Computer Vision:
Algorithms and Deep Learning

Eyal Gur

November 3, 2024

Contents

Part I

Classical Computer Vision

Chapter 1

Image Filtering

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.

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.

Run in Colab Figure 1.1.1 illustrates all of the point operations discussed above. Of course, there are many other types of point processing operations, but they all follow similar principles.

Figure 1.1.1: 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 $\text{ONES}\,(n, n)\,/n^2$, and the strides are 1.

Run in Colab

Figure 1.2.1 displays the result of applying a box filter. 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.

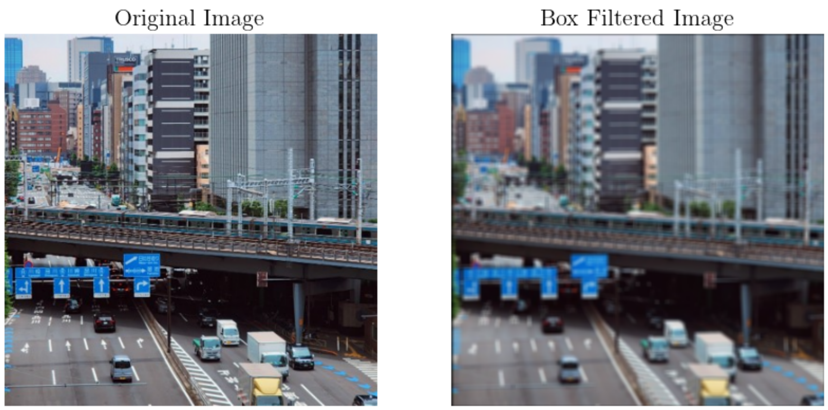
1.2.2 Separable Filters

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 $\mathbf{1}_n \mathbf{1}_n^T$.

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 $(k_H + k_W)\,HWC$.

Figure 1.2.1: Box filtering by applying a 5×5 kernel matrix.



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:

$$\mathbf{G}(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) = \mathbf{G}^1(x) \cdot \mathbf{G}^1(y), \tag{1.2.1}$$

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 $\mathbf{G} = \mathbf{G}^1 (\mathbf{G}^1)^T$. 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.

Run in Colab

Figure 1.2.2 shows the result of applying a Gaussian filter. 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 1.2.2: Gaussian filtering by applying a 5×5 kernel matrix, with two different standard deviations determining the level of blurring.

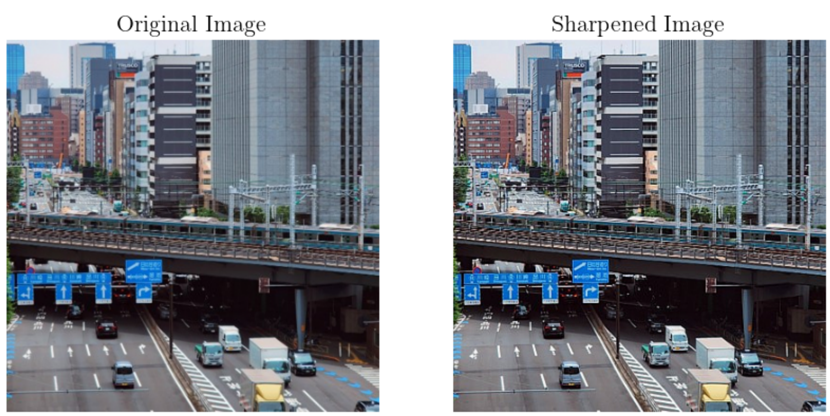
Sharpening Filter. One example of such a filter in the 3×3 case is constructed as follows:

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

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.

Run in Colab Figure 1.2.3 displays the result of applying a 3×3 sharpening filter, and it utilizes the `cv2` package to perform the convolution using the `cv2.filter2D` function (we can also apply two 1D convolutions as discussed above).

Figure 1.2.3: Sharpening filter by applying a 3×3 kernel matrix.



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:

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

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 $f : \mathbb{R} \to \mathbb{R}$ 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:

$$\frac{\partial \mathbf{v}}{\partial x}(x) = \frac{\mathbf{v}(x+h) - \mathbf{v}(x)}{h}, \tag{1.2.2}$$

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 (1.2.2) 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:

$$\frac{\partial \mathbf{v}}{\partial x} = (0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0) = \mathbf{k} * \mathbf{v}. \tag{1.2.3}$$

Notice that $\mathbf{k} = [1, -1]$ represents the coefficient of $\mathbf{v}(x)$ and $\mathbf{v}(x+1)$ in the approximation (1.2.3), 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 (1.2.2):

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

and plugging $h = -1$.

We can derive other discrete differentiation kernels using the same methodology. For instance, consider the alternative definition (1.2.2):

$$\frac{\partial \mathbf{v}}{\partial x}(x) = \frac{\mathbf{v}(x+0.5h) - \mathbf{v}(x-0.5h)}{h}, \tag{1.2.4}$$

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 1.3.1 shows the result of applying the Sobel and Prewitt filters for image edge detection. 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)

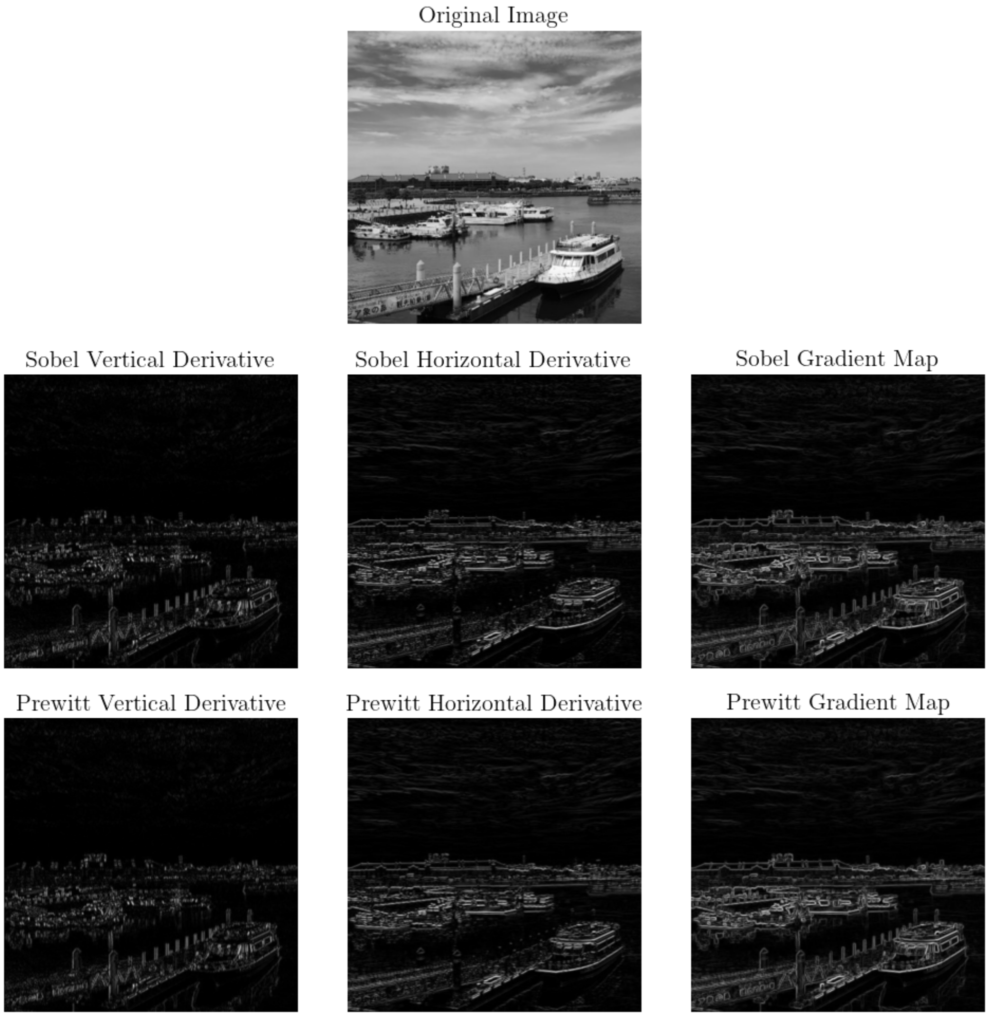
A problem occurs when attempting to detect edges in a noisy image. In this scenario, the changes in intensity might not indicate actual edges but rather discontinuities caused by the noise. Consequently, plotting the gradient map could highlight unnecessary noise that does not necessarily correspond to true edges in the image. For instance, see Figure 1.3.2 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 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:

$$\frac{\partial (\mathbf{G} * \mathbf{I})}{\partial x} \approx \mathbf{K}_x * (\mathbf{G} * \mathbf{I}) = (\mathbf{K}_x * \mathbf{G}) * \mathbf{I} \approx \frac{\partial \mathbf{G}}{\partial x} * \mathbf{I}, \tag{1.3.1}$$

Figure 1.3.1: 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.



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, following (1.2.1), the 2D DoGs are

$$\frac{\partial\mathbf{G}}{\partial x}(x,y) = -\frac{x}{\sigma^2}\mathbf{G}(x,y) \quad \text{and} \quad \frac{\partial\mathbf{G}}{\partial y}(x,y) = -\frac{y}{\sigma^2}\mathbf{G}(x,y). \tag{1.3.2}$$

Recall that we began this discussion with differentiation kernels $\mathbf{K}_x$ and $\mathbf{K}_y$ in (1.3.1). 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 1.3.2: Sobel gradient generated by a noisy input image.

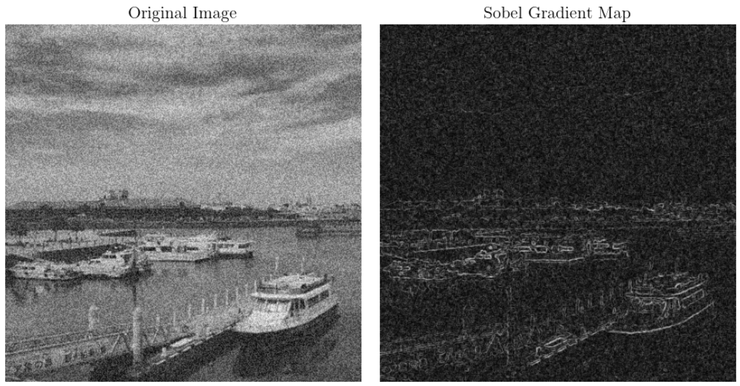
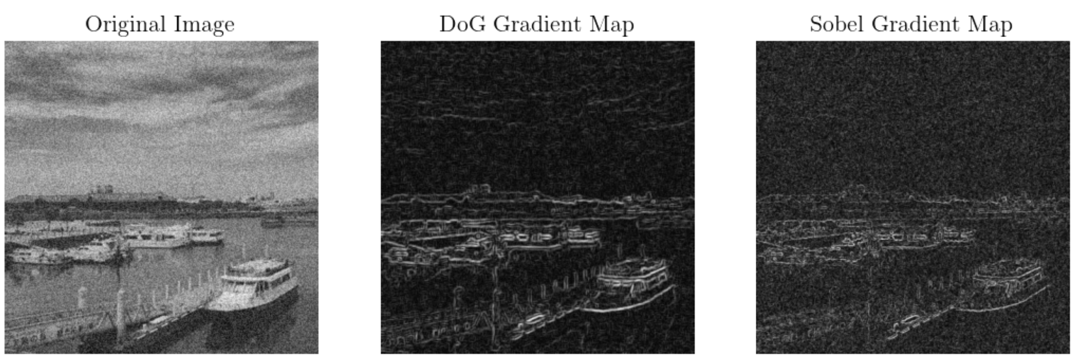


Figure 1.3.3 displays the gradient map for edge detection of a noisy input image, generated using the DoG filter and the Sobel filter for comparison. It is evident that the DoG filter yields significantly better results. The DoG filter is implemented by performing two 1D convolutions in both the x and 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 (1.3.2) as a column vector, followed by convolving with the 1D Gaussian kernel (1.2.1) 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 1.3.3: 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:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}, \tag{1.3.3}$$

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 (1.2.4), we have:

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

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*:

$$\frac{\partial^2 \mathbf{I}}{\partial x^2}(x,y) + \frac{\partial^2 \mathbf{I}}{\partial y^2}(x,y) \tag{1.3.4}$$

Following (1.3.3), a second-order approximation kernel matrix corresponding to (1.3.4) for a 2D image is:

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

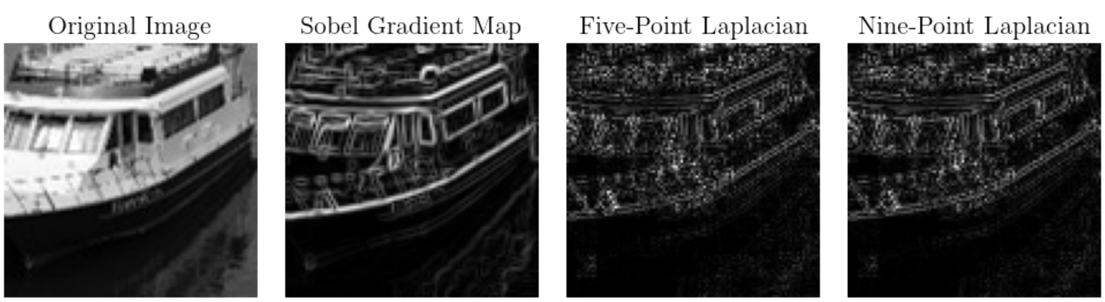
known as the *Five-Point Stencil* filter, as it consists of a grid including the central point and its four neighbors. An alternative second-order approximation of the sum in (1.3.4) involves using all neighboring points, hence it is called the *Nine-Point Stencil* filter, defined as:

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

Run in Colab

Figure 1.3.4 shows the two Laplacian filters alongside the gradient map of the Sobel filter. 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).

Figure 1.3.4: Two variations of the second-order Laplacian filter, compared with the first-order Sobel filter for edge detection.



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.

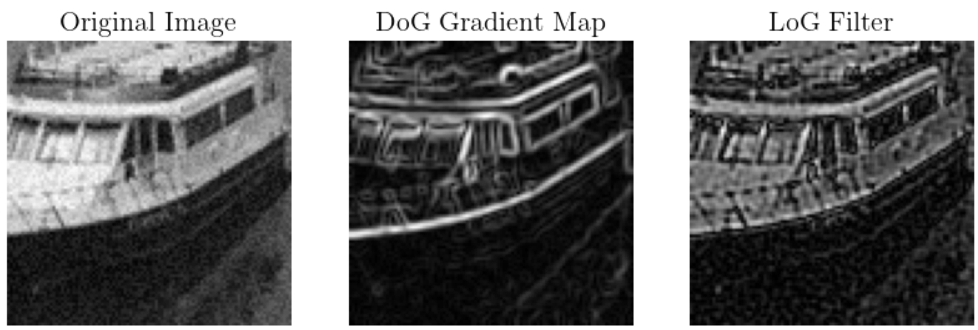
The second-order derivative of the Gaussian filter is known as the Laplacian of Gaussian (LoG), and it is defined for a 2D Gaussian kernel as follows:

$$\frac{\partial^2 \mathbf{G}}{\partial x^2}(x, y) + \frac{\partial^2 \mathbf{G}}{\partial y^2}(x, y) = \frac{x^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y) + \frac{y^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y) = \frac{x^2 + y^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y)\,. \tag{1.3.5}$$

Run in Colab

Figure 1.3.5 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 performs the LoG filtering using two 1D convolutions, as discussed in the DoG implementation in Section 1.3.2.

Figure 1.3.5: The absolute values of the LoG filter (zero crossings) and the gradient map produced by the DoG filter when applied to a noisy image.



Chapter 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.

Run in Colab

Figure 2.0.1 (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.

Figure 2.0.1: 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.

2.1 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.

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.

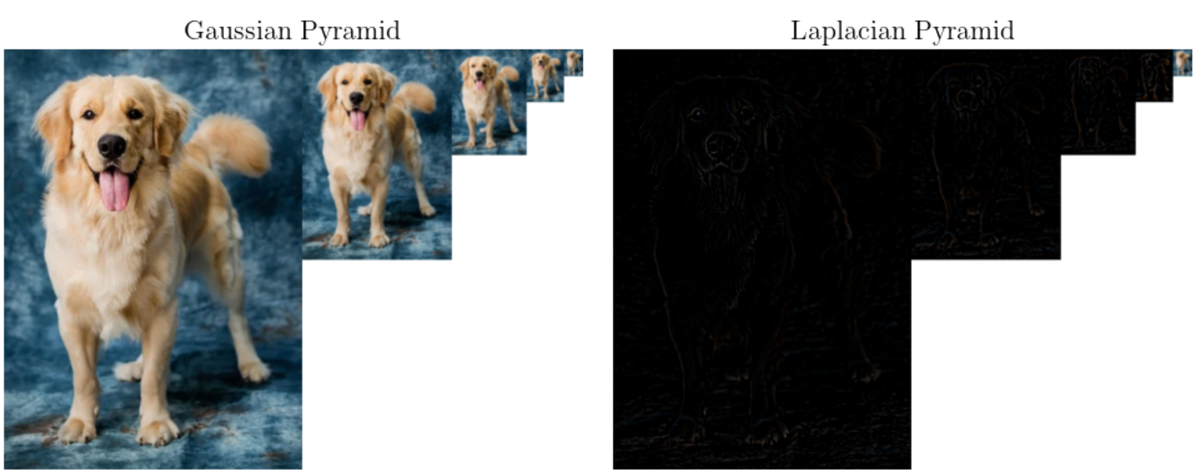
   An example of a bandpass pyramid is the *Laplacian pyramid*, which uses the Gaussian pyramid as its underlying lowpass pyramid. To calculate pixel-wise differences between levels, a common approach is to use *nearest-neighbor interpolation*. In this method, when new pixel rows and columns are added between 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.

2.1.1 The Gaussian and Laplacian Pyramids

Run in Colab

Figure 2.1.1 displays the Gaussian and Laplacian pyramids. In the Gaussian pyramid, the image is progressively downsampled and blurred at each step, moving from left to right in Figure 2.1.1. 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 2.1.1. 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 2.1.1: Gaussian and Laplacian pyramids.



Interestingly, 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:

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

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

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

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 (1.3.5), 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.

2.1.2 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 2.1.2 illustrates the process of reconstructing the original image from its smallest (most downsampled) version using a Laplacian pyramid. 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.

Figure 2.1.2: 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.

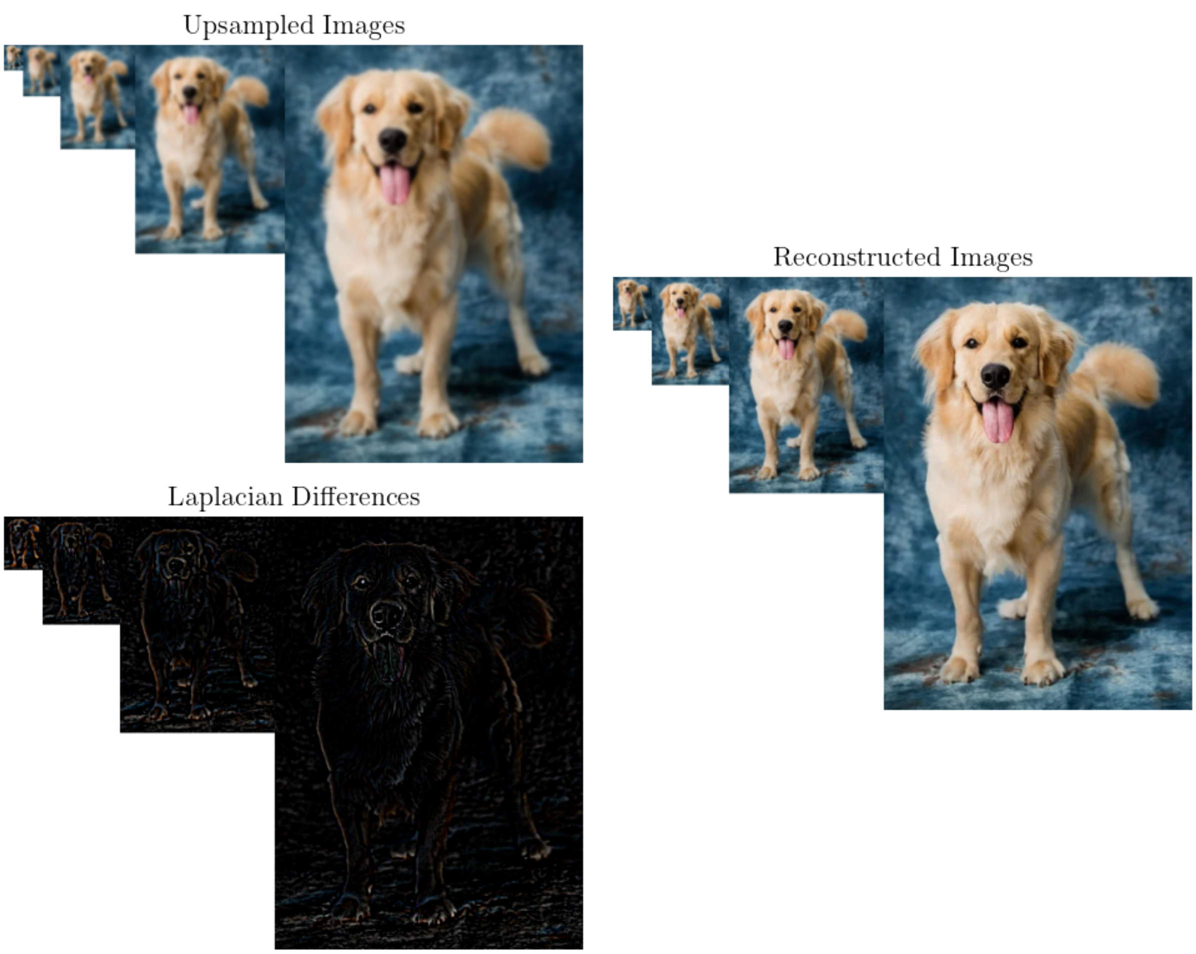


Image Blending. Laplacian pyramids can be used for image blending because they allow for integration of two images by decomposing them into different frequency components. In this process, corresponding levels from the two Laplacian pyramids (on of each image) are combined, often by concatenating or 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.

Run in Colab

Figure 2.1.3 illustrates the blending process of two images. The two original images to be blended are displayed at the top of Figure 2.1.3, together with their naive blending. The middle section of Figure 2.1.3 shows the blended Laplacian differences, where the images are stretched for visualization purposes. Initially, 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 2.1.3 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.

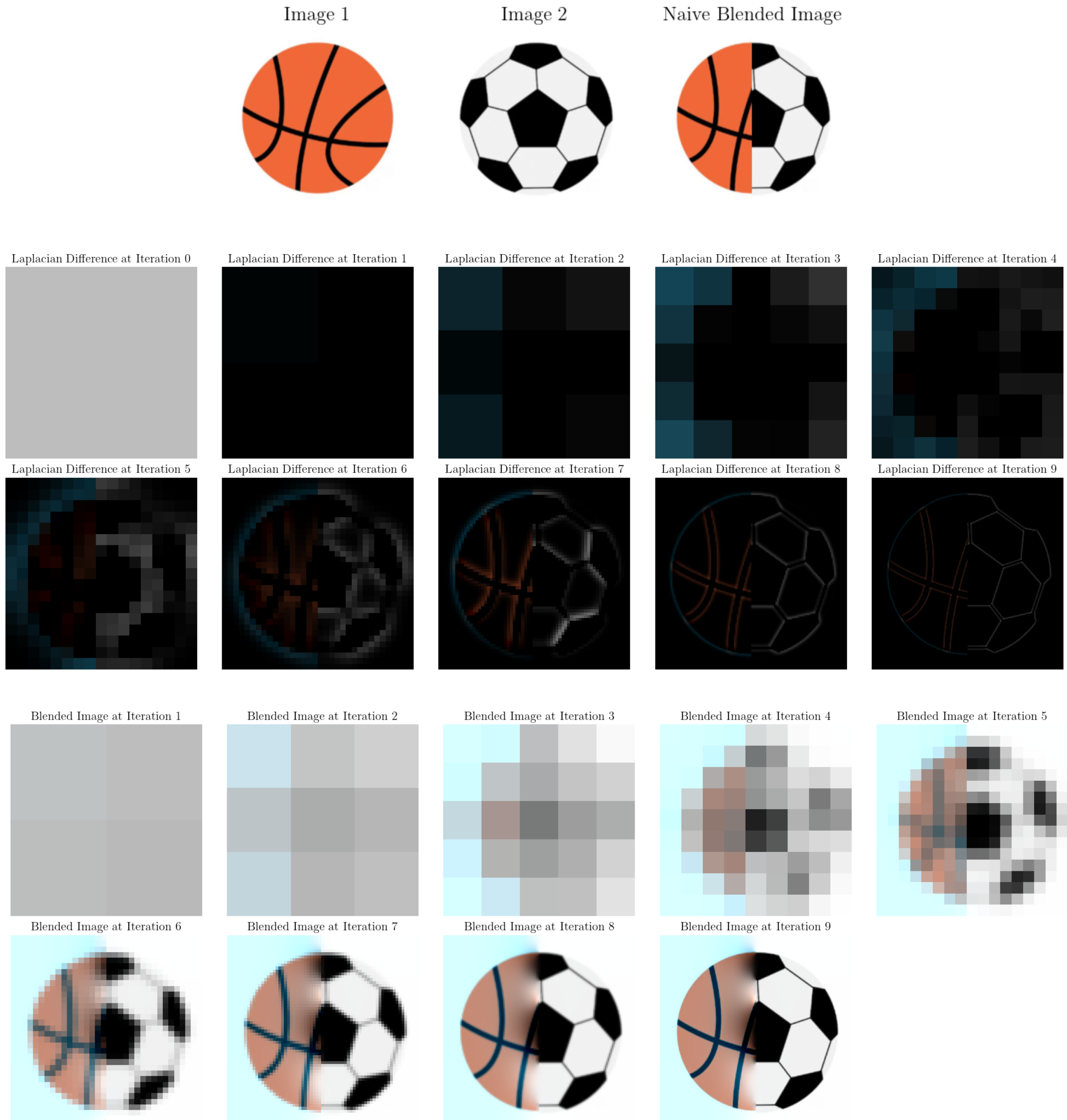
It is important to note that as we go deeper with downsampling, we progressively lose higher-frequency details, making the reconstruction process more challenging. In the example shown in Figure 2.1.3, we see that the downsampling went too deep (resulting in a single pixel at the last level), which led 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.

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 simi-

Figure 2.1.3: 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.



larity 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:

$$s(t) = D_0 + \sum_{k=1}^{\infty} D_k \cos\left(\frac{2\pi kt}{T} - \phi_k\right) = D_0 + \sum_{k=1}^{\infty} D_k \sin\left(\frac{2\pi kt}{T} - \phi_k + \frac{\pi}{2}\right), \tag{2.2.1}$$

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

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

Hence, up to amplitude scaling and phase shifting (that is, setting A and ϕ above), equation (2.2.1) 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 $\cos(2\pi kt/T)$ and the signal. Following the definition of cross-correlation, we define $\tilde{X}_k(\phi)$ as the cross-correlation between the signal s and $\cos(2\pi kt/T)$. That is:

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

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:

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

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

$$\cos(\alpha - \beta) = \cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\beta), \tag{2.2.2}$$

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:

$$A_k \cos(\phi) - B_k \sin(\phi) = 0, \tag{2.2.3}$$

where we define

$$\begin{aligned} A_0 &= \frac{1}{T} \int_T s(t)\, dt, \\ A_k &= \frac{2}{T} \int_T s(t) \cos\left(\frac{2\pi kt}{T}\right) dt, \quad k \geq 1, \\ B_k &= \frac{2}{T} \int_T s(t) \sin\left(\frac{2\pi kt}{T}\right) dt, \quad k \geq 1. \end{aligned} \tag{2.2.4}$$

These last equations are known as the *Fourier series analysis*. So, following (2.2.4) we get that the optimal phase is given by:

$$\phi_k = \operatorname{atan2}(B_k, A_k),$$

where

$$\operatorname{atan2}(y, x) \equiv \begin{cases} \arctan(y/x), & x > 0, \\ \arctan(y/x) + \pi, & x < 0, y \geq 0 \\ \arctan(y/x) - \pi, & x < 0, y < 0 \\ \operatorname{sign}(y) \cdot \pi/2, & x = 0, y \neq 0 \\ \text{undefined}, & x = y = 0. \end{cases} \tag{2.2.5}$$

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

$$D_0 = A_0 \quad \text{and} \quad D_k = \cos(\phi_k) A_k + \sin(\phi_k) B_k = \sqrt{A_k^2 + B_k^2}. \tag{2.2.6}$$

Finally, we point out that the coefficient $D_0 = A_0$ (the 0 frequency) as defined in (2.2.4), is the average value of the signal s over the time interval T.

Sine-Cosine Form. From (2.2.2) we get that

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

Plugging the above into (2.2.6) we get that

$$A_k = D_k \cos(\phi_k) \quad \text{and} \quad B_k = D_k \sin(\phi_k). \tag{2.2.7}$$

Hence, using the trigonometric identity (2.2.2) together with (2.2.7), we get that the series (2.2.1) can be equivalently expressed in the *sine-cosine form*:

$$s(t) = A_0 + \sum_{k=1}^{\infty} \left(A_k \cos\left(\frac{2\pi kt}{T}\right) + B_k \sin\left(\frac{2\pi kt}{T}\right)\right). \tag{2.2.8}$$

Exponential Form. Using Euler's formula

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

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 (2.2.1), we equivalently get the Fourier series in its *exponential form*:

$$s(t) = \sum_{k=-\infty}^{\infty} C_k e^{\frac{j2\pi kt}{T}}, \tag{2.2.9}$$

where, following (2.2.1), (2.2.7) and (2.2.8) we get that:

$$C_0 = D_0 = A_0, \quad C_{k\geq 1} = \frac{D_k e^{-j\phi_k}}{2} = \frac{(A_k - jB_k)}{2} \quad \text{and} \quad C_{k\leq -1} = C^*_{-k}. \tag{2.2.10}$$

Additionally, note that now it follows that:

$$A_k = C_k + C_{-k} \quad \text{and} \quad B_k = j(C_k - C_{-k}) \tag{2.2.11}$$

Following (2.2.4), (2.2.9) and (2.2.10), we can write the Fourier series analysis in exponential form:

$$C_k = \frac{1}{T}\int_T s(t)\, e^{\frac{-j2\pi kt}{T}}\, dt, \quad \forall k \in \mathbb{Z}. \tag{2.2.12}$$

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

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

From the Fourier series analysis we get:

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

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

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

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

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

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 $\operatorname{Re}(C_k)$ and $\operatorname{Im}(C_k)$), which contain both amplitude and phase information. In this context, recall that the amplitude of the k-the harmonic is (see (2.2.6) and (2.2.10))

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

Additionally, recall that for any $k \geq 1$ it holds that (see (2.2.10))

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

hence it follows for $k \geq 1$ that (see (2.2.6) and (2.2.11))

$$\begin{aligned}\mathrm{Phase}_k = \phi_k = \mathrm{atan2}\left(B_k, A_k\right) &= \mathrm{atan2}\left(j\left(C_k - C_{-k}\right), C_k + C_{-k}\right)\\ &= \mathrm{atan2}\left(-2\cdot\mathrm{Im}\left(C_k\right), 2\cdot\mathrm{Re}\left(C_k\right)\right)\\ &= \mathrm{atan2}\left(2\cdot\mathrm{Im}\left(C_{-k}\right), 2\cdot\mathrm{Re}\left(C_{-k}\right)\right) \equiv \arg\left(2C_{-k}\right).\end{aligned} \tag{2.2.13}$$

Run in Colab

Figure 2.2.1 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 (2.2.13)).

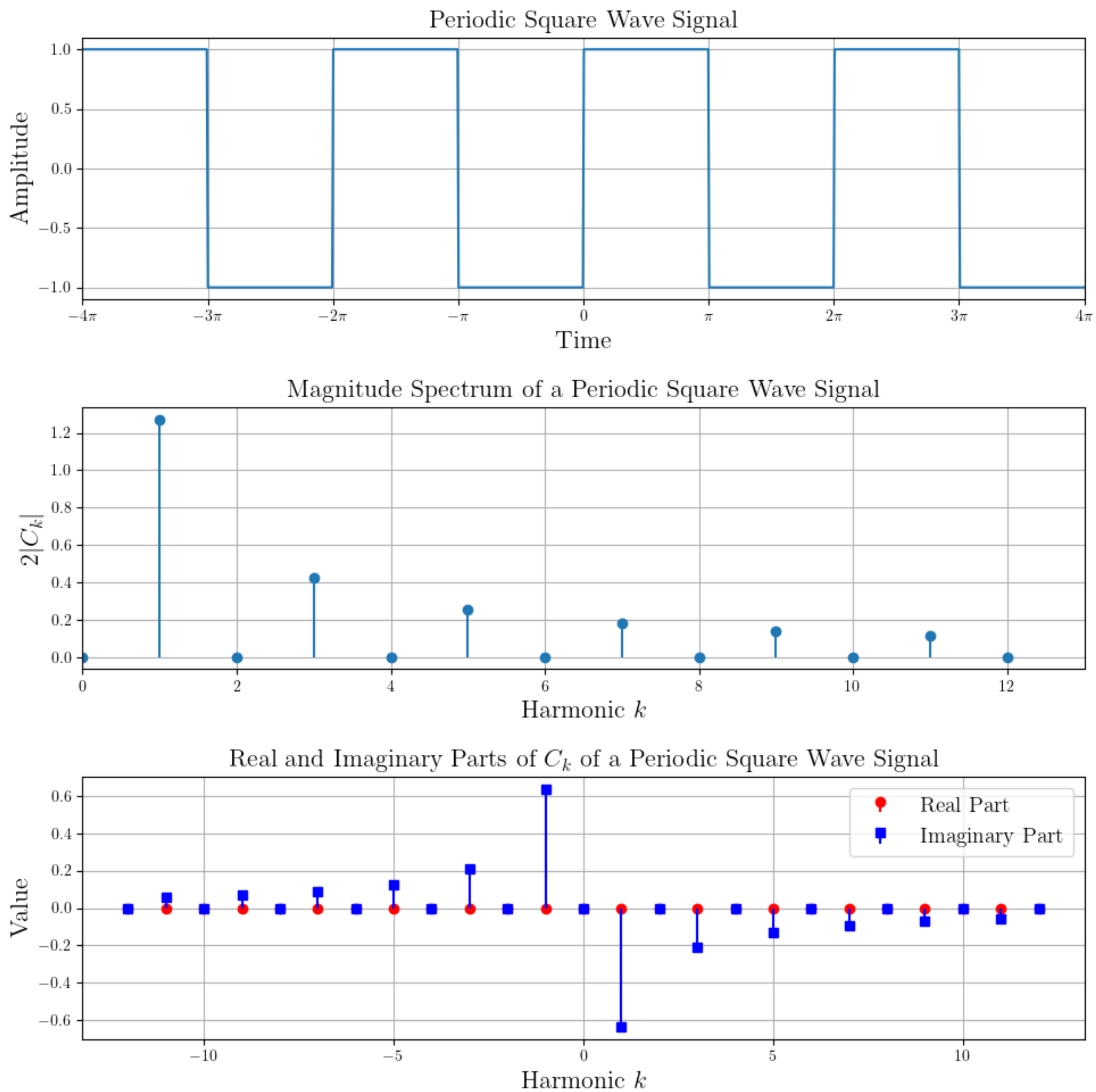
When interpreting the frequency domain graph, a magnitude (amplitude) value at a particular harmonic indicates the contribution of that frequency component to the overall signal. For example, if the graph shows a significant value at the third harmonic, it means that the signal has a strong component at three times the fundamental frequency $1/T$ (middle plot in Figure 2.2.1). The real and imaginary parts describe how it is phased relative to the other components (bottom plot in Figure 2.2.1).

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

Figure 2.2.1: 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.



(2.2.9), is given by the DFS:

$$s\left(nI\right) = \sum_{k=-\infty}^{\infty} C_k e^{\frac{j2\pi knI}{T}}, \quad \forall n \in \mathbb{Z}, \tag{2.2.14}$$

where, analog to equation (2.2.12), the coefficients C_k are defined using the analysis formula:

$$C_k = \frac{1}{T}\sum_{n=0}^{N} s\left(nI\right)e^{\frac{-j2\pi knI}{T}}, \quad \forall k \in \mathbb{Z}, \tag{2.2.15}$$

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 (2.2.1), 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 (2.2.12) while letting $k/T \to \xi$ for some frequency $\xi \in \mathbb{R}$, leading to the *Fourier transform* formula:

$$\hat{s}\left(\xi\right) = \int_{-\infty}^{\infty} s\left(t\right)e^{-j2\pi\xi t}dt. \tag{2.3.1}$$

The corresponding decomposition, analogous to (2.2.1) and (2.2.9), can be proved to be given by the *inverse Fourier transform* formula in its exponential form:

$$s\left(t\right) = \int_{-\infty}^{\infty} \hat{s}\left(\xi\right)e^{j2\pi\xi t}d\xi, \quad \forall t \in \mathbb{R}. \tag{2.3.2}$$

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 $s\left(t\right) \overset{\mathcal{F}}{\longleftrightarrow} \hat{s}\left(\xi\right)$. In the n-dimensional Euclidean space the pair takes the form

$$\hat{s}\left(\boldsymbol{\xi}\right) = \int_{\mathbb{R}^n} s\left(\mathbf{t}\right)e^{-j2\pi\boldsymbol{\xi}^T\mathbf{t}}d\mathbf{t} \quad \text{and} \quad s\left(\mathbf{t}\right) = \int_{\mathbb{R}^n} \hat{s}\left(\boldsymbol{\xi}\right)e^{j2\pi\boldsymbol{\xi}^T\mathbf{t}}d\boldsymbol{\xi}. \tag{2.3.3}$$

The intuitive interpretation of the Fourier transform can be understood through the effect of multiplying the signal s by the complex exponential $\exp\left(-j2\pi\xi t\right)$ (see (2.3.1)). 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

$$s\left(t\right)=\cos\left(2\pi 3t\right)e^{-\pi t^2}=\frac{1}{2}e^{-\pi t^2-j2\pi 3t}+\frac{1}{2}e^{-\pi t^2+j2\pi 3t}. \tag{2.3.4}$$

When multiplying the signal by $\exp\left(-j2\pi\xi t\right)$ (see (2.3.1)), we simply add $-j2\pi\xi t$ to each exponent (shifting) and get:

$$\begin{aligned} s\left(t\right)\cdot e^{-j2\pi\xi t} &= \cos\left(2\pi 3t\right)e^{-\pi t^2}=\frac{1}{2}e^{-\pi t^2-j2\pi(3+\xi)t}+\frac{1}{2}e^{-\pi t^2+j2\pi(3-\xi)t} \\ &= e^{-\pi t}\cos\left(6\pi t\right)\cos\left(2\pi\xi t\right)+je^{-\pi t^2}\cos\left(6\pi t\right)\sin\left(2\pi\xi t\right). \end{aligned} \tag{2.3.5}$$

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.

Run in Colab

In Figure 2.3.1, the top left plot displays the original signal (2.3.4). The top right and bottom left plots show the result of multiplying this signal by $\exp\left(-j2\pi\xi t\right)$ as in (2.3.5), 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.

2.3.1 Properties of the Fourier Transform

Let $s\left(t\right)\xleftrightarrow{\mathcal{F}}\hat{s}\left(\xi\right)$. Then,

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

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

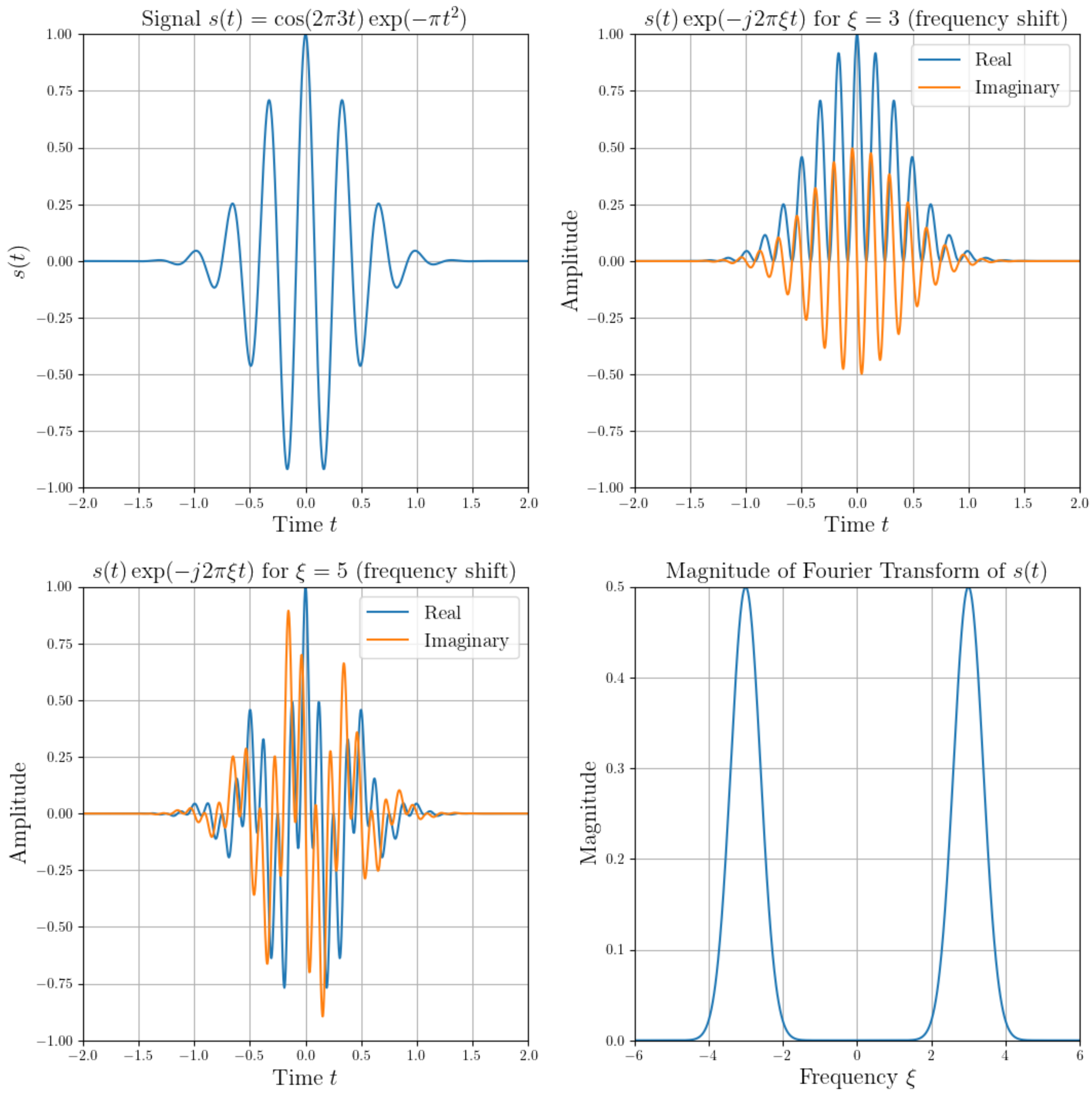
2. Time shifting: for $t_0\in\mathbb{R}$

$$s\left(t-t_0\right)\xleftrightarrow{\mathcal{F}}\hat{s}\left(\xi\right)e^{-j2\pi\xi t_0}.$$

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

$$s\left(t\right)e^{j2\pi\xi_0 t}\xleftrightarrow{\mathcal{F}}\hat{s}\left(\xi-\xi_0\right).$$

Figure 2.3.1: The signal described in (2.3.4) 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 $\xi = 5$ (refer to (2.3.5)). The magnitude (absolute value) of the Fourier transform of the signal is depicted in the bottom right plot.



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 $u(t) \xleftrightarrow{\mathcal{F}} \hat{u}(\xi)$

$$(s * u)(t) \xleftrightarrow{\mathcal{F}} \hat{s}(\xi)\,\hat{u}(\xi) \quad \text{and} \quad s(t)\,u(t) \xleftrightarrow{\mathcal{F}} (\hat{s} * \hat{u})(\xi)\,.$$

Analogously, for cross-correlation we have

$$(s \star u)(t) \xleftrightarrow{\mathcal{F}} \hat{s}(\xi)^{*}\,\hat{u}(\xi)\,.$$

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 (2.3.1), is:

$$C_k = \sum_{n=0}^{N-1} s_n e^{\frac{-j2\pi kn}{N}}, \quad k = 0, 1, \ldots, N-1, \tag{2.3.6}$$

and the *inverse discrete Fourier transform* (IDFT), analog to equation (2.3.2), is:

$$s_n = \frac{1}{N} \sum_{k=0}^{N-1} C_k e^{\frac{j2\pi kn}{N}}, \quad n = 0, 1, \ldots, N-1. \tag{2.3.7}$$

The proofs that the DFT and the IDFT are special cases of the continuous Fourier transform equations (2.3.1) and (2.3.2) are beyond the scope of this discussion. It is important to note that, similar to (2.2.14) and (2.2.15), 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 (2.2.14) and (2.2.15), apply to discrete and periodic (hence not finite) signals, while the DFT and IDFT, as developed in (2.3.6) and (2.3.7), 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 (2.3.3)):

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

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 (2.3.6) and (2.3.7) from 1D to 2D as follows:

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

2.4.1 DFT of 2D Images

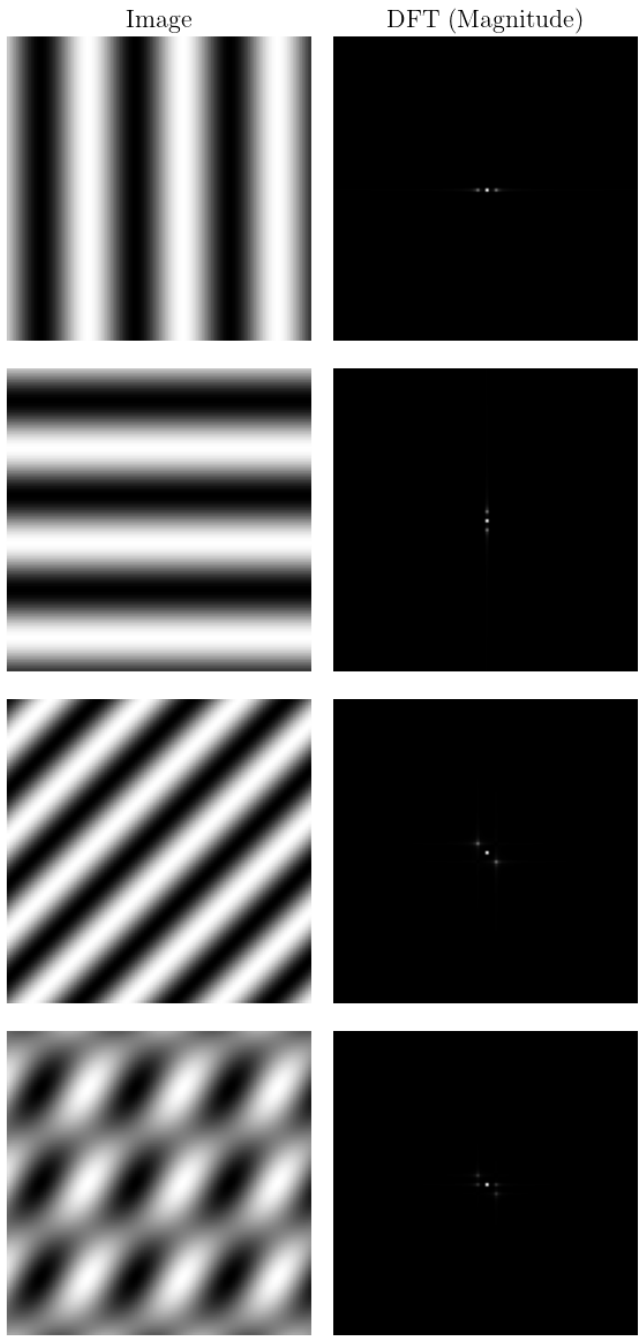
How do we interpret the DFT of a 2D image? In the frequency domain, low frequencies are found near the center of the spectrum after `fftshift`. Recall that these frequencies correspond to smooth variations or large-scale structures in the image. As we move away from the center towards the edges, high-frequency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

Run in Colab

In Figure 2.4.1, 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 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.

Figure 2.4.1: The left column displays low-frequency images with various orientations, while the right column shows the magnitudes of their corresponding DFT.



Run in Colab

In Figure 2.4.2, 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. The 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 2.4.2 we highlight 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 $\exp\left(j \cdot \arg\left(\mathrm{DFT}\right)\right)$, 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.

2.4.2 Frequency-Domain Filtering

Recall the convolution property of the Fourier transform

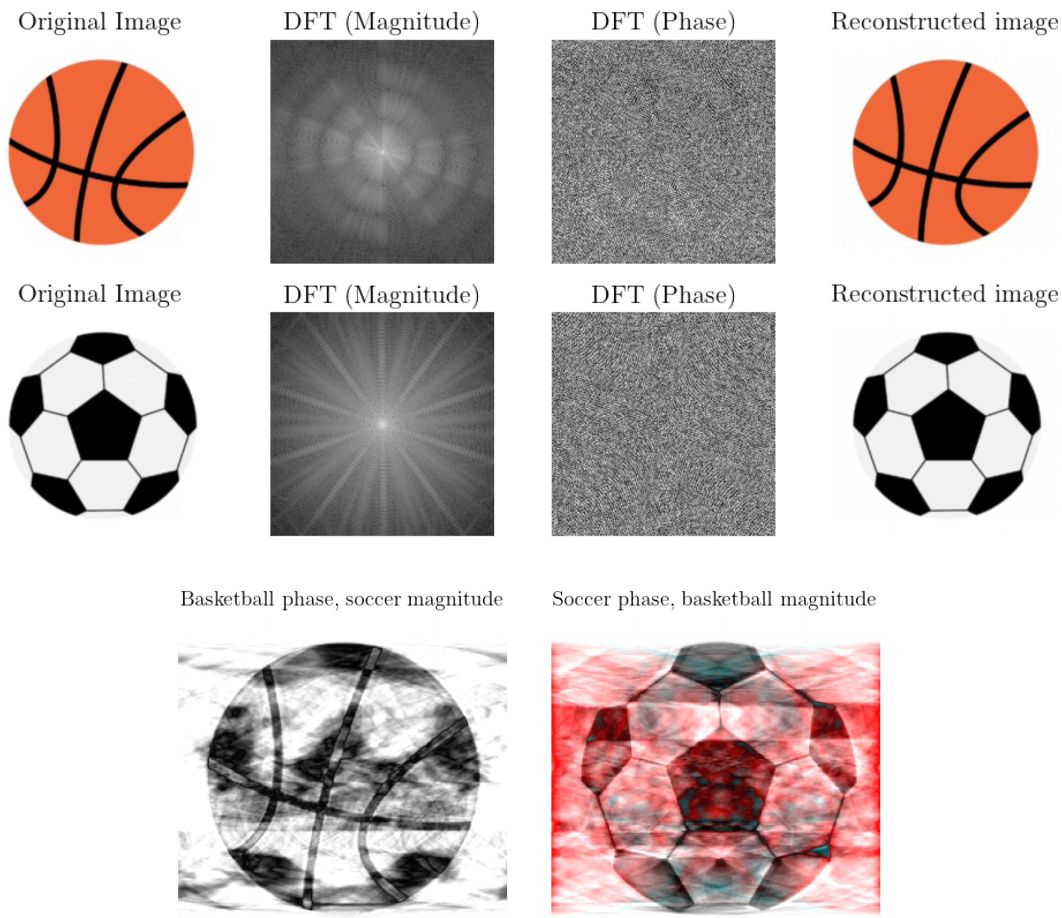
$$\left(s * u\right)(t) \xleftrightarrow{\mathcal{F}} \hat{s}\left(\xi\right)\hat{u}\left(\xi\right).$$

This means that the Fourier transform of the convolution of two functions is the product of their Fourier transforms (equivalently, the inverse Fourier transform of the product of two Fourier transforms is the convolution of the two inverse Fourier transforms). Hence, convolution in spatial domain is equivalent to multiplication in frequency domain. This means that we can interpret and implement all linear shift-invariant filtering as multiplication in frequency domain.

This approach allows us to perform image filtering in the frequency domain instead of the typical spatial domain method. First, we compute the Fourier transform of both the image and the filter. Note that the filter is padded with zeros to match the size of the image. The padding is done such that the original filter is positioned at the center of the padded array. It can be shown that, following the convolution theorem above, element-wise multiplication of this padded filter with the Fourier-transformed image will yield the same result as convolving the original image with the smaller filter using a sliding window. If the filter is padded with the non-zero values positioned differently, the resulting filtered image will appear tiled according to that position. Finally, we apply the inverse Fourier transform to the filtered image.

The advantage of this method is that performing element-wise multiplication once requires fewer computational operations than repeatedly performing convolution with sliding windows. However, it is important to note that this approach still involves applying both the `fft` and `ifft` functions (which also require the `fftshift` and `ifftshift` functions according to the `numpy` implementation).

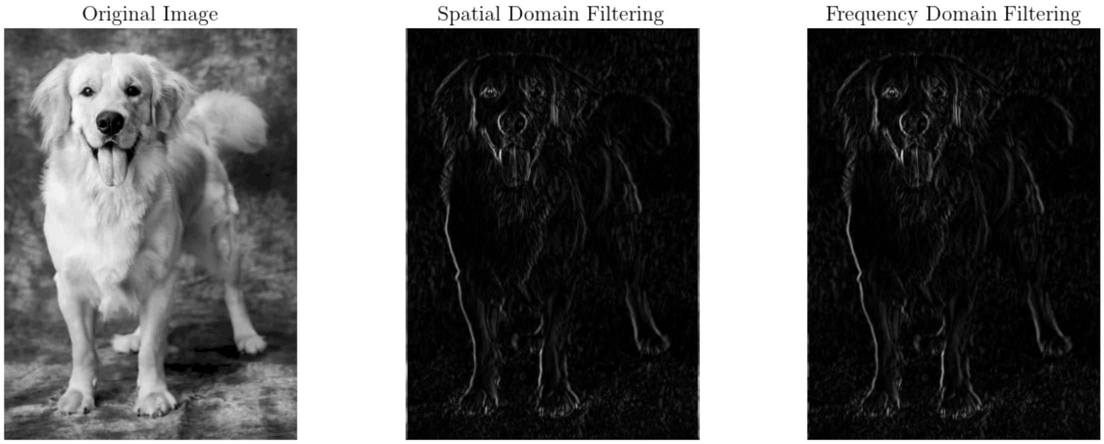
Figure 2.4.2: The left column displays low-frequency images with various orientations, while the right column shows the magnitudes of their corresponding DFT.



Run in Colab In Figure 2.4.3 below, we present the results of applying a 3×3 edge detection filter using both the convolution operation in the spatial domain and element-wise multiplication in the frequency domain. For the frequency domain approach, the filter is first padded to match the dimensions of the original image. We then apply `fft2` and `fftshift` to both the padded filter and the image, perform element-wise multiplication, and subsequently apply `ifft2` and `ifftshift` to obtain the filtered image. As expected, both methods produce the same edge-detected result.

Run in Colab With frequency-domain filtering, we can better understand why the Gaussian blur produces a smooth image, while the box blur filter tends to create sharper artifacts when zoomed in. In 2.4.4, we show the magnitude output of filtering an image in the frequency domain using both the Gaussian and box blurring kernels. Additionally, we plot the magnitudes of the Gaussian and box filters (after padding and applying `fft2` and `fftshift`). For the Gaussian blur, the magnitudes are concentrated around the center of the Fourier transform, indicating that it acts as a low-pass filter. That is, when applying the element-wise multiplication, the high-frequency components are essentially eliminated

Figure 2.4.3: Edge detection filter applied using convolution with a sliding window and using frequency domain filtering. The magnitudes of the resulting filtered images are displayed.



as they are not present in the Fourier transform of the Gaussian blur. In contrast, the box blur shows more pronounced magnitudes towards the edges of the Fourier transform, indicating that it behaves more like a high-pass filter – as the high-frequency components are nit eliminated when performing the element-wise multiplication, which explains the sharper artifacts observed in the blurred image.

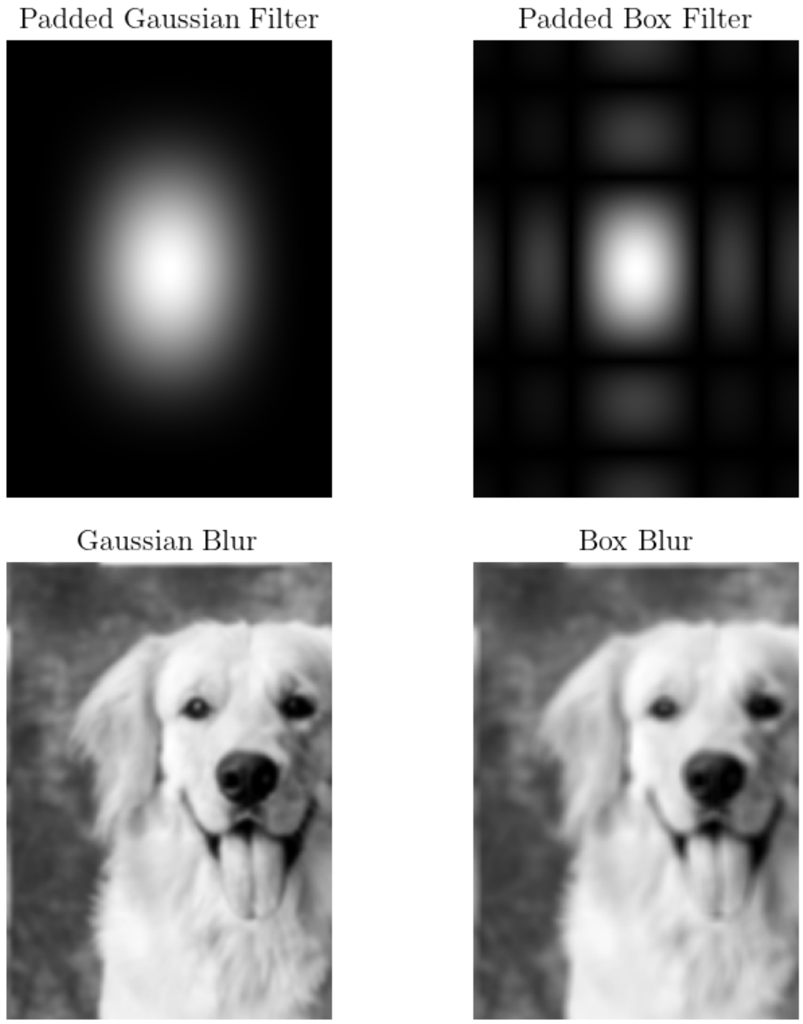
2.4.3 The Nyquist Frequency

The *Nyquist-Shannon sampling theorem* states that a continuous signal can be perfectly reconstructed from its discrete samples using linear interpolation if it is sampled at a rate that is at least twice the highest frequency present in the signal. This rate is known as the *Nyquist frequency*. In other words, if the highest frequency component of a signal is $f_{\max}$, the signal must be sampled at a frequency $f_s \geq 2f_{\max}$ to avoid aliasing. That is, aliasing does not occur if samples are taken at the Nyquist frequency or higher.

Recall that when constructing a Gaussian pyramid, an image is repeatedly smoothed (blurred) and then downsampled (reduced in size). The smoothing step is performed using a Gaussian filter, which acts as a low-pass filter to remove high-frequency components from the image, as explained in Figure 2.4.4 using the Fourier transform. The downsampling step involves reducing the image resolution by a factor of 2 (e.g., taking every second row and column). This means that the sampling rate is halved with each level of the pyramid.

Without smoothing, as shown in Figure 2.0.1, downsampling directly leads to aliasing because the high-frequency components of the image violate the Nyquist criterion. This means that these high-frequency details would overlap into lower frequencies, creating distortions. By applying the Gaussian filter (which removes high-frequency components), the image becomes band-limited, ensuring that the remaining frequencies are below the new Nyquist frequency after downsampling. This prevents aliasing and allows for the image to be downsampled without introducing distortions. Since each level was downsampled without aliasing, the reconstruction process is possible (using the corresponding Laplacian pyramid)

Figure 2.4.4: Applying a 5×5 Gaussian blur and box blur filters in the frequency domain. The magnitudes of the Fourier transform of the padded filter kernels are plotted at the top.



without introducing frequency artifacts, as shown in Figure 2.1.2, where the linear interpolation used was the nearest-neighbor interpolation.

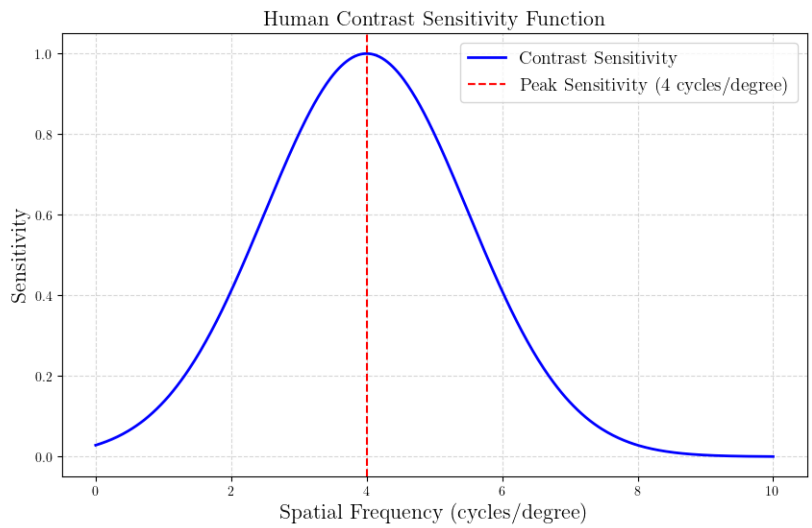
2.4.4 Frequency-Domain Filtering In Human Vision

The early stages of visual processing in the human brain involve breaking down an image into different spatial frequencies and orientations. The visual system essentially applies filters at different orientations (e.g., horizontal, vertical, diagonal) and scales (different levels of detail). This is akin to performing frequency-domain filtering, where the image is analyzed by decomposing it into components based on frequency and orientation.

Run in Colab Figure 2.4.5 illustrates the human contrast sensitivity function, which shows how the human visual system responds to different spatial frequencies. The x-axis represents the spatial frequency in cycles per degree, while the y-axis indicates the sensitivity of the human eye to contrast at each frequency.

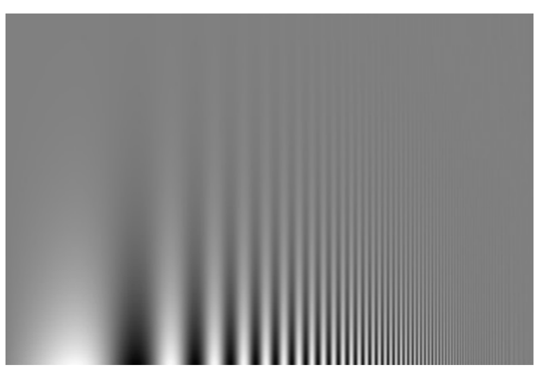
The peak sensitivity is centered around 4 cycles per degree, indicated by the red dashed line, showing that the human eye is most sensitive to mid-range frequencies. This sensitivity decreases for both lower and higher frequencies, meaning we perceive contrast less effectively at very low and very high frequencies. This highlights why mid-range frequencies are crucial for visual perception and why they dominate how we see and interpret visual scenes.

Figure 2.4.5: Human sensitivity function ans the peak spatial frequency.



We can also visualize this notion, as shown in Figure 2.4.6. In this figure, we can see that as the contrast decreases (higher along the y-axis, all frequencies are less visible. However, the mid-range frequencies are the most visible.

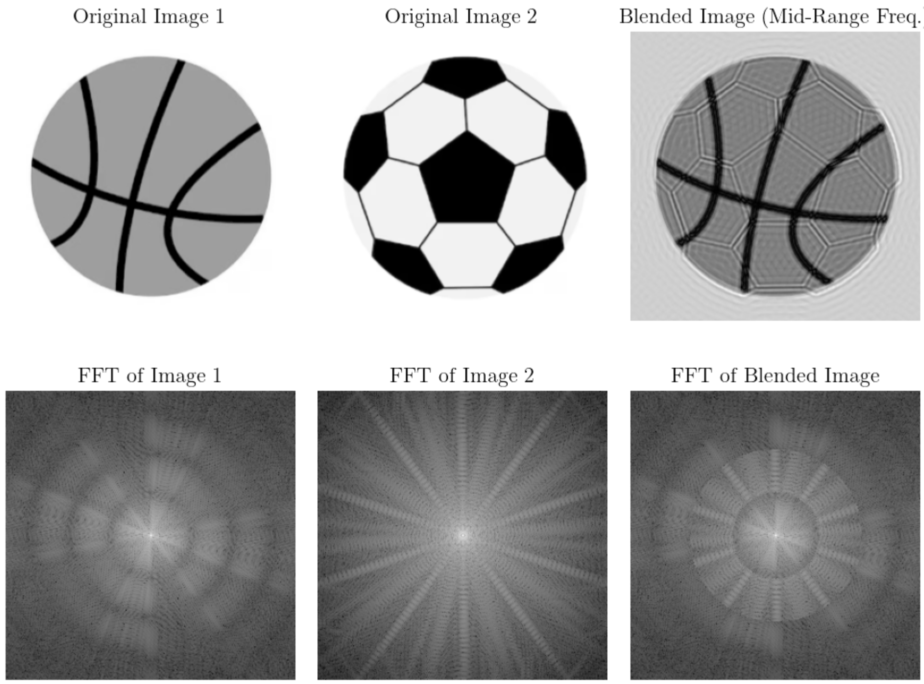
Figure 2.4.6: The x-axis are different frequencies, while along the y-axis the contrast decreases (changes in frequency are less visible).



Run in Colab

Finally, we can visualize the concept of mid-range frequencies through image blending. In Figure 2.4.7, we blend two images by replacing their mid-range frequencies, resulting in a blended image that retains the original structure of the first image but incorporates the mid-range frequencies from the second image. Specifically, we replaced frequencies with a distance between 40 and 80 from the center of the DFT. At the bottom, we display the magnitudes of the DFT (on a $\log_{10}$ scale) for the two original images and the blended image, where the replacement of the mid-range frequencies is clearly visible.

Figure 2.4.7: Image blending using by replacing mid-range frequencies of the DFTs.



Chapter 3

Hough Transform

When detecting edges or lines in images – using techniques such as the DoG, LoG, Sobel filter, etc. – we often obtain a scatter of points that do not perfectly align to form a line. This scattering can result from noise in the image or from edges that are difficult to detect. To address this, we can apply a line fitting technique that takes these scattered data points and fits a line that best represents the edge based on their distribution.

3.1 Linear Fitting

In the *data fitting* task we are given a dataset $\{\mathbf{x}^i\}_{i=1}^N \subset \mathbb{R}^{n+1}$, and we aim to find a *predicting function* that best describes an underlying relation between the coordinates of the data points:

$$0 \approx f\left(\mathbf{x}^i\right), \quad \forall i = 1, 2, \ldots, N, \tag{3.1.1}$$

where $f\colon \mathbb{R}^{n+1} \to \mathbb{R}$ is a function to be determined. In *line fitting* (or linear/vector/hyperplane fitting) we model the function f as a linear function.

Maximum-Likelihood Estimation and the Squared Error Function. Assume that the the function f is parameterized by θ. Following (3.1.1), we need to find such parameters (or coefficients) θ such that $f_\theta\left(\mathbf{x}^i\right)$ is as close to 0 as possible, for any data point $\mathbf{x}^i$. Treating $f_\theta\left(\mathbf{x}^i\right)$ as a random variable dependent on θ, we assume that it follows a Gaussian distribution. That is,

$$f_\theta\left(\mathbf{x}^i\right) \sim \text{Gauss}\left(0, \sigma^2\right), \quad \forall i = 1, 2, \ldots, N,$$

for some $\sigma > 0$. Using the maximum-likelihooed estimation approach to find the optimal θ, we maximize the the likelihood function (the joint density function):

$$\max_\theta \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2} f_\theta\left(\mathbf{x}^i\right)^2\right).$$

Taking the log of the likelihood we get

$$N \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \sum_{i=1}^N f_\theta\left(\mathbf{x}^i\right)^2.$$

Therefore, the likelihood maximization problem is equivalent to the following *least squares* minimization problem:

$$\min_{\theta} \sum_{i=1}^{N} f_{\theta}\left(\mathbf{x}^{i}\right)^{2},$$

which is known as minimizing the *squared error function*, as it measures the square difference of $f_{\theta}\left(\mathbf{x}^{i}\right)$ away from 0.

3.1.1 Simple Linear Model (Linear Least Squares)

One option for line fitting is taking the model

$$f\left(\mathbf{x}\right) = \mathbf{c}^{T}\mathbf{x} + c_0, \tag{3.1.2}$$

for scalar coefficients $\mathbf{c} = (c_1, c_2, \ldots, c_{n+1}) \in \mathbb{R}$ and $c_0 \in \mathbb{R}$ to be determined. Notice that the trivial coefficients $c_0 = c_1 = \ldots = c_{n+1} = 0$ always satisfy (3.1.1). Hence, if we can assume that $c_{n+1} \neq 0$, then dividing (3.1.1) by this non-zero scalar and redefining the scalars $a_j \coloneqq -\text{sign}\left(c_{n+1}\right) c_j / c_{n+1}$ for all $j = 0, 1, \ldots, n$, gives the linear relation

$$\mathbf{x}_{n+1}^{i} \approx a_0 + a_1\mathbf{x}_1^{i} + \cdots + a_n\mathbf{x}_n^{i}, \quad \forall i = 1, 2, \ldots, N. \tag{3.1.3}$$

In this scenario, we can find a suitable coefficients vector $\mathbf{a} = (a_0, a_1, \ldots, a_n) \in \mathbb{R}^{n+1}$ by minimizing the squared error function, that in the case of (3.1.3) takes the following *linear least squares* form:

$$\min_{\mathbf{a}} \left\|\mathbf{A}\mathbf{a} - \mathbf{y}\right\|^{2} = \sum_{i=1}^{N} \left(a_0 + a_1\mathbf{x}_1^{i} + \cdots + a_n\mathbf{x}_n^{i} - \mathbf{x}_{n+1}^{i}\right)^{2}, \tag{3.1.4}$$

where we define

$$\mathbf{A} \equiv \begin{bmatrix} 1 & \mathbf{x}_1^1 & \cdots & \mathbf{x}_n^1 \\ 1 & \mathbf{x}_1^2 & \cdots & \mathbf{x}_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \mathbf{x}_1^N & \cdots & \mathbf{x}_n^N \end{bmatrix} \in \mathbb{R}^{N \times (n+1)} \quad \text{and} \quad \mathbf{y} \equiv \begin{bmatrix} \mathbf{x}_{n+1}^1 \\ \mathbf{x}_{n+1}^2 \\ \vdots \\ \mathbf{x}_{n+1}^N \end{bmatrix}.$$

Notice that if the $n+1$-coordinate is considered as the label of each data point, then the optimization problem in (3.1.4) minimizes the squared difference (error) between each data point and its label. The optimal solution to (3.1.4) is given explicitly by the *normal equations*:

$$\mathbf{A}^{T}\mathbf{A}\mathbf{a} = \mathbf{A}^{T}\mathbf{y} \quad \Longrightarrow \quad \mathbf{a} = \left(\mathbf{A}^{T}\mathbf{A}\right)^{-1}\mathbf{A}^{T}\mathbf{y} \quad \text{if } \mathbf{A} \text{ is of full column rank.}$$

Summarizing the above derivations, the obtained algorithm for finding the line fit is given as follows:

1. Construct the matrix $\mathbf{A}$ and the vector $\mathbf{y}$.

2. Calculate $\mathbf{a}$ according to the normal equations.

3. Return the linear fit $\mathbf{x} \mapsto a_0 + a_1\mathbf{x}_1 + \ldots + a_n\mathbf{x}_n$ for any $\mathbf{x} \in \mathbb{R}^{n+1}$.

However, what if this optimal approximation for the model in (3.1.2) necessitates that $a_{n+1} = 0$ or is very close to 0? In this scenario, the model in (3.1.2) may no longer be valid, or it could result in a significant squared error. This suggests that alternative linear models for f might produce more satisfactory results.

3.1.2 Projection Linear Model (PCA)

One such alternative is to find a vector $\mathbf{v} \in \mathbb{R}^{n+1}$ such that the projection of each data point on the affine vector $\mathbf{v}+\boldsymbol{\alpha}$ (which is a vector that do not necessarily pass through the origin), for $\boldsymbol{\alpha} \in \mathbb{R}^{n+1}$, that minimizes the squared error between the projection and the data point. Both $\mathbf{v}$ and the translation vector $\boldsymbol{\alpha}$ are to be determined. This can be achieved by taking the linear model

$$f\left(\mathbf{x}\right) = \boldsymbol{\alpha} + \frac{\mathbf{v}^T\left(\mathbf{x} - \boldsymbol{\alpha}\right)}{\left\|\mathbf{v}\right\|^2}\mathbf{v} - \mathbf{x}, \tag{3.1.5}$$

In this case, following (3.1.1), each data point satisfies the linear relation

$$\mathbf{x}^i \approx \boldsymbol{\alpha} + \frac{\mathbf{v}^T\left(\mathbf{x}^i - \boldsymbol{\alpha}\right)}{\left\|\mathbf{v}\right\|^2}\mathbf{v}, \quad \forall i = 1, 2, \ldots, N. \tag{3.1.6}$$

Indeed, it can be shown that the projection of some vector $\mathbf{x}$ onto the affine vector $\mathbf{v} + \boldsymbol{\alpha}$ is given by the right hand side of (3.1.6).

As in the previous model discussed above, in this scenario we can also find suitable vector $\mathbf{v} \in \mathbb{R}^{n+1}$ and a translation vector $\boldsymbol{\alpha} \in \mathbb{R}^{n+1}$ by minimizing the squared error function, where in the case of (3.1.6) takes the following *non-linear least squares* form:

$$\min_{\mathbf{v},\boldsymbol{\alpha}} \sum_{i=1}^{N} \left\|\boldsymbol{\alpha} + \frac{\mathbf{v}^T\left(\mathbf{x}^i - \boldsymbol{\alpha}\right)}{\left\|\mathbf{v}\right\|^2}\mathbf{v} - \mathbf{x}^i\right\|^2. \tag{3.1.7}$$

Notice that the optimization problem in (3.1.7) minimizes the squared difference (error) between each data point and its projection onto the affine vector to be determined $\mathbf{v} + \boldsymbol{\alpha}$. Assuming without the loss of generalization that $\|\mathbf{v}\| = 1$, then solving (3.1.7) for $\boldsymbol{\alpha}$ yields after simple algebraic manipulations that

$$\boldsymbol{\alpha} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}^i \equiv \bar{\mathbf{x}}.$$

Now, we get that (3.1.7) reduces to

$$\min_{\|\mathbf{v}\|=1} \left\{\sum_{i=1}^{N}\left\|\mathbf{x}^i - \bar{\mathbf{x}}\right\|^2 - \mathbf{v}^T\sum_{i=1}^{N}\left(\mathbf{x}^i - \bar{\mathbf{x}}\right)\left(\mathbf{x}^i - \bar{\mathbf{x}}\right)^T\mathbf{v}\right\}, \tag{3.1.8}$$

It can be shown that the optimal solution to this problem is the normalized eigenvector that corresponds to the largest eigenvalue of the matrix $\mathbf{X}\mathbf{X}^T \in \mathbb{R}^{(n+1)\times(n+1)}$, where

$$\mathbf{X} \equiv \begin{bmatrix} | & | & & | \\ \mathbf{x}^1 - \bar{\mathbf{x}} & \mathbf{x}^2 - \bar{\mathbf{x}} & \cdots & \mathbf{x}^N - \bar{\mathbf{x}} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{(n+1)\times N} \Longrightarrow \mathbf{X}\mathbf{X}^T = \sum_{i=1}^{N} \left(\mathbf{x}^i - \bar{\mathbf{x}}\right)\left(\mathbf{x}^i - \bar{\mathbf{x}}\right)^T.$$

Of course, finding the normalized eigenvector corresponding to the largest eigenvalue $\lambda_{\max}\left(\mathbf{X}\mathbf{X}^T\right)$ is a more computationally demanding task than solving the normal equations in the simpler linear model discussed above. However, it allows for more fitting flexibility that could be beneficial. We should mention that generalization of this technique to higher-dimension, where we project onto an affine hyper-plane, can be derived. This technique is also called *Principal Component Analysis* (PCA).

Summarizing the above derivations, the obtained algorithm for finding the line fit is given as follows:

1. Construct the average vector $\bar{\mathbf{x}}$ and the matrix $\mathbf{X}$.

2. Calculate $\mathbf{v}$ the normalized eigenvector of $\mathbf{X}\mathbf{X}^T$ corresponding to $\lambda_{\max}\left(\mathbf{X}\mathbf{X}^T\right)$.

3. Return the linear fit $\mathbf{x} \mapsto \bar{\mathbf{x}} + \mathbf{v}^T\left(\mathbf{x} - \bar{\mathbf{x}}\right)\mathbf{v}$ for any $\mathbf{x} \in \mathbb{R}^{n+1}$.

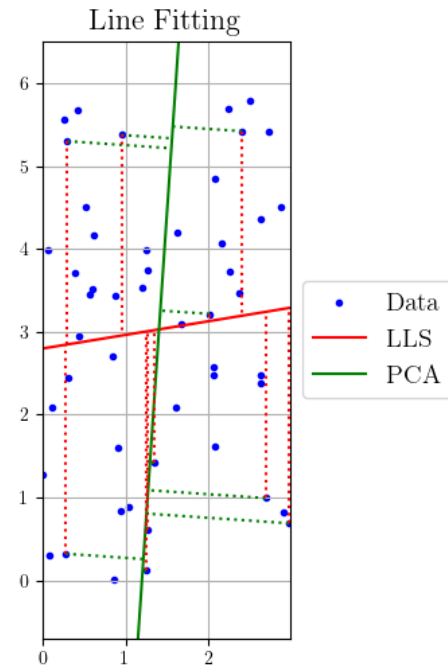
Relation to maximal variance: inspecting (3.1.8) we see that the maximal variance of the projections of the data points onto an affine vector is achieved for the optimal $\mathbf{v}$ and $\boldsymbol{\alpha} = \bar{\mathbf{x}}$ found above. To see this, notice that

$$\begin{aligned}
\mathrm{Var}\left[\left\{\mathbf{v}^T\left(\mathbf{x}^i - \bar{\mathbf{x}}\right)\right\}_{i=1}^{N}\right] &\equiv \frac{1}{N}\sum_{i=1}^{N} \mathbf{v}^T\left(\mathbf{x}^i - \bar{\mathbf{x}}\right)\left(\mathbf{x}^i - \bar{\mathbf{x}}\right)^T \mathbf{v} - \left(\frac{1}{N}\sum_{j=1}^{N} \mathbf{v}^T\left(\mathbf{x}^j - \bar{\mathbf{x}}\right)\right)^2 \\
&= \frac{1}{N}\mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v} - \left(\frac{\mathbf{v}^T}{N}\sum_{j=1}^{N}\mathbf{x}^j - \frac{\mathbf{v}^T}{N}\sum_{j=1}^{N}\left(\frac{1}{N}\sum_{l=1}^{N}\mathbf{x}^l\right)\right)^2 \\
&= \frac{1}{N}\mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v},
\end{aligned}$$

which is the objective function under maximization in (3.1.8), up to a positive constant.

Run in Colab In Figure 3.1.1, we illustrate the line fitting results from both methods: the simpler linear least squares (LLS) and the PCA linear projection model. The differences minimized by each method are represented by dashed lines. It is evident that PCA offers a superior line fit, as also indicated by the squared errors – approximately 142 for LLS and 42 for PCA. However, due to the steep slope obtained from PCA, in the context of predicting label values, the LLS method may still yield better results for unseen data. Nevertheless, because PCA maximizes the variance of the projected points, it is a more effective choice for dimensionality reduction, ensuring that the least amount of information is lost during projection.

Figure 3.1.1: Image blending using by replacing mid-range frequencies of the DFTs.



3.2 Line Fitting Using Parameter Spaces

As discussed earlier, identifying the optimal model for line fitting is challenging, particularly for vertical lines. Therefore, we need alternative techniques to achieve a suitable line fit.

3.2.1 Line Equations in $\mathbb{R}^2$

To begin, recall that a line L in $\mathbb{R}^2$ can be expressed in the *standard form* as follows:

$$L = \{(x, y) : Ax + By = C\},$$

where $A, B, C \in \mathbb{R}$ and at least one of A and B is non-zero. This equation represents all solutions to a linear equation. In this representation, horizontal lines are obtained when $A = 0$ and vertical lines when $B = 0$. If $C = 0$, the line passes through the origin. In addition, C/A is interpreted geometrically as the x-intercept (which does not exist for horizontal lines) and C/B as the y-intercept (which does not exist for vertical lines).

When $B \neq 0$ (i.e., the line is non-vertical), we can express the line in the following *slope-intercept form*:

$$L = \{(x, y) : y = mx + b\}, \tag{3.2.1}$$

where $m \equiv -A/B$ and $b \equiv C/B$. Here, m is the gradient with respect to x, which is the slope of the line, and b is the y-intercept.

It is important to note that in both the standard and slope-intercept forms, the parameters can take any value in $\mathbb{R}$, which can have numerical implications, as will be discussed later. Alternatively, one can divide the standard form by $\operatorname{sign}(C)\sqrt{A^2+B^2}$, where we define $\operatorname{sign}(0) \equiv 1$. This results in the equation:

$$\frac{\operatorname{sign}(C)\,A}{\sqrt{A^2+B^2}}x + \frac{\operatorname{sign}(C)\,B}{\sqrt{A^2+B^2}}y = \frac{|C|}{\sqrt{A^2+B^2}}. \tag{3.2.2}$$

In this formulation, the coefficients of x and y can take any value in the range $[-1,1]$, while the constant term can be any positive number. Additionally, since the sum of the squares of the coefficients of x and y equals 1, the line can be equivalently expressed in the *Hesse normal form*:

$$L = \{(x,y) : \cos(\theta)\,x + \sin(\theta)\,y = r\}, \tag{3.2.3}$$

for $\theta \in [0, 2\pi)$ and $r \geq 0$. In this representation, the parameter θ is bounded, and r has a lower bound of zero.

Interestingly, the values of r and θ possess a geometric interpretation. By solving a straightforward optimization problem, it can be shown that the point (x, y) on the line $Ax + By = C$ that is closest to the origin is given by:

$$\left(\frac{AC}{A^2+B^2}, \frac{BC}{A^2+B^2}\right).$$

The distance of this point from the origin, or its norm, is simply

$$r = \frac{|C|}{\sqrt{A^2+B^2}}, \tag{3.2.4}$$

as in (3.2.2). Furthermore, if θ represents the angle between the x-axis and this point, then it can be expressed as:

$$\theta = \operatorname{atan2}\left(\frac{BC}{A^2+B^2}, \frac{AC}{A^2+B^2}\right), \tag{3.2.5}$$

where atan2 is defined in (2.2.5). Consequently, according to the definition of atan2, we can derive that $\sin(\theta) = \operatorname{sign}(C)\,B/\sqrt{A^2+B^2}$ and $\cos(\theta) = \operatorname{sign}(C)\,A/\sqrt{A^2+B^2}$, as in (3.2.2).

3.2.2 Hough Transform in Slope-Intercept Parameter Space

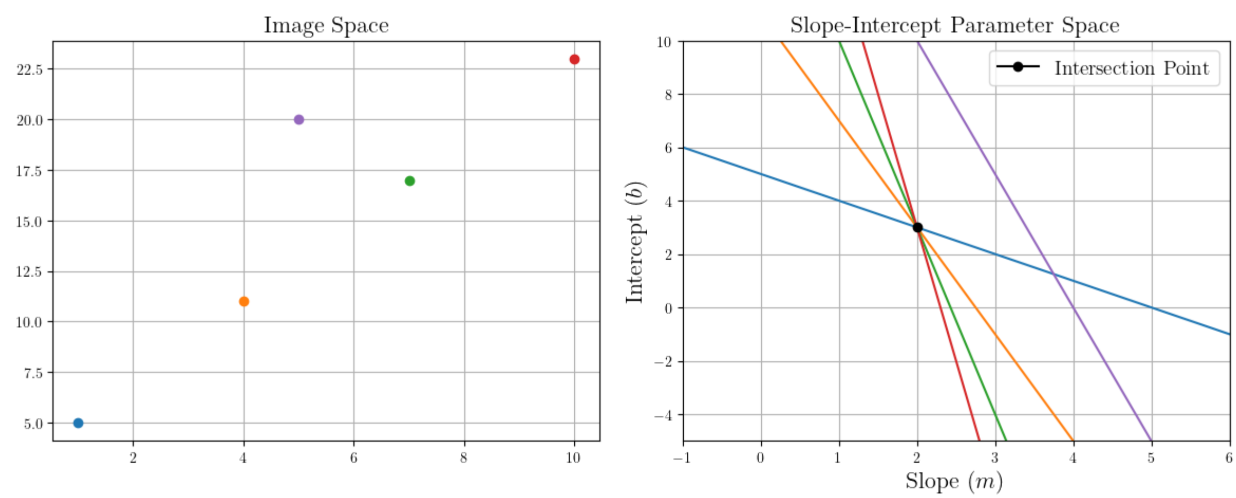
In the context of line fitting, the *slope-intercept parameter space* is defined by the parameters (m, b) as in (3.2.1). Each point in the parameter space corresponds to a unique line in the image space, where the coordinates (x, y) represent the Cartesian coordinates of points on that line. Thus, for any given slope m and intercept b, there exists a line in the image space that can be defined by all points (x, y) satisfying the line equation. Conversely, a line in the image space corresponds to a unique point in the parameter space.

Note that a single point in the image space can correspond to any slope m, with the intercept adjusted accordingly. For instance, the point $(1, 1)$ in the image space, when paired with a slope m results in an intercept $b = 1 - m$. Thus, a single point in the image space defines a line in the parameter space. Conversely, a line in the parameter space corresponds to a unique point in the image space.

How can we detect co-linearity using the slope-intercept parameter space? Consider a set of points in the image space that lie on the same line, such as those identified by an edge detector with minimal or no noise. As discussed above, each of these points is transformed into a line in the slope-intercept parameter space, representing by all possible values of (m, b) for which the point corresponds to a line in the image space with slope m and intercept b. Since all points are co-linear, they share the same slope and intercept in the image space. Consequently, when these points are transformed into the parameter space, the lines they represent will intersect at a single point (m, b), which reflects the equation of the line on which all the points lie.

Run in Colab This concept is illustrated in Figure 3.2.1. In this figure, we create a scatter plot of five points in the image space, four of which are co-linear and presumably represent detected edge pixels. Each point is represented in a different color, and its corresponding line $b = y - mx$ with the same color is plotted in the slope-intercept parameter space, where (x, y) denotes the point. The intersection of two lines in the parameter space indicates that the values of m and b at this intersection point define the slope and intercept of a line equation on which both points reside in the image space. The intersection point with the highest number of line intersections in the parameter space corresponds to the line equation on which the majority of the points in the image space are co-linear. In this case, this point is $(m, b) = (2, 3)$.

Figure 3.2.1: Image space showing points along with their corresponding lines in the slope-intercept parameter space, facilitating edge detection through line fitting in the image space via the intersection point with the highest number of line intersections. In this scenario, the resulting line in the image space has a slope of $m = 3$ and an intercept of $b = 2$.



Line Detection Using Hough Transform in Slope-Intercept Space. The process described above leads to the *Hough Transform* line detection algorithm. This algorithm identifies intersection points in the slope-intercept parameter space where the highest number of lines converge, representing lines or edges in the image space. The Hough Transform algorithm in the slope-intercept space is outlined in Algorithm 3.2.1.

Algorithm 3.2.1 Hough Transform – Slope-Intercept Parameter Space

Quantize the parameter space (m, b).
Initialize an accumulator array $A(m, b) = 0$ for all (m, b).
for (x_i, y_i) in image edge **do**
  for m in quantized space **do**
    Find b in quantized space for which $y_i - mx_i$ belongs.
    $A(m, b) := A(m, b) + 1$
  end for
end for
Find local maxima in A.

Given that the parameter space is continuous and infinite, measurement noise or other factors may prevent us from locating intersection points with the maximum number of lines passing through. To address this, we first quantize the parameter space, transforming it into a finite and discrete representation. Consequently, a neighborhood of points in the parameter space is consolidated into a single point, simplifying the detection of intersection points. This quantization also enhances the robustness of the process against measurement noise and outliers, as minor variations in slope or intercept become indistinguishable in the quantized space. However, it is important to note that finer quantization reduces robustness to measurement noise, while coarser quantization increases robustness but may lead to detecting features that do not correspond to actual edges in the image space.

We also mention that Hough Transform – compared to the linear least squares or PCA techniques introduced above – is good in scenarios where data points are noisy or contain outliers. Since the Hough Transform accumulates votes in a parameter space, the influence of outliers is minimized because they are less likely to contribute to peaks in the accumulator array. The least squares and PCA methods can be heavily affected by outliers, as they minimize the overall error. A few erroneous points can skew the results significantly.

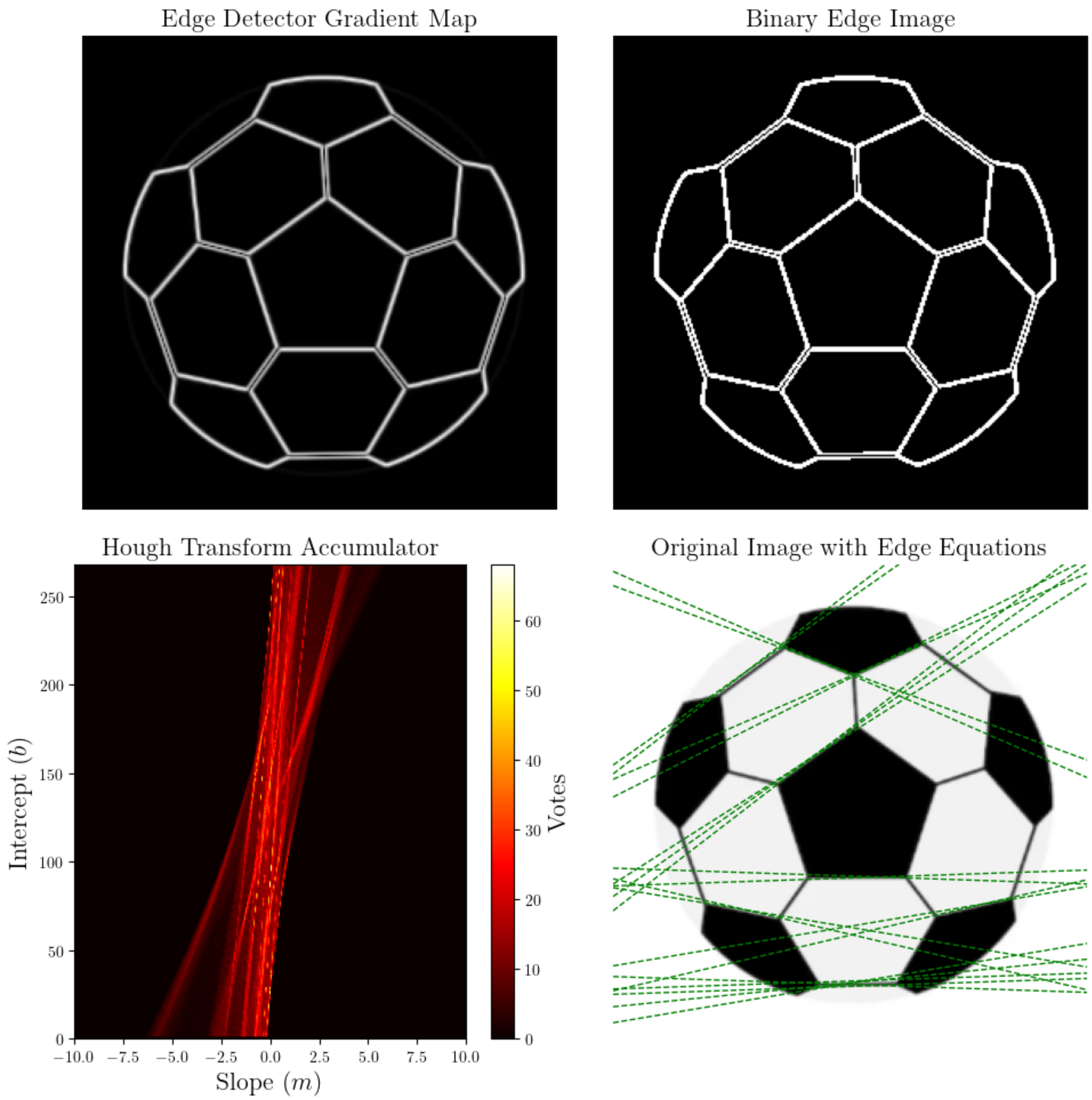
Another benefit of the Hough Transform is that the collection of points representing a single edge in an image does not need to be explicitly identified by the user, as is required with other line fitting methods. Specifically, when using linear least squares or PCA, the algorithm must be provided with the scatter of points. In contrast, the Hough Transform detects edges by counting the number of lines that intersect at various points.

Run in Colab

In Figure 3.2.2, we demonstrate the process of line detection using the Hough Transform. First, we utilize the Prewitt filter for edge detection to generate a gradient map (see Section 1.3.1), which identifies the pixels associated with edges in the image. Next, we apply a binary thresholding criterion to the gradient map, assigning a value of 1 to pixels that exceed the threshold and a value of 0 to those that do not. This step may result in the loss of pixels that are affected by noise. The remaining pixels now represent the image space. In the case of the specific image in Figure 3.2.2, the binary thresholding parameter is set to 1 (maximal value).

Next, we populate the accumulator according to Algorithm 3.2.1. To achieve this, we define a quantization of the slope-intercept parameter space. Specifically, we set 300 equally spaced values for m in the range $[-10, 10]$. For the b values, since the maximum shift along the y-axis cannot exceed the image row size, we set `image.shape[0]` equally spaced values for

Figure 3.2.2: Hough Transform in the slope-parameter space applied to extract edge equations from an image. The gradient map generated by the Prewitt filter is utilized to produce a binary edge image representing the image space. The accumulator (quantized slope-intercept space) is utilized to derive the edge equations.



b within the range 0 to `image.shape[0]`. In Figure 3.2.2, we display the resulting accumulator, using a colormap that reflects the number of votes in each cell. The number of votes simply represents the count of lines that intersect at the corresponding point of each cell. In it is important to note that the values in the accumulator are significantly influenced by the level of quantization and the selected values of m and b.

Once the accumulator is filled, we proceed to identify local maxima within it. Recall that local maxima indicate intersection points where the greatest number of lines pass through. A local maximum is defined as a cell with a value higher than that of its neighboring cells. To filter out local maxima associated with weak edges, we only examine those that exceed a specified vote threshold. In this instance, we consider local maxima with more than

70% of the maximum value found in the accumulator (alternatively, we could simply select the top k cells as local maxima). After identifying the local maxima – dependent on the methodology used and the thresholds applied – we can plot the edge equations corresponding to these maxima on the original image. In Figure 3.2.2, we observe that we obtain good edge equations that cover most of the edges in the image. To detect shorter edges, we may need to lower the thresholds. Conversely, for thicker edges with multiple pixels, we should increase the threshold to derive a single edge equation.

We reiterate that several practical factors influence the results obtained from the Hough Transform algorithm. These include the binary threshold of the edge detector filter, the refinement and boundaries of quantization, and the threshold for detecting local maxima in the accumulator.

3.2.3 Hough Transform in Hough Space

A significant challenge when applying the Hough Transform algorithm in the slope-intercept parameter space, is that vertical lines require an infinite m value, or at the very least, a very large value of m for nearly vertical lines. Since we quantize the parameter space, achieving this is only possible with either a very large accumulator or very coarse quantization. Consequently, this presents a significant computational limitation.

For these computational reasons, we consider the *Hough parameter space*, which is the space of parameters (r, θ) as in (3.2.3). The benefit of utilizing the Hough parameter space instead of the slope-intercept space is that the accumulator in this space is bounded. Specifically, $0 \leq \theta < 2\pi$ and $0 \leq r \leq r_{\max}$, where $r_{\max}$ represents the maximum possible radius based on the image dimensions. In addition, vertical lines can be easily described, simply by taking $\theta \in \{0, \pi\}$ and obtaining the vertical line $x = \pm r$.

As in the slope-intercept space, each point in the Hough space corresponds to a unique line in the image space. Thus, for any given slope $\theta \in [0, 2\pi)$ and radius $r \geq 0$, there exists a line in the image space that can be defined by all points (x, y) satisfying the line equation. Conversely, a line in the image space corresponds to a unique point in the Hough space.

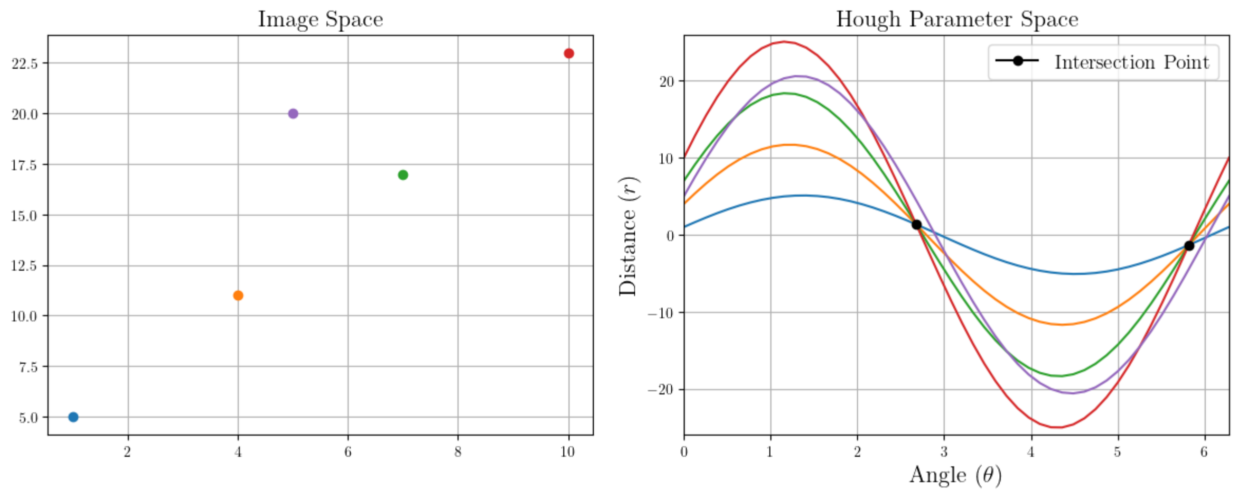
Note that a single point in the image space can correspond to any angle θ, with the radius r adjusted accordingly. For instance, the point $(1, 1)$ in the image space, when paired with an angle θ results in an radius $r = \cos(\theta) + \sin(\theta)$. Thus, a single point in the image space defines a sinusoidal function in the Hough space. Conversely, a line in the Hough space corresponds to a unique point in the image space.

How can we detect co-linearity using the Hough parameter space? Similarly to the slope-intercept space, if a scatter of points is co-linear, then there is a line equation passes through all of which, hence they share the same θ and r, which will be seen an intersection point in the Hough space.

Run in Colab

This concept is illustrated in Figure 3.2.3. In this figure, we present the same scatter plot as in 3.2.1, with the corresponding sinusoidal functions for each point plotted in the Hough parameter space. The intersection of two sinusoids in the Hough space signifies that the values of θ and r at this intersection point define the angle and distance of a line equation that both points lie on in the image space. The intersection point with the greatest number of line intersections in the parameter space represents the line equation on which most of the points in the image space are co-linear. In this case, since the equation of the co-linear line is $-2x + y = 3$, the corresponding intersection point, according to (3.2.4) and (3.2.5), is $(r_0, \theta_0) \approx (1.34, 2.68)$.

Figure 3.2.3: Image space showing points along with their corresponding lines in the Hough parameter space, facilitating edge detection through line fitting in the image space via the intersection point with the highest number of line intersections. In this scenario, the resulting line in the image space has an angle $\theta \approx 2.68$ and a distance $r \approx 1.34$.



It is important to note that there is another intersection point in Figure 3.2.3 located at $(-r_0, \theta_0 + \pi) \approx (-1.34, 5.82)$. This occurs because of the relationships:

$$\sin(\theta \pm \pi) = -\sin(\theta) \quad \text{and} \quad \cos(\theta \pm \pi) = -\cos(\theta),$$

which indicates that adding $\pm\pi$ to the angle results in $-r$. This implies that when we quantize the Hough space, we can either define the domain as $\theta \in [0, 2\pi)$ and $r \in [0, r_{\max}]$ for some suitable $r_{\max} \geq 0$ while searching for intersection points, or we can quantize the space as $\theta \in [0, \pi)$ and $r \in [-r_{\max}, r_{\max}]$. Both quantization domains are always valid, and the choice depends on the value of $r_{\max}$, as it influences the size of the search range for r.

Line Detection Using Hough Transform in Hough Space. Similarly to Algorithm 3.2.1, we devise an algorithm that identifies intersection points in the Hough parameter space where the highest number of lines converge, representing lines or edges in the image space. The Hough Transform algorithm in the Hough space is outlined in Algorithm 3.2.2.

Run in Colab

In Figure 3.2.4, we illustrate the process of line detection using the Hough Transform and the corresponding Hough parameter space. Similar to Figure 3.2.2 for the slope-intercept space, we apply the Prewitt filter for edge detection to create a gradient map and implement a binary thresholding criterion with a threshold set to 1. For further details, refer to the explanation in Figure 3.2.1.

Subsequently, we populate the accumulator following Algorithm 3.2.2. This involves defining a quantization for the Hough space, where here we specify 200 equally spaced values for θ within the range $[0, \pi]$. For the r values, since the maximum distance cannot

Algorithm 3.2.2 Hough Transform – Hough Parameter Space

```
Quantize the parameter space (θ, r).
Initialize an accumulator array A(θ, r) = 0 for all (θ, r).
for (x_i, y_i) in image edge do
    for θ in quantized space do
        Find r in quantized space for which x_i cos(θ) + y_i sin(θ) belongs.
        A(θ, r) := A(θ, r) + 1
    end for
end for
Find local maxima in A.
```

exceed the maximum length of the image, we also define 200 equally spaced values for r within the range $[-r_{\max}, r_{\max}]$, where $r_{\max}$ is `sqrt(image.shape[0]**2 +image.shape[1]**2)`. As mentioned above, we can alternatively quantize the space to $0 \leq \theta \leq 2\pi$ and $0 \leq r \leq r_{\max}$.

In Figure 3.2.4, we present the resulting accumulator, utilizing a colormap that indicates the number of votes in each cell. For this figure, a local maximum in the accumulator corresponds to values exceeding 82% of the maximum value found. It is evident from Figure 3.2.4 that we successfully obtain robust edge equations that encompass most of the edges in the image, detecting more edges (particularly vertical edges) than what was observed in Figure 3.2.2 in the slope-intercept space.

Once again, we emphasize that the effectiveness of detection of edge equations is significantly influenced by the hyperparameters, making it challenging to establish universal guidelines for their selection. These practical considerations include the binary threshold for the edge detector filter, the refinement and limits of quantization, and the threshold used for identifying local maxima in the accumulator.

3.3 More Complex Formations With Hough Transform

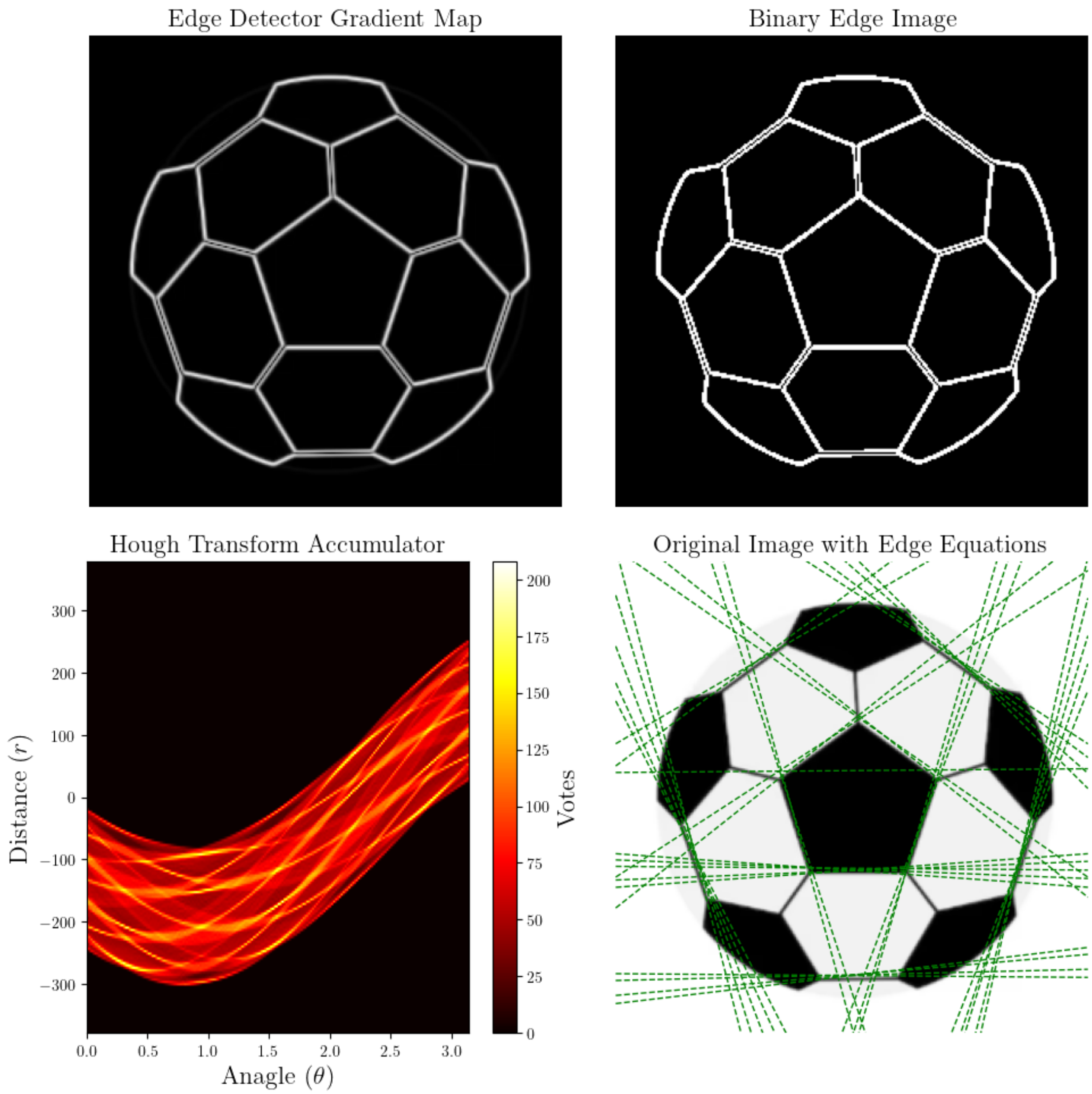
Run in Colab

In Figure 3.3.1, we present three additional plots for edge line detection using the Hough Transform in the Hough space (see Algorithm 3.2.2). For the triangle image, we obtain line equations corresponding to its three edges. Notice that for the horizontal edge of the triangle, the closest point to the origin (bottom-left corner) is its y-intercept, which has an angular value of $\theta = \pi/2$. As a result, the accumulator plot shows a value of r corresponding to this angular value, representing the distance (in pixels) from the origin. The threshold used for identifying local maxima in the triangle accumulator is set to %30 of the of the maximal vote (largest accumulator value).

For the rectangle image, we observe line equations corresponding to the local maxima for $\theta = \pi/2$ in the accumulator, representing horizontal lines. The vertical lines are represented by values at $\theta = 0$, as the closest point to the origin for these lines is the x-intercept. There are also similar lines at the angular value $\theta = \pi$, though we only need to quantize the interval $[0, \pi)$. The local maxima threshold used in the rectangular accumulator is set to %50 of the of the maximal vote.

For the circle image, the detected lines are approximately those generated by any two

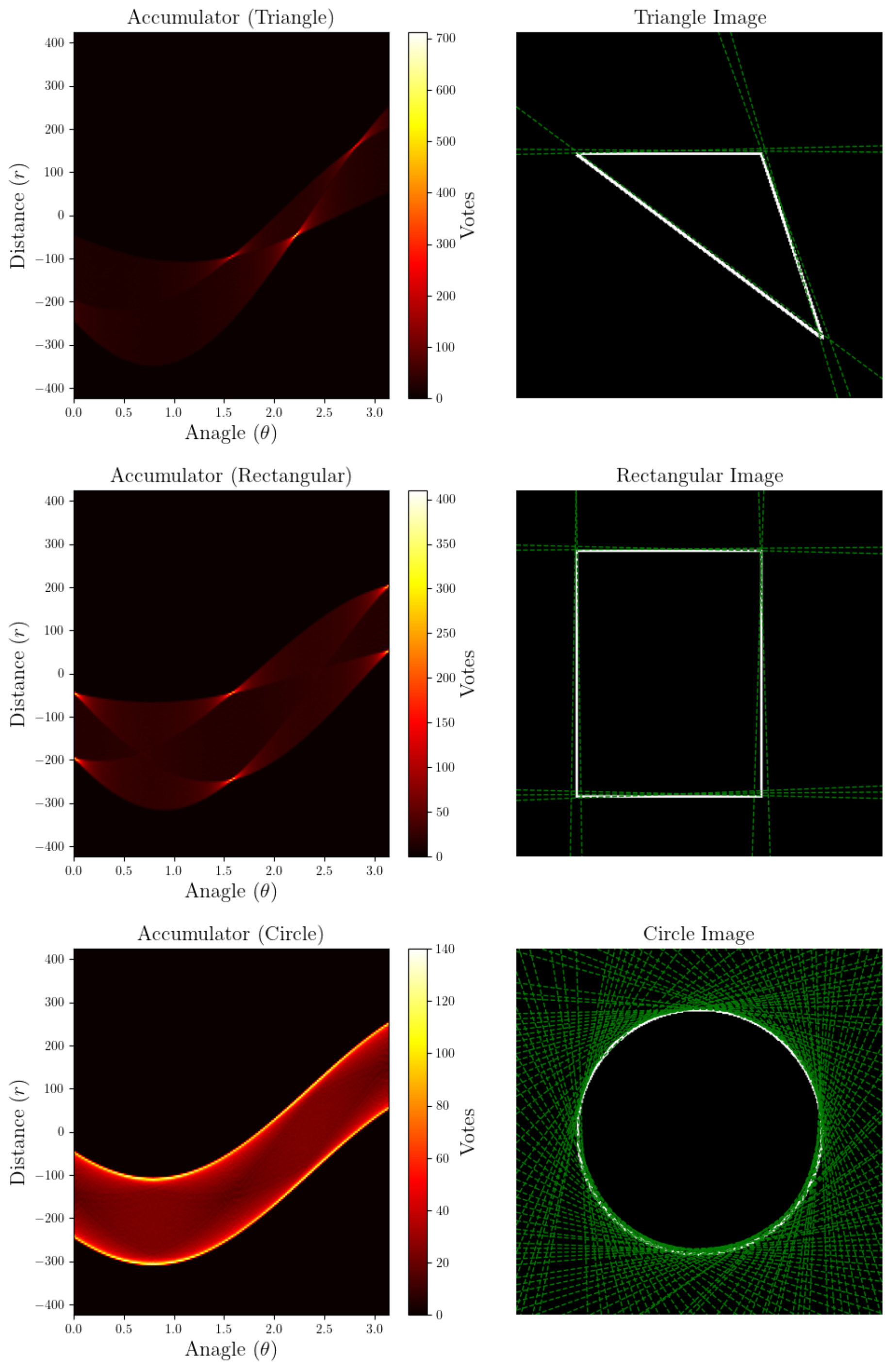
Figure 3.2.4: Hough Transform in the Hough space applied to extract edge equations from an image. The gradient map generated by the Prewitt filter is utilized to produce a binary edge image representing the image space. The accumulator (quantized Hough space) is utilized to derive the edge equations.



adjacent pixels along the boundary of the circle boundary. Thus, for any value of angular value θ, two local maxima are generated, representing a tangent line and its parallel counterpart. The local maxima threshold used in the circle accumulator is set to %80 of the of the maximal vote.

Notice that again we face a practical difficulty, as we had to set differently the local maxima threshold for each of the three shape images for better detection.

Figure 3.3.1: Hough Transform applied to three basic shapes in 300 × 300 pixel images using the Hough parameter space.

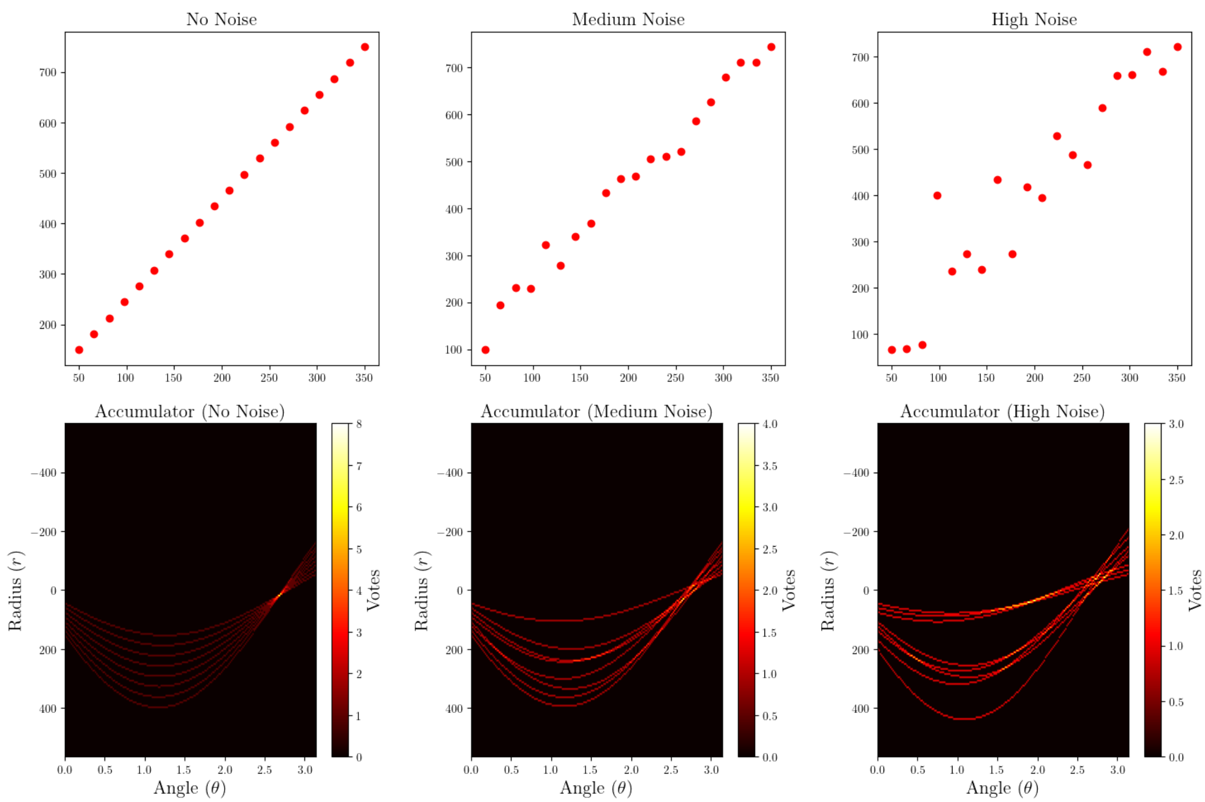


3.3.1 Effect of Noise on Hough Transform

In practice, edges in images are often noisy due to various factors such as sensor noise, lighting variations, and imperfections in edge detection algorithms. This noise can affect the results of the Hough Transform by spreading votes across multiple bins in the accumulator, making it difficult to identify the correct line parameters.

Run in Colab To better understand the impact of noise, we demonstrate in Figure 3.3.2 how the accumulator changes as noise is introduced to a line, showing cases with no noise, low noise, and higher noise levels. It can be seen that as the nose increases, the number of votes in the correct bin decreases, and for high noise the number of wrong votes in a certain bin can even surpass the number of votes in the correct bin.

Figure 3.3.2: Hough Transform applied to a line with tree different noise levels.

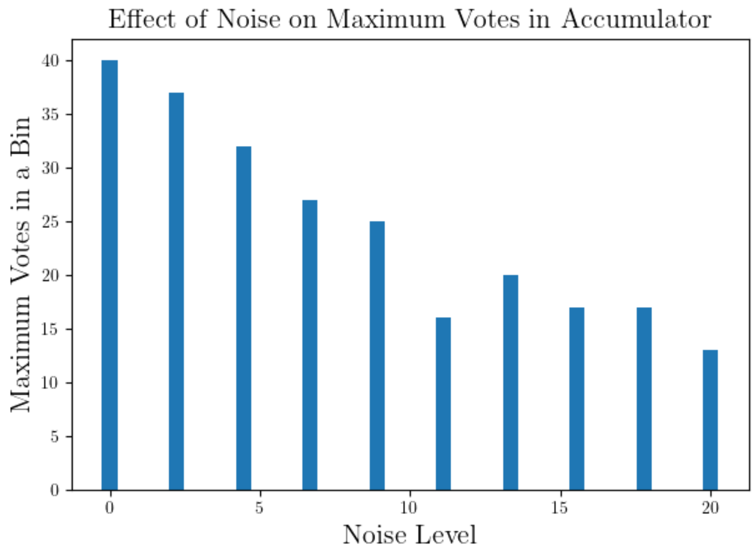


Run in Colab As discussed above, noise can significantly affect the detection of lines in an image. To better understand this effect, we can examine how the maximum number of votes in accumulator, across all bins, changes as noise is increased. This is illustrated in Figure 3.3.3. For a line consisting of 100 points, we plot the maximum number of votes in the accumulator as the noise level increases. The maximum number of votes clearly decreases, which indicates that the number of votes in the correct bin also diminishes. As a result, the votes are spread more broadly, reducing concentration around the correct bin and making line detection more challenging.

3.3.2 Hough Circles

The Hough Circle Transform is an extension of the Hough Transform used to detect circles in an image. It works by representing a circle in parametric form and finding the best circle

Figure 3.3.3: The maximum number of votes in a bin of the Hough Transform accumulator for a line formed by a scatter of 100 points.



parameters that fit a given set of edge points. Unlike line detection, the parameter space for circles in $\mathbb{R}^2$ involves three parameters, which are the coordinates of the center of circle (a, b), and its radius $r \geq 0$:

$$(x-a)^2 + (y-b)^2 = r^2.$$

For each edge point (x, y), there are infinitely many possible circles that could pass through it. The Hough Circle Transform identifies potential circles by voting in an accumulator space for various values of (a, b, r). For each detected edge point, the algorithm considers a range of possible radii and updates the accumulator for each potential circle center. The Hough Circle Transform is outlined in Algorithm 3.3.1.

Algorithm 3.3.1 Hough Circle Transform

Quantize the parameter space (a, b, r).
Initialize an accumulator array $A(a, b, r) = 0$ for all (a, b, r).
for (x_i, y_i) in image edge **do**
    for a in quantized space **do**
        for b in quantized space **do**
            Find r in quantized space for which $\sqrt{(x_i-a)^2 + (y_i-b)^2}$ belongs.
            $A(a, b, r) := A(a, b, r) + 1$
        end for
    end for
end for
Find local maxima in A.

Run in Colab In Figure 3.3.4, we illustrate the process of circle detection using the Hough Circle Transform. Similar to Figures 3.2.2 and 3.2.4 for line detection, we apply the Prewitt filter for edge detection to create a gradient map and implement a binary thresholding criterion with a threshold set to 1. For further details, refer to the explanation in Figure 3.2.1.

Subsequently, we populate the accumulator following Algorithm 3.3.1. To this end, we need to quantize the parameter space (a, b, r), where here we set 100 equally spaced values for a within the range $(0, a_{\max})$ where $a_{\max}$ is `image.shape[1]` (recall that a represents the x-axis value of the center of the circle). For b we also set 100 equally spaced values, but within the range $(0, b_{\max})$ where $b_{\max}$ is `image.shape[0]`. Last, for r we set 50 equally spaced within the range $[0, r_{\max}]$ where $r_{\max}$ is `sqrt(image.shape[0]**2 +image.shape[1]**2)`.

To visualize the accumulator of the Hough Circle Transform, we need to represent 4D data: the parameters a, b, r, and the number of votes in each bin. This is similar to the 3D data used for line detection, where we plotted the slope, intercept (or polar coordinates), and the corresponding votes. Visualizing this 4D data can be challenging. However, in Figure 3.3.4, we display the accumulator using a colormap that represents the r value with the maximum number of votes for each (a, b) pair. For example, it is clear that the detected circles in the image have centers located roughly near the center of the 268×268 pixel image, with radii around 100. Although this specific accumulator plot does not show the bins with local maxima, we define a local maximum as any value exceeding 90% of the highest vote count. From Figure 3.3.4, it is clear that we successfully identified robust circle equations that match the circles present in the image.

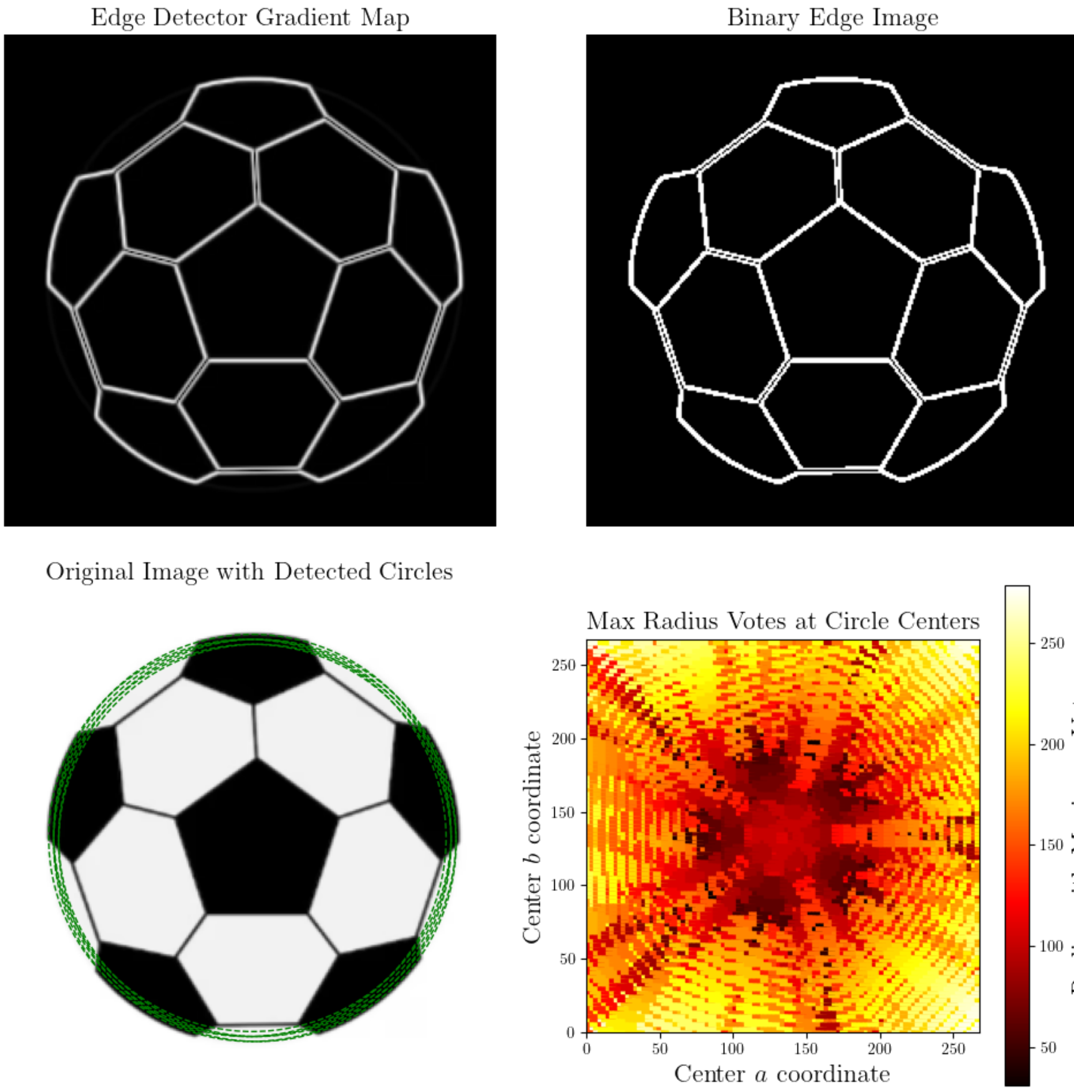
We point out that one of the key challenges of using the Hough Circle Transform is dealing with the 3D parameter space, which can be computationally intensive. Hence, this method is particularly useful for detecting circles of known or predictable sizes in an image (which limits the size of the quantized search space). However, it can be sensitive to noise and clutter, which may lead to false positives or missed detections if not properly tuned. The Hough Circle Transform is commonly used in applications like medical imaging (e.g., detecting cells) and in detecting circular objects in industrial inspection tasks.

Summarizing the Hough Transforms. The Hough Transform is a powerful technique for detecting shapes, such as lines or circles, in an image. It handles occlusion well because even if part of the shape is missing or obstructed, the voting process in the accumulator space can still gather enough evidence from the visible parts of the shape. As long as sufficient edge points align with the parametric form of the shape, the algorithm can detect the shape despite partial occlusion.

The method can also detect multiple instances of a shape because each possible shape generates its own peak in the accumulator. The algorithm can identify different peaks corresponding to different instances of the same shape, allowing it to detect multiple occurrences.

The Hough Transform is relatively robust to noise, as noise typically leads to scattered votes in the accumulator, which generally does not produce strong peaks. The votes of the shape tend to accumulate at the correct parametric values, while noise generally does not generate a coherent accumulation in the voting space. This helps filter out noise during the detection process.

Figure 3.3.4: Hough Circle Transform applied to extract circle equations from an image. The gradient map generated by the Prewitt filter is utilized to produce a binary edge image representing the image space. A visualization of the 4D accumulator using a colormap that represents the r value with the maximum number of votes for each (a, b) pair is plotted.



However, the computational complexity of the Hough Transform is a major drawback. The algorithm must consider many possible parametric values (e.g., slope and intercept for lines, or center coordinates and radii for circles), leading to a high-dimensional accumulator space. This can be computationally expensive, especially for large images or more complex shapes, as every edge point contributes to votes across a large number of bins in the accumulator. The complexity grows with the resolution of the quantized parameter space and the number of edge points, which can slow down the process and require significant memory.

Chapter 4

Corner Detection

In corner detection, the goal is to identify points in an image where the intensity changes significantly multiple directions. Applying a corner detection algorithm to the local image gradients is an effective way to determine the dominant directions of intensity change and detect whether a region contains a corner.

4.1 Identifying Corners Using Gradients

A corner in an image can be defined as a point for which there are two dominant and different edge directions in a local neighborhood of the point. Hence, we recall the definition of image gradients of an image $\mathbf{I} \in \mathbb{R}^{H \times W}$, as introduced in Section 1.3.1:

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

where $\mathbf{K}_x$ and $\mathbf{K}_y$ are the horizontal and vertical components of some discrete differentiation filter. The gradient at a point $(x, y) \in \mathbf{I}$ is then defined as:

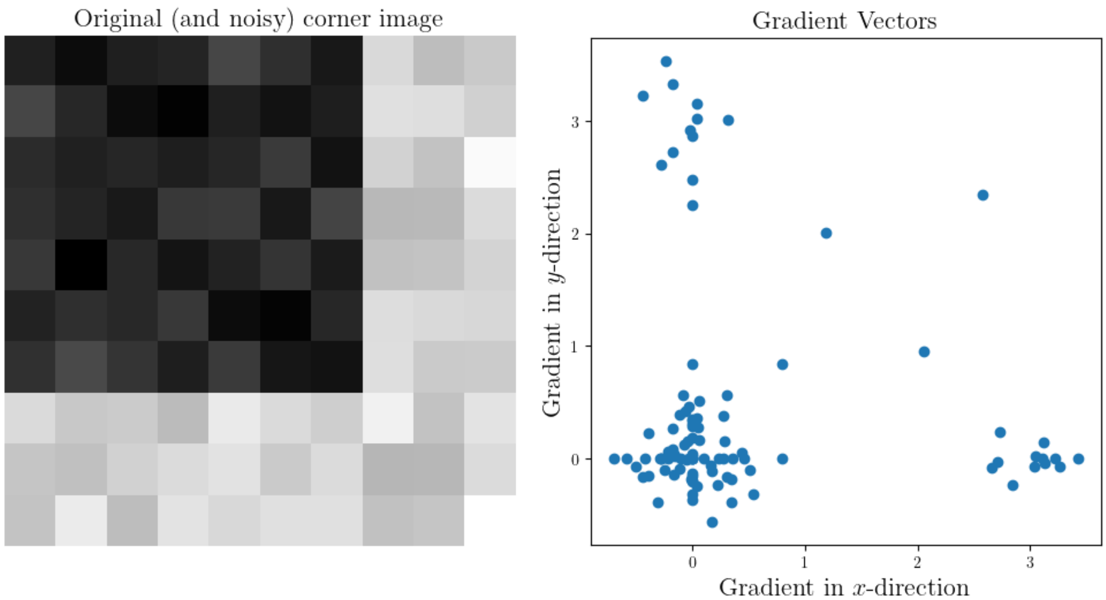
$$\nabla \mathbf{I}(x, y) \equiv \left(\frac{\partial \mathbf{I}}{\partial x}(x, y), \frac{\partial \mathbf{I}}{\partial y}(x, y) \right) \in \mathbb{R}^2, \tag{4.1.1}$$

where unless stated otherwise, the gradient of the point (x, y) is taken when the point is at the center of the filter.

Run in Colab

In Figure 4.1.1, we plot a 10×10 pixel image displaying a (noisy) corner. We then scatter the 2D points representing the gradients $\nabla \mathbf{I}(x, y)$ of each the 100 pixels (x, y) in the image, where the underlying edge detector is the 3×3 Prewitt filter. It can be seen that there is a cluster of points around the origin, representing many pixels for which the gradient along both axes is small – these are the pixels in flat areas of the image with little variation. The clusters around the points $(3, 0)$ and $(0, 3)$ represent the two edges in the image. Lastly, the points roughly in the center of the scatter, where both gradient components have a relatively high value, represent the corner. This occurs because the corner is an intersection of two edges in different directions, contributing to both the x-axis and the y-axis gradients.

Figure 4.1.1: A 10×10 pixel image depicting a corner with added Gaussian noise. The gradient vectors $\nabla \mathbf{I}(x, y)$ for each pixel (x, y), obtained by applying the 3×3 Prewitt filter, are scattered.



Eigenvalues and Eigenvectors of Quadratic Forms. Before applying image gradients to develop an algorithm for corner detection, let us revisit some fundamental properties of quadratic forms defined by a symmetric matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$:

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x} \in \mathbb{R}, \quad \forall \mathbf{x} \in \mathbb{R}^n.$$

Using spectral decomposition, we can express $\mathbf{A} = \mathbf{U} \boldsymbol{\Lambda} \mathbf{U}^T$, where $\boldsymbol{\Lambda} \in \mathbb{R}^{n \times n}$ is a diagonal matrix containing the eigenvalues of $\mathbf{A}$, and $\mathbf{U} \in \mathbb{R}^{n \times n}$ is an orthogonal containing the corresponding eigenvectors.

The level sets of the quadratic form f defined by $\mathbf{A}$ are ellipsoids (in n dimensions). The principal axes of these ellipsoids are aligned with the eigenvectors, while the intensity of the gradient along each axis is dictated by the eigenvalues. Specifically, we have:

$$\max_{\mathbf{x} \in \mathbb{R}^n} \|\nabla f(\mathbf{x})\| = \max_{\mathbf{x} \in \mathbb{R}^n} \left\|\mathbf{U} \boldsymbol{\Lambda} \mathbf{U}^T \mathbf{x}\right\| = \max_{\mathbf{y} \in \mathbb{R}^n} \|\boldsymbol{\Lambda} \mathbf{y}\| = \lambda_{\max}(\mathbf{A}),$$

where this maximum value is attained for the eigenvector corresponding to the largest eigenvalue. Similarly, $\min_{\mathbf{x} \in \mathbb{R}^n} \|\nabla f(\mathbf{x})\| = \lambda_{\min}(\mathbf{A})$ is attained for the eigenvector corresponding to the smallest eigenvalue.

4.1.1 PCA for Corner Detection

In the previous example, we used a scatter plot of the image gradients to identify a corner. However, to handle larger images and make the scatter analysis more straightforward, we need a more systematic method that formalizes these observations. By leveraging the insights from the image gradients, we can develop an algorithm based on PCA (refer to Section 3.1.2) to effectively detect corners in an image. This algorithm has five steps.

Step 1: Compute Image Gradients Over a Small Region. To detect a corner, we need to quantify how much the pixel intensity changes in a given local area. Mathematically, this change is captured by the image gradients as defined in (4.1.1). Large $\partial \mathbf{I}/\partial x$ indicates a significant horizontal intensity change, while large $\partial \mathbf{I}/\partial y$ indicates a significant vertical intensity change. If there is substantial change in both directions, it is indicative of a corner.

Step 2: Subtract the Mean from Each Image Gradient. Given a point (x, y) in the image, we denote the gradients of a small region (e.g., 5×5 window) around this point by $\nabla \mathbf{I}_i \in \mathbb{R}^2$, $i = 1, 2, \ldots, N$, where N is the number of pixels in the window (hence, we are taking the gradients of all corresponding points in the window). For ease of derivations, we center the gradients around their mean:

$$\nabla_i \mathbf{I} \coloneqq \nabla \mathbf{I}_i - \frac{1}{N}\sum_{j=1}^{N} \nabla \mathbf{I}_j \in \mathbb{R}^2, \quad \forall i = 1, 2, \ldots, N.$$

Intuitively, remember that our goal is to analyze the variance in pixel intensities. For instance, if we observe high variance in the gradients along the x-direction, it indicates a significant change in intensity in that direction. Subtracting the mean, as typically done in PCA, is helpful because it emphasizes the local variations in intensity, rather than focusing solely on the absolute gradient values.

Step 3: Compute the Covariance Matrix. To understand how the gradients vary in a local window, we compute the covariance matrix of the mean-centered gradients:

$$\mathbf{C}(x, y) \equiv \frac{1}{N}\sum_{i=1}^{N} \nabla \mathbf{I}_i \nabla \mathbf{I}_i^T = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{yx} & \sigma_{yy} \end{pmatrix} \in \mathbb{R}^{2\times 2},$$

where we denote

$$\sigma_{xx} \equiv \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial \mathbf{I}_i}{\partial x}\right)^2, \quad \sigma_{yy} \equiv \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial \mathbf{I}_i}{\partial y}\right)^2, \quad \text{and} \quad \sigma_{xy} = \sigma_{yx} \equiv \frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathbf{I}_i}{\partial x}\frac{\partial \mathbf{I}_i}{\partial y},$$

and the partial derivatives are with respect to the corresponding i-th pixel in the window.

The covariance matrix $\mathbf{C}(x, y)$ captures how the gradients vary together in the local region of (x, y). Its diagonal elements σ_{xx} and σ_{yy} measure the variance of the gradients in the x and y directions, respectively. Its off-diagonal element σ_{xy} measures variance in both x and y directions (their correlation).

Step 4: Compute Eigenvectors and Eigenvalues. To fully understand the nature of variations in an image, we need to identify both the principal directions and the magnitudes of these variations. This can be accomplished by calculating the eigenvectors and eigenvalues of the covariance matrix $\mathbf{C}(x, y)$. As previously discussed, if both eigenvalues are large, this indicates substantial variation in two orthogonal directions, which is characteristic of a corner. Conversely, if one eigenvalue is significantly larger than the other, it implies dominant variation in a single direction, suggesting the presence of an edge. If both eigenvalues are small, it means there is minimal variation, indicating a flat region.

It is also crucial to consider that if a corner has a very small angle, making it a sharp and acute corner, there may be a pronounced change in one direction but only a subtle change in the perpendicular direction. This can pose challenges for corner detection algorithms that depend on significant intensity changes in both directions.

Step 5: Use Threshold on Eigenvalues to Detect Corners. The final step is to apply a threshold to the eigenvalues to classify the type of point:

$$\lambda_i \left(\mathbf{C} \left(x, y \right) \right) > T, \quad i = 1, 2,$$

where $T \geq 0$ is a threshold that determines whether the variation is significant enough to qualify as a corner (recall that any covariance matrix is positive semi-definite). If both eigenvalues exceed the threshold, this indicates significant variation in both directions, suggesting the presence of a corner. In other words, when decomposing the intensity changes at the pixel to intensities along the x and y directions, we obtain eigenvalues that are both substantial. If one eigenvalue is large while the other is small, it signifies significant variation in just one direction, pointing to an edge. If both eigenvalues are small, it indicates minimal variation, characteristic of a flat region.

Run in Colab

In Figure 4.1.2, we implement all five steps of the PCA-based algorithm for corner detection. Initially, we use a 3×3 Prewitt filter to calculate the gradients at each pixel in both the x and y directions, working on the grayscale version of the image. Next, we consider a 5×5 window around each pixel in both gradient images, with the current pixel located at the center of these windows. This process results in two gradient images of the same dimensions as the original.

For each pixel, we collect the corresponding 25 gradient values from both directions within the window and compute their mean (note that this step can be skipped by adjusting the threshold accordingly). We then construct the covariance matrix, find its two eigenvalues, and apply thresholds of $T = 0.5$ and $T = 1.5$ to the minimum eigenvalue. Finally, any pixel that yields a minimal eigenvalue above the threshold is plotted in green over the original color image.

It can be observed that with a lower threshold, more corners are detected, though more pixels at each corner are classified as corner points. As the threshold increases, fewer corners are detected, but the precision of locating the exact corner pixels improves. Unfortunately, there is no universal rule of thumb for setting the threshold for a general image.

Figure 4.1.2: CCorner detection using a PCA-based algorithm. Gradients are computed with a 3×3 Prewitt filter, and the window size is to 5×5. The threshold is applied to the minimum eigenvalue, with values $T = 0.5$ and $T = 1.5$. Pixels with eigenvalues exceeding the threshold are highlighted in green.