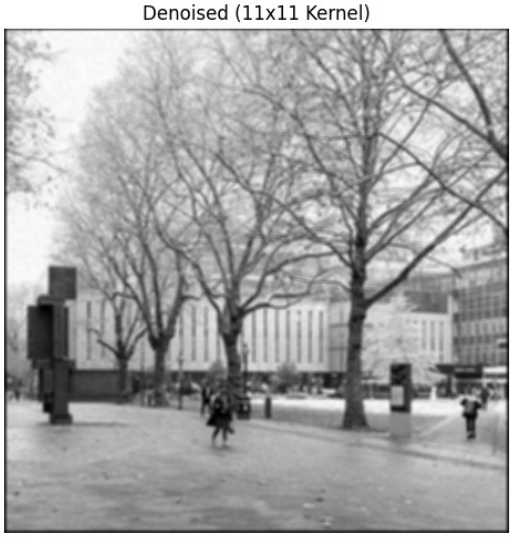
```
import matplotlib.pyplot as plt

# Create a figure to display all images
plt.figure(figsize=(15, 10))  # Adjust the figure size as needed

# Display the original image
plt.subplot(2, 2, 1)  # 2 rows, 2 columns, 1st position
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')  # Hide axes

# Display the noisy image
```

```
plt.subplot(2, 2, 2)  # 2 rows, 2 columns, 2nd position
plt.imshow(image_noisy, cmap='gray')
plt.title('Noisy Image')
plt.axis('off')  # Hide axes

# Display the image filtered with 3x3 kernel
plt.subplot(2, 2, 3)  # 2 rows, 2 columns, 3rd position
plt.imshow(image_filtered_3x3, cmap='gray')
plt.title('Denoised (3x3 Kernel)')
plt.axis('off')  # Hide axes

# Display the image filtered with 11x11 kernel
plt.subplot(2, 2, 4)  # 2 rows, 2 columns, 4th position
plt.imshow(image_filtered_11x11, cmap='gray')
plt.title('Denoised (11x11 Kernel)')
plt.axis('off')  # Hide axes

# Show the plot
plt.tight_layout()  # Adjust spacing between subplots
plt.show()
```









1.3 Comment on the filtering results. How do different kernel sizes influence the filtering results?

1.3.1 **Filtering Results**

- **3x3 Kernel**:
  - **Mild smoothing**: Reduces noise while preserving edges and details.
  - **Output**: Denoised image retains sharpness but still has some noise.
- **11x11 Kernel**:
  - **Strong smoothing**: Removes most noise but blurs edges and fine details.
  - **Output**: Denoised image appears over-smoothed with significant loss of sharpness.

1.3.2. **Influence of Kernel Size**

- **Small Kernel (e.g., 3x3)**:
  - Averages over a small region.

  - **Pros**: Preserves details and edges.
  - **Cons**: Limited noise reduction.
  - **Boundary Effect**: Convolve2D uses zero-padding by default to maintain output resolution. This results in a darker boundary around the output image, which becomes more pronounced as the filter size increases. This occurs because edge pixels are averaged with zero-padded values, reducing their intensity.
- **Large Kernel (e.g., 11x11)**:
  - Averages over a large region.
  - **Pros**: Effective noise reduction.
  - **Cons**: Blurs edges and loses fine details.

Conclusion

- **3x3 Kernel**: Best for mild noise reduction with detail preservation.
- **11x11 Kernel**: Best for heavy noise reduction at the cost of image sharpness.

2. Edge detection (56 points).

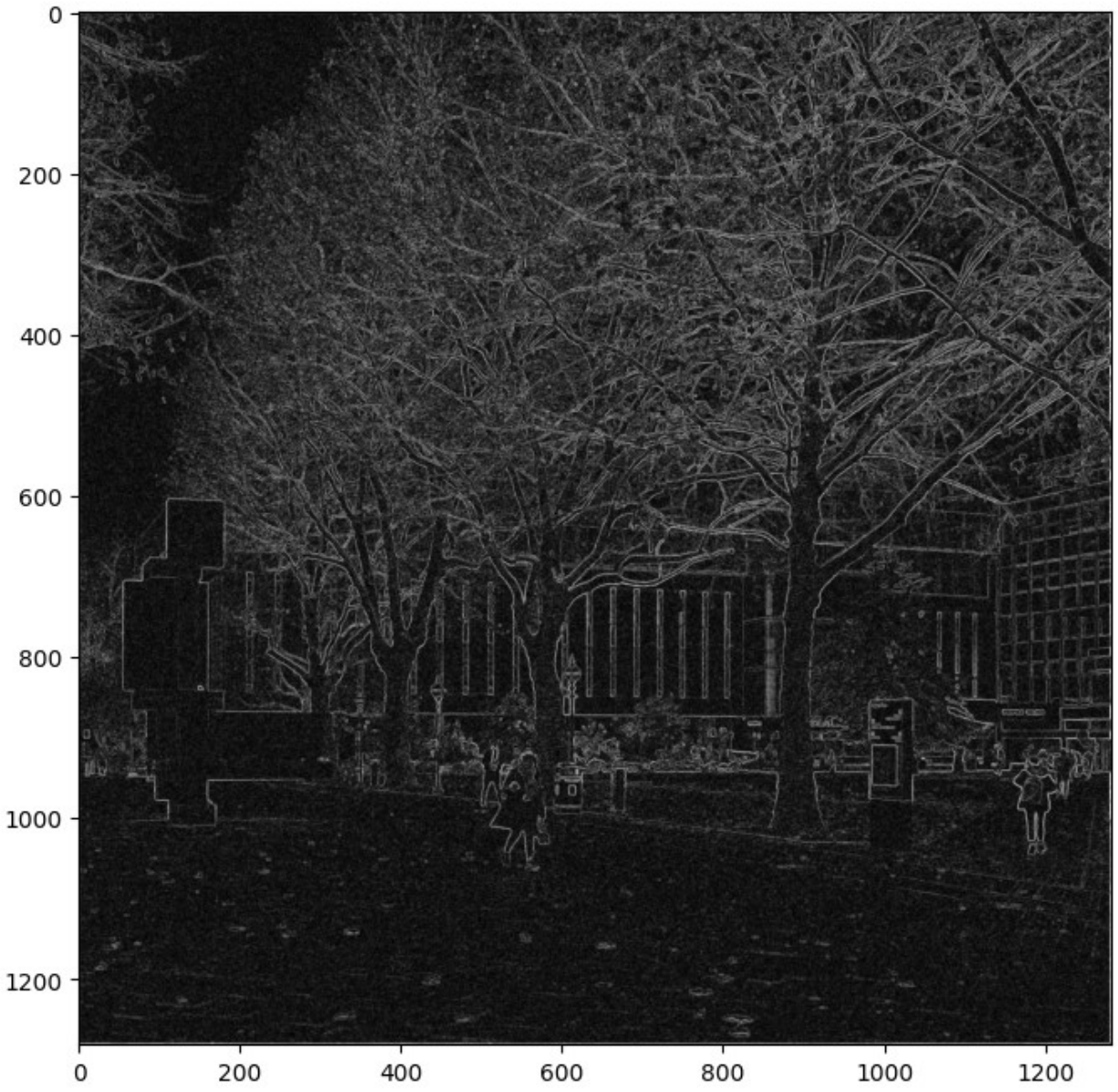
Perform edge detection using Sobel filtering, as well as Gaussian + Sobel filtering.

2.1 Implement 3x3 Sobel filters and convolve with the noisy image.

```
# Design the filters. Defined as 2D numpy arrays.
### Insert your code ###
sobel_x = np.array([[1, 0, -1],
                    [2, 0, -2],
                    [1, 0, -1]])
sobel_y = sobel_x.T

# Image filtering
### Insert your code ###
grad_x = scipy.signal.convolve2d(image_noisy, sobel_x)
grad_y = scipy.signal.convolve2d(image_noisy, sobel_y)

# Calculate the gradient magnitude
### Insert your code ###
grad_mag = np.sqrt(grad_x**2 + grad_y**2)

# Print the filters (provided)
print('sobel_x:')
print(sobel_x)
print('sobel_y:')
print(sobel_y)

# Display the magnitude map (provided)
plt.imshow(grad_mag, cmap='gray')
plt.gcf().set_size_inches(8, 8)
```

```
sobel_x:
[[ 1  0 -1]
```

```
 [ 2  0 -2]
 [ 1  0 -1]]
sobel_y:
[[ 1  2  1]
 [ 0  0  0]
 [-1 -2 -1]]
```

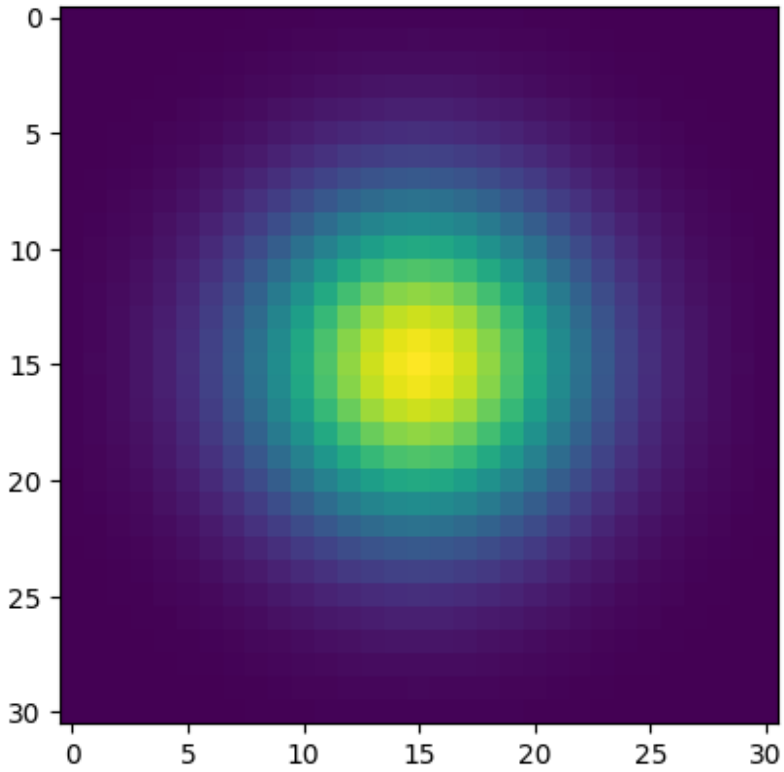


2.2 Implement a function that generates a 2D Gaussian filter given the parameter σ.

```
# Design the Gaussian filter
def gaussian_filter_2d(sigma):
```

```
    # sigma: the parameter sigma in the Gaussian kernel (unit: pixel)
    #
    # return: a 2D array for the Gaussian kernel

    ### Insert your code ###
    kernel_size = int(6 * sigma + 1)

    # Create a 2D grid for the kernel
    half_size = kernel_size // 2
    x, y = np.meshgrid(np.arange(-half_size, half_size + 1),
                       np.arange(-half_size, half_size + 1))

    # Compute the Gaussian kernel
    h = np.exp(-(x**2 + y**2) / (2 * sigma**2)) / (2 * np.pi *
sigma**2)

    # Normalise the kernel
    h = h / np.sum(h)

    return h

# Visualise the Gaussian filter when sigma = 5 pixel (provided)
sigma = 5
h = gaussian_filter_2d(sigma)
plt.imshow(h)
```

```
<matplotlib.image.AxesImage at 0x15c34cfa0>
```

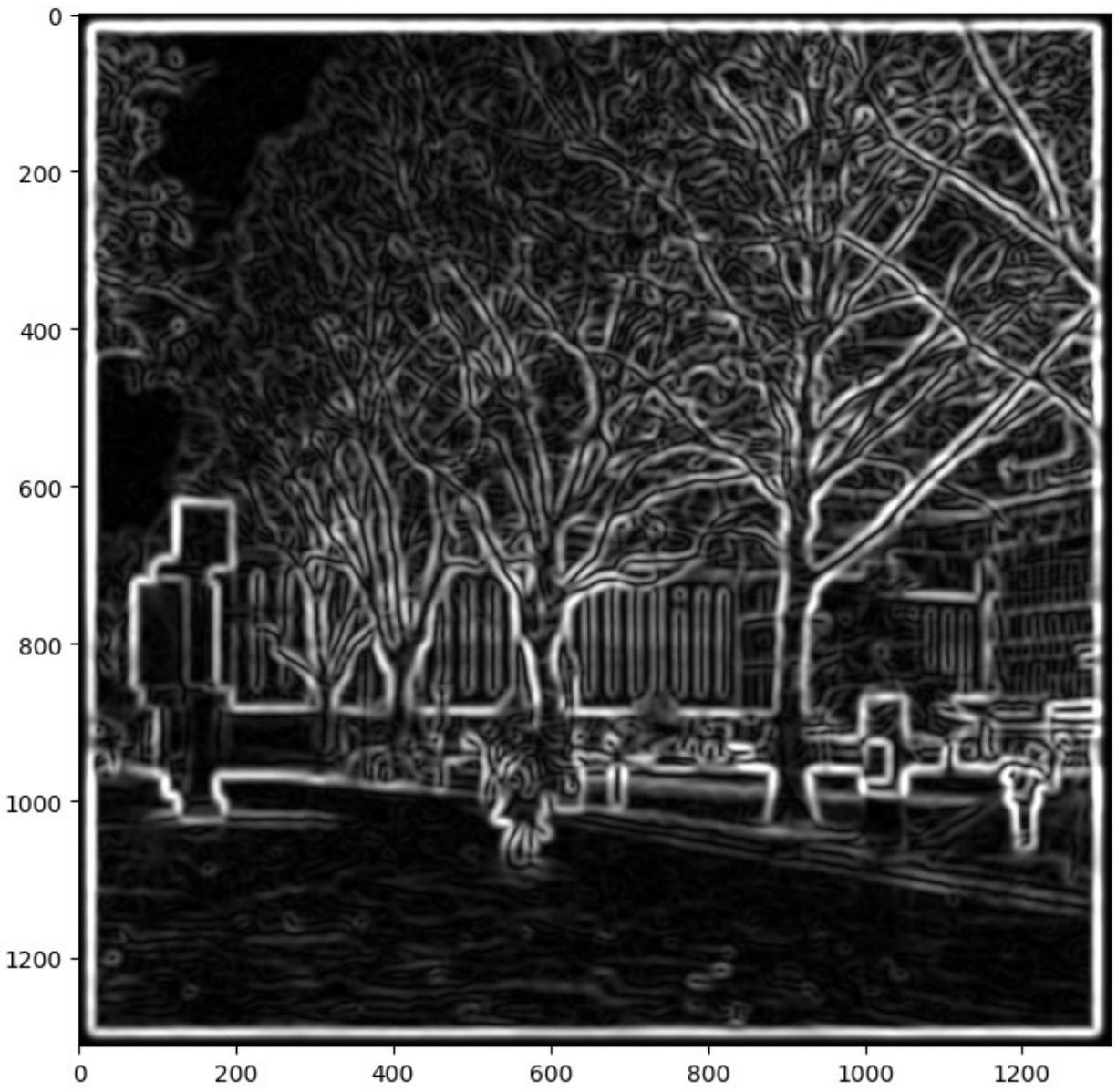


2.3 Perform Gaussian smoothing (σ = 5 pixels) and evaluate the computational time for Gaussian smoothing. After that, perform Sobel filtering and show the gradient magintude map.

```
# Construct the Gaussian filter
### Insert your code ###
sigma = 5
h = gaussian_filter_2d(sigma)

# Perform Gaussian smoothing and count time
### Insert your code ###
start_time = time.time()
image_filtered_gaussian = scipy.signal.convolve2d(image_noisy, h)
end_time = time.time()
print('Time taken to smoothen using Gaussian filter: %f seconds' %
(end_time - start_time))

# Image filtering
### Insert your code ###
grad_x = scipy.signal.convolve2d(image_filtered_gaussian, sobel_x)
grad_y = scipy.signal.convolve2d(image_filtered_gaussian, sobel_y)

# Calculate the gradient magnitude
### Insert your code ###
grad_mag = np.sqrt(grad_x**2 + grad_y**2)
```

```
# Display the gradient magnitude map (provided)
plt.imshow(grad_mag, cmap='gray', vmin=0, vmax=100)
plt.gcf().set_size_inches(8, 8)
```

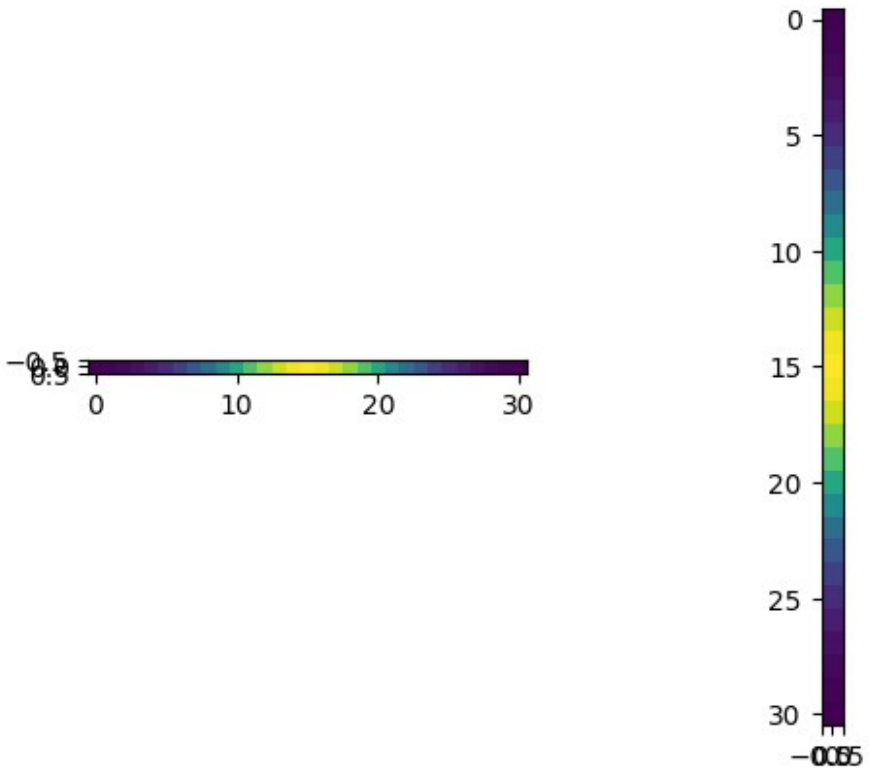
```
Time taken to smoothen using Gaussian filter: 5.141542 seconds
```



2.4 Implement a function that generates a 1D Gaussian filter given the parameter σ. Generate 1D Gaussian filters along x-axis and y-axis respectively.

```
# Design the Gaussian filter
def gaussian_filter_1d(sigma):
    # sigma: the parameter sigma in the Gaussian kernel (unit: pixel)
    #
    # return: a 1D array for the Gaussian kernel

    ### Insert your code ###

    # Determine the kernel size
    kernel_size = int(6 * sigma + 1)
    if kernel_size % 2 == 0:
        kernel_size += 1

    # Create a 1D grid for the kernel
    half_size = kernel_size // 2
    x = np.arange(-half_size, half_size + 1)

    # Compute the Gaussian kernel weights
    h = np.exp(-x**2 / (2 * sigma**2)) / (np.sqrt(2 * np.pi) * sigma)

    # Normalize the kernel
    h = h / np.sum(h)

    return h

# sigma = 5 pixel (provided)
sigma = 5

# The Gaussian filter along x-axis. Its shape is (1, sz).
### Insert your code ###
h_x = gaussian_filter_1d(sigma).reshape(1, -1)  # Reshape to a row
vector

# The Gaussian filter along y-axis. Its shape is (sz, 1).
### Insert your code ###
h_y = gaussian_filter_1d(sigma).reshape(-1, 1) # Reshape to a column
vector

# Visualise the filters (provided)
plt.subplot(1, 2, 1)
plt.imshow(h_x)

plt.subplot(1, 2, 2)
plt.imshow(h_y)
```

```
<matplotlib.image.AxesImage at 0x15c474850>
```

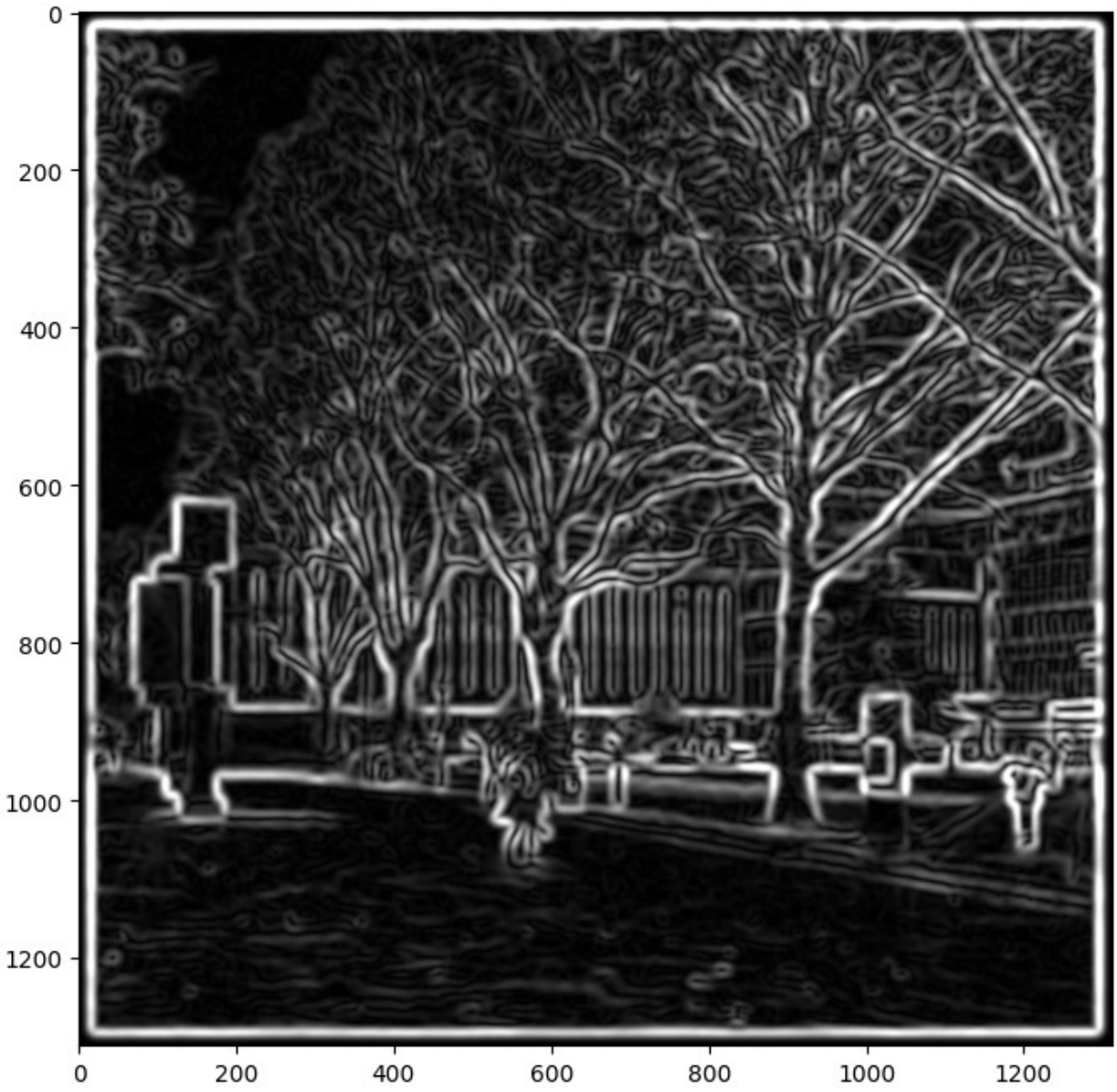


2.5 Perform Gaussian smoothing (σ = 5 pixels) using two separable filters and evaluate the computational time for separable Gaussian filtering. After that, perform Sobel filtering, show the gradient magnitude map and check whether it is the same as the previous one without separable filtering.

```
# Perform separable Gaussian smoothing and count time
### Insert your code ###
start_time = time.time()

image_smoothed_x = scipy.signal.convolve2d(image_noisy, h_x) # Apply
1D Gaussian filter aka Convolve along the x-axis
image_smoothed = scipy.signal.convolve2d(image_smoothed_x, h_y) #
Apply 1D Gaussian filter aka convolve along the y-axis

end_time = time.time()
separable_time = end_time - start_time
print('Time taken to smoothen using separable Gaussian filter: %f
seconds' % separable_time)

# Image filtering
### Insert your code ###
grad_x = scipy.signal.convolve2d(image_smoothed, sobel_x)
grad_y = scipy.signal.convolve2d(image_smoothed, sobel_y)
```

```python
# Calculate the gradient magnitude
### Insert your code ###
grad_mag2 = np.sqrt(grad_x**2 + grad_y**2)

# Display the gradient magnitude map (provided)
plt.imshow(grad_mag2, cmap='gray', vmin=0, vmax=100)
plt.gcf().set_size_inches(8, 8)

# Check the difference between the current gradient magnitude map
# and the previous one produced without separable filtering. You
# can report the mean difference between the two.
### Insert your code ###
mean_error = np.mean((grad_mag - grad_mag2))
mean_abosolute_error = np.mean(np.abs(grad_mag - grad_mag2))
print(f'The mean difference between the two gradient magnitude maps:
{mean_error}')
print(f'Mean absolute difference between the two gradient magnitude
maps: {mean_abosolute_error}')
```

```
Time taken to smoothen using separable Gaussian filter: 0.444123
seconds
The mean difference between the two gradient magnitude maps:
3.116483431009624e-15
Mean absolute difference between the two gradient magnitude maps:
4.061877347079493e-13
```



2.6 Comment on the Gaussian + Sobel filtering results and the computational time.

Gaussian Smoothing

- **Separable Filtering**:
  - The 1D Gaussian filters (`h_x` and `h_y`) were applied sequentially along the x-axis and y-axis.
  - This approach is **computationally more efficient** than using a 2D Gaussian filter because it reduces the number of operations from (O(n^2)) to (O(2n)).
  - Intensity gradients along the boundary of the image are amplified due to the use of 0-padding.

  - Blurring slightly widens gradient lines, which can be mitigated by subsequent steps like Non-Maximum Suppression (NMS).
- **Computational Time**:
  - The time taken for separable filtering was significantly **less** than for non-separable filtering.
  - Example: Separable filtering took ~`0.46 seconds`, while non-separable filtering took ~`4.81 seconds`.

Sobel Filtering

- The Sobel filters were applied to the Gaussian-smoothed image to compute the gradient magnitude.
- The gradient magnitude map highlights edges in the image.

Comparison of Results

- The gradient magnitude maps produced by separable and non-separable filtering were **almost identical**.
- The mean difference between the two maps was very small with a mean absolute error of ~`4.07e-13`, indicating that separable filtering produces **equivalent results** to non-separable filtering, and the difference was just floating point error.

Conclusion

- **Separable filtering** is **faster** and **equally effective** for Gaussian smoothing.
- It is a preferred method for large images or real-time applications where computational efficiency is critical.

3. Challenge: Implement 2D image filters using Pytorch (24 points).

Pytorch is a machine learning framework that supports filtering and convolution.

The Conv2D operator takes an input array of dimension NxC1xXxY, applies the filter and outputs an array of dimension NxC2xXxY. Here, since we only have one image with one colour channel, we will set N=1, C1=1 and C2=1. You can read the documentation of Conv2D for more detail.

```
# Import libaries (provided)
import torch
import torch.nn as nn
```

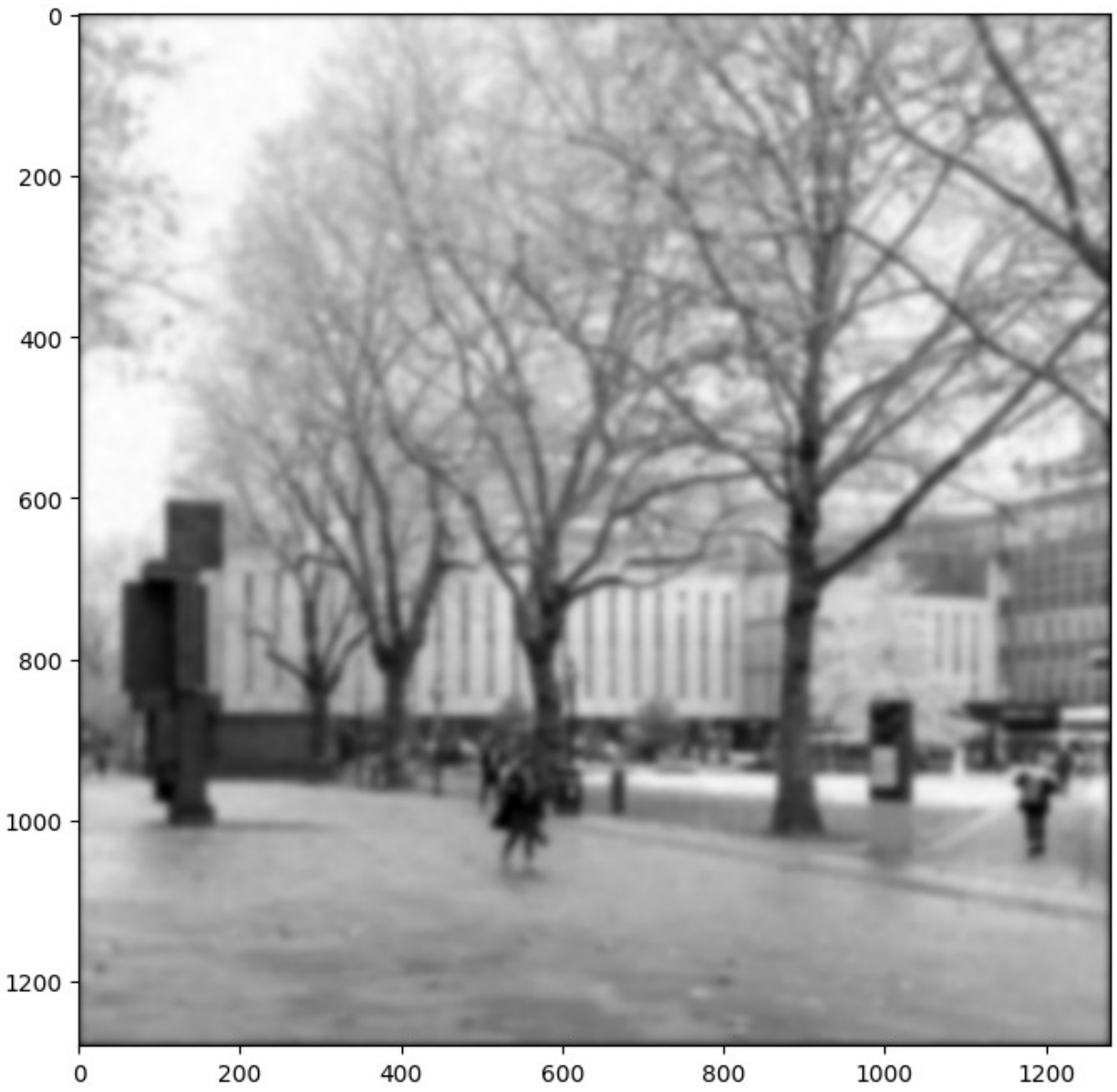
3.1 Expand the dimension of the noisy image into 1x1xXxY and convert it to a Pytorch tensor.

```
# Expand the dimension of the numpy array
### Insert your code ###

# Expand Dimensions:
# The noisy image is a 2D array (height × width). We need to expand it
to a 4D array with dimensions (N, C, H, W), where:
```

```
# N = 1 (batch size),
# C = 1 (number of channels),
# H = height,
# W = width.
image_noisy_expanded = np.expand_dims(image_noisy, axis=(0, 1))


# Convert to a Pytorch tensor using torch.from_numpy
### Insert your code ###
# Use torch.from_numpy() to convert the NumPy array to a PyTorch
tensor.
image_noisy_tensor = torch.from_numpy(image_noisy_expanded).float()
```

3.2 Create a Pytorch Conv2D filter, set its kernel to be a 2D Gaussian filter and perform filtering.

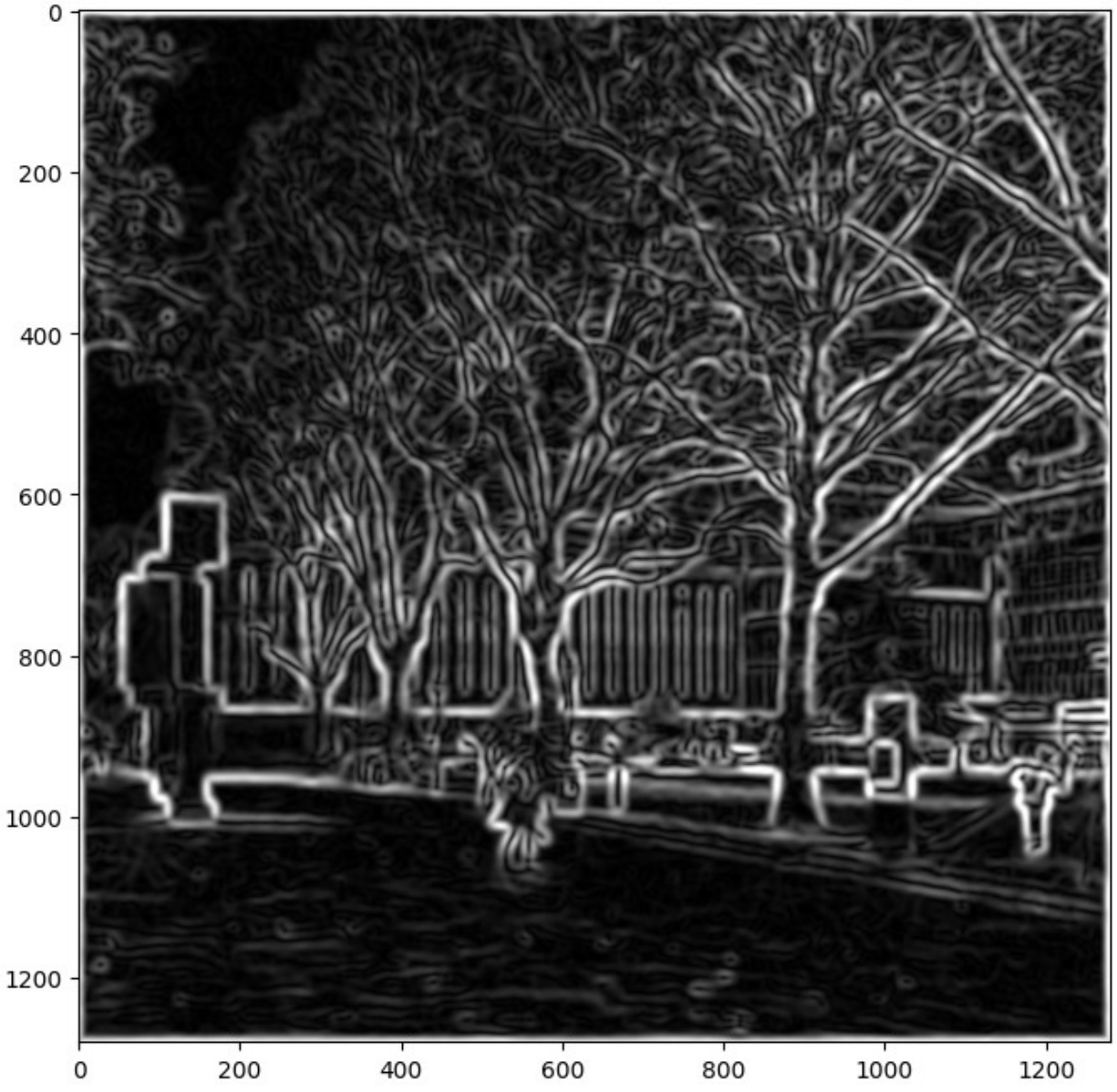
```
# A 2D Gaussian filter when sigma = 5 pixel (provided)
sigma = 5
h = gaussian_filter_2d(sigma)

# Create the Conv2D filter
### Insert your code ###
conv_gaussian = nn.Conv2d(in_channels=1, out_channels=1,
kernel_size=h.shape, padding=h.shape[0] // 2, bias=False)

# Set the kernel weights to the Gaussian kernel
with torch.no_grad():
    conv_gaussian.weight =
nn.Parameter(torch.from_numpy(h).float().unsqueeze(0).unsqueeze(0))

# Filtering
### Insert your code ###
with torch.no_grad():
    image_filtered_tensor = conv_gaussian(image_noisy_tensor)

# Convert the result back to a NumPy array
image_filtered = image_filtered_tensor.squeeze().numpy()

# Display the filtering result (provided)
plt.imshow(image_filtered, cmap='gray')
plt.gcf().set_size_inches(8, 8)
```



3.3 Implement Pytorch Conv2D filters to perform Sobel filtering on Gaussian smoothed images, show the gradient magnitude map.

```
# Create Conv2D filters
### Insert your code ###
conv_sobel_x = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3,
padding=1, bias=False)
conv_sobel_y = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3,
padding=1, bias=False)

# Set the kernel weights
with torch.no_grad():
    conv_sobel_x.weight =
```

```
nn.Parameter(torch.from_numpy(sobel_x).float().unsqueeze(0).unsqueeze(
0))
    conv_sobel_y.weight =
nn.Parameter(torch.from_numpy(sobel_y).float().unsqueeze(0).unsqueeze(
0))

# Perform filtering
### Insert your code ###
grad_x_tensor = conv_sobel_x(image_filtered_tensor)
grad_y_tensor = conv_sobel_y(image_filtered_tensor)

# Calculate the gradient magnitude map
### Insert your code ###
grad_mag3 = torch.sqrt(grad_x_tensor**2 +
grad_y_tensor**2).squeeze().detach().numpy()

# Visualise the gradient magnitude map (provided)
plt.imshow(grad_mag3, cmap='gray', vmin=0, vmax=100)
plt.gcf().set_size_inches(8, 8)
```


0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200