Computer Vision – Algorithms and Deep Learning

Eyal Gur

September 5, 2024

Part I
Classical Computer Vision

An *image* is a tensor of numbers, where each channel is a 2D array that represents the intensity of the pixels within that channel. A common pixel format is the 8-bit integer (byte image), giving a range of values from 0 (which is typically black) to 255 (typically white), and this range can be scaled to the range $[0, 1]$. Consequently, a grayscale image can be represented as a single 2D channel, while a color image can be represented as a 3D tensor, with each channel corresponding to a grayscale 2D image. The three channels in a color image – red, green, and blue (RGB) – can be combined and colorized to produce the final visual representation.

Since an image can be represented as a tensor, it can be viewed as a function F that maps a domain $D \subset \mathbb{R}^2$ (representing the pixel locations) to intensity values:

$$F(D) = (F_1(D), F_1(D), \ldots, F_C(D)),$$

where C is the number of channels, and F_i is the intensity map of channel i.

1 Image Filtering

There are two primary types of transformations that can be applied to an image: *filtering*, which involves transforming the pixel values (i.e., altering the range of F), and *warping*, which involves transforming the pixel locations (i.e., altering the domain D of F). Filtering is further categorized into *point processing* operations and *neighborhood processing* operations.

1.1 Point Processing

Point processing operations are transformations applied to each pixel of an image independently, without considering its neighboring pixels. For instance, given an image $\mathbf{x} \in \mathbb{R}^{h \times w \times m}$ with pixel values in the range $[0, 1]$, the element-wise operation $\mathbf{x} - 0.5$ uniformly darkens the image (with values outside the $[0, 1]$ range being clipped to the nearest boundary, either

0 or 1). Similarly, the operation $\mathbf{x} + 0.5$ uniformly brightens the image. The operation $1 - \mathbf{x}$ inverts the image colors.

The transformation $2\mathbf{x}$ increases the contrast of the image contrast by brightening all pixels, with the originally brighter pixels becoming even brighter relative to the darker ones. Conversely, the transformation $\mathbf{x}/2$ decreases contrast by darkening all pixels, making the originally brighter pixels less distinct from the darker ones.

Non-linear point operations can also be applied, such as $\mathbf{x}^2$, which darkens the image while leaving the white and black pixels unchanged. Another example is $\sqrt{\mathbf{x}}$, which brightens the image, again with white and black pixels remaining unaffected.

Figure 1 illustrates all of the discussed point operations and are implemented in the Google Colab notebook. Of course, there are many other types of point processing operations, but they all follow similar principles.

Figure 1: Point processing operations: applying linear or non-linear point operations for image filtering.

1.2 Linear Shift-Invariant Image Filtering

A linear shift-invariant filtering involves applying operations that take into account the neighborhood of each pixel, where each pixel is replaced by a linear combination of its neighboring pixels and itself.

1.2.1 Convolution In Image Processing

The Convolution Operation. An operation that combines two functions $f\colon \mathbb{R}^d \to \mathbb{R}$ and $g\colon \mathbb{R}^d \to \mathbb{R}$ to produce a third function $f * g\colon \mathbb{R}^d \to \mathbb{R}$. This operation is defined as the integral (or summation in the discrete case) of the product of the two functions after one function is reflected about the y-axis and then shifted (effectively flipping its domain). Mathematically, given two functions $f, g\colon \mathbb{R}^d \to \mathbb{C}$, their convolution is expressed as:

$$(f * g)(\mathbf{x}) = \int_{\mathbb{R}^d} f(\mathbf{y})\, g(\mathbf{x} - \mathbf{y})\, d\mathbf{y} = \int_{\mathbb{R}^d} f(\mathbf{x} - \mathbf{y})\, g(\mathbf{y})\, d\mathbf{y}.$$

A particularly interesting case is the two-dimensional discrete scenario, where the two functions being convolved can be represented as matrices. Given two matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m \times n}$, their convolution, according to the previous definition, is expressed as:

$$\mathbf{X} * \mathbf{Y} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \mathbf{X}_{i,j} \mathbf{Y}_{m-1-i,n-1-j}.$$

Notice that the convolution operation is commutative. Additionally, this operation flips the order of the matrix $\mathbf{Y}$ and reverses its elements. The reason for this flip, is that in many physical systems the output at any given time depends on past inputs, not future ones. If the order is not reversed, the operation is known as *cross-correlation*.

In the special case where one of the matrices is symmetric around its vertical and horizontal axes, a situation known as *symmetric convolution*, the convolution simplifies to their Frobenius inner product.

Using the definition of the convolution operation, we can introduce the *kernel convolution* operation, often simply referred to as convolution or filtering in image processing. This operation can be seen as a generalization of the mathematical convolution when the two matrices involved do not have the same shape. The operation involves taking a kernel matrix and convolving it with portions of the image that match the shape of the kernel, then shifting the kernel to convolve with other parts of the image.

Let $\mathbf{I} \in \mathbb{R}^{H \times W}$ be a 2D image (or a single channel of a multi-channel image), and let $\mathbf{K} \in \mathbb{R}^{k_H \times k_W}$ be a kernel matrix. Denote the *strides* along the height and width by $s_H, s_W \in \mathbb{N}$. The kernel convolution of $\mathbf{I}$ with the kernel $\mathbf{K}$ and stride (s_H, s_W) is defined as:

$$(\mathbf{K} * \mathbf{I})_{p,q} = \sum_{i=0}^{k_H-1} \sum_{j=0}^{k_W-1} \mathbf{K}_{ij} \mathbf{I}_{ps_H-1-i,qs_W-1-j} = \mathbf{K} * \mathbf{I}[ps_H - k_K : ps_H, qs_W - k_W : qs_W],$$

where we have applied the previously introduced definition of (mathematical) convolution. This operation is repeated to each channel. Commonly, the kernel a is square matrix with an odd dimension $2k + 1$, with the convention that the $(0, 0)$ point is its *center* element. In this case, the kernel convolution can be written conveniently as

$$(\mathbf{K} * \mathbf{I})_{p,q} = \sum_{i,j=-k}^{k} \mathbf{K}_{ij} \mathbf{I}_{ps_H-i,qs_W-j} = \mathbf{K} * \mathbf{I}[ps_H - k : ps_H + k, qs_W - k : qs_W + k].$$

The resulting convolved image $\mathbf{K} * \mathbf{I}$ (or single channel), referred to as the *filtered image*, has dimensions given by:

$$\left(\left\lfloor \frac{H - k_H}{s_H} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{W - k_W}{s_W} \right\rfloor + 1 \right).$$

Edge Handling. Kernel convolution often requires accessing pixels beyond the image boundaries. There are three primary methods to address this:

1. *Skipping*: Any pixel in the output image that requires values from outside the boundaries is skipped. This involves adjusting the kernel's position so that it only operates within the image area. As a result, the output image may be smaller, with cropped edges. This method is commonly used in machine learning.

2. *Kernel Cropping*: Any part of the kernel extending beyond the image boundaries is ignored during convolution, and the remaining values are normalized to compensate.

3. *Padding*: A widely used method where the image borders are extended to provide the necessary values for convolution. There are several types of padding:

   - Wrapping: the image is tiled, and values are taken from the opposite edge.
   - Mirroring: the image is mirrored at the edges, so accessing pixels beyond the edge reads from within the image.
   - Constant Padding: the added pixels have a constant value, usually zero.

   When padding by p_H pixels along the height and p_W pixels along the width, the output size of the convolved image is given by:

$$\left(\left\lfloor \frac{H - k_H + 2p_H}{s_H} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{W - k_W + 2p_W}{s_W} \right\rfloor + 1 \right).$$

Box Filtering. A simple linear filter where each pixel in the output image is computed as the average of its neighboring pixels in the input image. An $n \times n$ kernel matrix for this filter is $\textsc{ones}(n, n)/n^2$, and the strides are 1.

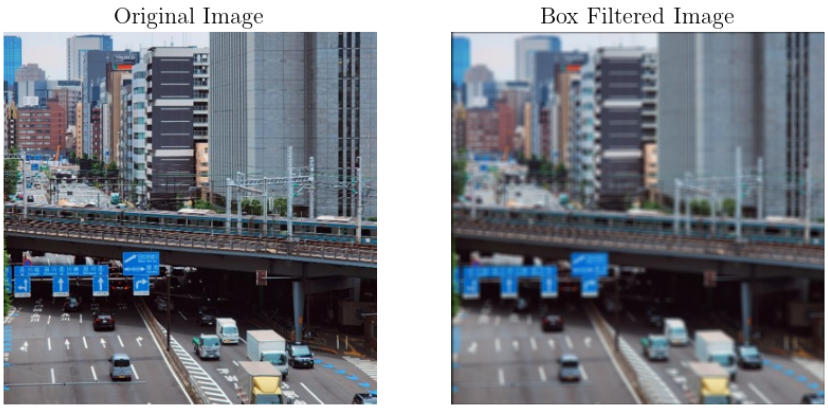
Figure 2 displays the result of applying a box filter, as implemented in the Google Colab notebook. This implementation automatically calculates the necessary constant padding to ensure that the output image retains the same dimensions as the input. The convolution is performed using `numpy`, although built-in functions are also available for this purpose.

1.2.2 Separable Filters

If the kernel matrix has dimensions $k_H \times k_W$ and the image has dimensions $H \times W \times C$ (where C is the number of channels), the computational cost of the kernel convolution is $k_H k_W HWC$. This is because the kernel requires $k_H k_W$ multiplications for each pixel.

This computational cost can be reduced if the kernel matrix is *separable*, meaning it can be expressed as the outer product of two vectors. For example, a box filter kernel of size $n \times n$ can be written as $\mathbf{1}_n \mathbf{1}_n^T$.

Figure 2: Box filtering by applying a 5×5 kernel matrix.



Suppose the kernel matrix $\mathbf{K} \in \mathbb{R}^{k_H \times k_W}$ can be expressed as $\mathbf{K} = \mathbf{k}\mathbf{l}^T$ where $\mathbf{k} \in \mathbb{R}^{k_H}$ and $\mathbf{l} \in \mathbb{R}^{k_W}$. In that case, it can be shown that $\mathbf{K} * \mathbf{I} = \mathbf{k} * \left(\mathbf{l}^T * \mathbf{I}\right)$, where $\mathbf{I}$ is a 2D image (or a single channel). Here, the 1D convolutions require a total of $k_H + k_W$ mmultiplications per pixel, resulting in a reduced overall cost of $(k_H + k_W)\, HWC$.

Gaussian Filter. The box filter computes the average pixel value at each pixel, treating all neighboring pixels equally. In contrast, the Gaussian filter averages pixel values with their neighbors, but it gives more weight to pixels closer to the center of the kernel, resulting in a more natural and realistic blur.

The x coordinate of the 1D Gaussian kernel is defined as:

$$\mathbf{G}^1(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right),$$

where 0 is the center element of the kernel, and $\sigma > 0$ is the standard deviation that controls the level of smoothing. The (x, y) coordinates of the 2D Gaussian kernel are defined as:

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

where $(0, 0)$ is the center element of the kernel. The equality above also demonstrates that this kernel is separable, by treating the first multiplicand as a column vector and the second as a row vector. In other words, the 2D Gaussian kernel is outer product of two 1D Gaussian kernels, that is $\mathbf{G} = \mathbf{G}^1 \left(\mathbf{G}^1\right)^T$. For instance, an approximation of a 3×3 Gaussian kernel matrix is:

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{16}\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

Theoretically, the Gaussian kernel is infinite, but in practice it is truncated to a certain radius from the center. Typically, the kernel values are considered negligible beyond about three standard deviations from the mean, allowing the kernel to be truncated at that point.

Figure 3 shows the result of applying a Gaussian filter, as implemented in the Google Colab notebook. This implementation accepts the kernel size and standard deviation as inputs and calculates the 1D vector for the convolution by leveraging the separability property for improved performance. The convolution is performed using `numpy`, although built-in functions are also available for this task.

Figure 3: Gaussian filtering by applying a 5×5 kernel matrix, with two different standard deviations determining the level of blurring.

Sharpening Filter. One example of such a filter in the 3×3 case is constructed as follows:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix},$$

which is also a separable filter (though there are other variants of the sharpening filter). With a stride of 1, this filter first amplifies the pixel by doubling its value, and then subtracts the average of the pixel values of its neighbors (subtracting a box filter). In regions with constant neighborhood values ("flat" areas), where subtracting the average from the doubled value results in no change, the filter has no effect. However, at edges, this process increases the pixel values in the output image, resulting in sharpening.

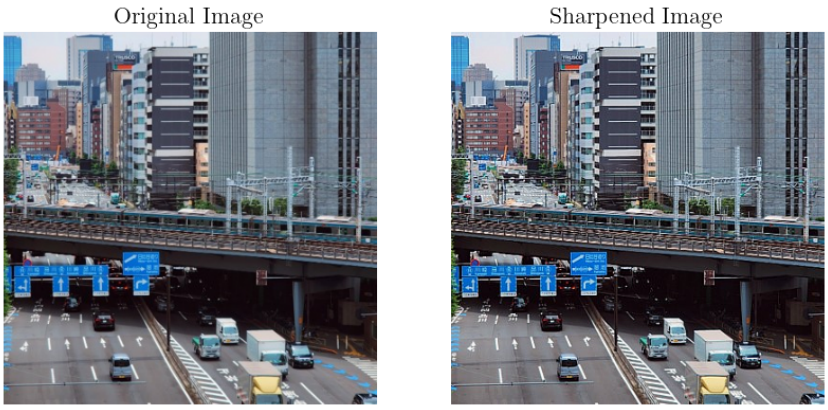
Figure 4 displays the result of applying a 3×3 sharpening filter, as implemented in the Google Colab notebook. This implementation utilizes the `cv2` package to perform the convolution using the `cv2.filter2D` function (we can also apply two 1D convolutions as discussed above).

1.2.3 Discrete Differentiation Filters

Discrete differentiation allows for detecting changes in pixel intensity, which correspond to edges and other important features in an image. For instance, consider the vector image $\mathbf{v} = (0, 0, 0, 1, 1, 1, 1, 0, 0, 0)$ and the kernel vector $\mathbf{k} = [1, -1]$. Using a stride of 1 and applying zero padding of size 1 (so that all elements of $\mathbf{v}$ are multiplied by the center of the kernel, which we define it to be the element 1), the convolution yields:

$$\mathbf{k} * \mathbf{v} = (0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0)\,.$$

Figure 4: Sharpening filter by applying a 3×3 kernel matrix.



This result demonstrates that the convolved output reflects the rate of change in pixel values within $\mathbf{v}$ when imagining moving from left to right through the input data. Similarly, taking the kernel $[-1, 1]$ reflects changes when moving from right to left (remember that during convolution, the multiplication is computed in a reversed order).

We can show that the discrete differentiation kernel $\mathbf{k} = [1, -1]$ is derived from the definition of the derivative. Recall the definition:

$$\frac{\partial f}{\partial x}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

where $f \colon \mathbb{R} \to \mathbb{R}$ is some continuous (and differentiable) function. In the discrete case, for a discrete signal $\mathbf{v}$, the discrete analogue is obtained by removing the limit:

$$\frac{\partial \mathbf{v}}{\partial x}(x) = \frac{\mathbf{v}(x+h) - \mathbf{v}(x)}{h}, \tag{2}$$

where $h \in \mathbb{R}$ is small enough, and $\mathbf{v}(x+h)$ represents the value of $\mathbf{v}$ at the coordinate $x+h$. An approximation of $\partial \mathbf{v}/\partial x$ can be obtained by choosing specific values of h. For example, substituting $h = 1$ in (2) we get:

$$\frac{\partial \mathbf{v}}{\partial x}(x) \approx \mathbf{v}(x+1) - \mathbf{v}(x).$$

Iterating through the coordinates of $\mathbf{v}$ with a stride of 1 and zero padding, we indeed obtain:

$$\frac{\partial \mathbf{v}}{\partial x} = (0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0) = \mathbf{k} * \mathbf{v}. \tag{3}$$

Notice that $\mathbf{k} = [1, -1]$ represents the coefficient of $\mathbf{v}(x)$ and $\mathbf{v}(x+1)$ in the approximation (3), with -1 corresponding to $\mathbf{v}(x)$ and 1 corresponding to $\mathbf{v}(x+1)$, but in a reversed order. Since the coefficient of $\mathbf{v}(x-1)$ is 0, then the kernel matrix $[1, -1, 0]$ is equivalent to $\mathbf{k} = [1, -1]$, up to padding. Similarly, the kernel $[-1, 1]$ (right to left derivative) is obtained by looking at the equivalent definition of (2):

$$\frac{\partial \mathbf{v}}{\partial x}(x) = \frac{\mathbf{v}(x-h) - \mathbf{v}(x)}{h},$$

and plugging $h = -1$.

We can derive other discrete differentiation kernels using the same methodology. For instance, consider the alternative definition (2):

$$\frac{\partial \mathbf{v}}{\partial x}(x) = \frac{\mathbf{v}(x + 0.5h) - \mathbf{v}(x - 0.5h)}{h}, \tag{4}$$

which leads to the kernel $[0.5, 0, -0.5]$ by plugging $h = 2$. This suggests that the kernels $[1, 0, -1]$ and $[-1, 0, 1]$ are discrete approximations of twice the derivative.

The Prewitt Filter. A type of discrete differentiation and separable filter is defined for vertical and horizontal directions. Utilizing the discrete approximation matrix $[1, 0, -1]$, which represents twice the derivative as derived above, we define the 2D kernels of the Prewitt filter as follows:

$$\mathbf{K}_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \mathbf{K}_x^T.$$

This means that $\mathbf{K}_x$ first approximates the derivative along the rows, and then increases the pixel intensity of the derivatives along the columns. It is evident that if, for example, the vertical kernel matrix $\mathbf{K}_x$ returns a positive pixel value after convolution, it indicates a transition from dark to light when moving from right to left. Conversely, if it returns a negative value, it signifies a transition from light to dark.

The Sobel Filter. Like the Prewitt filter, this is a discrete differentiation and separable filter that measures intensity transitions when moving from left to right or from top to bottom. However, it detects more complex transitions, might leading to a more cluttered convolved image. It is defined for vertical and horizontal directions as follows:

$$\mathbf{K}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \mathbf{K}_x^T.$$

Other Discrete Filters. The Scharr Filter is defined as

$$\mathbf{K}_x = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \mathbf{K}_x^T,$$

and it also measured light transitions from left to right or from top to bottom. The Roberts Filter is defined as

$$\mathbf{K}_x = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix},$$

and it measures light transition along diagonals.

1.3 Edge Detection

Image edges are sharp discontinuities or sudden changes in pixel intensity (rate of change). These discontinuities can be detected using discrete approximations of the first-order or second-order derivatives of the pixels, representing the rate of change in their intensity. These derivatives can are visualized, for example using a *gradient map*, thereby enabling edge detection.

1.3.1 Calculating First-Order Gradient Maps

To compute such maps, follow these steps:

1. Select a discrete differentiation filter for the vertical and horizontal directions, $\mathbf{K}_x$ and $\mathbf{K}_y$ (although other directions can also be used).

2. Convolve the image $\mathbf{I} \in \mathbb{R}^{H \times W}$ with these kernels to produce two filtered images:

$$\frac{\partial \mathbf{I}}{\partial x} \approx \mathbf{K}_x * \mathbf{I} \in \mathbb{R}^{H \times W} \quad \text{and} \quad \frac{\partial \mathbf{I}}{\partial y} \approx \mathbf{K}_y * \mathbf{I} \in \mathbb{R}^{H \times W}.$$

3. Calculate the gradient magnitude (the gradient map)

$$\|\nabla \mathbf{I}\| \equiv \sqrt{\left(\frac{\partial \mathbf{I}}{\partial x}\right)^2 + \left(\frac{\partial \mathbf{I}}{\partial y}\right)^2} \in \mathbb{R}^{H \times W},$$

   where the operations are performed element-wise.

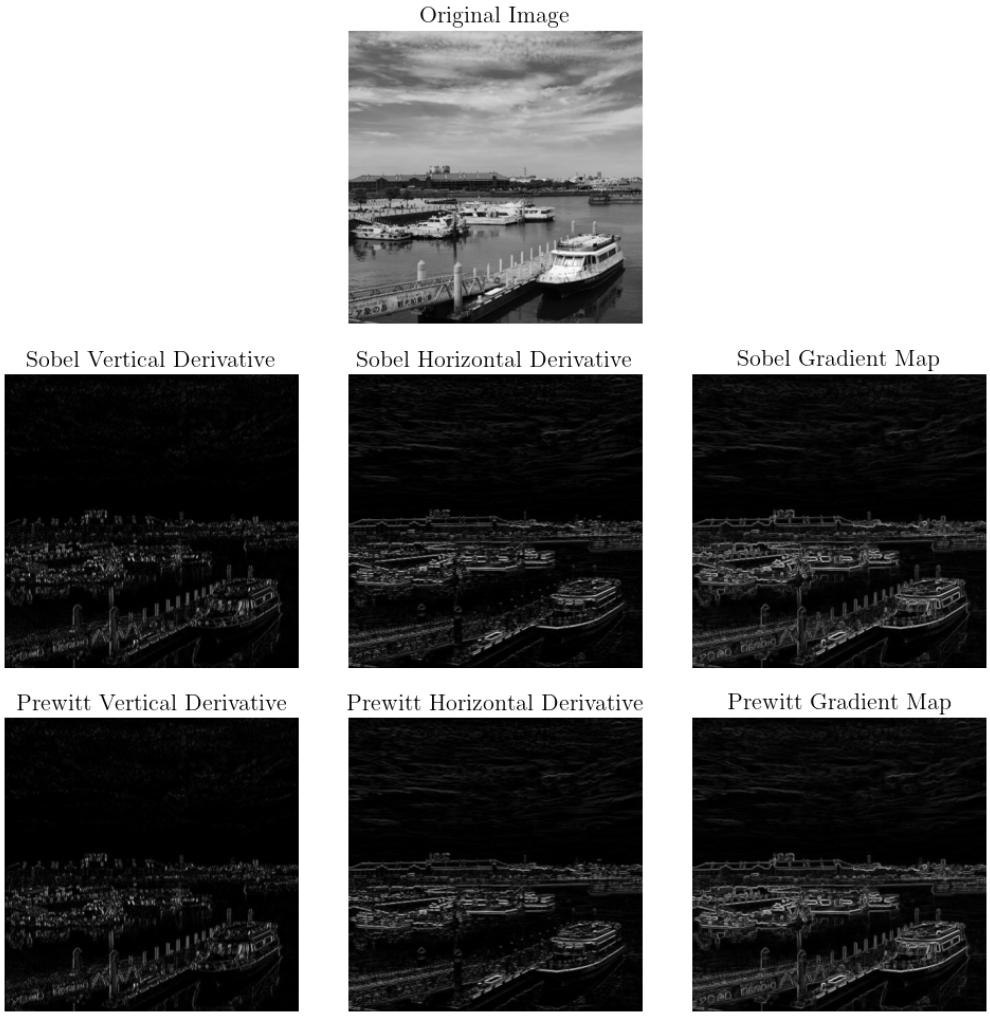
Figure 5 shows the result of applying the Sobel and Prewitt filters for image edge detection, as implemented in the Google Colab notebook. In this implementation, the filters are applied to generate the filtered images (the derivatives), which are then used to create the gradient maps for edge detection. It is important to note that the filtered images, unlike the gradient maps, contain both positive and negative values. However, when plotting the vertical or horizontal edges (the derivatives), we take the absolute values of the filtered images, since the sign of the derivative (positive or negative) is not relevant for edge detection, as it only indicates whether the edge transitions from light to dark or from dark to light. Additionally, the original image used is in grayscale, as edge detection focuses on transitions in pixel intensity rather than on any specific color channel.

1.3.2 Derivative of Gaussian (DoG)

A problem occurs when attempting to detect edges in a noisy image. In this scenario, the changes in intensity might not indicate actual edges but rather discontinuities caused by the noise. Consequently, plotting the gradient map could highlight unnecessary noise that does not necessarily correspond to true edges in the image. For instance, see Figure 6 with its implementation in the Google Colab notebook, that shows the output Sobel gradient map of a noisy input image.

To address this issue, we can first apply a Gaussian filter $\mathbf{G}$ to the noisy image $\mathbf{I}$, and calculate the gradient map of this filtered image instead using some discrete differentiation

Figure 5: Gradient maps for edge detection: Displaying the gradient maps for edge detection created using the Sobel and Prewitt filters. The absolute values of the vertical and horizontal derivatives (i.e., the filtered images from each kernel matrix) are also displayed.



kernels $\mathbf{K}_x$ and $\mathbf{K}_y$. The blurring effect of the Gaussian filter helps to smooth out random noise, allowing us to focus on true changes in pixel intensity.

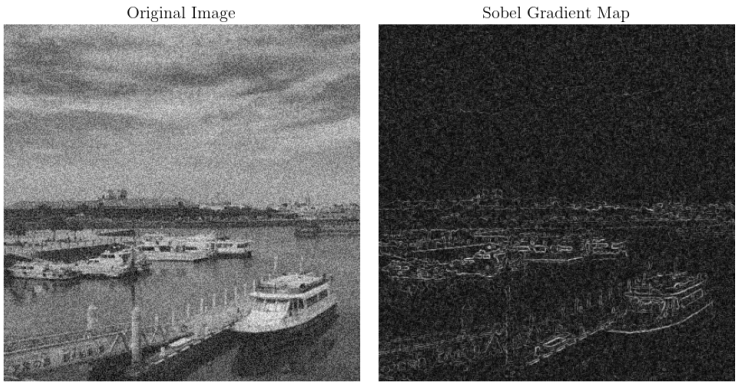
Let $\mathbf{G}$ the Gaussain filter, $\mathbf{K}_x$ and $\mathbf{K}_y$ be discrete differentiation kernels, and $\mathbf{I}$ the noisy image. Mathematically, the resulting operations are:

$$\frac{\partial\left(\mathbf{G} * \mathbf{I}\right)}{\partial x} \approx \mathbf{K}_x * \left(\mathbf{G} * \mathbf{I}\right) = \left(\mathbf{K}_x * \mathbf{G}\right) * \mathbf{I} \approx \frac{\partial \mathbf{G}}{\partial x} * \mathbf{I}, \tag{5}$$

and similarly for $\mathbf{K}_y$. This implies that instead of first convolving $\mathbf{G}$ with $\mathbf{I}$ and then convolving the result with $\mathbf{K}_x$, we can convolve $\partial\mathbf{G}/\partial x$ (which is a constant matrix independent of the choice of discrete differentiation kernel) directly with $\mathbf{I}$.

The matrix $\partial\mathbf{G}/\partial x$ is called the *Derivative of Gaussian* (DoG) kernel matrix, and it automatically blurs (smooths) the image, and calculates the derivatives. Mathematically,

Figure 6: Sobel gradient generated by a noisy input image.



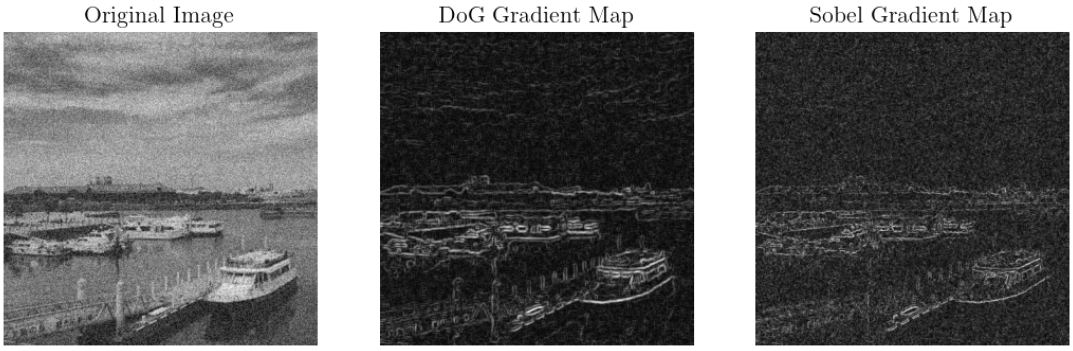
following (1), the 2D DoGs are

$$\frac{\partial \mathbf{G}}{\partial x}(x,y) = -\frac{x}{\sigma^2}\mathbf{G}(x,y) \quad \text{and} \quad \frac{\partial \mathbf{G}}{\partial y}(x,y) = -\frac{y}{\sigma^2}\mathbf{G}(x,y)\,. \tag{6}$$

Recall that we began this discussion with differentiation kernels $\mathbf{K}_x$ and $\mathbf{K}_y$ in (5). It can indeed be shown that for discrete differentiation kernel matrices, such as the Sobel kernels and others, the relationship $\mathbf{K}_x * \mathbf{G} \approx \partial\mathbf{G}/\partial x$ (and similarly for $\mathbf{K}_y$) holds true. These can act as an approximation of the DoG matrices.

Figure 7 displays the gradient map for edge detection of a noisy input image, generated using the DoG filter and the Sobel filter for comparison, as implemented in the Google Colab notebook. It is evident that the DoG filter yields significantly better results. The DoG filter is implemented by performing two 1D convolutions in both the x and y directions for faster results. This can be efficiently computed as follows: for the x direction, the process involves first convolving with the 1D DoG kernel from (6) as a column vector, followed by convolving with the 1D Gaussian kernel (1) treated as a row vector. For the y direction, we first convolve with the 1D Gaussian kernel as a column vector, and then convolve with the 1D DoG kernel treated as a row vector.

Figure 7: Sobel gradient generated by a noisy input image.



1.3.3 Laplacian Filters

When using first-order filters (such as Sobel, Prewitt, or DoG) for edge detection, the edges are highlighted in the resulting gradient map, but the exact coordinates of the edges are not precisely defined. Second-order filters allow for more accurate localization of the edges.

We begin by introducing *discrete second-order differentiation filters*, which are similar to the first-order filters discussed in Section 1.2.3. These filters are linear shift-invariant and can be derived in a similar manner to the first-order case. A common approximation of the second-order derivative of a twice-differentiable function $f\colon \mathbb{R} \to \mathbb{R}$ is given by:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f\left(x+h\right) - 2f\left(x\right) + f\left(x-h\right)}{h^2}, \tag{7}$$

though other approximations also exist. Following a similar derivation as in Section 1.2.3 for first-order differentiation of discrete signals, we arrive at the discrete second derivative approximation kernel $\mathbf{k}^2 = [1, -2, 1]$.

To illustrate this, consider the vector image $\mathbf{v} = (0,0,0,1,1,1,1,0,0,0)$ from Section 1.2.3 with the discrete differentiation kernel vector $\mathbf{k} = [1, -1]$, where the convolution yielded:

$$\mathbf{k} * \mathbf{v} = (0,0,0,1,0,0,0,-1,0,0,0)\,.$$

Now, applying the kernel vector $\mathbf{k}$ to the convolved data again, with a stride of 1 and zero padding of size 1, we obtain:

$$\mathbf{k} * (\mathbf{k} * \mathbf{v}) = (0,0,0,1,-1,0,0,0,-1,1,0,0,0)\,,$$

and it becomes clear that $\mathbf{k} * (\mathbf{k} * \mathbf{v}) = \mathbf{k}^2 * \mathbf{v}$, , which is a discrete approximation of the second-order derivative of $\mathbf{v}$, when imagining moving along $\mathbf{v}$ from right to left.

More On Discrete Approximations. There are many possible generalizations of this concept. For instance, with first-order filters, we approximated the derivatives along the x and y directions and then combined these results in a gradient map by summing their squares and taking the square root. However, it is also possible to combine these results differently – such as by simply adding the two derivatives together using the kernel matrix $\mathbf{K}_x + \mathbf{K}_y$. Since this kernel inherently accounts for both directions, constructing a gradient map is unnecessary (recall that the gradient map was a method for combining results from different directions into a single 2D representation). To better illustrate this, consider a 2D image $\mathbf{I}$. Following (4), we have:

$$\frac{\partial \mathbf{I}}{\partial x}(x,y) + \frac{\partial \mathbf{I}}{\partial y}(x,y) \approx \frac{\mathbf{I}\left(x+0.5h,0\right) - \mathbf{I}\left(x-0.5h,0\right) + \mathbf{I}\left(0,y+0.5h\right) - \mathbf{I}\left(0,y-0.5h\right)}{h}.$$

Plugging $h = 2$ (as done in the derivations in Section 1.2.3), we obtain the discrete approximation first-order kernel matrix:

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

Similarly, we can also take the sum of the derivatives along the x and y axes, together with the sum of directional derivatives along the two directions $(1, \pm 1)$, resulting with the kernel

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

which is also the kernel obtained by summing the two Sobel kernels (for the two directions), up to a scalar multiplication.

Deriving Laplacian Kernels. Second-order filters are filters that utilize this concept of approximations but with second-order derivatives of discrete signals. Specifically, Laplacian filters approximate the sum of the unmixed second-order derivatives, known as the *Laplacian*:

$$\frac{\partial^2 \mathbf{I}}{\partial x^2}(x, y) + \frac{\partial^2 \mathbf{I}}{\partial y^2}(x, y) \tag{8}$$

Following (7), a second-order approximation kernel matrix corresponding to (8) for a 2D image is:

$$\mathbf{K}^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

known as the *Five-Point Stencil* filter, as it consists of a grid including the central point and its four neighbors. An alternative second-order approximation of the sum in (8) involves using all neighboring points, hence it is called the *Nine-Point Stencil* filter, defined as:

$$\mathbf{K}^2 = \begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & -3 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{bmatrix}$$

Figure 8 shows the two Laplacian filters alongside the gradient map of the Sobel filter as implemented in the Google Colab notebook. It is clear that while the first-order Sobel gradient map highlights the edges, the second-order Laplacian filters have *zero crossings* at the edges, enabling more precise edge localization, though this method can be less convenient. Since the output of the Laplacian filter can be negative, and we only care about the absolute rate of change, we plot their map of absolute values (as we also did for the Sobel and Prewitt directional maps).

1.3.4 Laplacian of Gaussain (LoG)

Similarly to the first-order DoG discussed in Section 1.3.2, for noisy images, we can first apply a Gaussian filter for smoothing, and then use second-order derivative kernels for zero-crossing edge detection. Like the derivation of the DoG filter, this process involves taking the second-order derivative of the Gaussian filter and convolving it with the noisy image.

The second-order derivative of the Gaussian filter is known as the Laplacian of Gaussian (LoG), and it is defined for a 2D Gaussian kernel as follows:

$$\frac{\partial^2 \mathbf{G}}{\partial x^2}(x, y) + \frac{\partial^2 \mathbf{G}}{\partial y^2}(x, y) = \frac{x^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y) + \frac{y^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y) = \frac{x^2 + y^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y). \tag{9}$$

Figure 8: Two variations of the second-order Laplacian filter, compared with the first-order Sobel filter for edge detection.

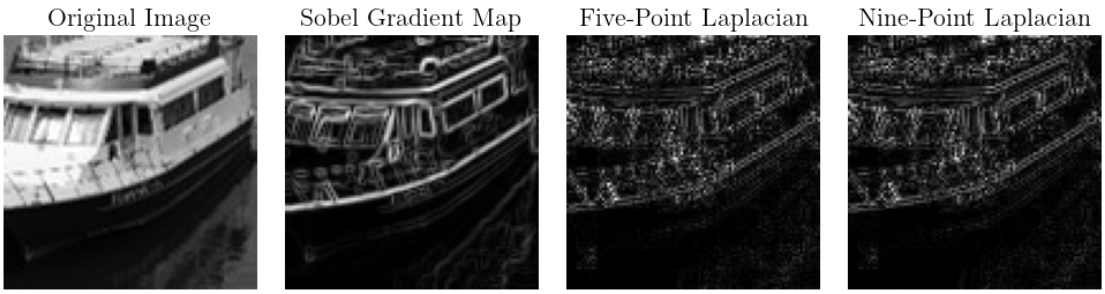
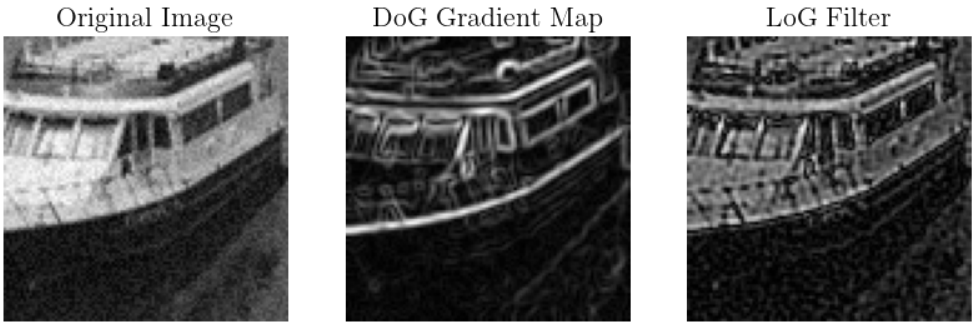


Figure 9 shows the results of applying the LoG filter. Since the LoG filter can produce negative values, the absolute values are plotted, as we are primarily interested in the magnitude of the change. The figure compares the LoG output to the DoG gradient map, and it is clear that the LoG filter exhibits zero-crossings at the edges. The implementation in the Google Colab notebook performs the LoG filtering using two 1D convolutions, as discussed in the DoG implementation in Section 1.3.2.

Figure 9: The absolute values of the LoG filter (zero crossings) and the gradient map produced by the DoG filter when applied to a noisy image.



2 Image Pyramids and Frequency Domains

In signal processing, *sampling* refers to the conversion of a continuous-time signal into a discrete-time signal. A *sample* is the value of the signal at a specific point in time. For time-varying functions, let $s(t)$ represent a continuous signal that is sampled every T seconds, known as the *sampling interval*. The resulting sampled function is expressed as the sequence $s(nT)$ for $n \in \mathbb{N}$. The *sampling frequency* f_s is the average number of samples taken per second, given by $f_s = 1/T$, typically measured in samples per second, or *hertz*. For instance, a sampling rate of 48 kHz corresponds to 48000 samples per second.

Undersampling occurs when we do not collect enough samples from the continuous signal, leading to a loss of information about the original signal. As a consequence, undersampling

can cause the signal to be mistaken for a lower-frequency one, a phenomenon known as *aliasing*. There are two primary strategies to address aliasing: *oversampling* the signal, which incurs a higher sampling cost, and *smoothing* the signal by filtering out details that cause aliasing. While smoothing results in some loss of information, it is preferable to the distortions introduced by aliasing.

In image processing, the term *frequency* describes the rate at which pixel values change across the image. Specifically:

- *Low-frequency components* correspond to slow or gradual changes in pixel values, such as smooth gradients or areas of uniform color.
- *High-frequency components* are associated with rapid changes in pixel values, typically found in areas with sharp edges or detailed patterns. For example, at a sharp edge in an image, where a black area meets a white area, the pixel intensity changes abruptly over a few pixels, representing a high-frequency component.

Figure 10 (as implemented in the Google Colab notebook) illustrates the impact of aliasing during downsampling and the use of smoothing as an anti-aliasing technique. Each downsampling step removes every other row and column, which leads to an image that appears blocky and pixelated due to aliasing. These artifacts arise because high-frequency details like sharp edges are sampled too sparsely, causing visual distortions and misinterpretation as lower frequencies. Conversely, when downsampling is combined with smoothing (such as using a Gaussian blur filter), many fine details are lost, but the severity of aliasing artifacts is reduced.

By reducing the intensity of high-frequency components, blurring ensures that these components do not get misrepresented (aliased) when the image is downsampled. The Gaussian blur acts as a *low-pass filter*, allowing only lower-frequency components (which can be more accurately sampled) to pass through.

2.1 Pyramids in Image Processing

Pyramid refers to a multi-scale image representation technique where an image undergoes repeated smoothing and subsampling. There are two primary types of pyramids:

1. *Lowpass pyramid*: This type is created by repeatedly applying a smoothing filter followed by subsampling, typically by a factor of two along each coordinate direction. It is termed a lowpass pyramid because the smoothing filter functions as a low-pass filter, allowing only lower-frequency components to pass through. As a result, the final levels of the pyramid primarily preserve large uniform regions of the original image.

   An example of a lowpass pyramid is the *Gaussian pyramid*, where smoothing is done using a Gaussian blur filter, and each step removes every other row and column. Such pyramids are useful for tasks like thumbnail generation, where only lower-resolution levels are stored. However, the original image cannot be reconstructed from these lower-resolution images.

Figure 10: Downsampling an image by removing every other row and column at each step. The top row shows the results without anti-aliasing, while the bottom row includes a Gaussian blur filter serving as a low-pass filter for anti-aliasing.

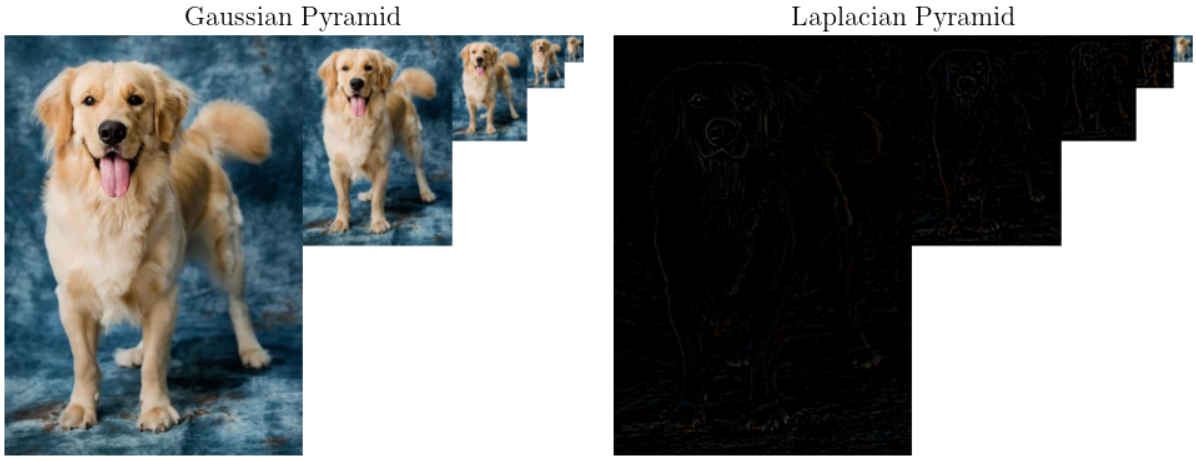
2. *Bandpass pyramid*: This pyramid is constructed by forming the difference between images at adjacent levels in the lowpass pyramid. To calculate these differences, image interpolation is performed between adjacent resolution levels, allowing pixel-wise differences to be computed. No subtraction is performed at the last level.

   An example of a bandpass pyramid is the *Laplacian pyramid*, which uses the Gaussian pyramid as its underlying lowpass pyramid. To calculate pixel-wise differences between levels, a common approach is to use *nearest-neighbor interpolation*. In this method, when new pixel rows and columns are added between existing ones during upsampling, each new pixel in the enlarged image is assigned the value of the nearest pixel from the image before upsampling. The Laplacian pyramid is useful for image compression because the original image can be reconstructed from the final level of the pyramid and the differences stored between levels.

Figure 11 displays the Gaussian and Laplacian pyramids, as implemented in the Google Colab notebook). In the Gaussian pyramid, the image is progressively downsampled and blurred at each step, moving from left to right in Figure 11. For the Laplacian pyramid, which is derived from the Gaussian pyramid, the process starts with the smallest image in the Gaussian pyramid. At each step, the image is upsampled using nearest-neighbor interpolation and the pixel-wise differences with the next larger image in the Gaussian pyramid are calculated, moving from right to left in Figure 11. Notice that the Laplacian pyramid

contains large black areas (pixel values are 0) because most of the regions in the difference between two Gaussian-blurred images (after interpolation of the smaller one) remain relatively unchanged.

Figure 11: Gaussian and Laplacian pyramids.



nterestingly, a Laplacian difference approximates the difference between two Gaussian filters applied to an image, each with a different standard deviation $\sigma > 0$. To see this, recall that if $\mathbf{G}_l(x, y, \sigma)$ rrepresents the Gaussian pyramid at level $l \in \mathbb{N}$ at coordinates (x, y) with standard deviation $\sigma > 0$, the Laplacian pyramid at that same level $\mathbf{L}_l(x, y)$ is computed as:

$$\mathbf{L}_l(x, y) = \mathbf{G}_l(x, y, \sigma) - \textsc{Upsample}\left(\mathbf{G}_{l+1}(x, y, \sigma)\right).$$

The upsampling process involves interpolating the lower-resolution image to match the size of the higher-resolution image. Since the lower-resolution image $\mathbf{G}_{l+1}(x, y, \sigma)$ was already blurred with a Gaussian filter during the downsampling process, when we upscale it, the upsampled image approximates the higher-resolution image $\mathbf{G}_l(x, y, \sigma)$ but with less blurring (smaller σ). This means that

$$\textsc{Upsample}\left(\mathbf{G}_{l+1}(x, y, \sigma)\right) \approx \mathbf{G}_l(x, y, \sigma - h)$$

for small enough $h > 0$. Equivalently, we have:

$$\mathbf{L}_l(x, y) \approx \mathbf{G}_l(x, y, \sigma + h) - \mathbf{G}_l(x, y, \sigma).$$

It can be shown that:

$$\frac{\partial \mathbf{G}_l}{\partial \sigma^2} = \lim_{h \to 0} \frac{\mathbf{G}_l(x, y, \sigma + h) - \mathbf{G}_l(x, y, \sigma)}{h} = \frac{1}{2}\left(\frac{\partial^2 \mathbf{G}_l}{\partial x^2}(x, y, \sigma) + \frac{\partial^2 \mathbf{G}_l}{\partial y^2}(x, y, \sigma)\right),$$

where following (9), this is half the Laplacian of the Gaussian $\mathbf{G}_l$ (LoG) with standard deviation $\sigma > 0$. Therefore, it follows that:

$$\mathbf{L}_l(x, y) \approx \frac{h}{2}\left(\frac{\partial^2 \mathbf{G}_l}{\partial x^2}(x, y, \sigma) + \frac{\partial^2 \mathbf{G}_l}{\partial y^2}(x, y, \sigma)\right),$$

for small $h > 0$. This implies that the Laplacain difference image $\mathbf{L}_l$ is the Laplacian of the Gaussian $\mathbf{G}_l$ (LoG) up to scaling.

2.1.1 Applications of Laplacian Pyramids

Image Compression and Reconstruction. When constructing a Laplacian pyramid, the process involves subtracting an upsampled version of a lower-resolution image from its higher-resolution counterpart in the Gaussian pyramid. This subtraction operation captures the high-frequency details (edges, textures) that are lost during the downsampling process. Since the difference between two Gaussian-blurred images often results in large areas of near-zero values (because low-frequency information is similar at adjacent levels), the Laplacian pyramid representation is sparse, and can be stored more efficiently.

The original image can be reconstructed from the Laplacian pyramid by reversing the pyramid construction process. Starting from the smallest (coarsest) image in the Gaussian pyramid, each level is upsampled (using interpolation techniques) and added back to the corresponding Laplacian level. This process effectively reintroduces the high-frequency details that were captured in the Laplacian pyramid, allowing the original image to be reconstructed.

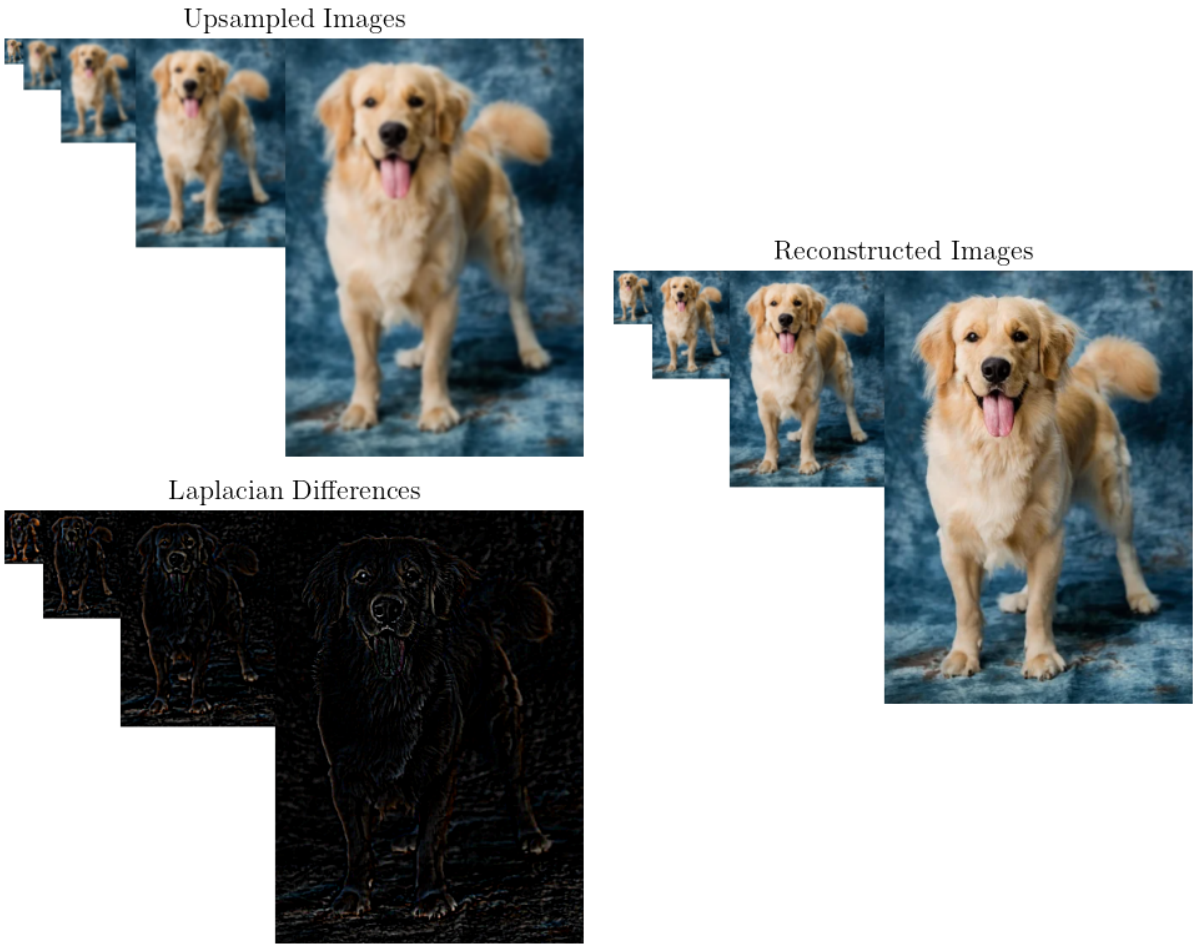
Remember that upsampling involves interpolation, which introduces new pixel values through approximation. The Laplacian differences capture small-scale, high-frequency details, and although upsampling adds some approximation, the overall structure of the image remains largely unaffected.

Figure 12 illustrates the process of reconstructing the original image from its smallest (most downsampled) version using a Laplacian pyramid, as implemented in the Google Colab notebook). The top figure displays the sequence of upsampled images, starting from the smallest (which is the last image of the Laplacian pyramid) to the most upsampled. The figure shows the corresponding Laplacian differences (displayed with higher contrast just for visualization), which are added to the upsampled images, while the right figure displays the step-by-step reconstruction of the original image. The process begins with the smallest image in the pyramid, which is successively upsampled by doubling its size through the insertion of new pixels between rows and columns using nearest-neighbor interpolation. After each upsampling step, the corresponding Laplacian difference is added to restore fine details, bringing the upsampled image closer to the original resolution. The process continues iteratively: upsampling the current reconstruction, adding the next Laplacian difference, and progressively reconstructing the image until we achieve the final version.

Image Blending. Laplacian pyramids can be used for image blending because they allow for integration of two images by decomposing them into different frequency components. In this process, corresponding levels from the two Laplacian pyramids (on of each image) are combined, often by concatenating or mixing regions, resulting in a single blended Laplacian pyramid. For example, you can take two Laplacian differences at the same level from each pyramid, and merge them by concatenating one half of one difference image with one half of the other. During the reconstruction phase, the low-frequency components establish a base that smooths out abrupt transitions between the blended regions. The more similar the blended regions are, the more seamless the overall outcome will be.

Figure 13 illustrates the blending process of two images, as implemented in the Google Colab notebook. The two original images to be blended are displayed at the top of Figure 13, together with their naive blending. The middle section of Figure 13 shows the blended Laplacian differences, where the images are stretched for visualization purposes. Initially,

Figure 12: Image reconstruction progresses from the most downsampled image (top left corner) to the fully reconstructed version (bottom right corner) by iteratively upsampling and adding the corresponding Laplacian difference.

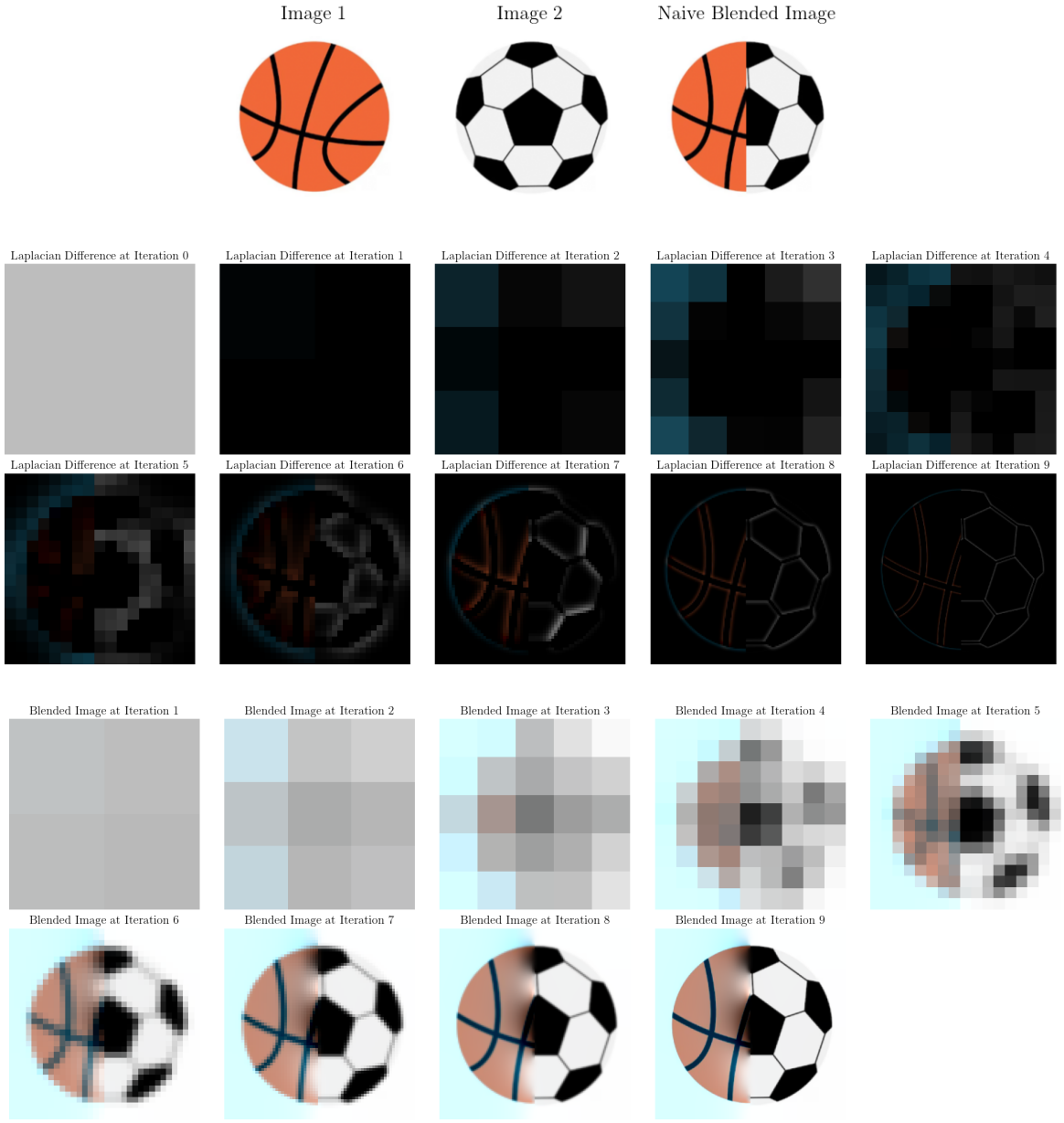


the image Laplacian Difference at Iteration 0 is just a single pixel (although typically, the downsampling process does not need to go that deep). As we upsample that image, its size is doubled, and the next Laplacian Difference at Iteration 1 is added, producing the first reconstructed image, which is shown at the bottom of the Figure 13 under Blended Image at Iteration 1. Early in the process, the images represent very low-frequency components with few sharp edges, resulting in almost smooth images. As the process continues, higher-frequency details are progressively added, blending the images while still maintaining the relatively smooth nature of the low-level images. However, if the original images have very different pixel values along the blending line, the final blended image may appear less natural as more high-frequency details are introduced.

It is important to note that as we go deeper with downsampling, we progressively lose higher-frequency details, making the reconstruction process more challenging. In the example shown in Figure 13, we see that the downsampling went too deep (resulting in a single pixel at the last level), which led to the emergence of a blue background, even though it was not

present in either of the original images. In this case, a shallower depth might have been more effective. This occurs because, during the construction of the Laplacian pyramid, Gaussian blur is applied, which calculates weighted average of pixel values. As we go deeper and lose more pixels, high-intensity pixels can dominate the average, even if they were not prominent in the original images.

Figure 13: Image blending process of two images (top), including their blended Laplacian pyramid (middle), and the reconstructed images at each iteration (bottom). In this example, the depth of the underlying Gaussian pyramid for each image is 10.



2.2 Fourier Series

A *Fourier series* is a series expansion of a periodic function into a sum of trigonometric functions, becoming easier to analyze because trigonometric functions are well understood. "Well-behaved" functions, for example smooth functions, have Fourier series that converge to the original function. Fourier series are closely related to the *Fourier transform*, which can be used to find information for functions that are not periodic.

The Cross-Correlation Operation. This operation is defined as the integral (or summation in the discrete case) of the product of two functions, where one function is shifted relative to the other. In signal processing, cross-correlation is used to measure the similarity between two signals as a function of the displacement of one signal relative to the other. It is often applied to detect the presence of a known feature within a longer signal.

Mathematically, given two functions $f, g \colon \mathbb{R}^d \to \mathbb{R}$, their cross-correlation is:

$$(f \star g)(\mathbf{x}) \equiv \int_{\mathbb{R}^d} f(\mathbf{y})\, g(\mathbf{y}+\mathbf{x})\, d\mathbf{y} = \int_{\mathbb{R}^d} f(\mathbf{y}-\mathbf{x})\, g(\mathbf{y})\, d\mathbf{y} = (f(-\mathbf{x}) * g(\mathbf{x}))(\mathbf{x}).$$

Note that if f is an even function, then $f \star g = f * g$. If both f and g are even, then $f \star g = g \star f$.

For finite discrete functions with N possible input vectors, their cross-correlation is:

$$(f \star g)(\mathbf{n}) \equiv \sum_{\mathbf{m}} f(\mathbf{n})\, g((\mathbf{m}+\mathbf{n}) \mod N),$$

where $\mathbf{n} \in \mathbb{R}^d$ is one of the N possible inputs, $\mathbf{m} \in \mathbb{R}^d$ is a shifting vector, and the modulus operation is applied element-wise.

Cross-correlation indeed measures similarity between two functions. Intuitively, when we multiply two functions point by point, we are essentially comparing their values at each point in time. If both functions have high values at the same time, the product will be large. If one function is large and the other is small (or negative), the product will be smaller or even negative. If they are out of phase (i.e., one is positive while the other is negative), the product will be negative. The integral then sums up the products over the entire time domain. Hence, by taking the maximum of the cross-correlation over all possible time shifts, we find the time shift (or phase) at which the two functions are most similar. This maximizing time shift indicates how much one function needs to be shifted relative to the other to achieve the best alignment, which corresponds to the point of maximum similarity.

2.2.1 Forms of the Fourier Series

Amplitude-Phase Form. Let $s \colon \mathbb{R} \to \mathbb{R}$ be a "well-behaved" periodic function with period T (the terms function and signal are used interchangeably). Then, there exist coefficients such that:

$$s(t) = D_0 + \sum_{k=1}^{\infty} D_k \cos\left(\frac{2\pi k t}{T} - \phi_k\right) = D_0 + \sum_{k=1}^{\infty} D_k \sin\left(\frac{2\pi k t}{T} - \phi_k + \frac{\pi}{2}\right), \tag{10}$$

which is known as the Fourier series of s in its *amplitude-phase form*. In this representation, the coefficients D_k are called the *amplitudes*, $f_k = k/T$ are the *frequencies*, $f_0 \equiv 1/T$ is the *fundamental frequency*, and ϕ_k are called the *phases*. The proof that an infinite sum of sinusoidal components indeed converges to the original function s is beyond the scope of these notes.

Recall that a sinusoidal function with frequency $f \in \mathbb{R}$ is any function of the form

$$D\cos\left(2\pi ft - \phi\right) = D\sin\left(2\pi ft - \phi + \frac{\pi}{2}\right).$$

Hence, up to amplitude scaling and phase shifting (that is, setting A and ϕ above), equation (10) implies that any well-behaved periodic function can be decomposed into a sum of all possible sinusoidal functions with period T and frequencies that are integer multiples of the fundamental frequency, known as *harmonics*. For example, the kth harmonic, $k \in \mathbb{N} \cup \{0\}$, corresponds to the frequency $kf_0 = k/T$.

Additionally, it can be shown that summing such sinusoidal functions with a sinusoidal function whose frequency is not an integer multiple of the fundamental frequency $f_0 = 1/T$ will not converge to a periodic signal with period T. This is why we only consider all possible harmonics. Each such sinusoidal function is then summed according to its relative contribution to the signal s, as represented by the coefficient D_k. Consequently, the coefficient D_k represents the degree of similarity between each of these sinusoidal functions and the signal.

Deriving the amplitudes D_k and the phases ϕ_k: For each harmonic $k \geq 0$, we need to find the time shift (phase) that maximizes the similarity between the sinusoidal function $\cos\left(2\pi kt/T\right)$ and the signal. Following the definition of cross-correlation, we define $\tilde{X}_k\left(\phi\right)$ as the cross-correlation between the signal s and $\cos\left(2\pi kt/T\right)$. That is:

$$\tilde{X}_k\left(\phi\right) \equiv \int_T s\left(t\right)\cos\left(\frac{2\pi k\left(t-\phi\right)}{T}\right)dt,$$

where $\int_T$ represents integration over the interval, usually $[0, T]$ or $[-T/2, T/2]$. Since the effective possible shifts of a sinusoidal function are $[0, 2\pi]$, then we can equivalently maximize the function:

$$X_k\left(\phi\right) \equiv \frac{2}{T}\int_T s\left(t\right)\cos\left(\frac{2\pi kt}{T} - \phi\right)dt, \quad \phi \in \left[0, 2\pi\right],$$

where the scaling factor $2/T$ is added solely for the purpose of convenience in the values of the coefficients (they will be scaled appropriately), and we mention that for $k = 0$ we set the scaling factor $1/T$. Using the trigonometric identity

$$\cos\left(\alpha - \beta\right) = \cos\left(\alpha\right)\cos\left(\beta\right) + \sin\left(\alpha\right)\sin\left(\beta\right), \tag{11}$$

and substituting $\alpha = 2\pi kt/T$ and $\beta = \phi$, we can maximize X_k with respect to ϕ (equating the derivative to 0) and get:

$$A_k\cos\left(\phi\right) - B_k\sin\left(\phi\right) = 0, \tag{12}$$

where we define

$$\begin{aligned} A_0 &= \frac{1}{T}\int_T s\left(t\right)dt, \\ A_k &= \frac{2}{T}\int_T s\left(t\right)\cos\left(\frac{2\pi kt}{T}\right)dt, \quad k \geq 1, \\ B_k &= \frac{2}{T}\int_T s\left(t\right)\sin\left(\frac{2\pi kt}{T}\right)dt, \quad k \geq 1. \end{aligned} \tag{13}$$

These last equations are known as the *Fourier series analysis*. So, following (13) we get that the optimal phase is given by:

$$\phi_k = \text{atan2}\left(B_k, A_k\right),$$

where

$$\text{atan2}\left(y,x\right) \equiv \begin{cases} \arctan\left(y/x\right), & x > 0, \\ \arctan\left(y/x\right) + \pi, & x < 0, y \geq 0 \\ \arctan\left(y/x\right) - \pi, & x > 0, y < 0 \\ \text{sign}\left(y\right)\cdot \pi/2, & x = 0, y \neq 0 \\ \text{undefined}, & x = y = 0. \end{cases}$$

Now, in order to derive the value of the coefficient D_k, following the explanations above we notice that $D_k \equiv X_k\left(\phi_k\right)$, and after some algebraic manipulations we get:

$$D_0 = A_0 \quad \text{and} \quad D_k = \cos\left(\phi_k\right)A_k + \sin\left(\phi_k\right)B_k = \sqrt{A_k^2 + B_k^2}. \tag{14}$$

Finally, we point out that the coefficient $D_0 = A_0$ (the 0 frequency) as defined in (13), is the average value of the signal s over the time interval T.

Sine-Cosine Form. From (11) we get that

$$A_k = \frac{B_k\cos\left(\phi_k\right)}{\sin\left(\phi_k\right)} \quad \text{and} \quad B_k = \frac{A_k\sin\left(\phi_k\right)}{\cos\left(\phi_k\right)}.$$

Plugging the above into (14) we get that

$$A_k = D_k\cos\left(\phi_k\right) \quad \text{and} \quad B_k = D_k\sin\left(\phi_k\right). \tag{15}$$

Hence, using the trigonometric identity (11) together with (15), we get that the series (10) can be equivalently expressed in the *sine-cosine form*:

$$s\left(t\right) = A_0 + \sum_{k=1}^{\infty}\left(A_k\cos\left(\frac{2\pi kt}{T}\right) + B_k\sin\left(\frac{2\pi kt}{T}\right)\right). \tag{16}$$

Exponential Form. Using Euler's formula

$$e^{jt} = \cos\left(t\right) + j\sin\left(t\right),$$

we get that

$$\cos\left(\frac{2\pi kt}{T}-\phi_k\right)=\left(\frac{1}{2}e^{-j\phi_k}\right)\cdot e^{\frac{j2\pi kt}{T}}+\left(\frac{1}{2}e^{-j\phi_k}\right)^*\cdot e^{-\frac{j2\pi kt}{T}}.$$

Plugging the above into (10), we equivalently get the Fourier series in its *exponential form*:

$$s\left(t\right)=\sum_{k=-\infty}^{\infty}C_k e^{\frac{j2\pi kt}{T}}, \tag{17}$$

where, following (10), (15) and (16) we get that:

$$C_0=D_0=A_0,\quad C_{k\geq 1}=\frac{D_k e^{-j\phi_k}}{2}=\frac{\left(A_k-jB_k\right)}{2}\quad\text{and}\quad C_{k\leq -1}=C_{-k}^*. \tag{18}$$

Additionally, note that now it follows that:

$$A_k=C_k+C_{-k}\quad\text{and}\quad B_k=j\left(C_k-C_{-k}\right) \tag{19}$$

Following (13), (17) and (18), we can write the Fourier series analysis in exponential form:

$$C_k=\frac{1}{T}\int_T s\left(t\right)e^{\frac{-j2\pi kt}{T}}dt,\quad \forall k\in\mathbb{Z}. \tag{20}$$

Square wave signal. For example, a *periodic square wave* signal $s\left(t\right)$ with period T and amplitude A is defined over $[0,T]$ as:

$$s\left(t\right)=A\cdot\operatorname{sign}\left(\sin\left(\frac{2\pi t}{T}\right)\right)=\begin{cases}A, & 0<t<T/2,\\ -A, & T/2<t<T,\\ 0, & \text{otherwise.}\end{cases}$$

From the Fourier series analysis we get:

$$\begin{aligned}A_0&=\frac{A}{T}\int_0^{T/2}1dt-\frac{A}{T}\int_{T/2}^{T}1dt=0\\ A_k&=\frac{2A}{T}\int_0^{T/2}\cos\left(\frac{2\pi kt}{T}\right)dt-\frac{2A}{T}\int_{T/2}^{T}\cos\left(kx\right)dx=0\\ B_k&=\frac{2A}{T}\int_0^{T/2}\sin\left(\frac{2\pi kt}{T}\right)dt-\frac{2A}{T}\int_{T/2}^{T}\sin\left(\frac{2\pi kt}{T}\right)dt=\begin{cases}\frac{4A}{\pi k}, & k\text{ odd},\\ 0, & k\text{ even}.\end{cases}\end{aligned}$$

Hence, the Fourier series of s with period T is given as:

$$s\left(t\right)=\frac{4A}{\pi}\sum_{k=1}^{\infty}\frac{1}{2k-1}\sin\left(\frac{2\pi\left(2k-1\right)t}{T}\right).$$

In exponential form we get $C_k=\left(A_k-jB_k\right)/2=-j2A/\pi k$ for $k\geq 1$ and odd, $C_k=j2A/\pi k$ for $k\leq -1$ and odd, and $C_k=0$ otherwise. Hence

$$s\left(t\right)=\frac{j2A}{\pi}\sum_{k=-\infty}^{\infty}\frac{-\operatorname{sign}\left(k\right)}{2k-1}e^{\frac{j2\pi(2k-1)t}{T}}.$$

2.2.2 Time-Domain and Frequency-Domain

The *time domain* is the domain in which signals or functions are expressed as they evolve over time t. The *frequency domain* represents signals in terms of their frequencies rather than time. Instead of showing how a signal changes over time, it shows how much of the amplitude lies within each given frequency over a range of frequencies. One of the main reasons for using a frequency-domain representation of a problem is to simplify the mathematical analysis.

For a periodic signal, the frequency domain plots can be obtained from its Fourier series, which represents the signal as a sum of sinusoids (sines and cosines) with different frequencies, known as *harmonics*. As introduced above, a harmonic is a sinusoidal component of the signal whose frequency is an integer multiple of the *fundamental frequency*, which is the inverse of the period of the signal $1/T$. In the frequency domain plot, the x-axis represents the harmonic index, corresponding to these integer multiples of the fundamental frequency. The y-axis can represent either the amplitude (magnitude) of the harmonics, or the the complex Fourier coefficients C_k (either the real and/or imaginary parts $\mathrm{Re}\,(C_k)$ and $\mathrm{Im}\,(C_k)$), which contain both amplitude and phase information. In this context, recall that the amplitude of the k-the harmonic is (see (14) and (18))

$$\text{Amplitude}_k \equiv D_k = \sqrt{A_k^2 + B_k^2} = 2\,|C_k|\,.$$

Additionally, recall that for any $k \geq 1$ it holds that (see (18))

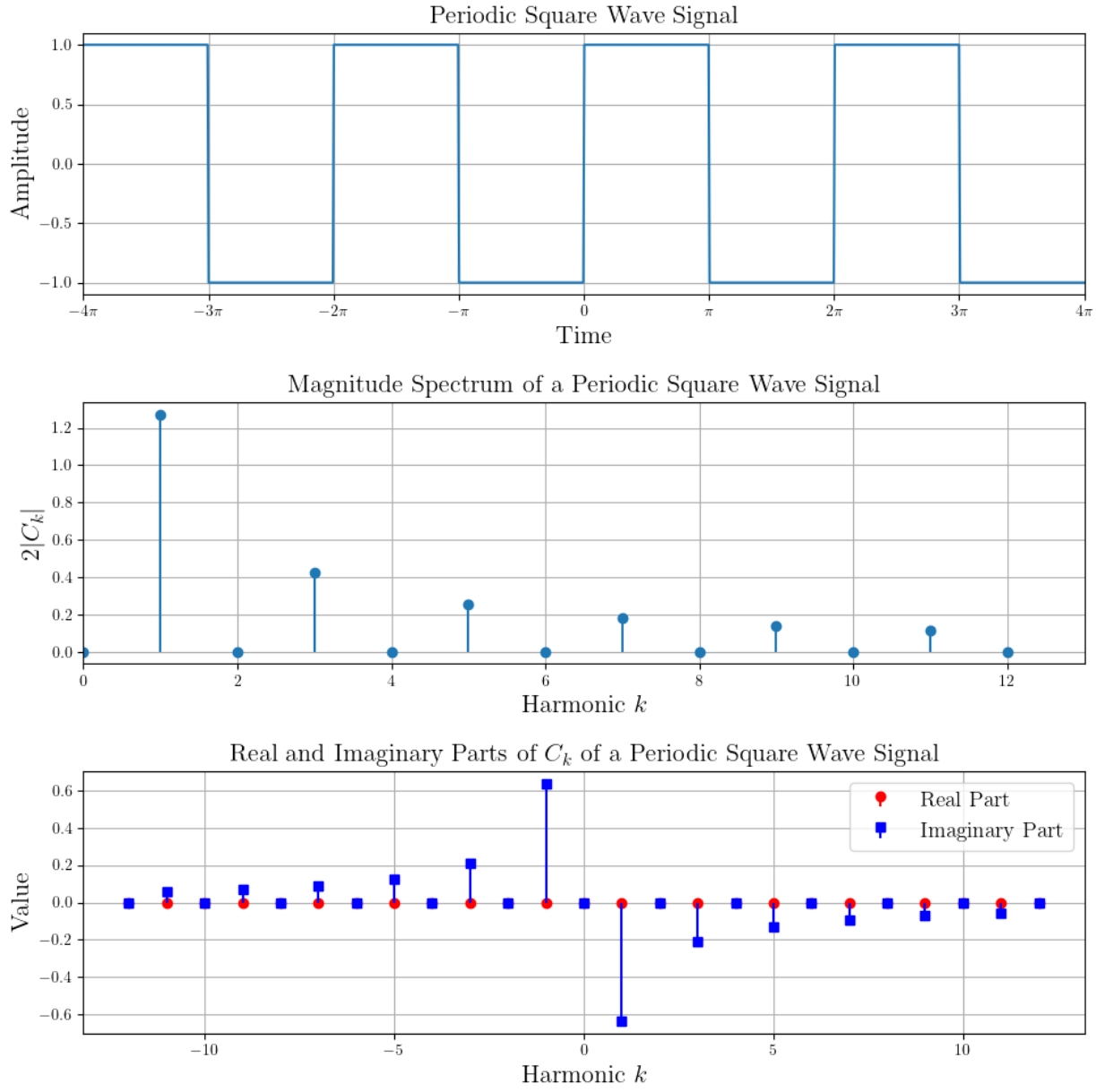
$$\mathrm{Re}\,(C_k) = A_k/2 = \mathrm{Re}\,(C_{-k}) \quad \text{and} \quad \mathrm{Im}\,(C_k) = -B_k/2 = -\mathrm{Im}\,(C_{-k})\,,$$

hence it follows for $k \geq 1$ that (see (14) and (19))

$$\begin{aligned}\text{Phase}_k = \phi_k &= \mathrm{atan2}\,(B_k, A_k) = \mathrm{atan2}\,(j\,(C_k - C_{-k})\,, C_k + C_{-k}) \\ &= \mathrm{atan2}\,(-2\cdot \mathrm{Im}\,(C_k)\,, 2\cdot \mathrm{Re}\,(C_k)) \\ &= \mathrm{atan2}\,(2\cdot \mathrm{Im}\,(C_{-k})\,, 2\cdot \mathrm{Re}\,(C_{-k})) \equiv \arg\,(2C_{-k})\,. \end{aligned} \tag{21}$$

Figure 14, as implemented in the Google Colab notebook, displays the time-domain and frequency-domain plots of the periodic square wave signal introduced above. The top plot displays a square wave with a period of $T = 2\pi$ and an amplitude of $A = 1$ (time-domain) ranging from $t = -4\pi$ to $t = 4\pi$. The middle plot (frequency-domain) shows the amplitudes of the sinusoidal components in the Fourier series of the square wave, ranging from the 0th to the 12th harmonic, although in theory the series includes harmonics from 0 to ∞. For example, the amplitude at the $k = 1$ harmonic is $4/\pi \approx 1.27$, and all even harmonics have an amplitude of 0. The bottom plot (frequency-domain) presents the real and imaginary values of the Fourier coefficients C_k from the -12th to the 12th harmonics, though these values extend from $-\infty$ to ∞. For both the real and imaginary parts, all values at even harmonics are 0. The phase ϕ_k of each harmonic $k \geq 1$ can be retrieved from the bottom plot by taking the argument of $2C_{-k}$, which gives the phase information necessary for reconstructing the original signal. In this case, all phases are 0 (see (21)).

Figure 14: A periodic square wave signal with period $T = 2\pi$ and amplitude $A = 1$. The top plot shows the time-domain of the signal. The middle plot show the amplitudes of the sinusoidal components of the Fourier series over the frequency-domain. The bottom plot shows the values of the complex coefficients C_k of the Fourier series over the frequency-domain.



When interpreting the frequency domain graph, a magnitude (amplitude) value at a particular harmonic indicates the contribution of that frequency component to the overall signal. For example, if the graph shows a significant value at the third harmonic, it means that the signal has a strong component at three times the fundamental frequency $1/T$ (middle plot in Figure 14). The real and imaginary parts describe how it is phased relative to the other components (bottom plot in Figure 14).

We point out that the amplitude spectrum plot (magnitude plot) shows the strength of

each harmonic, but it does not contain phase information. As a result, using the amplitude plot alone, we cannot fully reconstruct the original time-domain signal because phase information is crucial for proper alignment of the sinusoids. The real and imaginary parts of C_k, when plotted, provide the complete information needed to reconstruct the time-domain signal, as they capture both the amplitude and the phase of each harmonic.

2.2.3 Discrete Fourier Series

A *discrete Fourier series* (DFS) is a Fourier series in which the sinusoidal components are functions of discrete time rather than continuous time. Consider a periodic signal s with period T defined over a discrete time domain, where the time intervals are equally spaced (non-equally spaced time intervals are beyond the scope of this discussion). This implies that there exists a time interval $I \geq 0$ such that the discrete time samples are $\{nI\}_{n\in\mathbb{Z}}$. In this case, the discrete analog of the exponential form of the Fourier series, as seen in equation (17), is given by the DFS:

$$s(nI) = \sum_{k=-\infty}^{\infty} C_k e^{\frac{j2\pi knI}{T}}, \quad \forall n \in \mathbb{Z}, \tag{22}$$

where, analog to equation (20), the coefficients C_k are defined using the analysis formula:

$$C_k = \frac{1}{T}\sum_{n=0}^{N} s(nI)\, e^{\frac{-j2\pi knI}{T}}, \quad \forall k \in \mathbb{Z}, \tag{23}$$

where N is the maximal integer $n \geq 0$ such that $nI \leq T$ (hence, we are summing all time samples over the time period T. Notice that if T/I is an integer, then $N = T/I$ and the DFS can also be viewed as a series with a period of N and fundamental frequency $1/N$.

2.3 Fourier Transform

The Fourier series is used to analyze periodic functions, while the Fourier transform can be seen as a generalization that applies to any "well-behaved" function $s\colon \mathbb{R} \to \mathbb{R}$, not necessarily periodic in the time domain.

For non-periodic signals, the Fourier series decomposition into an infinite sum of harmonics, as shown in (10), no longer holds because, as discussed earlier, summing only harmonics results in a periodic signal. Therefore, for non-periodic signals, we need to sum over all possible frequencies across the entire time domain, not just harmonics within a single period. Formally, the "coefficients" in this case can be derived by taking the limit $T \to \infty$ in the analysis formula in (20) while letting $k/T \to \xi$ for some frequency $\xi \in \mathbb{R}$, leading to the *Fourier transform* formula:

$$\hat{s}(\xi) = \int_{-\infty}^{\infty} s(t)\, e^{-j2\pi\xi t} dt. \tag{24}$$

The corresponding decomposition, analogous to (10) and (17), can be proved to be given by the *inverse Fourier transform* formula in its exponential form:

$$s(t) = \int_{-\infty}^{\infty} \hat{s}(\xi)\, e^{j2\pi\xi t} d\xi, \quad \forall t \in \mathbb{R}. \tag{25}$$

The fact that these integrals indeed converge for "well-behaved" functions is beyond the scope of these notes. The two functions s and its Fourier transom $\hat{s}$ are known as the *Fourier transform pair*, commonly denoted as $s(t) \overset{\mathcal{F}}{\longleftrightarrow} \hat{s}(\xi)$. In the n-dimensional Euclidean space the pair takes the form

$$\hat{s}(\boldsymbol{\xi}) = \int_{\mathbb{R}^n} s(\mathbf{t})\, e^{-j2\pi\boldsymbol{\xi}^T\mathbf{t}} d\mathbf{t} \quad \text{and} \quad s(\mathbf{t}) = \int_{\mathbb{R}^n} \hat{s}(\boldsymbol{\xi})\, e^{j2\pi\boldsymbol{\xi}^T\mathbf{t}} d\boldsymbol{\xi}. \tag{26}$$

The intuitive interpretation of the Fourier transform can be understood through the effect of multiplying the signal s by the complex exponential $\exp(-j2\pi\xi t)$ (see (24)). By doing so, we effectively shift all the frequency components of s by $-\xi$. This shift means that any frequency component in s that was originally at frequency f is now at frequency $f-\xi$. In the Fourier transform, we are integrating this product over all time. For most frequency components in s, this shift results in oscillations that cancel out over time when integrated, leading to a value of zero. However, if there is a component of s that exactly matches the frequency ξ, then after the shift, that component produces a non-zero value of the infinite integral. This constant term does not cancel out when integrated, producing a non-zero value. Therefore, the Fourier transform at frequency ξ essentially measures how much of the original signal contains a frequency component at ξ.

As an example to illustrate the above intuition, consider the signal

$$s(t) = \cos(2\pi 3t)\, e^{-\pi t^2} = \frac{1}{2} e^{-\pi t^2 - j2\pi 3t} + \frac{1}{2} e^{-\pi t^2 + j2\pi 3t}. \tag{27}$$

When multiplying the signal by $\exp(-j2\pi\xi t)$ (see (24)), we simply add $-j2\pi\xi t$ to each exponent (shifting) and get:

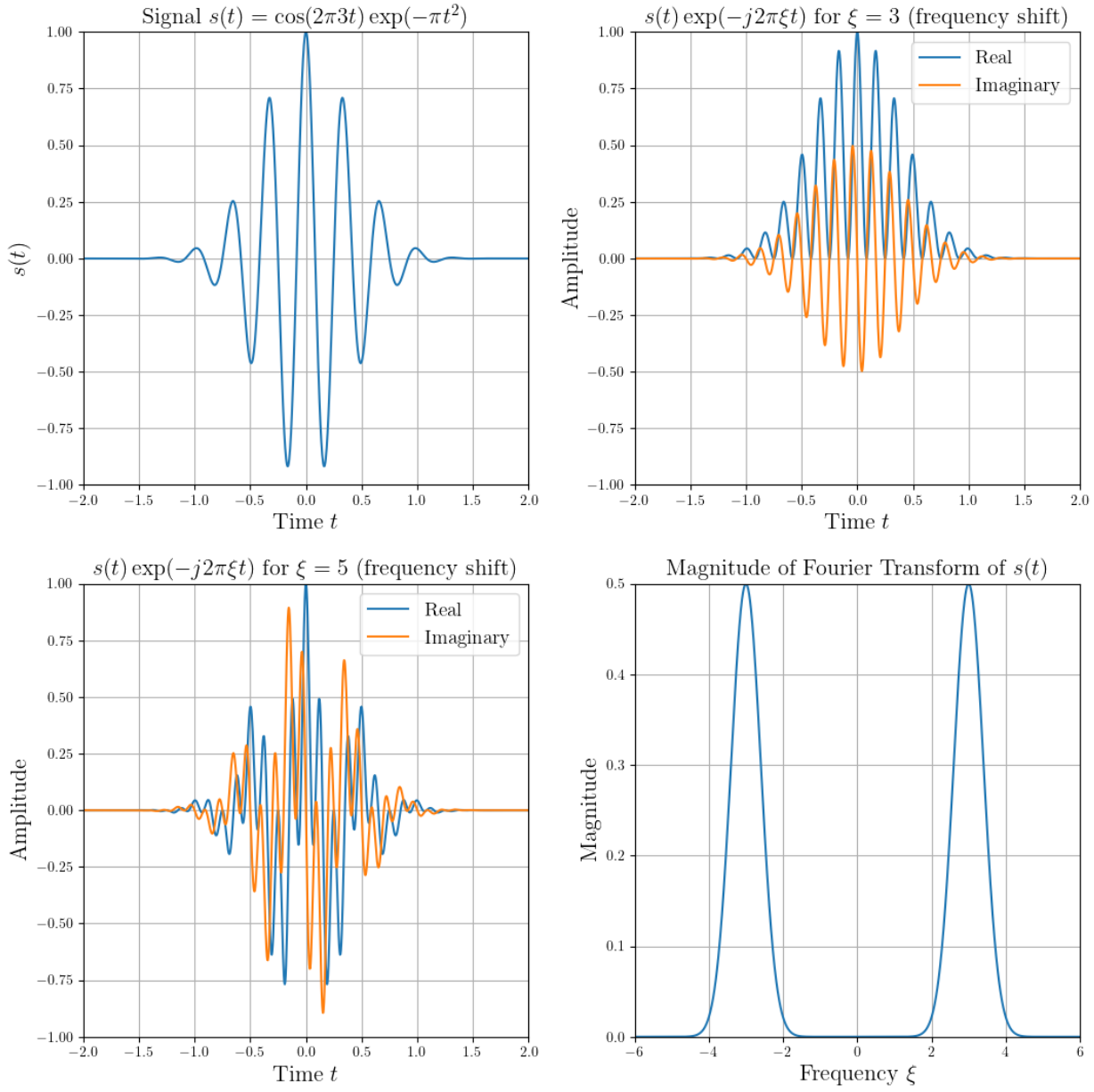
$$\begin{aligned} s(t)\cdot e^{-j2\pi\xi t} = \cos(2\pi 3t)\, e^{-\pi t^2} &= \frac{1}{2} e^{-\pi t^2 - j2\pi(3+\xi)t} + \frac{1}{2} e^{-\pi t^2 + j2\pi(3-\xi)t} \\ &= e^{-\pi t} \cos(6\pi t)\cos(2\pi\xi t) + j e^{-\pi t^2} \cos(6\pi t)\sin(2\pi\xi t). \end{aligned} \tag{28}$$

Now, we can observe that for $\xi = \pm 3$, the real part is non-negative for all $t \in \mathbb{R}$, resulting in a non-zero integral. Meanwhile, the imaginary part oscillates around zero due to the multiplication of two out-of-phase trigonometric functions, leading its integral to be zero or close to zero. When ξ is near ± 3, the integral of the real part approaches the value at ± 3 (as these functions are continuous), while the integral of the imaginary part remains zero or close to zero.

In Figure 15, as implemented in the Google Colab notebook, the top left plot displays the original signal (27). The top right and bottom left plots show the result of multiplying this signal by $\exp(-j2\pi\xi t)$ as in (28), with the real and imaginary parts plotted separately. For a frequency shift of $\xi = 3$ (top right plot), the real part of the product is non-negative, resulting in a positive integral value. The imaginary part oscillates around 0, contributing minimally to the integral (if contributes at all). Consequently, the total integral is close 0.5, as seen in the bottom right Fourier transform plot. The bottom left plot presents the same analysis for $\xi = 5$, where both the real and imaginary parts oscillate around zero, leading to an integral close to 0 (or exactly 0). The bottom right plot shows magnitude (absolute

value) of the Fourier transform of the signal, with peaks at $\xi = \pm 3$ and a near-zero value elsewhere, reflecting the frequency content of the original signal.

Figure 15: The signal described in (27) is shown in the top left plot. The top right plot displays the signal shifted by a frequency of $\xi = 3$ while the bottom left plot shows the signal shifted by $\xi = t$ (refer to (28)). The magnitude (absolute value) of the Fourier transform of the signal is depicted in the bottom right plot.



2.3.1 Properties of the Fourier Transform

Let $s(t) \overset{\mathcal{F}}{\longleftrightarrow} \hat{s}(\xi)$. Then,

1. Time scaling: for $\alpha \neq 0$

$$s(\alpha t) \overset{\mathcal{F}}{\longleftrightarrow} \frac{1}{|\alpha|}\hat{s}\left(\frac{\xi}{\alpha}\right).$$

2. Time shifting: for $t_0 \in \mathbb{R}$

$$s(t - t_0) \overset{\mathcal{F}}{\longleftrightarrow} \hat{s}(\xi)\, e^{-j2\pi\xi t_0}.$$

3. Frequency shifting: for $\xi_0 \in \mathbb{R}$

$$s(t)\, e^{j2\pi\xi_0 t} \overset{\mathcal{F}}{\longleftrightarrow} \hat{s}(\xi - \xi_0).$$

4. Linearity: for $u(t) \overset{\mathcal{F}}{\longleftrightarrow} \hat{u}(\xi)$ and $a, b \in \mathbb{C}$

$$as(t) + bu(t) \overset{\mathcal{F}}{\longleftrightarrow} a\hat{s}(\xi) + b\hat{u}(\xi).$$

5. Duality: when switching the time and frequency domains

$$\hat{s}(t) \overset{\mathcal{F}}{\longleftrightarrow} s(-\xi).$$

   For example, the pair $\operatorname{sinc}(t)$ and the rectangular function $\operatorname{rect}(t)$, and the pair $\operatorname{Gauss}(t, \sigma)$ and $\operatorname{Gauss}(t, 1/\sigma)$.

6. Convolution and cross-correlation: for $u(t) \overset{\mathcal{F}}{\longleftrightarrow} \hat{u}(\xi)$

$$(s * u)(t) \overset{\mathcal{F}}{\longleftrightarrow} \hat{s}(\xi)\,\hat{u}(\xi) \quad \text{and} \quad (s \star u)(t) \overset{\mathcal{F}}{\longleftrightarrow} \hat{s}(\xi)^*\,\hat{u}(\xi).$$

2.3.2 Discrete Fourier Transform

A *discrete Fourier transform* (DFT) converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of a complex-valued function of frequency (the non equally-spaced case is beyond the scope of this discussion). Mathematically, given a finite sequence of complex numbers $s_1, s_2, \ldots, s_{N-1}$, its DFT, analog to equation (24), is:

$$C_k = \sum_{n=0}^{N-1} s_n e^{\frac{-j2\pi kn}{N}}, \quad k = 0, 1, \ldots, N-1, \tag{29}$$

and the *inverse discrete Fourier transform* (IDFT), analog to equation (25), is:

$$s_n = \frac{1}{N}\sum_{k=0}^{N-1} C_k e^{\frac{j2\pi kn}{N}}, \quad n = 0, 1, \ldots, N-1. \tag{30}$$

The proofs that the DFT and the IDFT are special cases of the continuous Fourier transform equations (24) and (25) are beyond the scope of this discussion. It is important to note that, similar to (22) and (23), the IDFT can be considered a form of a DFS when we view the DFS in a more general context as a sum of sinusoidal functions. In this context, remember

that the equations for the DFS and its coefficients, as developed in (22) and (23), apply to discrete and periodic (hence not finite) signals, while the DFT and IDFT, as developed in (29) and (30), apply to discrete and finite (hence not periodic) signals.

For deriving the DFT, notice that it is simply the matrix multiplication $\mathbf{c} = \mathbf{W}\mathbf{s}$, where $\mathbf{c} \equiv (C_0, C_1, \cdots, C_{N-1})^T$, $\mathbf{s} \equiv (s_0, s_1, \cdots, s_{N-1})^T$, and where

$$\mathbf{W} \equiv \begin{pmatrix} w^{0\cdot 0} & w^{0\cdot 1} & \cdots & w^{0\cdot(N-1)} \\ w^{1\cdot 0} & w^{1\cdot 1} & \cdots & w^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w^{(N-1)\cdot 0} & w^{(N-1)\cdot 1} & \cdots & w^{(N-1)\cdot(N-1)} \end{pmatrix}$$

for $w \equiv \exp(-j2\pi/N)$. In practice, this matrix multiplication is calculated using *Fast Fourier Transform* (FFT) algorithms, which reduce the computation complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log(N))$.

We should mention that when we compute a FFT of a signal, for example using `numpy.fft` library, the zero-frequency component (corresponding to the average value of the signal) is located at the beginning of the array (index 0). The positive frequencies follow sequentially, and the negative frequencies are wrapped around to the end of the array. This layout is a consequence of how the FFT algorithm computes and stores the frequency components. The `numpy.fft.fftshift` function shifts the zero-frequency component back to the center of the spectrum and rearranges the frequencies accordingly. Similarly, we can apply an IFFT algorithm, for example using `numpy.fft.ifft`, that is applied after shifting the frequencies back to the output form of the FFT using `numpy.fft.ifftshift`.

2.4 2D Frequency Analysis

Focusing on image processing tasks, we emphasize 2D frequency analysis. For a 2D signal $s(x, y)$ in the time domain, the 2D Fourier Transform is given explicitly by (see (26)):

$$\hat{s}(u, v) = \int_{\mathbb{R}} \int_{\mathbb{R}} s(x, y)\, e^{-j2\pi(ux+vy)} dxdy \quad \text{and} \quad s(x, y) = \int_{\mathbb{R}} \int_{\mathbb{R}} \hat{s}(u, v)\, e^{j2\pi(ux+vy)} dudv.$$

For the discrete case, considering a 2D image $I \in \mathbb{R}^{M\times N}$, its transform $\hat{I} \in \mathbb{R}^{M\times N}$ and its inverse, are given by extending the DFT and IDFT equations (29) and (30) from 1D to 2D as follows:

$$\hat{I}_{pq} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I_{mn} e^{-j2\pi\left(\frac{pm}{M}+\frac{qn}{N}\right)} \quad \text{and} \quad I_{mn} = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \hat{I}_{pq} e^{j2\pi\left(\frac{pm}{M}+\frac{qn}{N}\right)}.$$

How do we interpret the DFT of a 2D image? In the frequency domain, low frequencies are found near the center of the spectrum after `fftshift`. Recall that these frequencies correspond to smooth variations or large-scale structures in the image. As we move away from the center towards the edges, high-frequency components can be found. Recall that these frequencies correspond to rapid changes or fine details in the image.

We also recall that for real images, the DFT is symmetric around the axes, so we can look at a portion of the transform for analysis, though usually the entire DFT is visualized.

In Figure 16, we showcase examples of 2D images and their corresponding DFT, obtained by applying a FFT, followed by the `fftshift` function, and then plotting the magnitudes using their absolute values. The process is implemented in the Google Colab notebook. The top image contains frequency components solely along the x-axis, as it was generated using the periodic function $f(x, y) = \sin(x)$ over a discrete domain. Consequently, its DFT reveals discrete sinusoidal components only along the x-axis, as expected. These are lower-frequency components since the image lacks fine details corresponding to higher frequencies. The second image similarly contains frequency components only along the y-axis, generated using the function $f(x, y) = \sin(y)$. The third image exhibits a diagonal pattern, indicating the presence of both vertical and horizontal frequency components in its spectrum. This image was generated using the function $\sin(x + y)$. Finally, the bottom image is a combination of the first and third images, represented by $\sin(x) + \sin(x + y)$, and its frequency spectrum is accordingly a sum of the corresponding spectra.

In Figure 17, we present two examples of real 2D images and their corresponding DFT (both magnitude and phase plots). The DFTs were computed by applying an FFT followed by the `fftshift` function. The images were reconstructed using the `fftshift` function. The reconstructed images were obtained by applying the `ifftshift` function, followed by the IFFT algorithm. This process is demonstrated in the linked Google Colab notebook. TThe magnitudes are simply the absolute values of each pixel in the DFT matrix, while the phases are the arguments of each pixel (represented as complex numbers). For RGB images, the FFT and IFFT were applied separately to each channel. In the magnitude and phase plots for the basketball and soccer images, the FFT and IFFT were applied to the grayscale versions for clearer illustration. Additionally, the magnitude plot is shown on a $\log_{10}$ scale for better visibility. The bottom portion of Figure 17 we highlights the importance of retaining both magnitude and phase information (captured in $\hat{s}$ or the coefficients C_k), as swapping the phases between images leads to incorrect reconstructions. The magnitude controls the strength of different frequency components, while the phase governs the spatial structure. To switch phases, we performed an element-wise multiplication of magnitudes of one of the images with $\exp(j \cdot \arg(\text{DFT}))$, where the DFT from the other image was used. It is easy to prove that this creates a combined matrix with the magnitude of the first DFT and the phase of the second DFT.

Figure 16: The left column displays low-frequency images with various orientations, while the right column shows the magnitudes of their corresponding DFT.

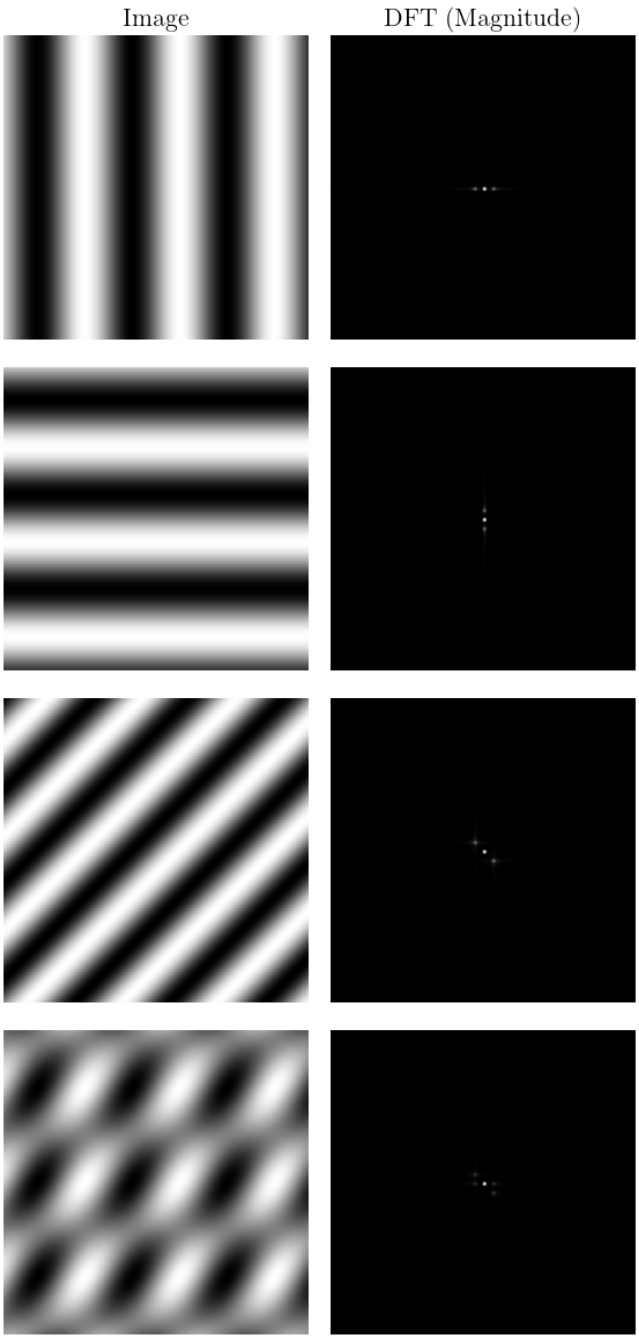


Figure 17: The left column displays low-frequency images with various orientations, while the right column shows the magnitudes of their corresponding DFT.