Laboratory 2. Image filtering

Operations like Image Filtering are often used as a preprocessing step in Computer Vision applications. The concept of signal frequency, known from 1D digital signal processing, can be extended to signals in a 2D space, meaning images. The term indicates in a 2D signal the spatial frequency. The spatial frequency is a measure of how often sinusoidal components (as determined by the Fourier transform) of the structure repeat per unit of distance. An image region is said to have low frequency information if it is smooth and does not have a lot of texture. An image area is considered to have high frequency information if it has a lot of texture (edges, corners etc.). In Figure1 are displayed the Fourier transforms corresponding to the logo images above them. In the center of the spectrum plots are situated the zero-frequency components, while going from the center to the exterior, higher frequency components are displayed with a color-coded amplitude. Larger dark blue areas (for low amplitude frequency components) correspond to an image that is more uniform, without significant texture. Warm colors in the spectrum indicate higher amplitudes. The original logo image contains more elements (contours, objects, …) and so the associated spectrum has also significantly more higher frequency components.

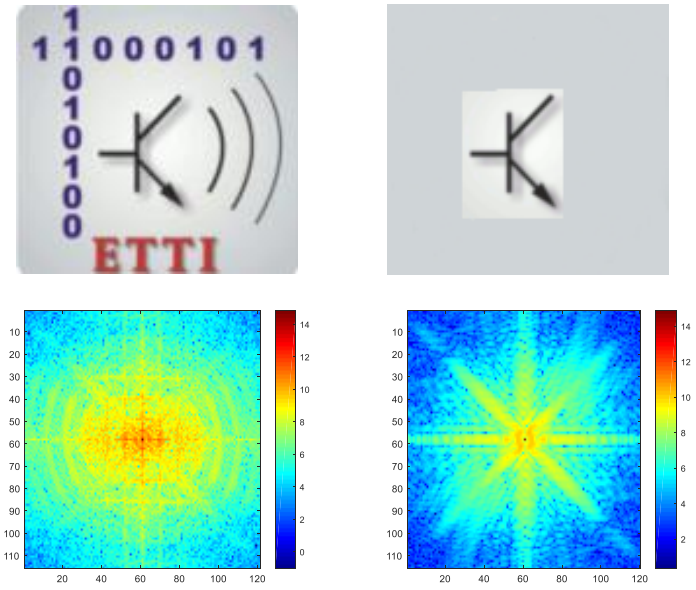

Figure 1. Input images (top) and their corresponding spectra (bottom)

Low Pass Filtering (LPF) is in essence the operation of blurring or smoothing of an input image. Blurring is obtained by discarding the fine texture: it allows the lower frequency information to pass and it blocks higher frequency information.

High Pass Filtering (HPF) represents a sharpening or edge enhancement type of operation. In this case, the low frequency information is suppressed and the high frequency information is preserved.

2.1 Image filtering

Image Filtering is a broad term applied to a variety of Image Processing techniques that enhance an image by eliminating unwanted characteristics (e.g. noise) or improve desired characteristics (e.g. better contrast). Blurring, edge detection, edge sharpening, and noise removal are all examples of image filtering.

Image filtering is a neighborhood (or local) operation. This means that the pixel value at location (x, y) in the output image depends on the pixels in a small neighborhood of location (x, y) in the input image. For example, image filtering using a 3x3 kernel (the 2D equivalent of the impulse response) would make the output pixel at location (x, y) depend on the input pixels at locations (x, y) and its 8 neighbors, as displayed in Figure 2.

197	198	203	205	199	145	75	58	55	53
201	203	205	205	198	136	84	108	81	54
203	205	205	205	202	178	168	179	103	56
204	205	205	206	206	203	205	191	103	56
204	205	205	204	205	204	205	191	102	56
203	201	202	203	203	203	205	191	103	56
203	201	202	203	203	203	206	192	105	58
204	202	203	203	203	204	205	199	150	123
205	202	203	203	203	204	206	206	200	198
203	203	203	203	204	204	206	206	207	207

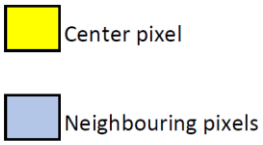

Figure 2. Example of an input image that will be filtered with a 3x3 kernel. One output pixel will be a weighted combination of its collocated pixel (yellow) and neighborhood values (blue). The weights will be specified in the kernel.

When the output pixel depends only on a linear combination of the input pixels, we call the filter a *Linear Filter*. Otherwise, it is called a *Nonlinear Filter*.

2.1.1 Convolution

Linear filters are implemented using 2D convolution. Operations like blurring and contour detection are realized using convolution. Let us consider a 3x3 kernel:

$$\mathrm{h} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

that we will use to filter the yellow center pixel in Figure 2. The output filtered pixel value is computed by multiplying corresponding elements of the input blue & yellow patch and the convolution kernel, followed by adding up all the products:

$$out_{center} = 206 \times (-1) + 203 \times 0 + 205 \times 1 + 205 \times (-2) + 204 \times 0 + 205 \times 2 + 203 \times (-1) + 203 \times 0 + 205 \times 1 = 1$$

In order to filter an entire image, this process is repeated pixel by pixel (the kernel slides through the image, left to right, top to bottom). For color images, the convolution is performed independently on each channel. At the boundary, the convolution is not uniquely defined. There are several options to choose from:

- Ignore the boundary pixels: by discarding the boundary pixels, the output image will be slightly smaller than the input image.
- Zero padding: the input image is padded with zeros at the boundary pixels to make it larger and then convolution is performed.

- Replicate border: another option is to replicate the boundary pixels of the input image and then perform the convolution operation on this larger image.
- Reflect border: the preferred option is to reflect the border about the boundary. Reflecting ensures a smooth intensity transition of pixels at the boundary.

In OpenCV, the convolution is performed using the function `filter2D`. The function syntax is:

```
dst = cv2.filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]])
```

with parameters:

`src` input image.

`dst` output image of the same size and the same number of channels as `src`.

`ddepth` desired depth of the destination image.

`kernel` convolution kernel, a single-channel floating point matrix; if you want to apply different kernels to different channels, split the image into separate color planes using split and process them individually.

`anchor` anchor of the kernel that indicates the relative position of a filtered point within the kernel; the anchor should lie within the kernel; the default value (-1,-1) means that the anchor is at the kernel center.

`delta` an optional value added to the filtered pixels before storing them in `dst`.

`borderType` pixel extrapolation method.

The optional parameters like `anchor` point, `delta` and `borderType` are almost never changed from their default values.

For the function `cv2.filter2D` in the OpenCV documentation, the term ***convolution*** is wrongly used instead of ***cross-correlation***. It actually performs cross-correlation, which is identical to convolution only if the filter kernel is symmetric!

Example:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

image = cv2.imread(”test.jpg”)

if image is None:       # Check if file is not present
    print("Could not open the image")
ker_size = 5
# Box kernel: 5*5 kernel with the sum of all the elements equal to 1
```

```
kernel = np.ones((ker_size, ker_size), dtype=np.float32) / ker_size**2

print (kernel)

result = cv2.filter2D(image, -1, kernel, (-1, -1), delta=0, borderType =
                                                        cv2.BORDER_DEFAULT)
plt.figure()
plt.subplot(121);plt.imshow(image[...,::-1]);plt.title("Original Image")
plt.subplot(122);plt.imshow(result[...,::-1]);plt.title("Convolution Result")
plt.show()
```

The second parameter of `cv2.filter2D` (`depth`) is set to `-1`, which means the bit-depth of the output image is the same as the input image. So if the input image is of type `uint8`, the output image will also be of the same type.

- **Ex. 2.1** Change the kernel used in the previous example and analyze the results. Try to use a kernel without normalizing the values by the number of elements in the kernel. How is this altering the output image brightness level compared to the input image?
- **Ex. 2.2** Modify the previous example by testing the following kernels on the input images "white_square.jpg" and "hop.jpg".

$$h_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad h_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad h_3 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad h_4 = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Describe the effect of each kernel on the input image.

2.1.2 Box blur

In image processing is often necessary to blur or smooth the input image, and this is a type of LPF. Box blur is a smoothing technique that reduces the different types of noise in the input image. It can be implemented in 2 ways. 1st method is to generate a kernel (let's have an example of 3x3)

$$\mathrm{h} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

and convolve the input image with this kernel using `filter2D`. 2nd option is to use the OpenCV function `blur` with the following syntax:

```
dst = cv2.blur( src, ksize[, dst[, anchor[, borderType]]] )
```

`src` – represents the input image and it can have any number of channels, which are processed independently, but the depth should be CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.

`dst` – output image of the same size and type as `src`.

`ksize` stands for the blurring kernel size

`anchor` – anchor point; default value is Point(-1,-1) means that the anchor is at the kernel center.

`borderType` – border mode used to extrapolate pixels outside of the image.

Example:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('logo_noise2.jpg')

# Apply box filter - kernel size 3
dst1 = cv2.blur(img,(3,3),(-1,-1))
# Apply box filter - kernel size 5
dst2 = cv2.blur(img,(5,5),(-1,-1))

plt.figure(figsize=[30,10])
plt.subplot(131);plt.imshow(img[...,::-1]);plt.title("Noisy Image")
plt.subplot(132);plt.imshow(dst1[...,::-1]);plt.title("Smoothed 3x3")
plt.subplot(133);plt.imshow(dst2[...,::-1]);plt.title("Smoothed 5x5")
plt.show()
```

- **Ex. 2.3** Describe both smoothing operations in view of the final results: How much smoother appears the logo background? How are the contours changing with the blurring kernel?

2.1.3 Gaussian filtering in OpenCV

The box kernel explained previously weights the contribution of all pixels in the neighborhood equally. A Gaussian Blur kernel, on the other hand, weights the contribution of a neighboring pixel based on the distance of the pixel from the center pixel. The shape of the curve is controlled by a single parameter called σ that controls how peaky the hill characteristic is:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

A bigger σ creates a kernel that blurs more. A 5x5 Gaussian kernel with $\sigma = 1$ is given by:

$$\frac{1}{337}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 20 & 33 & 20 & 4 \\ 7 & 33 & 55 & 33 & 7 \\ 4 & 20 & 33 & 20 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

It is obvious that the middle pixel gets the maximum weight, while the pixels farther away are given less weight.

An image blurred using the Gaussian kernel looks less blurry compared to a box kernel of the same size. A small amount of Gaussian blurring is frequently used to remove noise from an image. It is also applied to the image prior to a noise sensitive image filtering operation. For example, the Sobel kernel used for calculating the derivative of an image is a combination of a Gaussian kernel and a finite difference kernel.

The OpenCV function that implements the Gaussian filtering is `GaussianBlur` with the following syntax:

```
dst = cv2.GaussianBlur( src, ksize, sigmaX[, dst[, sigmaY[, borderType]]] )
```

`src` – represents the input image; the image can have any number of channels, which are processed independently, but the depth should be CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
`dst` – is the output image of the same size and type as `src`.
`ksize` – Gaussian kernel size. `ksize.width` and `ksize.height` can differ but they both must be positive and odd. Or, they can be zero's and then they are computed from sigma.
`sigmaX` – Gaussian kernel standard deviation in X direction.
`sigmaY` – Gaussian kernel standard deviation in Y direction; if `sigmaY` is zero, it is set to be equal to `sigmaX` – if both sigmas are zeros, they are computed from `ksize.width` and `ksize.height`, respectively; to fully control the result regardless of possible future modifications of all this semantics, it is recommended to specify all of `ksize`, `sigmaX`, and `sigmaY`.
`borderType` – pixel extrapolation method.

- **Ex. 2.4** Add the missing lines of code, indicated by the comments. Describe both smoothing operations comparing the final images: how much smoother appears the logo background? How are the contours in the image changing with the blurring kernel? Try Gaussian filtering for the test images “logo_noise2.jpg” and “lina_noise.jpg” and identify the best kernel size.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('logo_noise.jpg')

# Apply gaussian blur with kernel 5x5 and sigmaX=0, sigmaY=0; output image dst1
# Apply gaussian blur with kernel 15x15 and sigmaX=3, sigmaY=3; ; output image
# dst2

plt.figure()
plt.subplot(131);plt.imshow(img[...,::-1]);plt.title("Noisy Image")
plt.subplot(132);plt.imshow(dst1[...,::-1]);plt.title("Smoothed 5x5")
plt.subplot(133);plt.imshow(dst2[...,::-1]);plt.title("Smoothed 15x15")
plt.show()
```

2.1.4 Median filter

Median blur filtering is a *nonlinear* filtering technique that is mostly used to remove salt-and-pepper noise from images. In color images, salt-and-pepper noise may appear as small random colored spots. The median filter in OpenCV should have a square kernel with the square's side length an odd value. The median blurring filter replaces the value of the central pixel with the *median* of all the pixels within the kernel area. The median value is obtained by sorting the data in increasing order and taking the middle value as an estimate.

Let's consider a 3x3 area from an image affected by salt-and-pepper noise:

$$\begin{bmatrix} 60 & 62 & 59 \\ 61 & 255 & 65 \\ 65 & 60 & 63 \end{bmatrix}$$

It is easy to recognize that the value in the center pixel is way higher than the neighbors, so probably is affected by noise. If we were to use a Box Blur filter to smooth out this noise, the resulting center pixel value would be:

$$\frac{60 + 62 + 59 + 61 + 255 + 65 + 65 + 60 + 63}{9} = 83.3$$

so the patch after box filtering is

$$\begin{bmatrix} 60 & 62 & 59 \\ 61 & 83 & 65 \\ 65 & 60 & 63 \end{bmatrix}$$

The result is clearly an improvement, but still the center value remains higher than the neighbors. Using the median filter, the sorted pixels will be

$$[59\ 60\ 60\ 61\ \mathbf{62}\ 63\ 65\ 65\ 255]$$

and the value in the middle of the sorted list is now 62, so, after filtering the center pixel, the patch becomes

$$\begin{bmatrix} 60 & 62 & 59 \\ 61 & 62 & 65 \\ 65 & 60 & 63 \end{bmatrix}$$

This process is repeated in the same manner for all the pixels in the input image.

- **Ex. 2.5** Add the missing lines of code, indicated by the comments. What type of texture is affected the most by the salt-and-pepper noise? Which smoothing filter performs better on the edges? In order to preserve the edges, is it better to use a larger kernel or a smaller one? Which removes the noise most efficiently? Try experimenting with other kernel sizes as well!

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread('lina_noise.jpg')

# Apply gaussian blur with kernel 5x5 and sigmaX=0, sigmaY=0; output image dst1
```

```
kernelSize = 5
# Performing Median Blurring with kernelSize=5 and store it in numpy array
# "dst2"

plt.figure()
plt.subplot(131);plt.imshow(img[...,::-1]);plt.title("Noisy Image")
plt.subplot(132);plt.imshow(dst1[...,::-1]);plt.title("gaussian 5x5")
plt.subplot(133);plt.imshow(dst2[...,::-1]);plt.title("median 5x5")
plt.show()
```

2.1.5 Bilateral filter

Another *nonlinear* smoothing technique is the Bilateral filter. This filter preserves the edges and reduces the noise. Most smoothing filters (e.g. a Gaussian or a Box filter) have a parameter called σ_s (the s in the subscript stands for "spatial") that determines the amount of smoothing. Often this value is closely related to the kernel size. A typical smoothing filter replaces the intensity value of a pixel by the weighted sum of its neighbors. The bigger the neighborhood, the smoother the filtered image looks. The size of the neighborhood is directly proportional to the parameter σ_s.

In edge-preserving filters, there are 2 competing objectives:

- smooth the image.
- do not smooth the edges / color boundaries.

Let us consider a 3x3 image block with the following values:

$$\begin{bmatrix} 60 & 200 & 239 \\ 61 & 220 & 235 \\ 65 & 210 & 233 \end{bmatrix}$$

It is easy to notice that the values in the left column are much lower than the values in the center and the right columns. This block is part of a vertical edge. The center pixel is filtered in this case based on only the center and right-hand side columns, so that the edge is retained and not blurred-out. In bilateral filtering, while calculating the contribution of any pixel to the final output, we weigh the pixels that are close in terms of intensity to the center pixel higher as compared to the pixels whose intensities are very different from the center pixels. That weight is the following modified Gaussian function

$$G_{\sigma_r}(I_c - I_n)$$

where the intensity difference between the center pixel I_c and the neighboring pixel I_n is controlled by the parameter σ_r (the subscript r stands for '*range*').

Additionally, just like Gaussian filtering, we also want to weight the pixels that are closer to the center pixel higher than the pixels that are farther away, so the weights should depend on $\|c - n\|$. The weight in this case will be a Gaussian $G_{\sigma_s}(\|c - n\|)$.

Combining the two, a bilateral filter will output the following at center pixel c:

$$O_c = \frac{1}{W_c} \sum_c G_{\sigma_s}(\|c - n\|)\, G_{\sigma_r}(I_c - I_n)$$

where W_c is a normalization constant, G_{σ_s} represents the Spatial Gaussian kernel, G_{σ_r} represents the color / range Gaussian kernel, c is the center pixel position, n is the neighboring pixel position, I_c represents the intensity at the center pixel and I_n is the intensity at the neighboring location n.

If the neighborhood pixels are edges, the difference in intensity $(I_c - I_n)$ will be higher. Since the Gaussian is a decreasing function, $G_{\sigma_r}(I_c - I_n)$ will have lower weights for higher values. Hence, the smoothing effect will be lower for such pixels, preserving the edges.

There are 2 significant parameters in bilateral filtering: σ_s and σ_r . σ_s controls the amount of spatial smoothing, and σ_r controls how dissimilar colors within the neighborhood will be averaged. A higher σ_r results in larger regions of constant color. The OpenCV function for bilateral filtering is:

`dst = cv2.bilateralFilter(src, d, sigmaColor, sigmaSpace[, dst[, borderType]])`

with parameters

`src` – Source image – 8-bit or floating-point, 1-channel or 3-channel image.

`dst` – Destination image of the same size and type as `src`.

`d` – Diameter of each pixel neighborhood that is used during filtering. If it is non-positive, it is computed from `sigmaSpace`.

`sigmaColor` – Filter sigma in the color space. A larger value of the parameter means that farther colors within the pixel neighborhood (see `sigmaSpace`) will be mixed together, resulting in larger areas of semi-equal color.

`sigmaSpace` – Filter sigma in the coordinate space. A larger value of the parameter means that farther pixels will influence each other as long as their colors are close enough (see `sigmaColor`). When `d`>0, it specifies the neighborhood size, regardless of `sigmaSpace`. Otherwise, `d` is proportional to `sigmaSpace`.

`borderType` – border mode used to extrapolate pixels outside of the image.

- **Ex. 2.6** Add the missing lines of code, indicated by the comments. Read multiple images affected by different types of noise: salt-and-pepper noise is present in "logo_noise.jpg" and "lina_noise.jpg", while gaussian noise is present in "logo_noise2.jpg". Experiment with different values of kernel size and sigma and establish which type of filtering: median or bilateral is best in each case, depending on the noise type. Show the comparison results and comment on the best choice of parameters for each filter.

```
import cv2
import numpy as np
```

```
import matplotlib.pyplot as plt

img = cv2.imread('test_image.jpg')

# diameter of the pixel neighbourhood used during filtering
dia=15;

# Larger the value the distant colours will be mixed together
# to produce areas of semi equal colors
sigmaColor=80

# Larger the value more the influence of the farther placed pixels
# as long as their colors are close enough
sigmaSpace=80

#Apply bilateralFilter on img with parameters dia, sigmaColor and sigmaSpace

plt.figure()
plt.subplot(121);plt.imshow(img[...,::-1]);plt.title("Noisy Image")
plt.subplot(122);plt.imshow(dst1[...,::-1]);plt.title("Bilateral filter")
plt.show()
```

The following should be noted while deciding which filter to use:

- Median filtering is the best way to smooth images which have salt-pepper type of noise (sudden high / low values in the neighborhood of a pixel).
- Gaussian filtering can be used if there is low Gaussian noise.
- Bilateral Filtering should be used if there is high level of Gaussian noise, and you want the edges intact while blurring other areas.
- In terms of execution speed, Gaussian filtering is the fastest and Bilateral filtering is the slowest.

2.2 Image gradients

An image ***gradient*** is a directional change in intensity (or color) in an image. We need gradients in order to study filters that alter sharp intensity discontinuities. We will analyze a few examples marked with red dots from Figure 3:

- pixels A and B and their neighborhood are of the same color with no intensity change, so the gradient is 0 in those points.
- in pixel C there is an abrupt change in intensity from left to right, so pixel C has a high positive gradient in x-direction. Similarly, pixel D has a high positive gradient in y-direction.
- in pixel E the intensity changes from white to black, so the gradient is therefore negative in the x-direction. Similarly, pixel F has a negative gradient in the y-direction.

- pixels G and I have gradients in the x- and y-directions, respectively, but they are quite subtle. The gradient magnitude is no longer maximum, since the transition is between dark gray to lighter gray (not from black to white as before).
- pixel H has gradients in both directions, which is the case for most pixels in natural images.

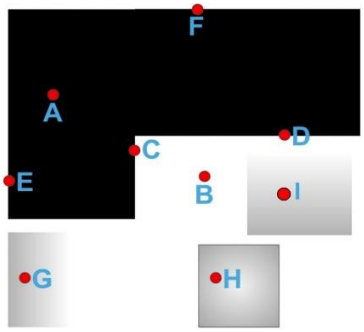

Figure 3. Image gradients

In color images, gradients can be computed for every color channel separately (1 pixel will have 6 values representing the 3 color gradients in x and y directions). In many applications color gradients are not needed, so usually color images are converted to grayscale and intensity gradients are computed instead.

Let us denote the gradient in the x-direction I_x and the gradient in the y-direction, I_y. A pixel gradient is therefore similar to a vector with x and y components. One pixel's gradient magnitude is given by

$$G = \sqrt{I_x^2 + I_y^2}$$

while the gradient's direction is:

$$\theta = arctg \frac{I_y}{I_x}$$

The gradient magnitude and direction are computed at every pixel in the image. I_x and I_y are the images obtained by running the Sobel filters for X and Y Gradients of the image.

2.2.1 Prewitt filter

For the X Gradient value of a pixel, I_x, we need to find the difference in intensity to the right and to the left of the current pixel position. The following Prewitt filter retrieves the X gradient:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

and for the Y gradient, the 2nd Prewitt filter kernel is

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Using Prewitt filters to compute the image gradients I_x and I_y, we actually computed the x and y derivatives of the image.

2.2.2 Sobel filter

Gradient calculations are more robust and noise-free if the image is Gaussian-blurred slightly before applying a gradient filter. The Sobel filters perform Gaussian smoothing implicitly. The Sobel filters for X Gradient and Y Gradient are shown below:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The OpenCV function for Sobel filtering is:

```
dst = cv2.Sobel(src,ddepth,dx,dy[,dst[,ksize[,scale[,delta[, borderType]]]]])
```

with parmeters:

`src` input image

`dst` output image of the same size and the same number of channels as `src`

`ddepth` output image depth,in the case of 8-bit input images it will result in truncated derivatives.

`dx` order of the derivative x.

`dy` order of the derivative y.

`ksize` size of the extended Sobel kernel; it must be 1, 3, 5, or 7.

`scale` optional scale factor for the computed derivative values; by default, no scaling is applied.

`delta` optional delta value that is added to the results prior to storing them in `dst`

`borderType` pixel extrapolation method.

- **Ex. 2.7**

  - Read the image "*Halep.jpg*" as grayscale.

  - Use the Sobel function to compute the X and Y gradients. Set the depth of the output images to `CV_32F` since the gradients can take negative values.

  - Use the function `cv2.normalize` to normalize the gradients, so that all pixel values lie between 0 and 1.

  - Display both gradient images. Can they be used as edge detectors?

2.2.3 Laplacian filter

The Laplacian filter is based on the second derivative and it is also used to recognize edges. Mathematically, the Laplacian operator or filter is given by

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

and the corresponding convolution kernel is

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

The Laplacian filter is very sensitive to noise and therefore it is important to smooth the image before applying it. The associated function in OpenCV is `cv2.Laplacian`.

- **Ex. 2.8**

  - Read the image "*Hummingbird.jpg*" as grayscale.

  - Apply gaussian blurring using `cv2.GaussianBlur` for a kernel size of 3x3 pixels.

  - Apply Laplacian filtering on the previous blurred image. The depth of the output image should be set to `CV_32F` since gradients can take negative values!

  - Use the function `cv2.normalize` to normalize the gradients, so that all pixel values lie between 0 and 1.

  - Display the final normalized image! What are the advantages of a Laplacian edge detector compared to Sobel filters?

  - Try also the Laplacian filtering without the Gaussian smoothing. Compare both filtered images!

2.3 Image sharpening

Image sharpening is meant to enhance the edges and highlight the underlying texture. The idea is not recent, it is actually derived from an old technique denoted ***unsharp masking***. The following steps are necessary to achieve the unsharp masking effect:

Step 1: Blur the input image to smooth out texture. The blurred image contains the low frequency information from the original image. Let I be the original input image and I_b – the blurred image.

Step 2: Subtract the blurred image from the original image, $I - I_b$, to obtain the high frequency information from the original image.

Step 3: Add back the high frequency information to the original image and control the amount of high-frequency texture added – using the α parameter. The final sharpened image is therefore:

$$I_s = I + \alpha(I - I_b)$$

Currently, sharpening is implemented using a simple ***sharpening kernel*** that approximates the above behavior. The input image is convolved with the following kernel:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The above kernel is obtained using $\alpha = 1$ and approximating $I - I_b$ using the Laplacian kernel.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Example:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread("tree1.jpg")
# Sharpen kernel
sharpen = np.array(( [0, -1, 0],
                     [-1, 5, -1],
                     [0, -1, 0]), dtype="int")
sharpImg = cv2.filter2D(img, -1, sharpen)

plt.figure(figsize=[20,10])
plt.subplot(121);plt.imshow(img[...,::-1]);plt.title("Original image")
plt.subplot(122);plt.imshow(sharpImg[...,::-1]);plt.title("Sharpen output")
plt.show()
```

- **Ex. 2.9** What are the texture components highlighted by the sharpening effect? Experiment also with other images (parrots.jpg). Describe the type of texture that is more prominent after the sharpen filtering.

Bibliography / Further reading

https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html

https://docs.opencv.org/4.1.0/d4/d13/tutorial_py_filtering.html