coursework_01

February 3, 2022

1 Coursework 1: Image filtering

In this coursework you will practice image filtering techniques, which are commonly used to smooth, sharpen or add certain effects to images. The coursework includes both coding questions and written questions. Please read both the text and code comment in this notebook to get an idea what you are expected to implement.

1.1 What to do?

- Complete and run the code using `jupyter-lab` or `jupyter-notebook` to get the results.
- Export (File | Export Notebook As…) or print (using the print function of your browser) the notebook as a pdf file, which contains your code, results and text answers, and upload the pdf file onto Cate.
- If Jupyter-lab does not work for you, you can also use Google Colab to write the code and export the pdf file.

1.2 Dependencies:

If you do not have Jupyter-Lab on your laptop, you can find information for installing Jupyter-Lab here.

There may be certain Python packages you may want to use for completing the coursework. We have provided examples below for importing libraries. If some packages are missing, you need to install them. In general, new packages (e.g. imageio etc) can be installed by running

```
pip3 install [package_name]
```

in the terminal. If you use Anaconda, you can also install new packages by running `conda install [package_name]` or using its graphic user interface.

[1]:
```
# Import libaries (provided)
import imageio
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import noise
import scipy
import scipy.signal
import math
```

```python
import time
# set print precision to ensure tidy print
np.set_printoptions(precision = 3)
```

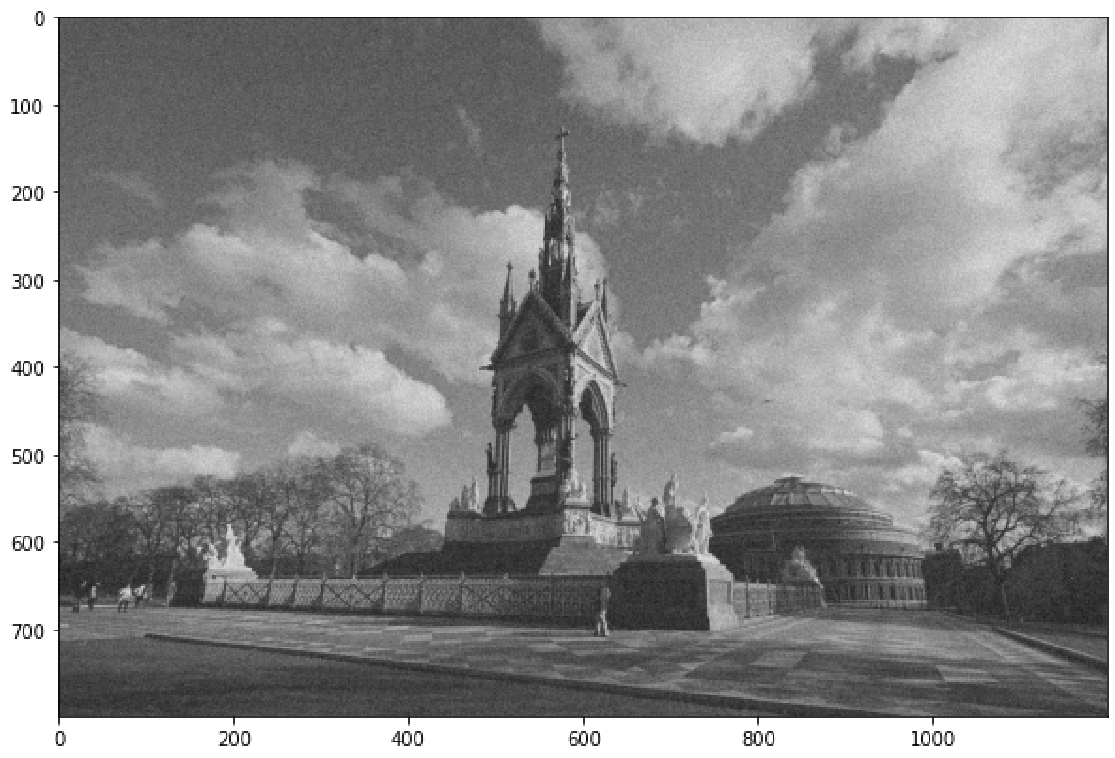
1.3 1. Moving average filter.

Read a specific input image and add noise to the image. Design a moving average filter of kernel size 3x3 and 11x11 respectively. Perform image filtering on the noisy image.

Design the kernel of the filter by yourself. Then perform 2D image filtering using the function `scipy.signal.convolve2d()`.

[2]:
```python
# Read the image (provided)
image = imageio.imread('hyde_park.jpg')
plt.imshow(image, cmap='gray')
# gcf() gets the current figure.
# set_size_inches by default follows _dpi_ratio in the given image
plt.gcf().set_size_inches(10, 8)
```

[3]:
```python
# 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(10, 8)
```



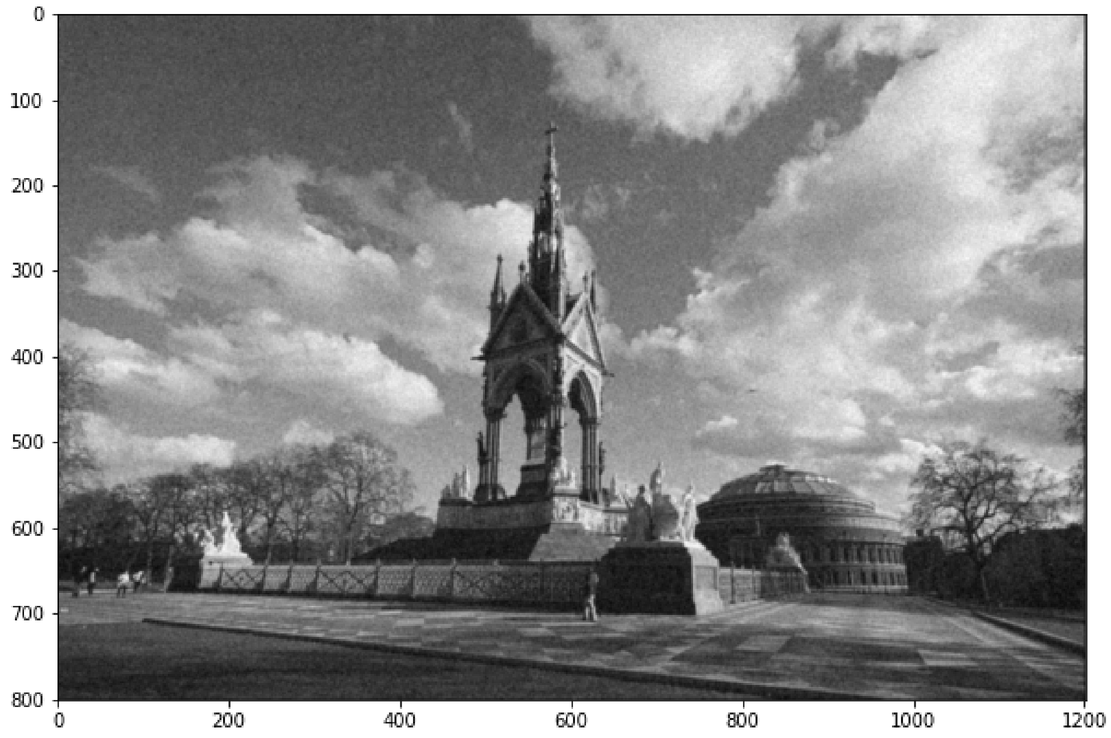
1.3.1 Note: from now on, please use the noisy image as the input for the filters.

1.3.2 1.1 Filter the noisy image with a 3x3 moving average filter. Show the filtering results. (5 points)

[4]:
```python
# Helper filter function
def generate_smooth_filter(size = 3, filter_type = "Uniform"):
    """
    generate 2-D Uniform/Gaussian filter

    Arguments:
    size (int): size of one dimension
    filter_type (string): Uniform or Gaussian

    Return:
    a 2-d smooth filter with shape (size, size)
    """
    assert(size%2 and size > 0)
    assert(filter_type == "Uniform" or filter_type == "Gaussian")
    if filter_type == "Uniform":
        h_x = np.ones(size).reshape(-1, size)
        h = np.outer(h_x, h_x)
    if filter_type == "Gaussian":
        sigma = (size-1)/6
        h_x = scipy.signal.gaussian(size, std = sigma).reshape(-1, size)
        h = np.outer(h_x, h_x)
    return h/h.sum()
```

[5]:
```
# Design the filter h
h = generate_smooth_filter(3, "Uniform")

# Convolve the corrupted image with h using scipy.signal.convolve2d function
image_filtered = scipy.signal.convolve2d(image_noisy, h)

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

# Display the filtering result (provided)
plt.imshow(image_filtered, cmap='gray')
plt.gcf().set_size_inches(10, 8)
```

```
Filter h:
[[0.111 0.111 0.111]
 [0.111 0.111 0.111]
 [0.111 0.111 0.111]]
```



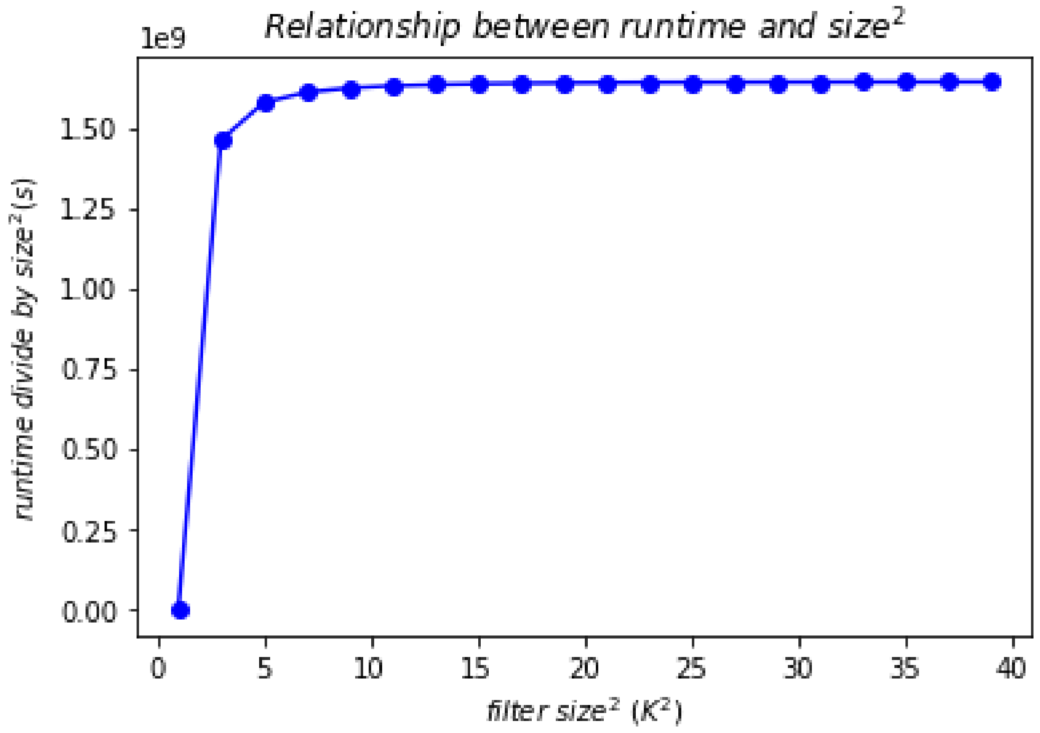
1.3.3 1.2 Filter the noisy image with a 11x11 moving average filter. (5 points)

[6]:
```
# Design the filter h
h = generate_smooth_filter(11, "Uniform")

# Convolve the corrupted image with h using scipy.signal.convolve2d function
image_filtered = scipy.signal.convolve2d(image_noisy, h)

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

# Display the filtering result (provided)
plt.imshow(image_filtered, cmap='gray')
plt.gcf().set_size_inches(10, 8)
```

```
Filter h:
[[0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]
 [0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008 0.008]]
```

1.3.4 1.3 Comment on the filtering results. How do different kernel sizes influence the filtering results? (10 points)

[7]:
```
# time complexity analysis, comment is in below box
sizes = []
times = []
for i in range(20):
    size = 2*i + 1
    h = generate_smooth_filter(size, "Uniform")
    start_time = time.time()
    image_filtered = scipy.signal.convolve2d(image_noisy, h)
    end_time = time.time()
    sizes.append(size)
    times.append(end_time - start_time/size**2)

fig = plt.figure()
plt.plot(sizes, times, 'b-o')
plt.xlabel(r"$filter \ size^2\ (K^2)$")
plt.ylabel(r"$runtime\ divide\ by\ size^2(s)$")
plt.title(r"$Relationship\ between\ runtime\ and\ size^2$")
plt.show()
```



1. In terms of smoothing level

- If the kernel is uniformly distributed, a larger kernel size means more pixels are included, resulting in a better blurring effect. Mathematically, if the original noise e in each pixel follows $N(0, \sigma^2)$, after using uniform filter, noise of average result (of N pixels) e' follows distribution $N(0, \frac{\sigma^2}{N})$ with a much smaller variance, hence noise is reduced.
- If the kernel is Gaussian distributed, the smoothing level also increases along with kernel size and sigma parameter. However, unlike a uniform filter that considers equal significance for each pixel, the Gaussian filter considers nearby pixels more important, which guarantees more features of a position.

2. In terms of time complexity

- It can be seen from the above figure that when filter size gets larger, time consumed grows in parallel with $size^2$, which verifies the deduction in tut that time complexity of using 2-d filter is $O(K^2N^2)$ for kernel size K and image size N.

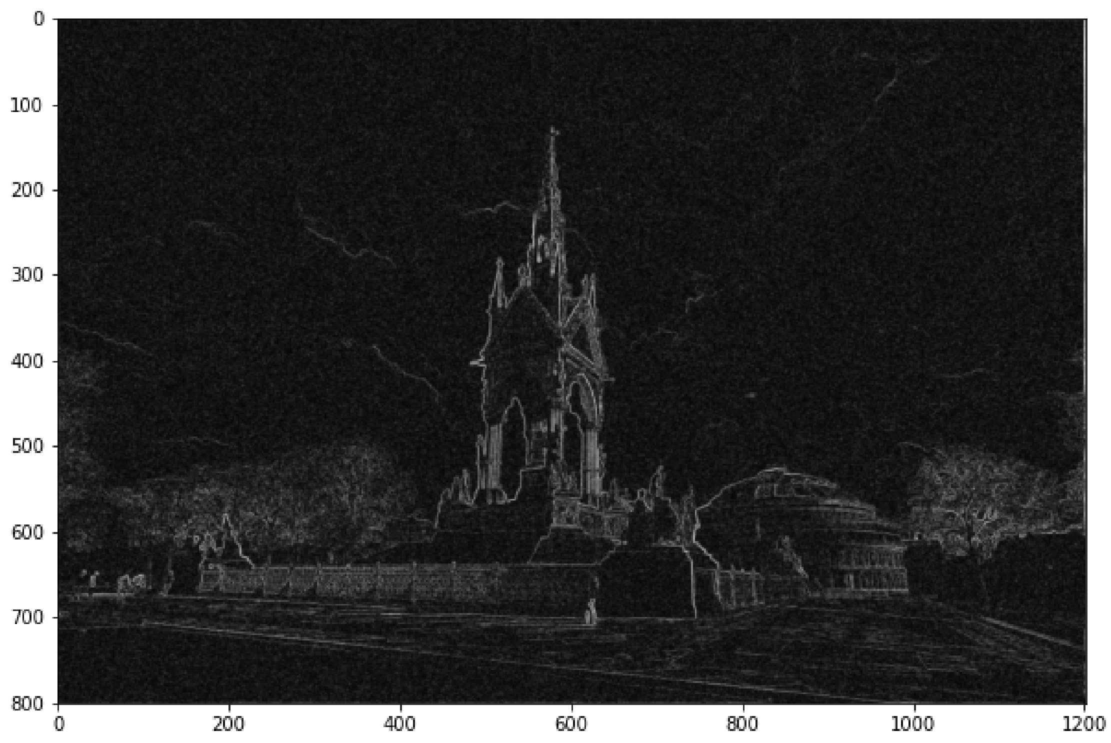
1.4 2. Edge detection.

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

1.4.1 2.1 Implement 3x3 Prewitt filters and convolve with the noisy image. (10 points)

[8]:
```
# Design the Prewitt filters
smooth = np.array([1,1,1]).reshape((-1, 3))
finite_diff = np.array([1,0,-1]).reshape((-1, 3))
prewitt_x = np.outer(smooth, finite_diff)
prewitt_y = np.outer(finite_diff, smooth)

# Prewitt filtering
image_edged_x = scipy.signal.convolve2d(image_noisy, prewitt_x)
image_edged_y = scipy.signal.convolve2d(image_noisy, prewitt_y)

# Calculate the gradient magnitude
grad_mag = np.sqrt(image_edged_x**2 + image_edged_y**2)

# Print the filters (provided)
print('prewitt_x:')
print(prewitt_x)
print('prewitt_y:')
print(prewitt_y)

# Display the gradient magnitude image (provided)
plt.imshow(grad_mag, cmap='gray')
plt.gcf().set_size_inches(10, 8)
```

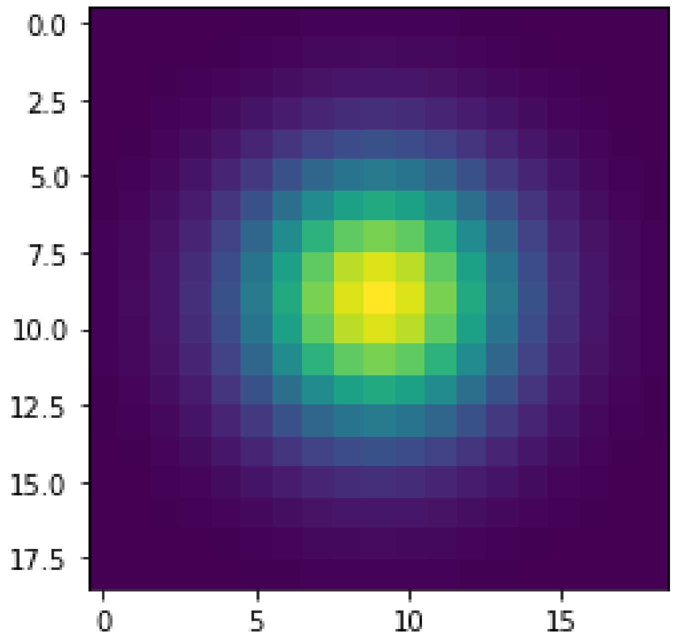
```
prewitt_x:
[[ 1  0 -1]
 [ 1  0 -1]
 [ 1  0 -1]]
prewitt_y:
[[ 1  1  1]
 [ 0  0  0]
 [-1 -1 -1]]
```



1.4.2 2.2 Implement a function that generates a 2D Gaussian filter given the parameter σ. (10 points)

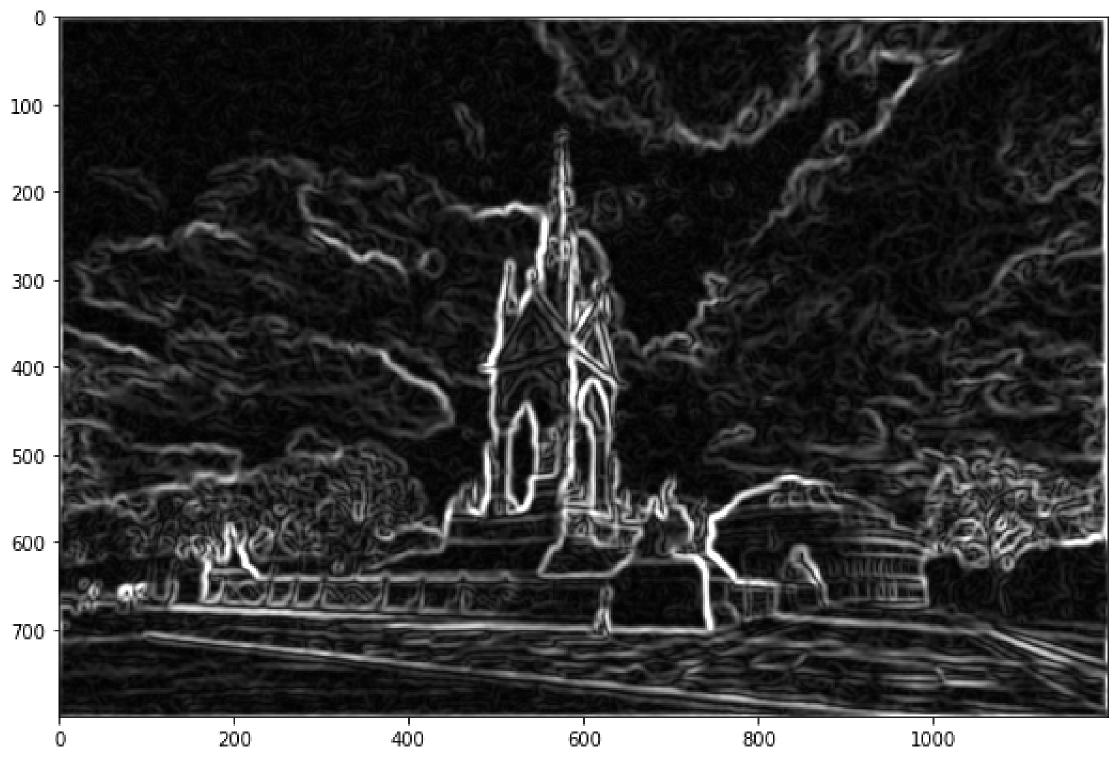
```
[9]: # Design the Gaussian filter
def gaussian_filter_2d(sigma):
    """"
    generate a 3x3 Gaussian filter with given sigma

    Arguments:
    sigma: the parameter sigma in the Gaussian kernel (unit: pixel)

    returns:
    a 2D array for the Gaussian kernel
    """
    # The helper func defined before can be used here
    size = (sigma)*6 + 1
    h = generate_smooth_filter(size, "Gaussian")
    return h

# Visualise the Gaussian filter when sigma = 3 pixel (provided)
sigma = 3
h = gaussian_filter_2d(sigma)
plt.imshow(h)
```

```
[9]: <matplotlib.image.AxesImage at 0x232b316eac0>
```



1.4.3 2.3 Perform Gaussian smoothing ($\sigma = 3$ pixels), followed by Prewitt filtering, show the gradient magnitude image. (5 points)

```
[10]: # Perform Gaussian smoothing before Prewitt filtering
h = gaussian_filter_2d(3)
image_filtered = scipy.signal.convolve2d(image_noisy, h, mode = 'same')

# Prewitt filtering
smooth = np.array([1,1,1]).reshape((-1, 3))
finite_diff = np.array([1,0,-1]).reshape((-1, 3))
prewitt_x = np.outer(smooth, finite_diff)
prewitt_y = np.outer(finite_diff, smooth)
image_edged_x = scipy.signal.convolve2d(image_filtered, prewitt_x, mode =␣
 ↪'same')
image_edged_y = scipy.signal.convolve2d(image_filtered, prewitt_y, mode =␣
 ↪'same')

# Calculate the gradient magnitude
grad_mag = np.sqrt(image_edged_x**2 + image_edged_y**2)

# Display the gradient magnitude image (provided)
plt.imshow(grad_mag, cmap='gray', vmin=0, vmax=100)
plt.gcf().set_size_inches(10, 8)
```

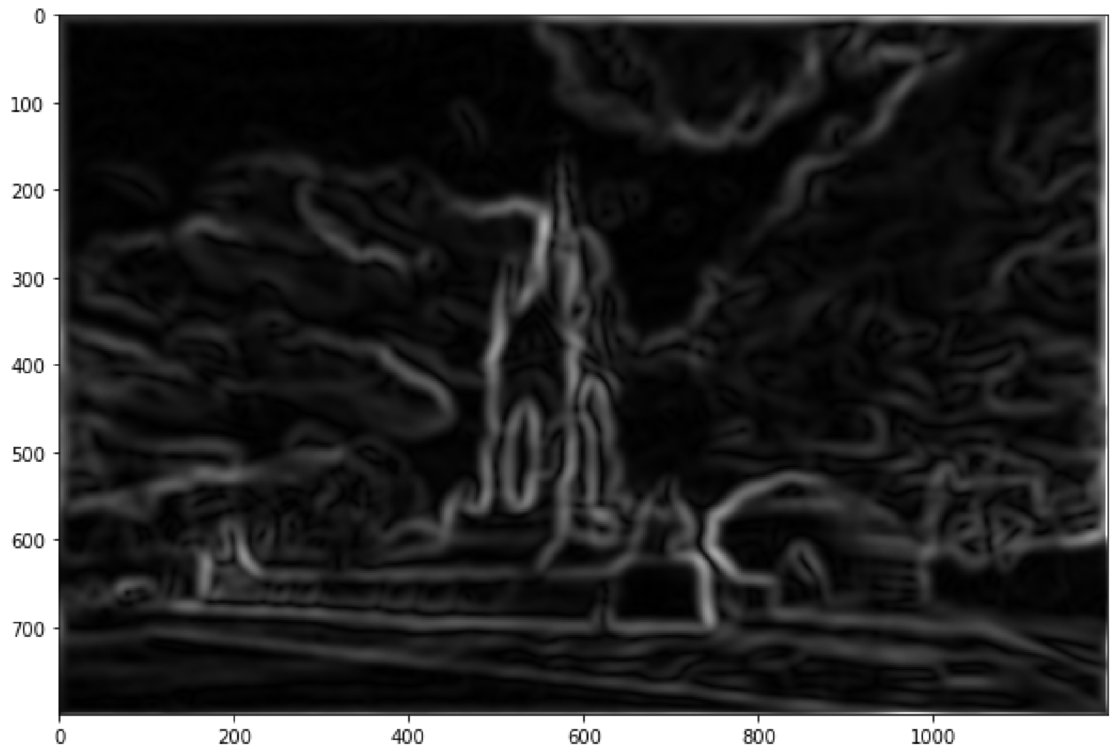


1.4.4 2.4 Perform Gaussian smoothing ($\sigma = 7$ pixels) and evaluate the computational time for Gaussian smoothing. After that, perform Prewitt filtering. (7 points)

[11]:
```
# Construct the Gaussian filter
h = gaussian_filter_2d(7)

# Perform Gaussian smoothing and count time
start_time = time.time()
image_filtered = scipy.signal.convolve2d(image_noisy, h, mode = "same")
end_time = time.time()
print(f"A 2D Gaussian filter operation with sigma {sigma} takes {end_time -␣
 ↪start_time:.2f} seconds")

# Prewitt filtering
smooth = np.array([1,1,1]).reshape((-1, 3))
finite_diff = np.array([1,0,-1]).reshape((-1, 3))
prewitt_x = np.outer(smooth, finite_diff)
prewitt_y = np.outer(finite_diff, smooth)
image_edged_x = scipy.signal.convolve2d(image_filtered, prewitt_x, mode =␣
 ↪'same')
image_edged_y = scipy.signal.convolve2d(image_filtered, prewitt_y, mode =␣
 ↪'same')
```

```
# Calculate the gradient magnitude
grad_mag = np.sqrt(image_edged_x**2 + image_edged_y**2)

# Display the gradient magnitude image (provided)
plt.imshow(grad_mag, cmap='gray', vmin=0, vmax=100)
plt.gcf().set_size_inches(10, 8)
```

```
A 2D Gaussian filter operation with sigma 3 takes 4.68 seconds
```

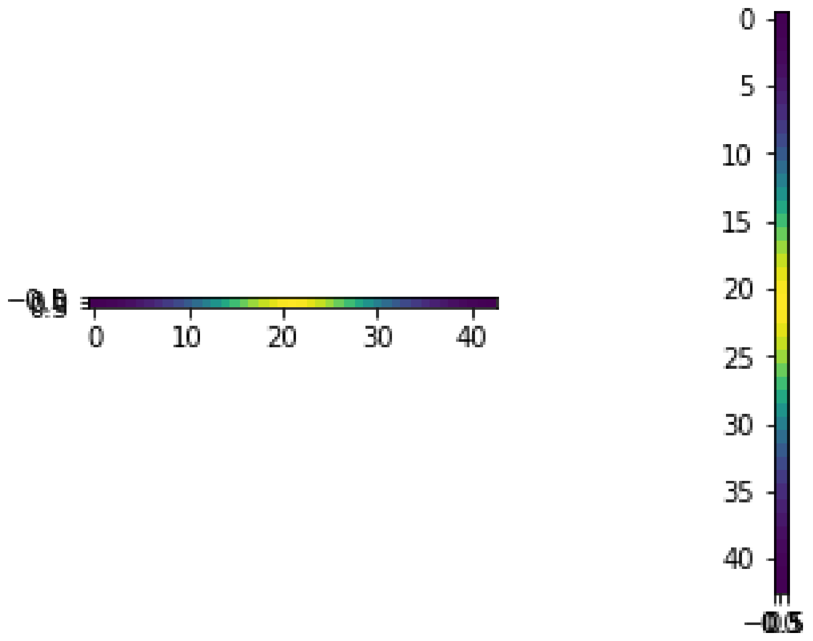


1.4.5 2.5 Implement a function that generates a 1D Gaussian filter given the parameter σ. Generate 1D Gaussian filters along x-axis and y-axis respectively. (10 points)

[12]:
```
# Design the Gaussian filter
def gaussian_filter_1d(sigma):
    """
    generate a 1D Gaussian kernel

    Arguments:
    sigma: the parameter sigma in the Gaussian kernel (unit: pixel)

    returns:
```

```
    a 1D array for the Gaussian kernel
    """
    size = 6*sigma + 1
    h = scipy.signal.gaussian(size, std = sigma)
    return h/h.sum()

# sigma = 7 pixel (provided)
sigma = 7
h = gaussian_filter_1d(sigma)

# The Gaussian filter along x-axis. Its shape is (1, sz).
h_x = h.reshape((-1, h.size))

# The Gaussian filter along y-axis. Its shape is (sz, 1).
h_y = h.reshape((h.size, -1))

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

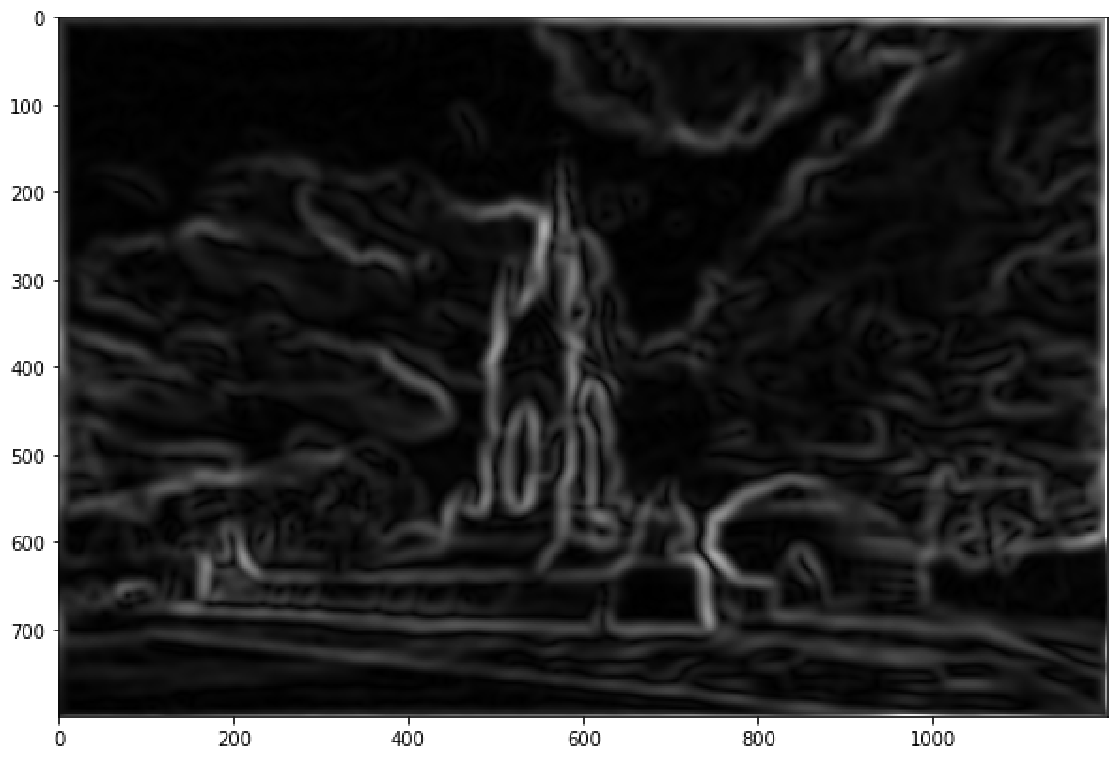
[12]:
```
<matplotlib.image.AxesImage at 0x232b54960d0>
```



1.4.6 2.6 Perform Gaussian smoothing ($\sigma = 7$ pixels) using two separable filters and evaluate the computational time for separable Gaussian filtering. After that, perform Prewitt filtering, show results and check whether the results are the same as the previous one without separable filtering. (9 points)

[13]:
```python
# Perform separable Gaussian smoothing and count time
sigma = 7
h = gaussian_filter_1d(sigma)
h_x = h.reshape((-1, h.size))
h_y = h.reshape((h.size, -1))

start_time = time.time()
image_filtered = scipy.signal.convolve2d(image_noisy, h_x, mode = 'same')
image_filtered = scipy.signal.convolve2d(image_filtered, h_y, mode = 'same')
end_time = time.time()
print(f"Two seperated 1D Gaussian filter operations with sigma {sigma} take␣
 ↪{end_time - start_time:.2f} seconds")

# Prewitt filtering
image_edged_x = scipy.signal.convolve2d(image_filtered, prewitt_x, mode =␣
 ↪'same')
image_edged_y = scipy.signal.convolve2d(image_filtered, prewitt_y, mode =␣
 ↪'same')

# Calculate the gradient magnitude
grad_mag2 = np.sqrt(image_edged_x**2 + image_edged_y**2)

# # Display the gradient magnitude image (provided)
plt.imshow(grad_mag2, cmap='gray', vmin=0, vmax=100)
plt.gcf().set_size_inches(10, 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.
mean_diff = np.abs(grad_mag - grad_mag2).mean()
print(f"The absolute mean difference between the two is {mean_diff:.4}, which␣
 ↪is negligible")
```

```
Two seperated 1D Gaussian filter operations with sigma 7 take 0.43 seconds
The absolute mean difference between the two is 2.997e-13, which is negligible
```



1.4.7 2.7 Comment on the Gaussian + Prewitt filtering results and the computational time. (9 points)

1. result

- Using Gaussian + Prewitt filtering does good smoothing and preserves features. It results in an less-noisy image compared with that uses prewitt only. Considering the property of 2-d Gaussian distribution, x and y are independent of each other, which indicates 2-d Gaussian filter is separable and can be divided into two 1-d Gaussian filters. The mean difference of result after one 2d filtering and two 1d filterings is negligible, proving the separability experimentally.

Mathematically, the separablility can be proved like this:

$f[x, y] * h[x, y]$

$= \Sigma_i \Sigma_j f[x-i, y-j] \cdot h[i, j]$

$= \Sigma_i \Sigma_j f[x-i, y-j] \cdot \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$

$= \Sigma_i (\Sigma_j f[x-i, y-j] \cdot \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{j^2}{2\sigma^2}}) \cdot \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{i^2}{2\sigma^2}}$

$= \Sigma_i (\Sigma_j f[x-i, y-j] \cdot h[j]) \cdot h[i]$

$= \Sigma_i (f * h_y)[x-i] \cdot h[i]$

$= (f * h_y) * h_x$

2. Computational time

- One 2D Gaussian filtering spends more time compared with two 1D Gaussian filterings. This is because doing one 2D filtering with kernel size K on the image with size N has $O(K^2N^2)$ time complexity while doing one 1D filtering has $O(K^2N^2)$ complexity.

1.5 3. Challenge: Implement a 2D Gaussian filter using Pytorch.

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.

1.5.1 3.1 Expand the dimension of the noisy image into 1x1xXxY and convert it to a Pytorch tensor. (7 points)

[14]:
```
# Import libaries (provided)
import torch
import torch.nn as nn
```

[15]:
```
# Expand the dimension of the numpy array
image_noisy_torch = torch.tensor(image_noisy).reshape(1, 1, image_noisy.
 ↪shape[0], image_noisy.shape[1])

# Convert to a Pytorch tensor using torch.from_numpy
def gaussian_filter_torch(sigma):
    """
    create a torch-version gaussian filter

    Arguments:
    sigma (int): the parameter sigma in the Gaussian kernel (unit: pixel)

    Returns:
    a 2D Gaussian kernel tensor.
    """
    size = sigma*6 +1
    k_1d = scipy.signal.gaussian(size, std = sigma).reshape(-1, size)
    k_2d = np.outer(k_1d, k_1d)
    return torch.from_numpy(k_2d/k_2d.sum())
```

1.5.2 3.2 Create a Pytorch Conv2D filter, set its kernel to be a 2D Gaussian filter. (7 points)

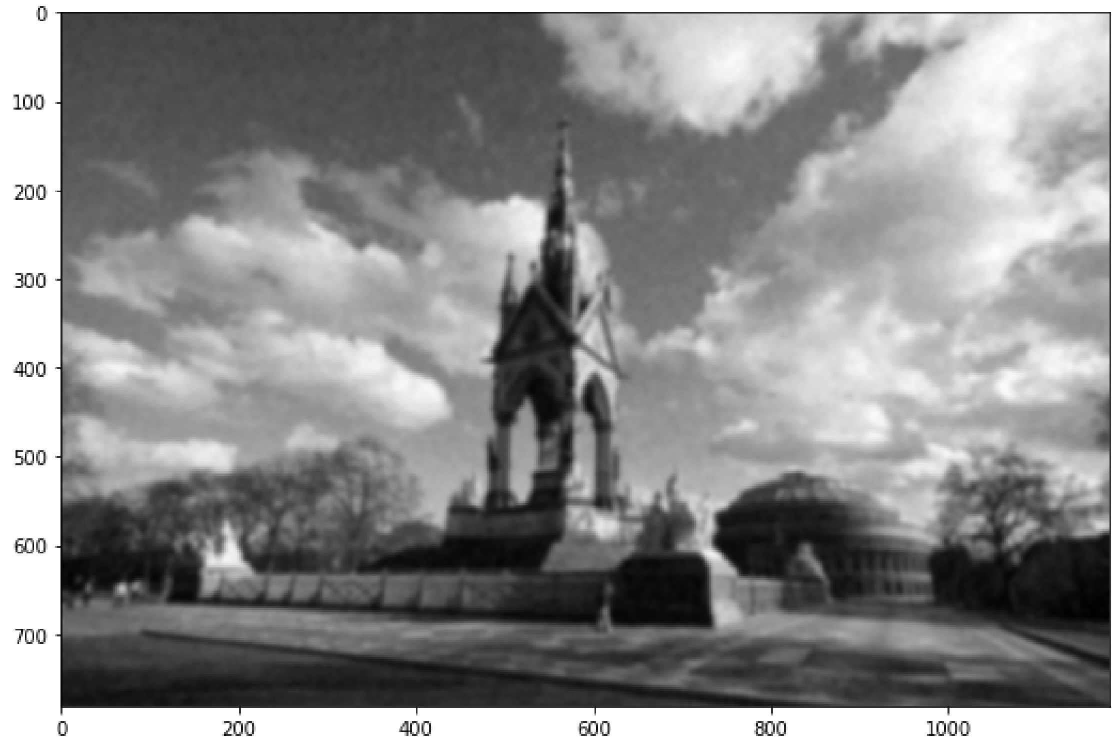
[16]:
```
# A 2D Gaussian filter when sigma = 3 pixel (provided)
sigma = 3
h = gaussian_filter_2d(sigma)
print(h.shape)
```

```
# Create the Conv2D filter (provided)
conv = nn.Conv2d(in_channels = 1, out_channels = 1, kernel_size = sigma,␣
↪bias=False)

# Set the kernel weight
with torch.no_grad():
    conv.weight = nn.Parameter(torch.tensor(h.reshape((1, 1, h.shape[0], h.
↪shape[1]))))
```

```
(19, 19)
```

1.5.3 3.3 Apply the filter to the noisy image tensor and display the output image. (6 points)

[17]:
```
# Filtering
image_filtered = conv.forward(image_noisy_torch)

# Display the filtering result (provided)
plt.imshow(image_filtered.detach()[0][0], cmap='gray')
plt.gcf().set_size_inches(10, 8)
```



1.6 4. Survey: How long does it take you to complete the coursework?

Around 8 hours