4. Image Convolution

Table of Contents

Importing Libraries

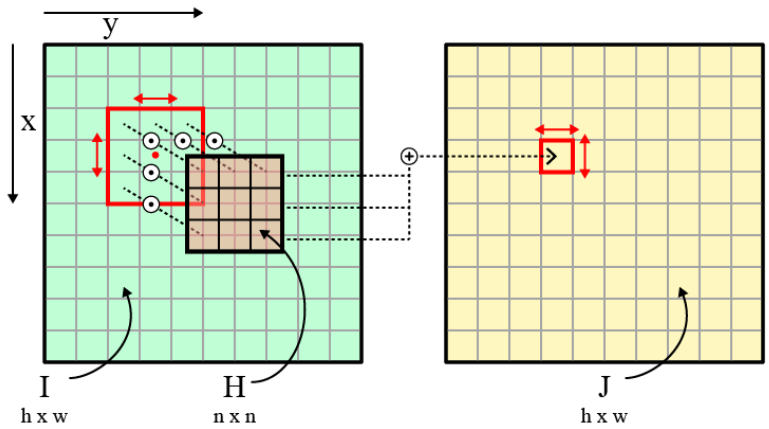
In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import cv2
```

Simple Convolution

Definition

- **I**: Image to convolve.
- **H**: filter matrix to convolve the image with.
- **J**: Result of the convolution.

The following graphics shows exemplary the mathematical operations of the convolution. The filter matrix **H** is shifted over the input image **I**. The values 'under' the filter matrix are multiplicated with the corresponding values in **H**, summed up and writen to the result **J**. The target position is usually the position under the center of **H**.



In order to implement the convolution with a block filter, we need two methods. The first

one will create the block filter matrix **H** depending on the filter width/height **n**.

A block filter holds the value $\frac{1}{n \cdot n}$ at each position:

In [ ]:
```
def block_filter(n):
    H = np.ones((n, n)) / (n * n) # each element in H has the value 1/(n*n)
    return H
```

We will test the method by creating a filter with `n = 5` :

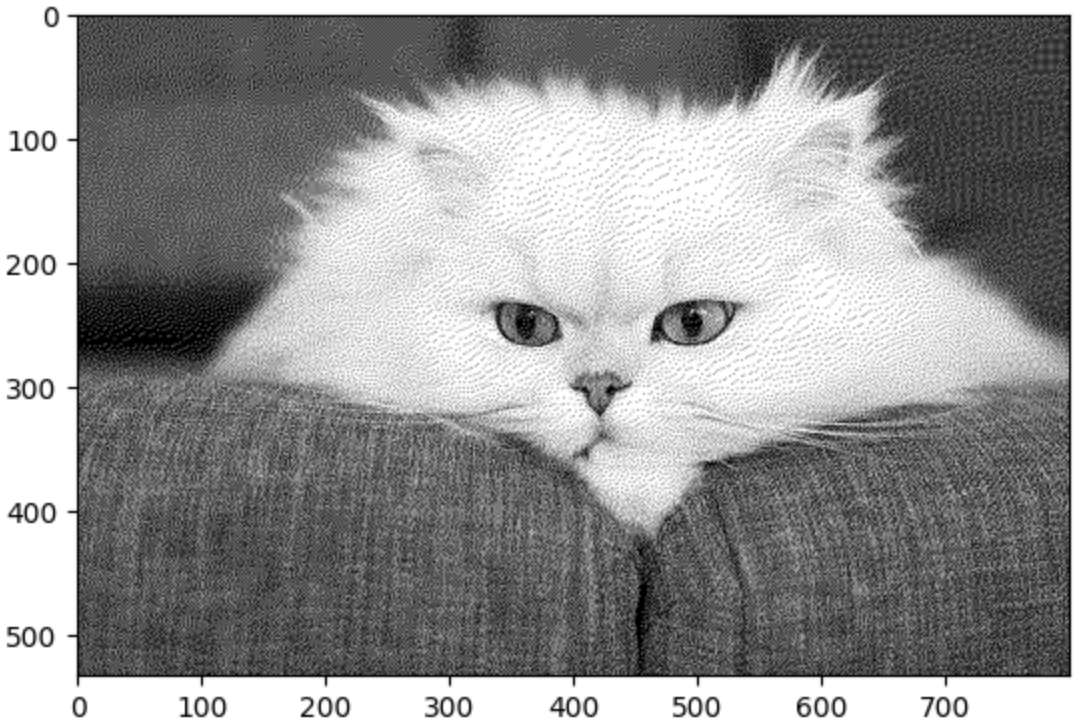
In [ ]:
```
H = block_filter(5)
print(H)
```

```
[[0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]]
```

Next, we define the actual convolution operation. To prevent invalid indices at the border of the image, we introduce the padding **p**.

In [ ]:
```
def apply_filter(I, H):
    h, w = I.shape                          # image dimensions (height, width
    n = H.shape[0]                          # filter size
    p = n // 2                              # padding size
    J = np.zeros_like(I)                    # output image, initialized with

    for x in range(p, h-p):
        for y in range(p, w-p):
            J[x, y] = np.sum(I[x-p:x+n-p, y-p:y+n-p] * H)
    return J
```

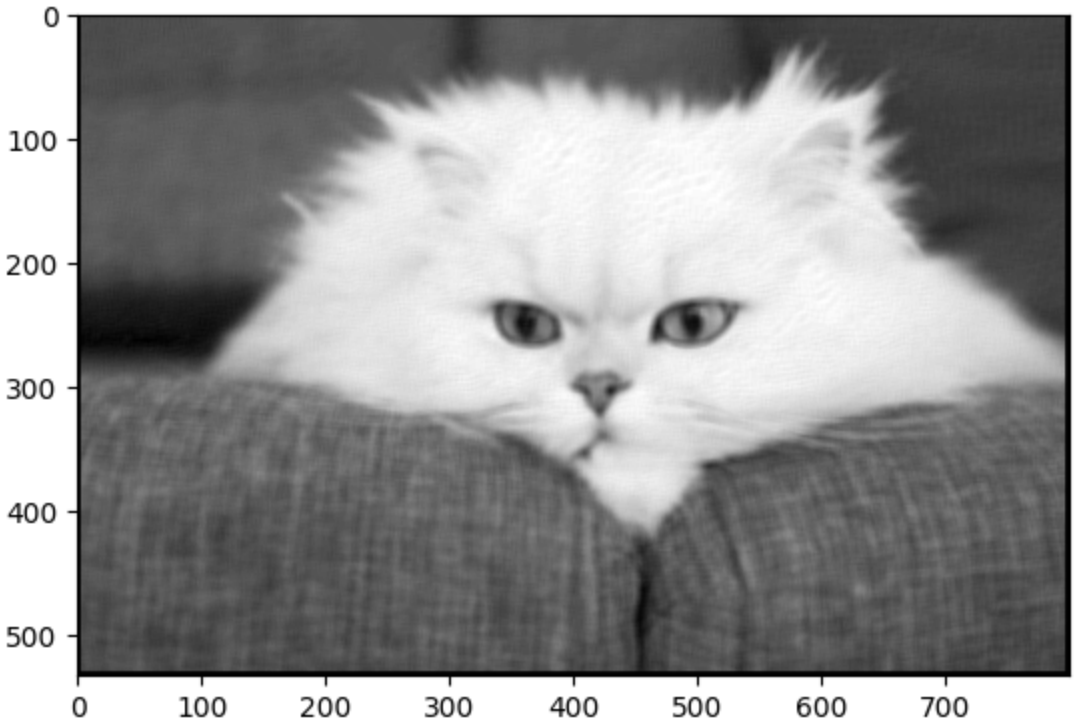
In [ ]:
```
image = Image.open('data/image.jpg')
image = image.convert('1') # convert image to black and white

image = np.array(image)

# image = np.zeros((200, 200), dtype=np.float)
# for x in range(200):
#     for y in range(200):
#         d = ((x-100)**2+(y-100)**2)**0.5
#         image[x, y] = d % 8 < 4

plt.imshow(image, cmap='gray',vmin=0.0, vmax=1.0)
plt.show()
```



In [ ]:
```
image = image.astype(float)
```

Next we test our implementation and apply a block filter with size 7

In [ ]:
```
n = 7
H = block_filter(n)
J = apply_filter(image, H)

plt.imshow(J, cmap='gray')
plt.show()
```



PyTorch Convolution

In [ ]:
```
from PIL import Image

img = Image.open('data/image.jpg')
img.thumbnail((256,256), Image.ANTIALIAS) # Resize to half to reduce the siz
```

```
<ipython-input-8-7c8c44ebc1df>:4: DeprecationWarning: ANTIALIAS is deprecate
d and will be removed in Pillow 10 (2023-07-01). Use LANCZOS or Resampling.L
ANCZOS instead.
  img.thumbnail((256,256), Image.ANTIALIAS) # Resize to half to reduce the s
ize of this notebook.
```

In [ ]:
```
img
```

Out[ ]:

In [ ]:
```
import torch, torchvision
from torchvision import transforms
from torch import nn
```

In [ ]:
```
to_tensor = transforms.Compose([
```

```
   transforms.Grayscale(),  # Convert image to grayscale.
   transforms.ToTensor()    # Converts a PIL Image in the range [0, 255] to
])

to_pil = transforms.Compose([
    transforms.ToPILImage()
])
```

In [ ]:
```
input = to_tensor(img)
input.shape
```

Out[ ]:
```
torch.Size([1, 171, 256])
```

In [ ]:
```
to_pil(input)
```

Out[ ]:

2D convolution over an input image:

- `in_channels = 1` : an input is a grayscale image
- `out_channels = 1` : an output is a grayscale image
- `kernel_size = (3, 3)` : the kernel (filter) size is 3 x 3
- `stride = 1` : the stride for the cross-correlation is 1
- `padding = 1` : zero-paddings on both sides for 1 point for each dimension
- `bias = False` : no bias parameter (for simplicity)

In [ ]:
```
conv = nn.Conv2d(1, 1, (3, 3), stride=1, padding=1, bias=False)
```

In [ ]:
```
# The code below does not work because the convolution layer requires the di
conv(input)
```

Out[ ]:
```
tensor([[[ 0.0119,  0.0131,  0.0134,  ...,  0.0095,  0.0095,  0.0729],
         [ 0.0630,  0.0031,  0.0041,  ...,  0.0041,  0.0041,  0.1083],
         [ 0.0615, -0.0004,  0.0029,  ...,  0.0019,  0.0006,  0.1036],
         ...,
         [ 0.0523,  0.0061,  0.0251,  ...,  0.0036,  0.0200,  0.0986],
         [ 0.0387, -0.0170,  0.0079,  ...,  0.0333, -0.0007,  0.1073],
         [ 0.0202, -0.1253, -0.1078,  ..., -0.1028, -0.1431, -0.0165]]],
       grad_fn=<SqueezeBackward1>)
```

We need to insert a dimension for a batch at dim=0.

In [ ]:
```
input = input.unsqueeze(0)
input.shape
```

Out[ ]: torch.Size([1, 1, 171, 256])

In [ ]:
```
output = conv(input)
output.shape
```

Out[ ]: torch.Size([1, 1, 171, 256])

Setting `padding=1` in the convolution layer, we obtain an image of the same size.

In [ ]:
```
output.shape
```

Out[ ]: torch.Size([1, 1, 171, 256])

We need to remove the first dimension before converting to a PIL object.

In [ ]:
```
output.data.squeeze(dim=0).shape
```

Out[ ]: torch.Size([1, 171, 256])

Display the output from the convolution layer by converting `output` to a PIL object.

In [ ]:
```
to_pil(output.data.squeeze(dim=0))
```

Out[ ]:

Clip every value in the output tensor within the range of [0, 1].

In [ ]:
```
to_pil(torch.clamp(output, 0, 1).data.squeeze(dim=0))
```

Out[ ]:

In [ ]:
```
def display(img1, img2):
    im1 = to_pil(torch.clamp(img1, 0, 1).data.squeeze(dim=0))
    im2 = to_pil(torch.clamp(img2, 0, 1).data.squeeze(dim=0))
    dst = Image.new('RGB', (im1.width + im2.width, im1.height))
    dst.paste(im1, (0, 0))
```

```
    dst.paste(im2, (im1.width, 0))
    return dst
```

In [ ]:
```
display(input, output)
```

Out[ ]:

Identity

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [0., 0., 0.],
    [0., 1, 0.],
    [0., 0., 0.],
]]])

output = conv(input)
display(input, output)
```

Out[ ]:

Brighten

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [0., 0., 0.],
    [0., 1.5, 0.],
    [0., 0., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[0.0000, 0.0000, 0.0000],
          [0.0000, 1.5000, 0.0000],
          [0.0000, 0.0000, 0.0000]]]])
```

Out[ ]:

Darken

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [0., 0., 0.],
    [0., 0.5, 0.],
    [0., 0., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[0.0000, 0.0000, 0.0000],
          [0.0000, 0.5000, 0.0000],
          [0.0000, 0.0000, 0.0000]]]])
```

Out[ ]:

Box blur

In [ ]:
```
conv.weight.data = torch.ones((1, 1, 3,3), dtype=torch.float) / 9.
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[0.1111, 0.1111, 0.1111],
          [0.1111, 0.1111, 0.1111],
          [0.1111, 0.1111, 0.1111]]]])
```

Out[ ]:

Gaussian blur

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [1., 2., 1.],
    [2., 4., 2.],
    [1., 2., 1.],
]]])/16.
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[0.0625, 0.1250, 0.0625],
          [0.1250, 0.2500, 0.1250],
          [0.0625, 0.1250, 0.0625]]]])
```

Out[ ]:

Sharpen

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [0., -1., 0.],
    [-1., 5., -1.],
    [0., -1., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

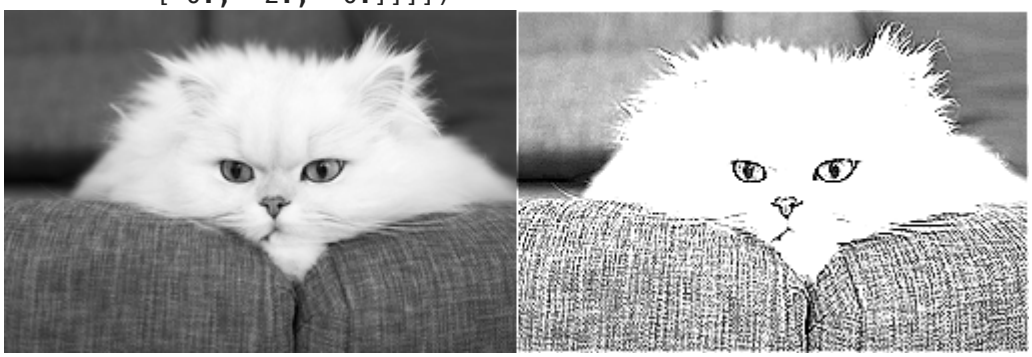
```
tensor([[[[ 0., -1.,  0.],
          [-1.,  5., -1.],
          [ 0., -1.,  0.]]]])
```

Out[ ]:

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [0., -2., 0.],
    [-2., 10., -2.],
    [0., -2., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[ 0., -2.,  0.],
          [-2., 10., -2.],
          [ 0., -2.,  0.]]]])
```

Out[ ]:



Edge detection

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [0., 1., 0.],
    [1., -4., 1.],
    [0., 1., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[ 0.,  1.,  0.],
          [ 1., -4.,  1.],
          [ 0.,  1.,  0.]]]])
```

Out[ ]:

In [ ]:
```
conv.weight.data = torch.tensor([[[
    [-1., -1., -1.],
    [-1., 8., -1.],
    [-1., -1., -1.],
]]])
output = conv(input)
display(input, output)
```

Out[ ]:

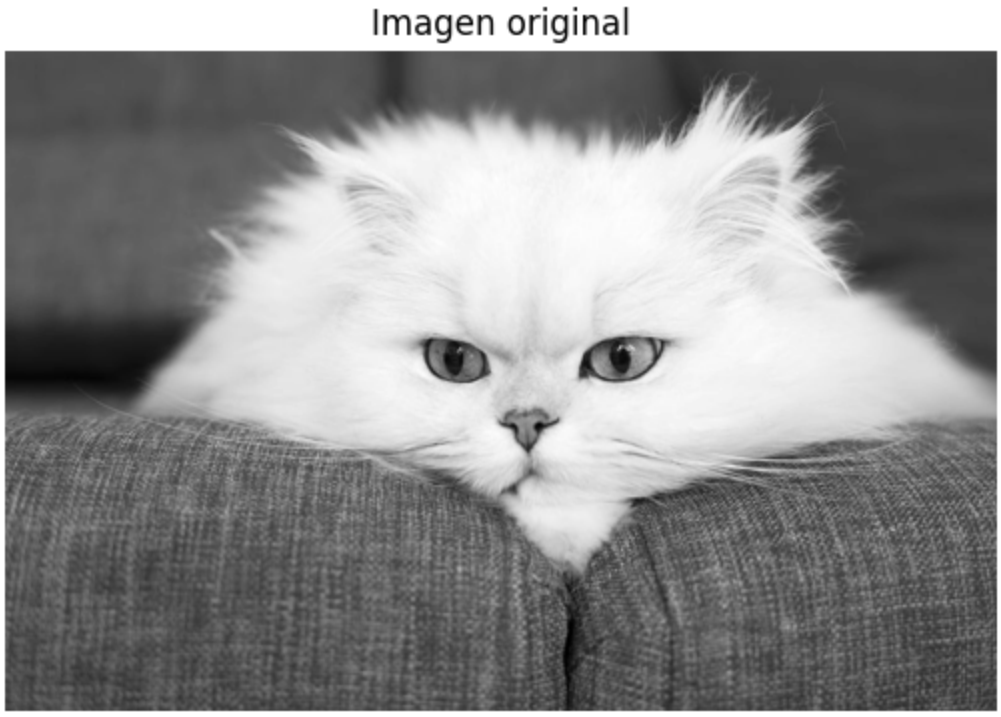
Homework: Additional exercises

1. Implement the following line detectors using Python code (i.e. without using OpenCV libraries): Prewitt, Sobel and Laplacian. Investigate the algorithmic complexity of these, which is more efficient?

In [ ]:
```
def display_image(image, title):
  plt.imshow(image, cmap='gray')
  plt.title(title)
  plt.axis('off')
```

In [ ]:
```
image = Image.open('data/image.jpg')
input = to_tensor(image)
input = input.squeeze(0).numpy()
```

In [ ]:
```
display_image(input, 'Imagen original')
```



In [ ]:
```python
# The same functions are used by the three operators, so we can define gener

def convolve(image, mask):
  return np.sum(np.multiply(image, mask[::-1, ::-1]))

def apply_mask(image, mask):
    m, n = mask.shape
    y, x = image.shape

    pad_height = (m - 1) // 2
    pad_width = (n - 1) // 2

    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width, pad_

    result = np.zeros((y, x))
    for i in range(y):
        for j in range(x):
            result[i, j] = convolve(padded_image[i:i+m, j:j+n], mask)

    return result
```

1.1. Prewitt

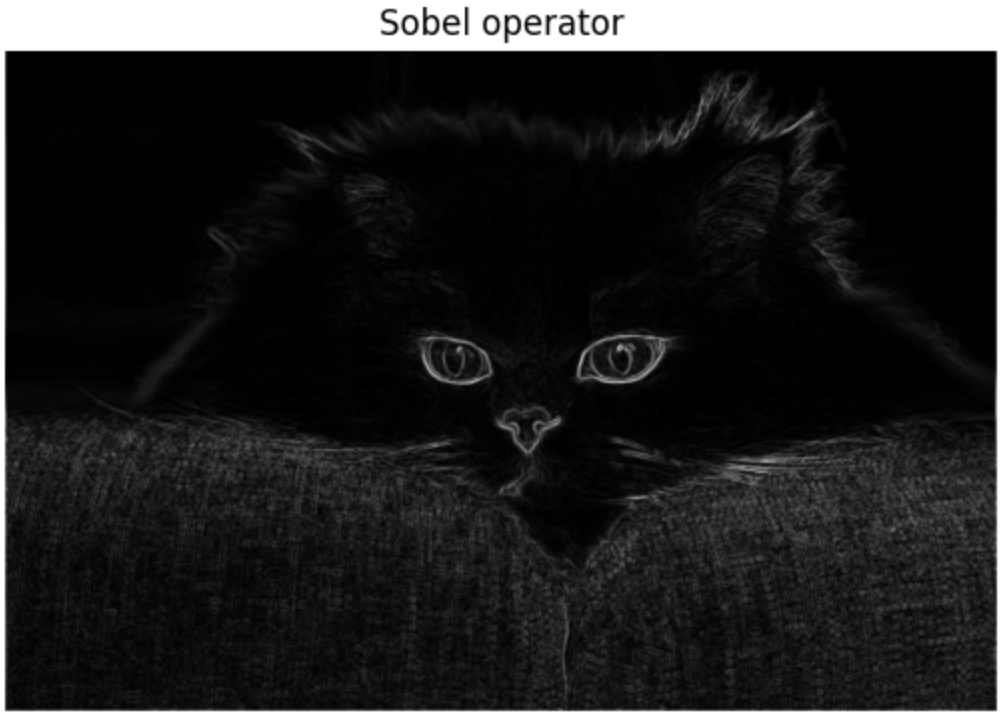
In [ ]:
```python
def prewitt_operator(image):
  vertical_mask = np.array([[-1, 0, 1]]*3) # Since for this operator the mat
  horizontal_mask = np.array([[-1]*3, [0]*3,[1]*3])

  gradient_x = apply_mask(image, vertical_mask)
  gradient_y = apply_mask(image, horizontal_mask)

  return np.sqrt(gradient_x**2 + gradient_y**2)
```

In [ ]:
```
prewitt_result = prewitt_operator(input)

display_image(prewitt_result, 'Prewitt Operator')
```

1.2. Sobel

In [ ]:
```
def sobel_operator(image):
  vertical_mask = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
  horizontal_mask = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

  gradient_x = apply_mask(image, vertical_mask)
  gradient_y = apply_mask(image, horizontal_mask)

  return np.sqrt(gradient_x**2 + gradient_y**2)
```

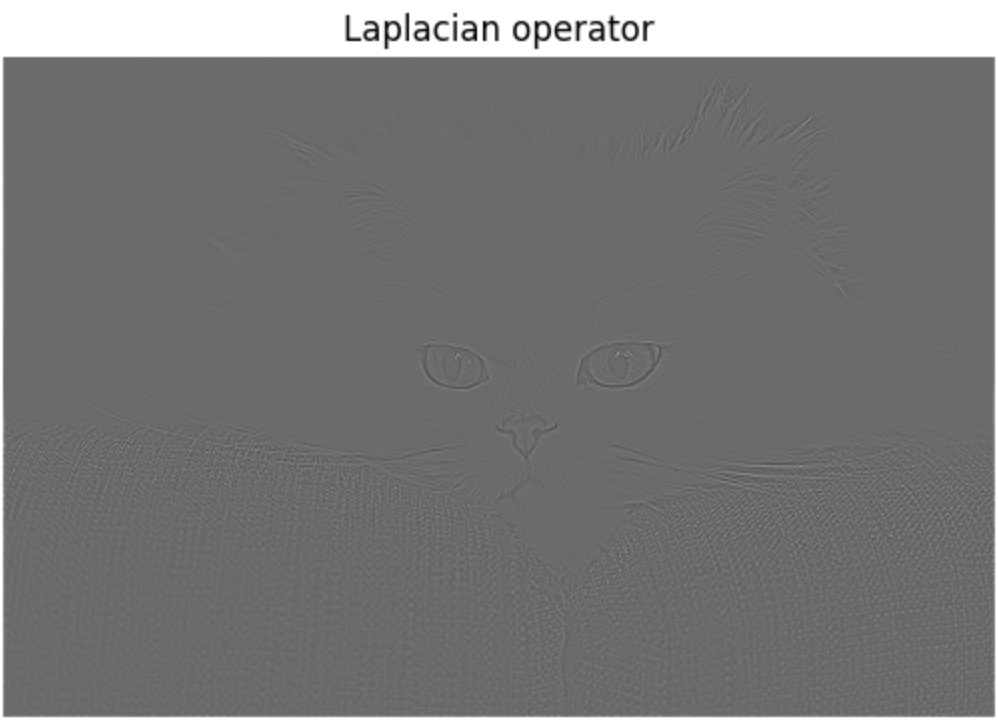
In [ ]:
```
sobel_result = sobel_operator(input)

display_image(sobel_result, 'Sobel operator')
```



1.3. Laplacian

In [ ]:
```
def laplacian_operator(image):
  laplacian_mask = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])

  laplacian_result = apply_mask(image, laplacian_mask)

  return laplacian_result
```

In [ ]:
```
laplacian_result = laplacian_operator(input)

display_image(laplacian_result, 'Laplacian operator')
```



The computational complexity of the Prewitt, Sobel and Laplacian convolution operators depends on the image size and the size of the mask used in the convolution. Therefore, the complexity of the image can be expressed using:

- `m x n` as the size of the image (pixels).
- The size of all the mask matrices is `3 x 3 = 9`.

1. **Prewitt Operator:**
   - Convolution using the Prewitt operator involves two separate steps for the horizontal and vertical derivatives. Each step involves a double loop that loops through the image and applies a kernel of size `3 x 3`.
   - The total computational complexity can be expressed as `O(2*(m * n * 9))`, simplified as `O(m * n * 18)`.
2. **Sobel Operator:**
   - Similar to Prewitt, the Sobel operator also performs two separate steps for the horizontal and vertical derivatives.
   - The computational complexity for this mask type is also `O(m * n * 18)`.
3. **Laplacian Operator:**
   - The Laplacian operator involves a single step that applies a similar mask of size 3x3.
   - Therefore, the complexity would be `O(m * n * 9)`.

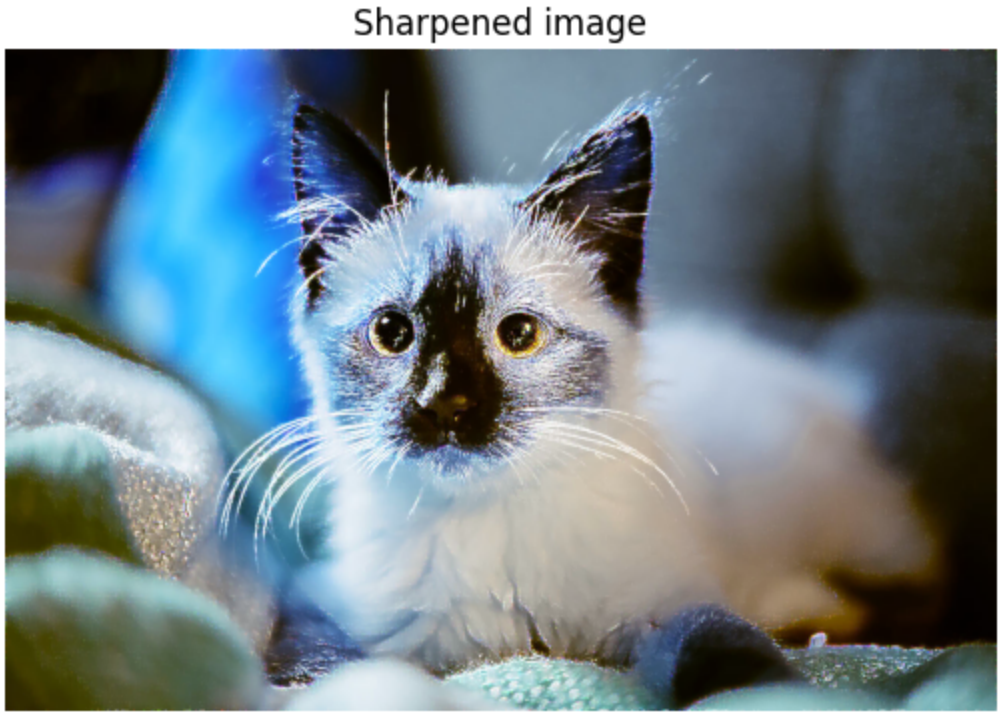
2. Implement an image enhancement or enhancement algorithm using an algorithm in which the lines are extracted from the image and then the difference with the original image is applied, multiplying the pixels of the "mask" image (the lines found) by an alpha factor greater than 1.

In [ ]:
```
image = cv2.imread('data/kitty-baseline.jpg')

display_image(image, 'Original image')
```

In [ ]:
```
# Use Sobel to get the edges.
edges = sobel_operator(cv2.cvtColor(image, cv2.COLOR_BGR2GRAY))

edges_normalized = cv2.normalize(edges, None, 0, 255, cv2.NORM_MINMAX).astyp
```

In [ ]:
```
# Apply Gaussian Blur to the color image.
blurred_image = cv2.GaussianBlur(image, (3, 3), 0)
mask = cv2.subtract(image, blurred_image)
edges_3channel = np.stack((edges_normalized, edges_normalized, edges_normali

# Ensure the mask and edges_3channel have the same shape.
mask_cropped = mask[1:-1, 1:-1]
mask_cropped_resized = cv2.resize(mask_cropped, (edges_3channel.shape[1], ed

# Combine the mask with the edges.
combined_mask = np.multiply(mask_cropped_resized, edges_3channel)
combined_mask_resized = cv2.resize(combined_mask, (image.shape[1], image.sha

# Apply the combined mask for sharpening.
alpha_factor = 3
sharpened_image = cv2.addWeighted(image, 1, combined_mask_resized, alpha_fac
```

In [ ]:
```
display_image(sharpened_image, 'Sharpened image')
```

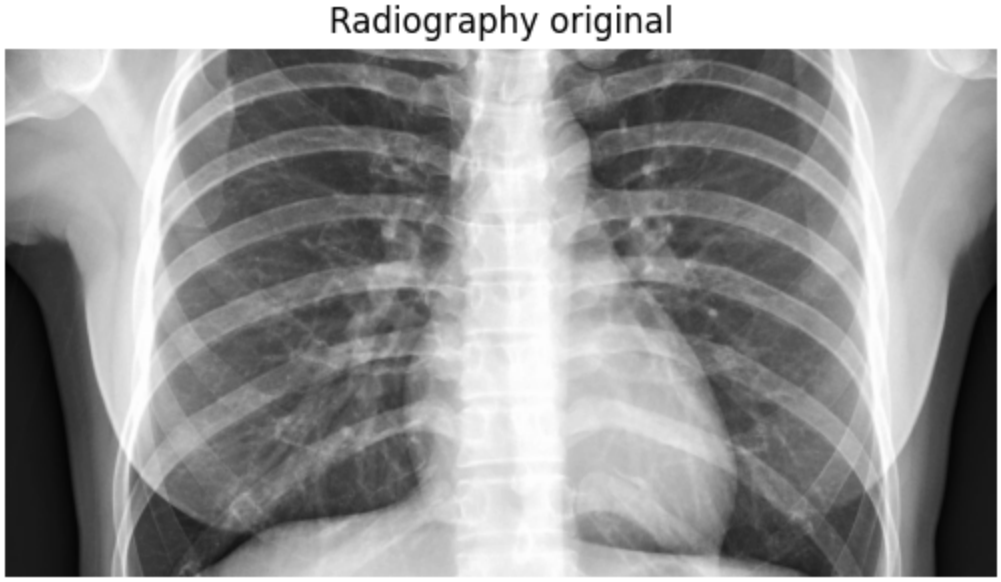


3. Find a medical application (Gonzalez's book includes several examples of PET images) and make enhancements using the technique in Figure 3.43 combining different stages of image processing

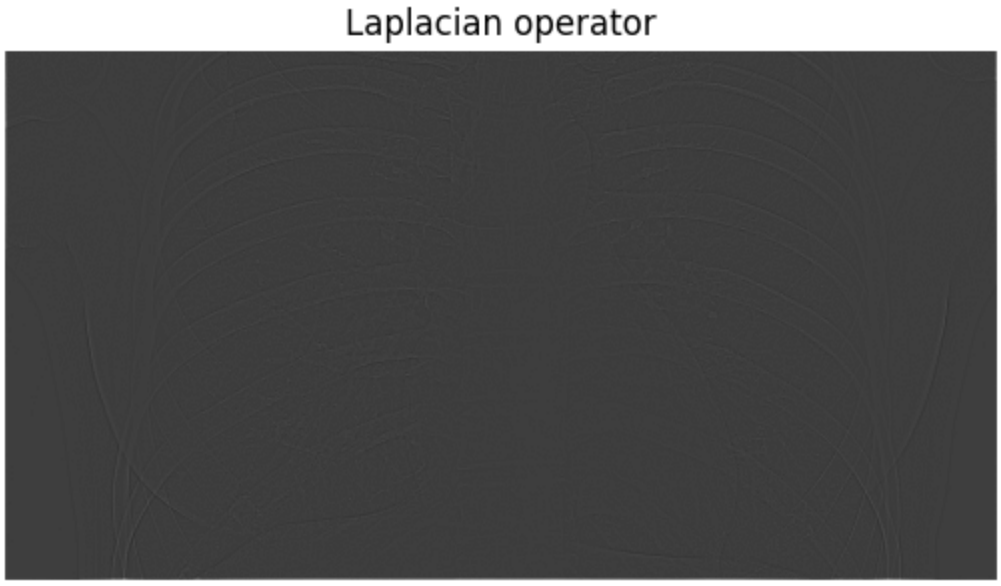
Let's enhance the radiography image of the 3rd section of the book.

In [ ]:
```
# Importing radiography image.
image_radiography = Image.open('data/radiography.jpg')
input_radiography = to_tensor(image_radiography)
input_radiography = input_radiography.squeeze(0).numpy()
```

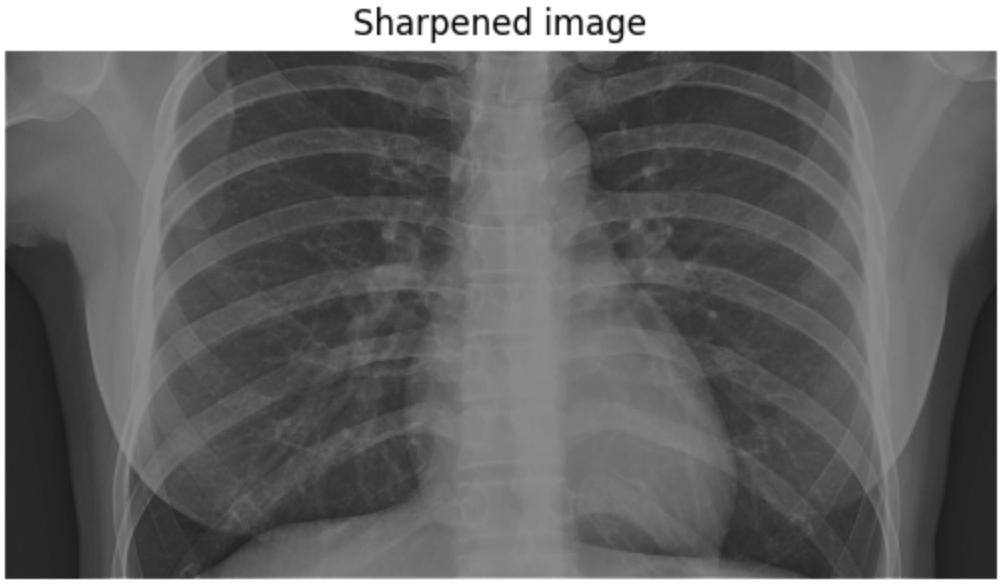
In [ ]:
```
display_image(input_radiography, 'Radiography original')
```



In [ ]:
```
# Applying Laplacian operator.
laplacian_result_radiography = laplacian_operator(input_radiography)
display_image(laplacian_result_radiography, 'Laplacian operator')
```



In [ ]:
```
# Sharpened image obtained by adding the original image with the Laplacian.
sharpened_image_radiography = input_radiography + laplacian_result_radiograp

# Min max normalization.
min_value = np.min(sharpened_image_radiography)
max_value = np.max(sharpened_image_radiography)
sharpened_image_radiography = (sharpened_image_radiography - min_value) / (m

display_image(sharpened_image_radiography, 'Sharpened image')
```



In [ ]:
```
# Adding Sobel operator to the original image.
sobel_result_radiography = sobel_operator(input_radiography)
display_image(sobel_result_radiography, 'Sobel operator')
```