Digital Image Processing - Lab Session 2

Sofia Noemi Crobeddu
student number: 032410554
International Master of Biometrics and Intelligent Vision - Université Paris-Est Cretéil (UPEC)

Introduction

In this work we analyze distances, difference, convolution operation and Dirac function in image processing. Moreover, there is a focus on Gaussian kernel.

1 Image distance

In this section we analyze the different distances in 2D images. In general, in image processing, the method of distance transform for binary images, assigns to each pixel a value that represents its distance to the nearest foreground pixel (usually denoted by 1).
In the first point of the task, the function *bwdist* is mentioned. It is a function of Matlab, and it computes the distance from each background pixel (value 0) to the closest foreground pixel (value 1). Since we are working with Python, we implement a similar approach as follow:

```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from scipy import ndimage

#bwdist is a function in Matlab. We can implement a similar one in python.
#Simulation of a binary image as a simple example
image_simulation = np.zeros((100, 100)) #Black color
image_simulation[30:80, 30:80] = 1  #A square in the center in white

#Calculating distances with the three different method: Euclidean, Manhattan, Chessboard distances
euclidean_dist_ex = ndimage.distance_transform_edt(image_simulation)
manhattan_dist_ex = ndimage.distance_transform_cdt(image_simulation, metric='taxicab')
chessboard_dist_ex = ndimage.distance_transform_cdt(image_simulation, metric='chessboard')

#Plot of the results for each distance for comparison
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
axes[0].imshow(euclidean_dist_ex, cmap='grey')
axes[0].set_title('Euclidean Distance')
axes[1].imshow(manhattan_dist_ex, cmap='grey')
axes[1].set_title('Manhattan Distance')
axes[2].imshow(chessboard_dist_ex, cmap='grey')
axes[2].set_title('Chessboard Distance')

for ax in axes:
    ax.axis('off') #Without any axes
plt.show()
```

In this simulation, first it is created a 100x100 binary black image putting all value to 0. After that, it is added a central white square (values 1). Finally, three distances are calculated using the function *distance_transform_edt*. We use three principal metrics:

- **Euclidean Distance**, also called **L2 norm** and defined as

$$d(x, y) = ||x - y||_2 = \sqrt{\sum_i (x_i - y_i)^2}.$$

  This distance performs a straight-line distance, and in this specific simulation example it produces a smooth gradient around the square. In fact, the central pixels have distance equal to 0, while going outward, we see that the distance increases. In terms of pixels in a 2D grid, it is the direct diagonal distance.
- **Manhattan Distance**, also called **L1 norm** and defined as

$$d(x, y) = ||x - y||_1 = \sum_i |x_i - y_i|.$$

  Since it does not take diagonals into account, it tends to give higher distances, compared to the Euclidean one, for points that are diagonally positioned relative to each other.
- **Chebyshev or Chessboard Distance**, also called **L$_\infty$ norm** and defined as

$$d(x, y) = ||x - y||_\infty = max|x_i - y_i|.$$

  It basically measures how many steps a king in a chess game would need to move from one square to another, and allows diagonal movement. Hence, the result is a pattern where the distance values grow equally in all directions (diagonal, vertical, horizontal).

In Figure 1 we can see the resulting plot of the simulated image for the example. Even if it is not so evident from eye, ass already said the Euclidean distance produces the smoothest distance. Manhattan distance, instead, shows a slightly increasing sharper (look at the center), while Chebyshev more equal increments in all directions.

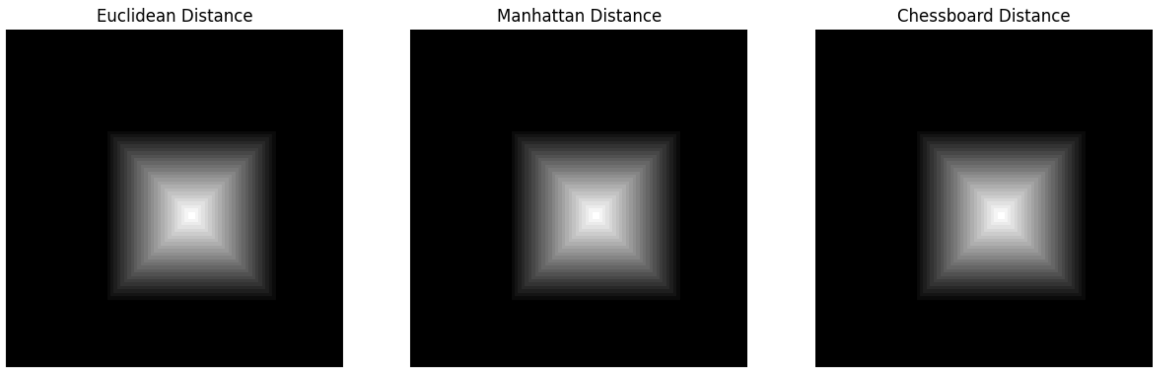

Fig. 1: Distances of the example image.

Continuing with the task, we analyze now the distance transforms for two different images: *leaf* and *leafnoisy*. The images are shown in Figures 2 and 4. The second image has noise, that in Figure 4 are evident as three white noise dots.
Before calculating the distances, we convert the images to gray scale and we threshold them in order to obtain binary images, with pixels set to 0 for background and to 1 for foreground (as before). After this, for both images we calculate the three metrics presented before. Figures 3 and 5 show the result. As we can see, in the noisy image, the three evident noisy points are smoothed through the Euclidean distance. In the case of Manhattan distance these points have a higher impact, making the distance field less smooth. Finally, in Chebyshev distance, the points have a less pronounced effect compared to Manhattan one. Hence, we can say that Chebyshev distance is the most robust to noise among the

three metrics, since it only considers the maximum of the horizontal and vertical distances. However, there is still distortion.
Below there are the blocks of codes for this analysis.

```python
#Loading the image
image_leaf = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 2 - IP\\IP2\\IP2\\leaf.png")

#Plot gray scale
plt.imshow(image_leaf, cmap='gray')
plt.axis("off")  #Don't show the axes
plt.show()
```

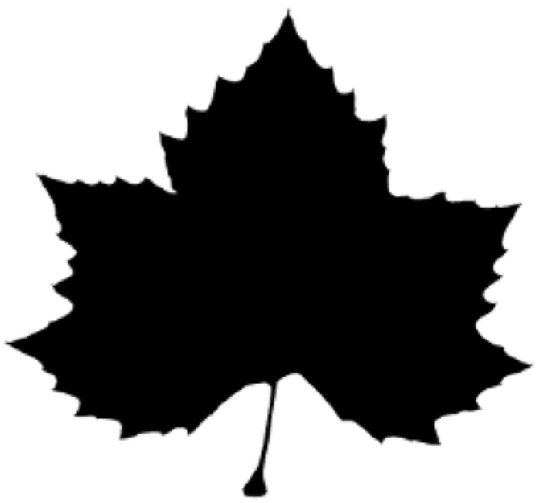

Fig. 2: Leaf image.

```python
#To array
image_leaf = np.array(image_leaf)
#Binary image tranformation
binary_leaf = image_leaf < 128

#Calculating distances
euclidean_leaf = ndimage.distance_transform_edt(binary_leaf)
manhattan_leaf = ndimage.distance_transform_cdt(binary_leaf, metric='taxicab')
chessboard_leaf = ndimage.distance_transform_cdt(binary_leaf, metric='chessboard')

#Plot of the results for each distance for comparison
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
axes[0].imshow(euclidean_leaf, cmap='grey')
axes[0].set_title('Euclidean Distance')
axes[1].imshow(manhattan_leaf, cmap='grey')
axes[1].set_title('Manhattan Distance')
axes[2].imshow(chessboard_leaf, cmap='grey')
axes[2].set_title('Chessboard Distance')
```

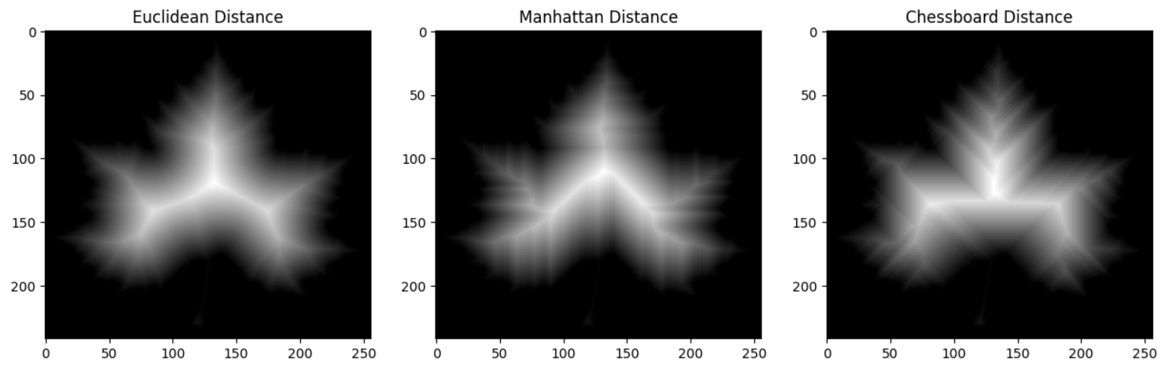

Fig. 3: Distances for Leaf Image.

```python
image_leafnoisy = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 2 - IP\\IP2\\IP2\\leafnoisy.png")

#Plot gray scale
plt.imshow(image_leafnoisy, cmap='gray')
plt.axis("off")  #Don't show the axes
plt.show()
```

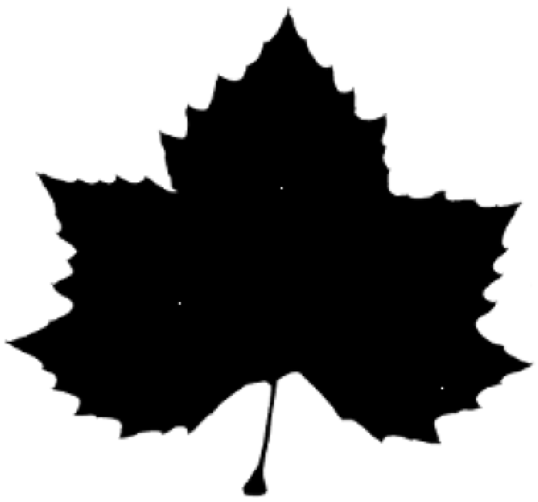

Fig. 4: Leafnoisy image.

```python
#To array
image_leafnoisy = np.array(image_leafnoisy)
#Binary image tranformation
binary_leafnoisy = image_leafnoisy < 128

#Calculating distances
euclidean_leafnoisy = ndimage.distance_transform_edt(binary_leafnoisy)
manhattan_leafnoisy = ndimage.distance_transform_cdt(binary_leafnoisy, metric='taxicab')
chessboard_leafnoisy = ndimage.distance_transform_cdt(binary_leafnoisy, metric='chessboard')

#Plot of the results for each distance for comparison
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
axes[0].imshow(euclidean_leafnoisy, cmap='grey')
axes[0].set_title('Euclidean Distance')
axes[1].imshow(manhattan_leafnoisy, cmap='grey')
axes[1].set_title('Manhattan Distance')
axes[2].imshow(chessboard_leafnoisy, cmap='grey')
axes[2].set_title('Chessboard Distance')
```

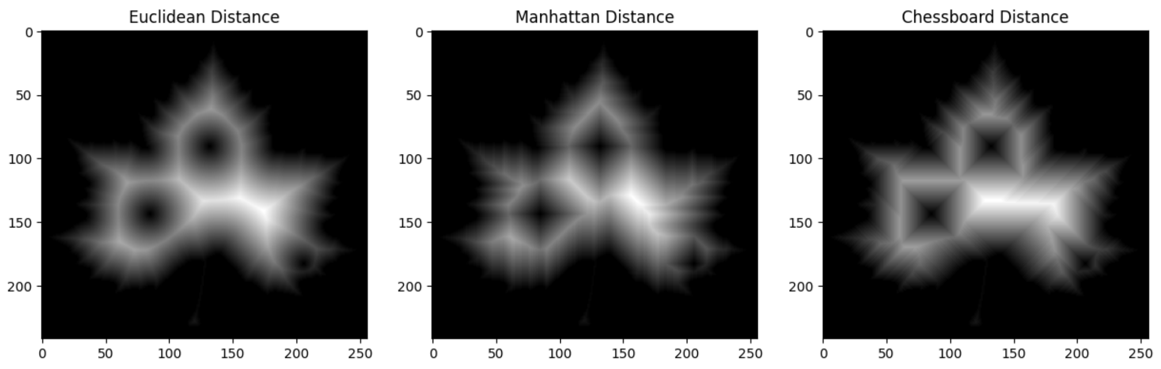

Fig. 5: Distances for Leafnoisy image.

2 Arithmetic operations

In the first part of this section we analyze two other images: *original* and *original2*, that are almost equal except some slight differences difficult to notice with eye. The code to read the files is shown below, and the images are shown in Figures 6 and 7.

```
#Loading the images
image_original = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 2 - IP\\IP2\\IP2\\original.bmp")
image_original2 = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 2 - IP\\IP2\\IP2\\original2.bmp")

#Plots gray scale
plt.imshow(image_original, cmap='gray')
plt.axis("off")  #Don't show the axes
plt.show()

plt.imshow(image_original2, cmap='gray')
plt.axis("off")  #Don't show the axes
plt.show()
```

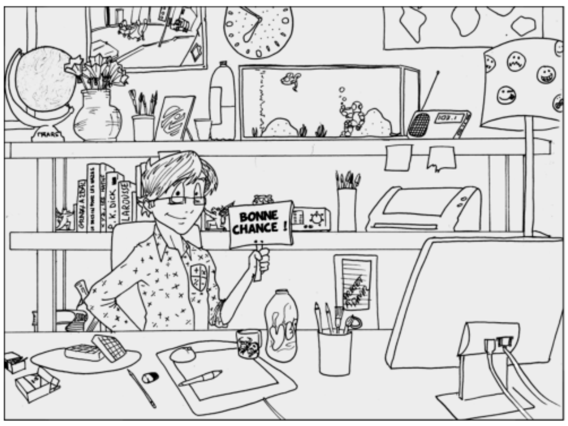

Fig. 6: Original image.

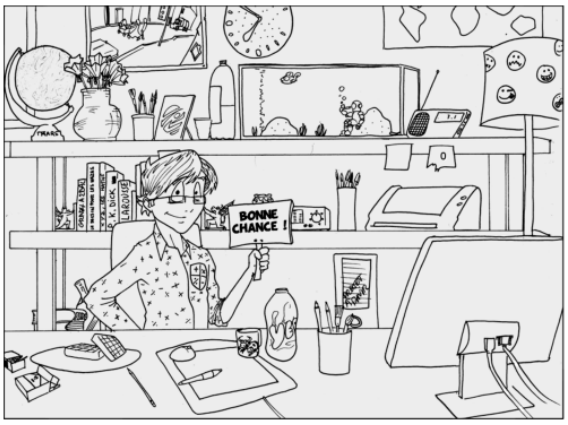

Fig. 7: Original2 image.

In image processing, to compare two images pixel by pixel, we use the operation of **difference**. We perform this using the absolute value of the subtraction between a pixel of the first image and its corresponding one in the second, i.e. every non-zero result represents a mismatch between pixels. In this way, we take into account just the intensity and not if they are increasing or decreasing.
To count the errors, a suitable method could be counting the non-zero differences, i.e. the non-zero pixels. In our case the different pixels are 792.
The code for this analysis is shown below, while Figure 8 represents the difference between the two images considered.

```python
#To array
image_array_original = np.array(image_original)
image_array_original2 = np.array(image_original2)

#Absolute difference between original and original2
difference_or = np.abs(image_array_original - image_array_original2)

#Plot of the difference
plt.imshow(difference_or, cmap='gray')
plt.axis("off") #Don't show the axes
plt.show()

#Counting the "error" pixels
num_errors = np.sum(difference_or > 0)
print(f"Number of different pixels: {num_errors}")
```

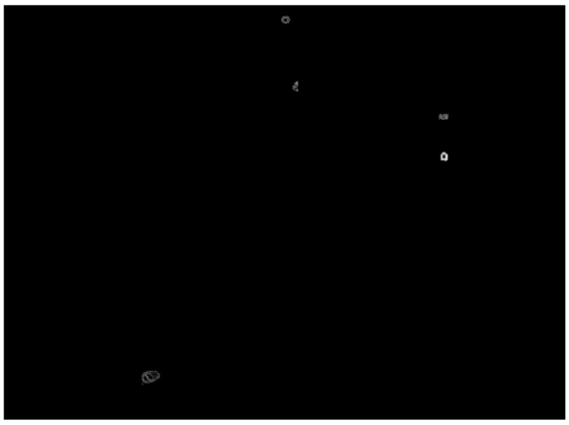

Fig. 8: Difference between Original and Original2.

In the second part of this section we analyze two other images: *chroOp* and *chroL*, and as the previous case they are almost. The code to read the files is shown below, and the images are shown in Figures 9 and 10. They could be microscope images, representing cell structures. They probably were constructed from the original image *chro*, and shown with a different color map.

```python
#Loading the images
image_chroOp = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 2 - IP\\IP2\\IP2\\chroOp.bmp")
image_chroL = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 2 - IP\\IP2\\chroL.bmp")

#Plots gray scale
plt.imshow(image_chroOp, cmap='gray')
plt.axis("off")  #Don't show the axes
plt.show()

plt.imshow(image_chroL, cmap='gray')
plt.axis("off")  #Don't show the axes
plt.show()
```

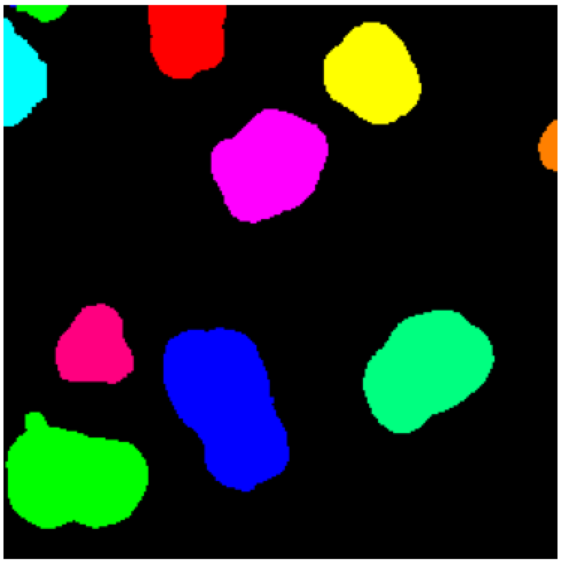

Fig. 9: ChroOp image.

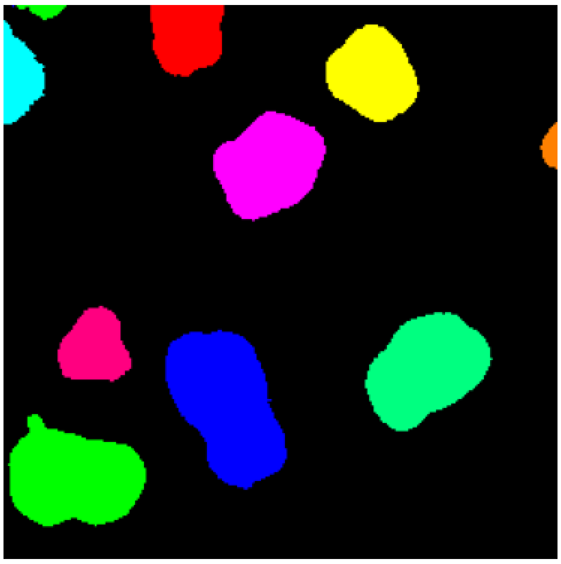

Fig. 10: ChroL image.

Hence, we perform their difference as we did before. The code is shown below and the difference is in Figure 11.

```python
#To array
image_array_chroOp = np.array(image_chroOp)
image_array_chroL = np.array(image_chroL)

#Dfference
difference_chro = np.abs(image_array_chroOp - image_array_chroL)

#Plot of the difference
plt.imshow(difference_chro, cmap='hot')
plt.axis('off')
plt.show()
```

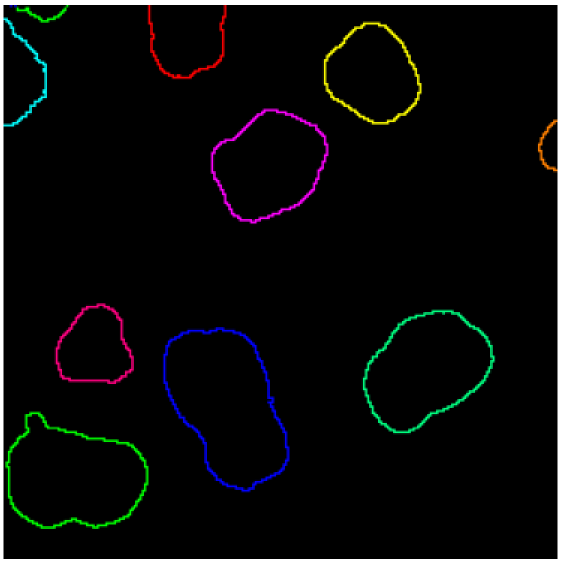

Fig. 11: Difference between chroOp and chroL.

After computing the difference, we load the original image *chro*, and we contert it first into RGB and after into an array. A mask is then created by selecting pixels in the difference image that are above a specific threshold (in this case we put a trivial value of 10) to filter out insignificant changes. This mask is finally applied to the original image, in order to highlight the areas where there is more difference between *chroOp* and *chroL*. The code for this analysis is below, and Figure 12 shows the result.

```python
#Loading the image
image_chro = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 2 - IP\\IP2\\IP2\\chro.tif")
#Converting the image into RGB
image_chro_rgb = image_chro.convert("RGB")
#To array
image_chro_array = np.array(image_chro_rgb)

#Now we mask the original image 'chro' with the image of the difference between chroOp and chroL.
#This is important to see the significative differences. We put a threshold to consider the
    significative ones:
threshold = 10
mask = difference_chro > threshold  #The mask has to be higher than the threshold.

#Here we create an image where the mask highlights the parts of difference.
#Creating an empty image, i.e. it is totally black. The dimension is the same as 'chro' image.
final_image = np.zeros_like(image_chro_array)
#Applying the mask on the original image in array format, and saving it into the empty image
    created before.
final_image[mask] = image_chro_array[mask]

#Plot
plt.imshow(final_image)
plt.axis('off')
plt.show()
```

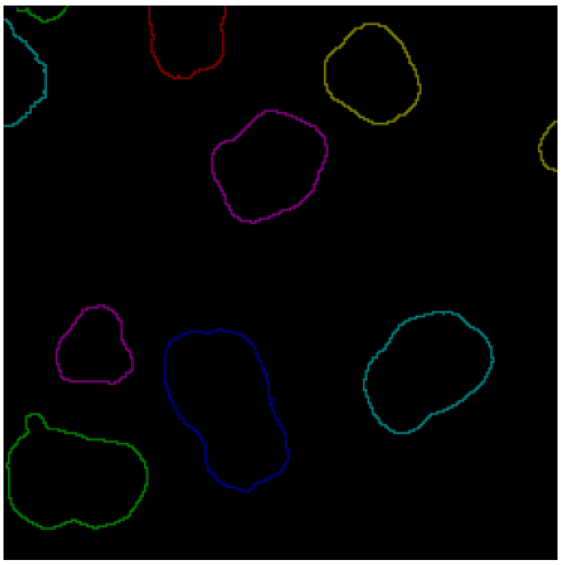

Fig. 12: Highlighted parts in chro.

3 Convolution

In this section we analyze the Gaussian kernel for images.
The preliminary theory for 1D dimension, regards the **convolution**, a mathematical operation that combines two signals or functions f and g, in order to produce a third one. In general it is defined as:

$$f * g = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau.$$

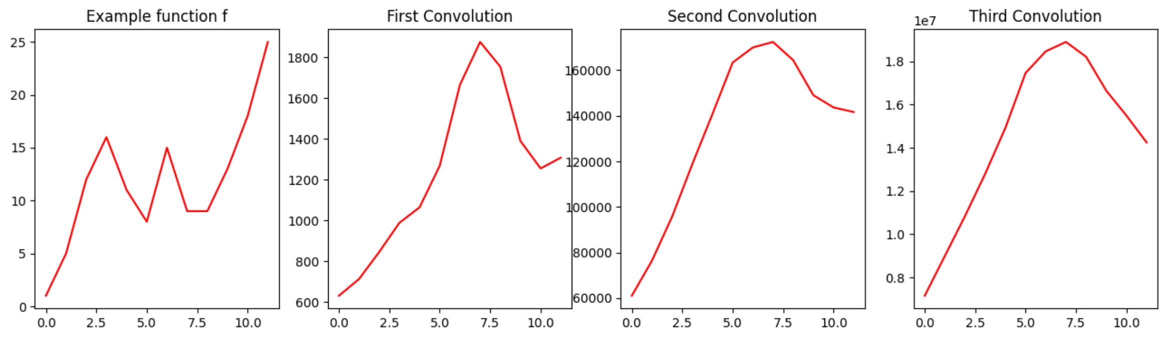

Fig. 13: Convolution of a function with itself.

In this way, we are applying a smoothing effect.

In our exercise we convolve a function f with itself for three times using the function *convolve*, in order to get g that will be our convolved final function such that

$$g(t) = (f * f)(t) = \int_{-\infty}^{+\infty} f(\tau)f(t-\tau)d\tau.$$

The code is shown below, and the resulting function after each convolution is shown in Figure 13.

```python
from scipy.ndimage import convolve

#Defining a signal
f = np.array([1, 5, 12, 16, 11, 8, 15, 9, 9, 13, 18, 25])

#Convolution (three times)
g1 = convolve(f, f, mode='constant')
g2 = convolve(g1, f, mode='constant')
g3 = convolve(g2, f, mode='constant')

#Plot
# Plot the results
plt.figure(figsize=(16, 4))
plt.subplot(1, 4, 1)
plt.plot(f, color='red')
plt.title('Example function f')
plt.subplot(1, 4, 2)
plt.plot(g1, color='red')
plt.title('First Convolution')

plt.subplot(1, 4, 3)
plt.plot(g2, color='red')
plt.title('Second Convolution')

plt.subplot(1, 4, 4)
plt.plot(g3, color='red')
plt.title('Third Convolution')

plt.show()
```

Going now to 2D dimension, in image processing a common way to apply convolution operation, is **Gaussian kernel**, which performs Gaussian smoothing. For the 2D case it is defined as

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

where σ is the standard deviation. This kernel is usually useful for blurring, edge detection, and noise reduction. In the code below, the function for the kernel is built, and right after it is applied to *leaf* image that we saw in section 1. The resulting convolved image is shown in Figure 14, and as we can see the image is more smoother and blurred compared to the original one.

```python
#Function for a Gaussian 2D kernel
def gaussian_kernel(size, sigma):
    ax = np.linspace(-(size // 2), size // 2, size)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-(xx**2 + yy**2) / (2. * sigma**2))
    return kernel / np.sum(kernel)  #Normalizing the resulting kernel

#Preliminary statements
size = 8  #Dimension of the kernel (3x3, 5x5, ...)
sigma = 1.0  #Standard deviation of the Gaussian
kernel = gaussian_kernel(size, sigma)

#Convolution between the image and the kernel
convolved_image = convolve(image_leaf, kernel)

#Plot of both the original image and the convolved
plt.figure(figsize=(10, 6))

plt.subplot(1, 2, 1)
plt.imshow(image_leaf, cmap='gray')
plt.title("Leaf Image - original.")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(convolved_image, cmap='gray')
plt.title("Leaf Image - convolved with Gaussian Kernel.")
plt.axis('off')

plt.show()
```

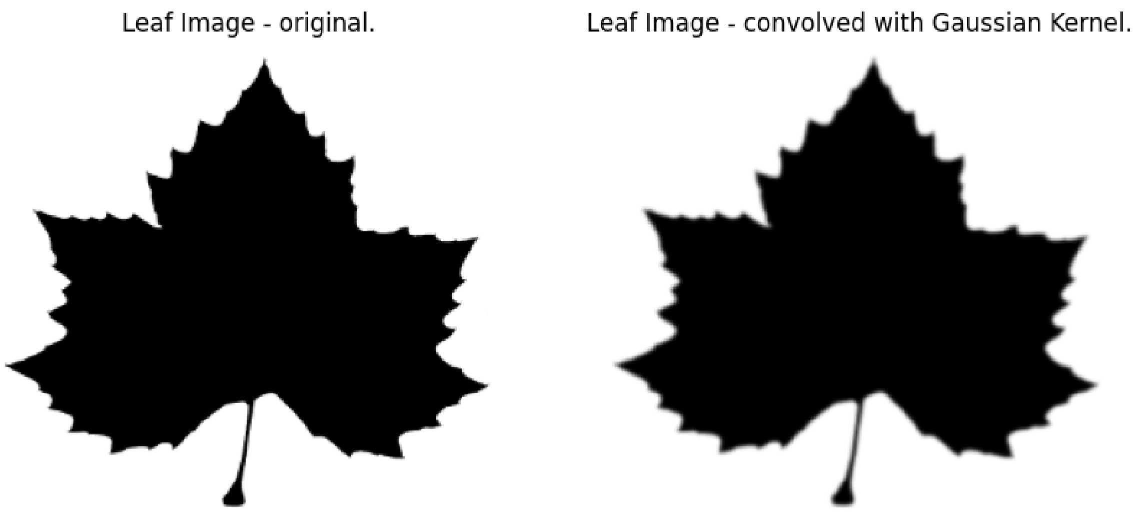

Fig. 14: Before and after Convolution on Leaf image.

In the last step of this section, instead, we convolve *leaf* image with itself, using the function *convolve2d*. Through this operation, every pixel is compared against all others, finding overlaps between the original image and shifted versions of itself. Since our original image has strong contrast, the resulting convolution, shown in Figure 15, amplifies the differences and produces an output that looks noisy.
In general, the convolution of an image with itself increases the image's size of blurred regions, since it mixes the image content with itself.
The code is shown below:

```python
from scipy.signal import convolve2d

#Convolution of the image with itself
convoluted_with_self = convolve2d(image_leaf, image_leaf, mode='same', boundary='fill', fillvalue
    =0)

#Plot
plt.imshow(convoluted_with_self, cmap='gray')
plt.axis('off')
plt.show()
```

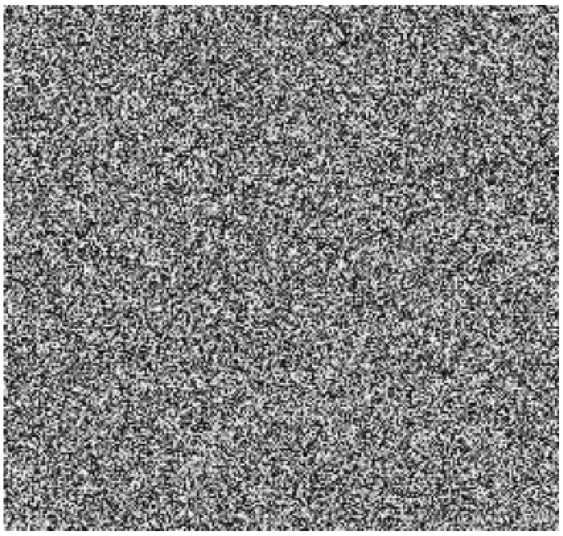
Fig. 15: Convolution of Leaf Image with itself.

4 Dirac Delta Exercises

In this final section we analyze some exercises about the Dirac delta function. In image and signal processing it is important since it represents through a mathematical function, the intensity source. The simplest function we use to model it, is the real Dirac function, equal to 0 everywhere except at the origin.

- 1) **Proposition**: $\int_{-\infty}^{+\infty} \delta(x)f(x)dx = f(0)$ for a continuous function $f(x)$.

  Proof:
  The Dirac delta function $\delta(x)$ is defined as:

  - $\delta(x) = \begin{cases} 0, & \text{for } x \neq 0 \\ +\infty, & \text{for } x = 0 \end{cases}$
  - $\int_{-\infty}^{+\infty} \delta(x)dx = 1$ (***Fundamental Property***).

  Since the delta function is equal to 0 in all the points except for $x = 0$, the product $\delta(x)f(x)$ contributes (i.e. it is not equal to 0) only at the point $x = 0$. Hence, we can rewrite the integral as:

  $$\int_{-\infty}^{+\infty} \delta(x)f(x)dx = f(0)\int_{-\infty}^{+\infty} \delta(x).$$

  Using the Fundamental Property, we have that:

  $$\int_{-\infty}^{+\infty} \delta(x)f(x)dx = f(0)\int_{-\infty}^{+\infty} \delta(x) = f(0) \cdot 1 = f(0).$$

  This means that the Dirac function has the **Sampling Property**, i.e. when it is integrated combined with a continuous function $f(x)$, the result is equal to the value of $f(x)$ in the point where $\delta(x)$ is not null.

  Now that we have proved the result, we need to prove the continuity. This proof in fact, to be valid, considers a function $f(x)$ which has to be continue in the neighborhood of $x = 0$. In this case, we can rewrite the function as:

  $$f(x) \approx f(0) + \epsilon(x)$$

where $\epsilon(x) \to 0$ as $x \to 0$.
Hence, we can rewrite the integral as:

$$\int_{-\infty}^{+\infty} \delta(x)f(x)dx = \int_{-\infty}^{+\infty} \delta(x)(f(0)+\epsilon(x))dx = \int_{-\infty}^{+\infty} \delta(x)f(0)dx + \int_{-\infty}^{+\infty} \delta(x)\epsilon(x)dx$$

where the term $\int_{-\infty}^{+\infty} \delta(x)\epsilon(x) \to 0$, since $\epsilon(x) \to 0$ as $x \to 0$.

Hence, we have proved that $\int_{-\infty}^{+\infty} \delta(x)f(x)dx = \int_{-\infty}^{+\infty} \delta(x)f(0)dx$ only when $f(x)$ is continue. ■

- 2) **Proposition**: $\int_{-\infty}^{+\infty} \delta(g(x))f(x)dx = \sum_i \frac{f(x_i)}{|g'(x_i)|}$ for $g(x)$ differentiable function and $f(x)$ a continuous function.

  Proof:
  From the proof of exercise 1), we know that $\int_{-\infty}^{+\infty} \delta(g(x))dx$ is not null only when $g(x) = 0$. Hence, $g(x)$ has some zero-points within x_1, x_2, x_3, ..., x_n, i.e. $g(x_i) = 0$. In these points the delta function has non-zero values.

  Extending the basic insights of **Sampling Property**, of the Dirac function proved in 1), we have that

$$\int_{-\infty}^{+\infty} \delta(x - x_i)f(x)dx = f(x_i).$$

  Hence, in the integral $\int_{-\infty}^{+\infty} \delta(g(x))f(x)dx$, the delta function samples the function $f(x)$ only in points where the argument of the delta function is 0. Then, considering $\delta(g(x))$, the Dirac function samples the values of the function $f(g(x))$ only where $g(x) = 0$. Now, operating a change of variable $u = g(x)$, where g is continuously differentiable, and $g'(x)$ is its derivative, we have that $du = g'(x)dx$. Hence we have:

$$\int_{-\infty}^{+\infty} \delta(g(x))f(x)dx = \int_{g(\mathbb{R})} \delta(u)f(x(u))|\frac{dx}{du}|du = \int_{-\infty}^{+\infty} \delta(g(x))f(g(x))|g'(x)|dx$$

  where $x(u)$ is the inverse function of $g(x)$, and $\frac{dx}{du} = \frac{1}{g'(x)}$ is such that the term $|g'(x)|$ is the one which takes into account the variation of x in respect to u.

  As we said before, Dirac function applied to the function $f(x)$, outputs the value of the function f in the point x_i, since it is not null just in that point. When we take into account a more complex function like $g(x)$, we have to consider more points, as expressed at the start of this proof, i.e. we consider all the i where $g(x_i) = 0$. In the neighborhood of every point x_i, we can approximate the function g using the Taylor expansion such that:

$$g(x) \approx g(x_i) + g'(x_i)(x - x_i)$$

  for x close to x_i.

  Considering now the change of variable made before using $u = g(x)$, we have that

$$\delta(g(x))dx = \delta(u)\frac{du}{|g'(x)|},$$

  but since $\delta(g(x))$ is not null only in x_i points where $g(x_i) = 0$, we can rewrite it as a sum of centered deltas in those points, with the normalization factor $\frac{1}{|g'(x_i)|}$, considering the variation of x in respect to u:

$$\delta(g(x)) = \sum_i \frac{\delta(x - x_i)}{|g'(x_i)|}$$

.
Hence, expanding this proof just performed, we have that our initial algorithm is:

$$\int_{-\infty}^{+\infty} \delta(g(x))f(x)dx = \int_{-\infty}^{+\infty} \sum_i \frac{\delta(x-x_i)}{|g'(x_i)|} f(x)dx = \sum_i \frac{f(x_i)}{|g'(x_i)|}.$$

■

- 3) **Proposition**: $\int_{-\infty}^{+\infty} \delta(ax-bx^2)f(x)dx = \frac{f(0)+f(\frac{a}{b})}{|a|}$ for a continuous function f.

  Proof:
  We can start taking into account the proof of proposition in 2). In this case 3) we have $g(x) = ax - bx^2$. The points where $g(x) = ax - bx^2 = 0$, are:

  $$ax - bx^2 = 0 \iff x(a - bx) = 0.$$

  This system has two solutions: $x_1 = 0$ and $x_2 = \frac{a}{b}$.

  Now, recalling the property of delta function proved in 2), we have that:

  $$\delta(g(x)) = \sum_i \frac{\delta(x-x_i)}{|g'(x_i)|}$$

  where x_i are the points where $g(x) = 0$, and $g'(x_i)$ is the derivative of $g(x)$ calculated in those points. Hence, in this case we have $g'(x) = a - 2bx$. We evaluate $g'(x)$ in the points $x_1 = 0$ and $x_2 = \frac{a}{b}$:

  – $g'(0) = a$ for $x_1 = 0$;
  – $g'(\frac{a}{b}) = a - 2a = -a$ for $x_2 = \frac{a}{b}$.

  Hence, using the property of delta function, we have that:

  $$\delta(ax - bx^2) = \frac{\delta(x-0)}{|a|} + \frac{\delta(x-\frac{a}{b})}{|-a|}.$$

  Then, by 2), we have that:

  $$\begin{aligned}\int_{-\infty}^{+\infty} \delta(ax-bx^2)f(x)\,dx &= \int_{-\infty}^{+\infty} \left(\frac{\delta(x)}{|a|} + \frac{\delta\left(x-\frac{a}{b}\right)}{|-a|}\right) f(x)\,dx \\ &= \int_{-\infty}^{+\infty} \frac{\delta(x)}{|a|} f(x)\,dx + \int_{-\infty}^{+\infty} \frac{\delta\left(x-\frac{a}{b}\right)}{|-a|} f(x)\,dx \\ &= \frac{f(0)}{|a|} + \frac{f\left(\frac{a}{b}\right)}{|a|}.\end{aligned}$$

  Since $|-a| = |a|$, the final result is:

  $$\int_{-\infty}^{+\infty} \delta(ax-bx^2)f(x)dx = \frac{f(0)}{|a|} + \frac{f(\frac{a}{b})}{|a|} = \frac{f(0)+f(\frac{a}{b})}{|a|}.$$

  ■

- 4) **Proposition**: $\int_{-\infty}^{+\infty} \delta(x^3 - x)e^{-x}dx = 1 + \frac{1}{2e} + \frac{e}{2}$.

Proof:
To start this proof, we need to determine the points where the argument of the delta function is 0. In this case we have that $g(x) = x^3 - x$. Hence,

$$x^3 - x = 0 \iff x(x^2-1) = 0 \iff x(x-1)(x+1) = 0.$$

So, the roots of the equation are $x_1 = 0$, $x_2 = 1$ and $x_3 = -1$. Regarding the derivative, we have that $g'(x) = 3x^2 - 1$. Evaluating it into the roots points, we obtain:

- $g'(0) = -1$ for $x_1 = 0$;
- $g'(1) = 2$ for $x_2 = 1$;
- $g'(-1) = 2$ for $x_3 = -1$.

Hence, by the property of the delta function, we have that:

$$\delta(x^3 - x) = \frac{\delta(x-0)}{|-1|} + \frac{\delta(x-1)}{|2|} + \frac{\delta(x+1)}{|2|}.$$

Then, by 2) we have that:

$$\begin{aligned}\int_{-\infty}^{+\infty} \delta(x^3 - x)e^{-x}\,dx &= \int_{-\infty}^{+\infty} \left(\frac{\delta(x-0)}{|-1|} + \frac{\delta(x-1)}{|2|} + \frac{\delta(x+1)}{|2|}\right) e^{-x}\,dx \\ &= \int_{-\infty}^{+\infty} \frac{\delta(x-0)}{|-1|}e^{-x}\,dx + \int_{-\infty}^{+\infty} \frac{\delta(x-1)}{|2|}e^{-x}\,dx + \int_{-\infty}^{+\infty} \frac{\delta(x+1)}{|2|}e^{-x}\,dx \\ &= \frac{f(0)}{|-1|} + \frac{f(1)}{|2|} + \frac{f(-1)}{|2|}.\end{aligned}$$

Since in our case $f(x) = e^{-x}$, we have that:

- $f(0) = e^0 = 1$;
- $f(1) = e^{-1}$;
- $f(-1) = e^1 = e$.

Hence, the final result is:

$$\int_{-\infty}^{+\infty} \delta(x^3 - x)e^{-x}dx = \frac{1}{|-1|} + \frac{e^{-1}}{|2|} + \frac{e}{|2|} = 1 + \frac{1}{2e} + \frac{e}{2}.$$

■

References

[1] Majidzadeh, F.: Digital Image Processing and Pattern Recognition, (2023)

[2] Dirac Delta Function. https://en.wikipedia.org/wiki/Dirac_delta_function

[3] NIKHEF: Exercises Lecture 5. Accessed: 2024-11-19 (2020). https://www.nikhef.nl/\texttildelowi93/MSP/2020/Exercises_Lecture5.pdf

[4] Convolution. https://en.wikipedia.org/wiki/Convolution

[5] Chung, M.: Diffusion, Gaussian, Kernel (n.d.). https://pages.stat.wisc.edu/\texttildelowmchung/teaching/MIA/reading/diffusion.gaussian.kernel.pdf.pdf