Chapter 13

Image Processing

The idea behind the discrete-time signals we've seen so far is that of a well-ordered sequence of scalar values; whether the result of a series of measurements (sampling) or the output of a signal generator (synthesis), the resulting sequence unfolds "chronologically" as a function of an integer index $n \in \mathbb{Z}$. While this model captures extremely well a very large number of physical frameworks, there are in fact situations in which the set of values of interest is best represented by a function of a multidimensional index. The everyday example is provided by images, which can be modeled as a light intensity value over a set of points in the two-dimensional plane. In the digital world, an "image signal" becomes a discrete set of grayscale or color values, a.k.a. *pixels*, which are ordered in space rather than in time; the corresponding signal can be written as $x[n_1, n_2]$, with n_1 and n_2 denoting the horizontal and vertical discrete coordinates.

Note that, although in the rest of this chapter we will concentrate on images, i.e. on two-dimensional signals, higher-dimensional signals do exist and are indeed quite common. Video, for instance, is a sequence of still pictures; as such it can be modeled as a signal defined over a triplet of indices, two for the image coordinates in each video frame and one for the frame number. Another class of three-dimensional signals emerges from volume rendering, where the data represents units of volume (or *voxels*) in three-dimensional space. Four dimensions naturally appear in time-varying vol-

ume rendering applications common in biomedical imaging. Most of the elementary concepts developed for one-dimensional signals admit a "natural" extension to the multidimensional domain, in the sense that the extension performs according to expectations; a multidimensional Fourier transform, for instance, decomposes a multidimensional signal into a set of multidimensional oscillatory basis functions. As the number of dimensions grows, however, intuition begins to fail us and high-dimensional signal processing quickly becomes extremely abstract (and quite cumbersome notation-wise); as a consequence, in this introductory material, we will focus solely on two-dimensional signals in the form of images.

Finally, a word of caution. With respect to 1D signal processing, image processing often appears less formal and less self contained; the impression is not altogether unfounded and there are two main reasons for that. To begin with, images are very specialized signals, namely, they are signals "designed" for the unique processing unit that is the human visual system[1]. With some rare exceptions such as barcodes or QR-codes, in order to make sense of an image one needs to deploy a level of semantic analysis that profoundly transcend the capabilities of the standard signal processing toolbox; although linear processing is an almost universal first step, it so happens that a handful of simple tools are all that's necessary in the vast majority of cases. The second reason, which is not at all unrelated to the first, is that Fourier analysis does not play a major role in image processing. Most of the information in an image is encoded by its edges (as we learn in infancy from coloring books), but edges represent signal discontinuities that, in the frequency domain, affect mostly the global phase response and are therefore hard to see. Also consider this: a spectral magnitude is an invaluable *visual* tool to quickly establish the properties of a 1D signal; but images are *already* visual entities, and their spectra does not represet their information in a more readable manner.

[1] For instance, if humans had bat-like ranging capabilities, 2D signals wouldn't be so special and 3D depth maps would be all the rage.

13.1 Preliminaries

An $N_1 \times N_2$ digital picture is a collection of $M = N_1 N_2$ values, called *pixels*[2], usually arranged in a regular way to form a rectangular image. Scalar values for the pixels give rise to grayscale images, with each pixel's value normally constrained between zero (black) and some maximum value (white). The most commom quantization scheme allocates 8 bits per pixel, which gives 256 levels of gray to choose from; this is adequate in most circumstances for standard viewing. To encode color images, we first need to define a *color space*; the popular RGB encoding, for instance, represents visible colors as a weighed sum of the three primary colors red, green and blue, so that each color pixel is a triple of (quantized) coordinates in RGB space. Many other color spaces exist, each with their pros and cons, but since colorimetry (the theory and practice of color perception and reproduction) is a very complete and rich discipline unto itself, we will not attempt even a brief summary here. For all practical purposes we will simply consider a color image as a collection of C superimposed scalar images, where C is the number of coordinates in the chosen color space. For 24-bit RGB images, for instance, the red, green and blue images will be handled as three independent 8-bit grayscale images.

13.1.1 Pixel Arrangement

From the point of view of its information content, a grayscale $N_1 \times N_2$ image is just a collection of $M = N_1 N_2$ scalar values; consequently, the image could be "unrolled" into a standard one-dimensional M-point signal in $\mathbb{R}^M$ and we could revert to the "classical" signal processing techniques of the previous chapters. And, as a matter of fact, there are several applications that do precisely that: the storage or the transmission of raw image, for instance, involves a *scanning* operation that serializes the set of image pixels into a 1-D signal. Normally, the serialization takes place in a row-by-row fashion, i.e., the image is converted to a *raster scan* format. Classic reproducing devices such as printers or cathode-ray tubes perform the inverse operation

[2] From *pic*ture *el*ement. For an interesting historical account of the word see "A Brief History of Pixel", by Richard F. Lyon, 2006.

by stacking the 1-D data back into rectangular form one row at a time. Nevertheless, in "true" image processing applications, we will always use a 2-D representation indexed by two distinct coordinates; the fundamental concept here is that images possess *local correlations* in the two-dimensional spatial domain and that those correlations would be lost in the scanning process. As a simple example imagine a $N \times N$ black image with a single white line. A row-by-row raster scan would produce very different length-N^2 1D signals according to the line's orientation: if the line is vertical, we would obtain an extremely sparse signal where only one every N samples is nonzero; a horizontal line, on the other hand, would produce a signal in which only a block of N contiguous pixels are nonzero; other orientations would produce intermediate results. It's clear that the spatial structure of this simple image is completely obfuscated by a one-dimensional representation and that's why fully two-dimensional operators are necessary.

Normally pixels are arranged on a regular two-dimensional array of points on the plane. Although different arrangements are possible (each corresponding to different *point lattices* in the plane), here we will consider only the "canonical" grid associated to the regular lattice $\mathbb{Z}^2$, i.e. each pixel will be associated to a pair of integer-valued coordinates.

13.1.2 Visualization

Since images can be modeled as real-valued functions on the plane, a formal graphical representation would require a three-dimensional plot such as the one in Figure 13.2, which is a natural extension of the Cartesian plots of 1D signals as functions of discrete time. Of course this kind of representation is useful primarily for "abstract" 2D sequences, i.e. 2D signals that do not encode semantically intuitive visual information; examples of the latter include 2D impulse and frequency responses and signal transforms. Figure 13.2, for example, depicts a portion of a Gaussian impulse response used in image blurring (see Section 13.4.2); since the impulse response is obtained from a 2D function sampled on the grid, it makes sense to represent the signal in such an abstract fashion.

"Normal" images, on the other hand, are usually represented by plotting a graylevel dot at each pixel location, where the shade of grey encodes

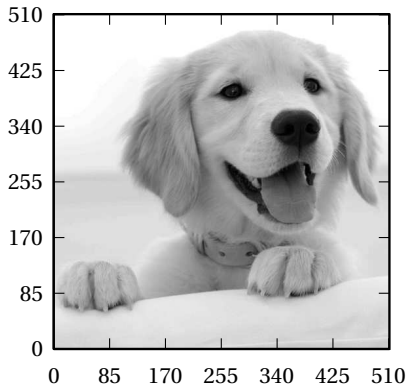


Figure 13.1: A prototypical image, shown as such.

the pixel's value; if the dots are closely spaced, the eye automatically interpolates the underlying pointwise structure and the plot gives the illusion of a "continuous" image[3]; perceived closeness is of course also dependent on viewing distance. A standard example of this "pictorial" representation of images is shown in Figure 13.1[4]. An important point to notice is that, since the range of gray levels available on print (or on a screen) is finite, most images will require a level adjustment in order to be readable. In fact, this is no different than rescaling a fixed-size axis in one-dimensional plots; in pictures, the representable range is normally standardized between zero, which is usually mapped to black, and one, which is mapped to white. Unless otherwise noted, all images in the rest of the chapter and scaled and shifted so that they occupy the full grayscale range. In other words, the represented signal is

$$y[n_1, n_2] = (x[n_1, n_2] - m)/(M - m)$$

[3] Of course, Pop artists (and Roy Lichtestein in particular) hit a gold mine when they realized that dots could be spaced far apart...

[4] For a nice and compact history of the "Lena" image please see `http://www.lenna.org/`; in spite of potential controversy, the author's feeling is that Lena is really the Mona Lisa of image processing, that is to say, a universally recognized icon that completely transcends its earthly begetting. As such, Lena is indeed quite perfect for a short introduction to image processing.

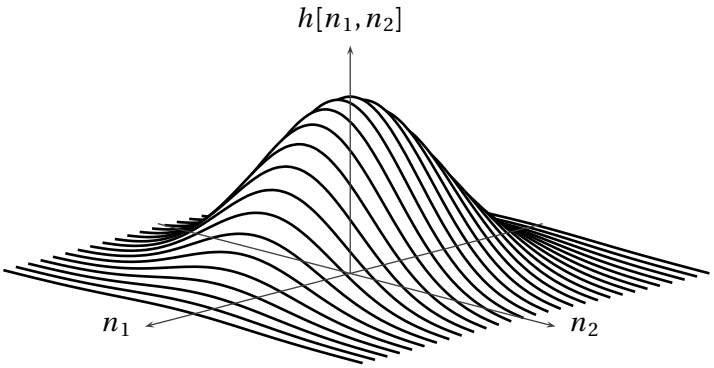


Figure 13.2: Example of two-dimensional Gaussian impulse response plotted as a 3D graph.

where $m = \min\{x[n_1, n_2]\}$ and $M = \max\{x[n_1, n_2]\}$. In Matlab, this rescaling is performed automatically when using the `imagesc` command.

A third, intermediate graphical option is given by "support plots". In these plots, we revert to a Cartesian viewpoint but we look at the 3D space "from the top", so to speak, and we only represent the nonzero values of a signal. The result is a 2D plot where the nonzero pixels appear as dots on the plane and examples can be seen in Figure . This representation focuses on the support of a signal and is particularly useful in describing finite impulse responses and other algorithmic techniques.

13.2 Basic Signals and Operators

The simplest two-dimensional signal is the 2D impulse:

$$\delta[n_1, n_2] = \begin{cases} 1 & \text{if } n_1 = n_2 = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{13.1}$$

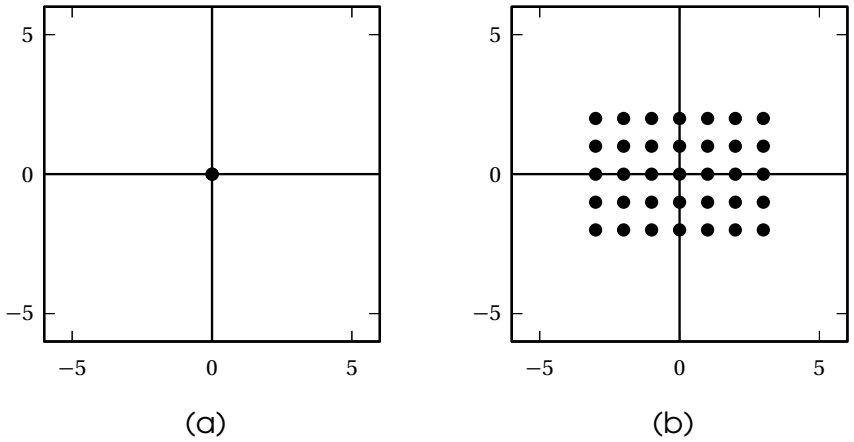


Figure 13.3: Two dimensional impulse (a) and rectangular signal for $N_1 = 4$ and $N_2 = 3$.

The 2D equivalent of a zero-centered rectangular signal is

$$\text{rect}\left(\frac{n_1}{2N_1}, \frac{n_2}{2N_2}\right) = \begin{cases} 1 & \text{if } |n_1| < N_1 \text{ and } |n_2| < N_2 \\ 0 & \text{otherwise;} \end{cases} \tag{13.2}$$

these two signals are represented in Figure 13.3 using support plots.

The impulse and the rect signals are examples of *separable* 2D signals, i.e. signals that can be written as the product of two independent 1D signals:

$$x[n_1, n_2] = x_1[n_1]x_2[n_2].$$

It is indeed obvious that

$$\delta[n_1, n_2] = \delta[n_1]\delta[n_1]$$

and that

$$\text{rect}\left(\frac{n_1}{2N_1}, \frac{n_2}{2N_2}\right) = \text{rect}\left(\frac{n_1}{2N_1}\right)\text{rect}\left(\frac{n_2}{2N_2}\right).$$

Although separable signals constitute a rather "artificial" subset of 2D signals, they are very convenient in conjunction with filtering operations, as we will see shortly. To realize that separability is entirely dependent on the reference system consider for instance the diamond-shaped signal

$$x[n_1, n_2] = \begin{cases} 1 & \text{if } |n_1| + |n_2| < N \\ 0 & \text{otherwise} \end{cases}$$

which shown in Figure 13.2-(a) for $N = 3$; clearly this is a 45-degree rotation of a square separable signal, yet the diamond signal is not separable. Similarly, the rectangular frame

$$x[n_1, n_2] = \text{rect}(n_1/2N, n_2/2N) - \text{rect}(n_1/2M, n_2/2M)$$

shown in Figure 13.2-(b) for $N = 4$ and $M = 2$, although obtained as a linear combination of separable signal is not itself separable.

The basic operations between 2D signals are the intuitive extension of their 1D counterparts. In particular, the convolution between two 2D signals, which defines the operation of processing an image with a linear, space-invariant system, is defined as

$$x[n_1, n_2] * h[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1, k_2] h[n_1 - k_1, n_2 - k_2]. \quad (13.3)$$

If $h[n_1, n_2] = h_1[n_1]h_2[n_2]$, i.e. if $h[n_1, n_2]$ is separable, we can rewrite (13.4) as

$$x[n_1, n_2] * h[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} h_1[n_1 - k_1] \sum_{k_2=-\infty}^{\infty} x[k_1, k_2] h_2[n_2 - k_2] \quad (13.4)$$

$$= h_1[n_1] * (h_2[n_2] * x[n_1, n_2]). \quad (13.5)$$

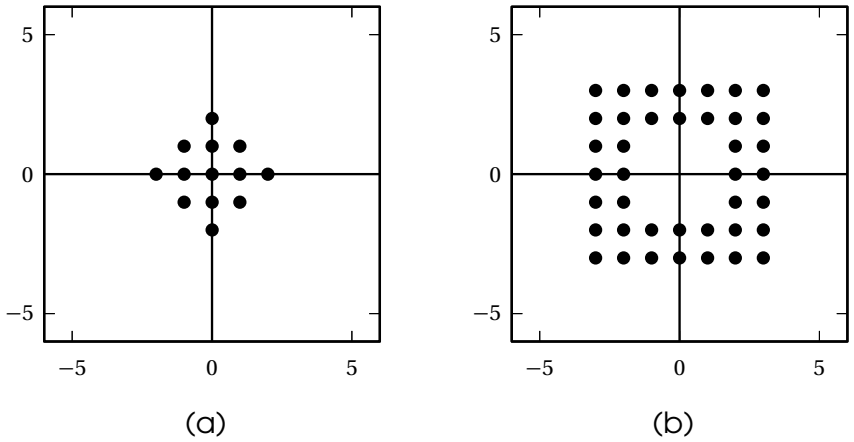


Figure 13.4: Diamond signal (a) and rectangular frame (b).

From the algorithmic point of view, therefore, the convolution with a separable signal is particularly simple since it becomes the cascade of two one-dimensional convolutions; stated differently, if $h[k_1, k_2]$ is a separable impulse response, the filtering operation is equivalent to filtering each image row with the first component of the impulse response, followed by filtering each column by the second component. In the case of FIR filters, this reduces the number of operations *per output sample* from $M_1 M_2$ to $M_1 + M_2$, where $M_1 \times M_2$ is the size of the filter's support.

13.2.1 Translation, Scaling and Rotations

An arbitrary function of two *real* variables $x(t_1, t_2)$ can be reshaped via an affine transform of the underlying coordinate system. An affine transform is defined by the formula

$$\begin{bmatrix} t_1' \\ t_2' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} - \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \mathbf{A} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} - \mathbf{d} \tag{13.6}$$

where **A** defines a linear mapping of the reference system while the vector **d** determines a translation of the origin. Particular forms for **A** and **d** give rise to special cases:

- **Translation:** if $\mathbf{A} = \mathbf{I}$, the result is a simple translation of the origin to the point (d_1, d_2).
- **Scaling:** if $\mathbf{A} = \text{diag}(a_1, a_2)$ the function is rescaled by a_1 and a_2 along the two dimensions. If $a_1 = a_2$ the scaling is uniform for both axes and the rescaling is said to preserve the *aspect ratio* of the original function.
- **Rotation:** if **A** is of the form

  $$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

  the result is a counterclockwise rotation around the origin by an angle θ.
- **Shearing:** if **A** is of the form

  $$\mathbf{A} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$

  the result is a horizontal shearing of the function, i.e. a horizontal slanting of the support aligned with the line $t_2 = s\,t_1$. The transpose of the above matrix produces a vertical shear.
- **Flips:** the matrix

  $$\mathbf{A} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

  flips the function around the t_2 axis (horizontal flip) while the matrix

  $$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

  flips the function around the t_1 axis (vertical flip)

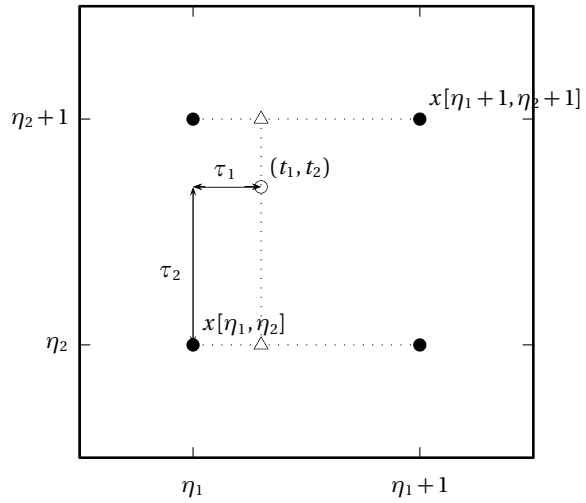


Figure 13.5: Bilinear interpolation framework.

In the case of a digital image $x[n_1, n_2]$ the coordinate indices must be integers so that, in principle, we can only use affine transforms for which $(\mathbf{A}[n_1, n_2]^T - \mathbf{d}) \in \mathbb{N}^2$. This limits our toolbox to integer translations, subsampling (scaling) by an integer factor, flips and rotations by multiples of 90 degrees only. A way out of the impasse is to consider a continuous-space interpolation of the digital image; the resulting function of two real variables can be then arbitrarily transformed and the result can be sampled back on the integer grid. Interpolation of 2D signals is conceptually identical to the 1D case, leading to a global 2D sinc interpolation formula and an associated sampling theorem for bandlimited 2D functions. Instead of following that route, however, we will concentrate here on purely algorithmic (and therefore *local*) interpolation schemes that allow us to apply arbitrary affine transforms to a digital image.

Given an $N_1 \times N_2$ image $x[n_1, n_2]$ and an $(\mathbf{A}, \mathbf{d})$ pair, call $y[m_1, m_2]$ the image we would obtain if we could apply the associated affine transform to x. The key idea is to consider the *inverse* transformation, which maps each

pixel coordinate in y back to the space where x lives:

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \mathbf{A}^{-1} \begin{bmatrix} m_1 + d_1 \\ m_2 + d_2 \end{bmatrix};$$

here t_1 and t_2 are obviously real variables and, as such, they will not fall on the original integer grid. If either t_1 or t_2 fall outside the $[0, N_1 - 1]$ and $[0, N_2 - 1]$ respectively, then we simply set $y[m_1, m_2] = 0$ (or any other pre-arranged value) and we're done. Otherwise we can express the coordinate pair as

$$(t_1, t_2) = (\eta_1 + \tau_1, \eta_2 + \tau_2)$$

with $\eta_{1,2} \in \mathbb{N}$ and with $0 \leq \tau_{1,2} < 1$. This notation highlights the fact that the coordinate pair points to a "subpixel" located at a distance (t_1, t_2) from $x[\eta_1, \eta_2]$ and contained within a four-pixel square bounded by $x[\eta_1, \eta_2]$ and $x[\eta_1 + 1, \eta_2 + 1]$ (see Figure 13.2.1). A popular method to obtain an estimate of the value for this subpixel is the *bilinear interpolation* algorithm, a straightforward extension of the first order interpolator in the one-dimensional domain. Imagine applying the 1D triangular interpolator a row in the image; the result is a straight line interpolation between neighboring pixels in the line so that for each τ between zero and one we can write

$$x_c(\eta_1 + \tau_1, n_2) = (1 - \tau_1)x[\eta_1, n_2] + \tau_1 x[\eta_1 + 1, n_2].$$

We can compute the above formula for $n_2 = \eta_2$ and $n_2 = \eta_2 + 1$ in order to obtain the interpolated points indicated by triangles in Figure 13.2.1; we can then apply the same first order interpolation column-wise, and interpolate between the "triangles" to obtain:

$$x_c(\eta_1 + \tau_1, \eta_2 + \tau_2) = (1 - \tau_2)x_c(\eta_1 + \tau_1, \eta_2) + \tau_2 x_c(\eta_1 + \tau_1, \eta_2).$$

By combining the two expressions we finally obtain the bilinear estimate for the transformed image as

$$\begin{aligned} y[m_1, m_2] = \quad & (1 - \tau_1)(1 - \tau_2)x[\eta_1, \eta_2] + \tau_1(1 - \tau_2)x[\eta_1 + 1, \eta_2] \\ & + (1 - \tau_1)\tau_2 x[\eta_1, \eta_2 + 1] + \tau_1\tau_2 x[\eta_1 + 1, \eta_2 + 2] \end{aligned} \quad (13.7)$$

which is a linear combination of the values of the pixels surrounding the target coordinate obtained by inverting the affine transform. As for all local interpolation schemes, many other methods exist to estimate the value of the subpixel, each with interesting properties and increasing complexity; it should be clear from the more complete analysis conducted in one dimension that in all cases we are dealing with localized approximations to an ideal sinc interpolation, affine transform and resampling.

13.3 Fourier Transforms

13.3.1 2D-DFT

As we have seen, the space of $N_1 \times N_2$ images is isomorphic to $\mathbb{C}^{N_1 N_2}$ (although we will only deal with real-valued images, so in fact we could consider $\mathbb{R}^{N_1 N_2} \subset \mathbb{C}^{N_1 N_2}$). Amongst all possible bases for $\mathbb{C}^{N_1 N_2}$, the Fourier basis holds a preeminent position and allows us to map an $N_1 \times N_2$ signal into a complex $N_1 \times N_2$ signal via the 2D-DFT as:

$$X[k_1,k_2] = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x[n_1,n_2] e^{-j\frac{2\pi}{N_1}n_1k_1} e^{-j\frac{2\pi}{N_2}n_2k_2}. \tag{13.8}$$

The set of N_1N_2 basis vectors

$$w^{k_1,k_2}_{N_1,N_2}[n_1,n_2] = e^{j\frac{2\pi}{N_1}n_1k_1} e^{j\frac{2\pi}{N_2}n_2k_2}. \tag{13.9}$$

forms an orthogonal set in $\mathbb{C}^{N_1N_2}$:

$$\sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} w^{k_1,k_2}_{N_1,N_2}[n_1,n_2] w^{h_1,h_2}_{N_1,N_2}[n_1,n_2] = N_1N_2\delta[k_1-h_1,k_2-h_2] \tag{13.10}$$

so that the 2D-DFT can be inverted as

$$x[n_1,n_2] = \frac{1}{N_1N_2} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} X[k_1,k_2] e^{j\frac{2\pi}{N_1}n_1k_1} e^{j\frac{2\pi}{N_2}n_2k_2}. \tag{13.11}$$

The basis vectors are planar oscillatory components whose frequency in the vertical and horizontal directions are determined by the indices k_1 and k_2. Some examples are displayed in Figures 13.3.1 to 13.3.1; the figures plot the magnitude of the basis vectors by mapping the maximum value (1) to white and the minimum value (0) to black. Note that the Fourier basis is separable and that therefore the 2D-DFT can be computed efficiently as the cascade of a row-by-row 1D transform followed by a column-by-column transform. This algorithmic advantage is however somewhat offset by the fact that separability privileges the directions parallel to the vertical and horizontal axes which, in natural images, are not "special" directions at all; as a consequence, the basis is somewhat ill-equipped to encode naturally occurring patterns.

13.3.2 Magnitude and Phase

Just as the ear is largely insensitive to phase information in sound, the opposite happens with the eye. To understand why, consider that most of the semantic information in a scene can be almost always intelligibly summed up by an outline, i.e. by the edges of the objects contained in the picture. As we have seen several times by now, a sharp transition in a signal is encoded mostly in the phase domain, since the wavefronts of the underlying Fourier basis vectors must align properly in order for their weighted sum to transition sharply. This is immediately apparent if we perform the following experiment: take an image, compute its DFT and then try to reconstruct it either only from the magnitude spectrum or from the phase information. The results are shown in Figure 13.9 where the picture on the left depicts

$$x_m[n_1, n_2] = \Re\{\text{IDFT}\{|X[k_1, k_2]|\}\}$$

while the image on the right shows

$$x_p[n_1, n_2] = \Re\{\text{IDFT}\{e^{j\angle X[k_1,k_2]}\}\}$$

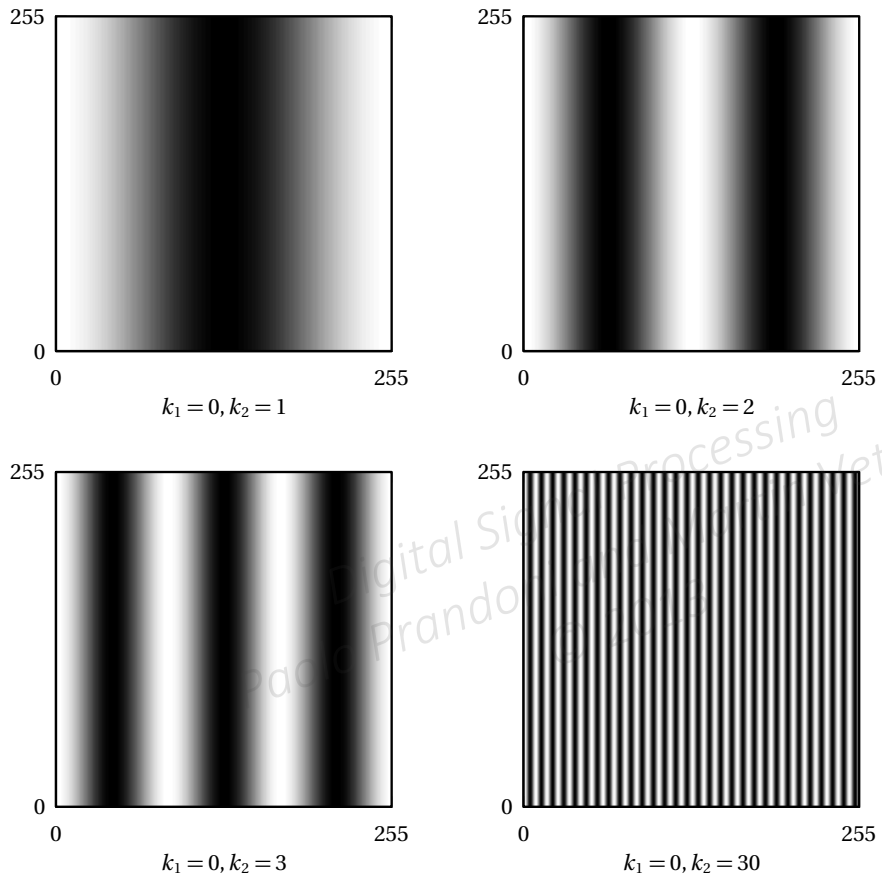


Figure 13.6: Some basis vectors for $\mathbb{C}^{256\times256}$ (real part).

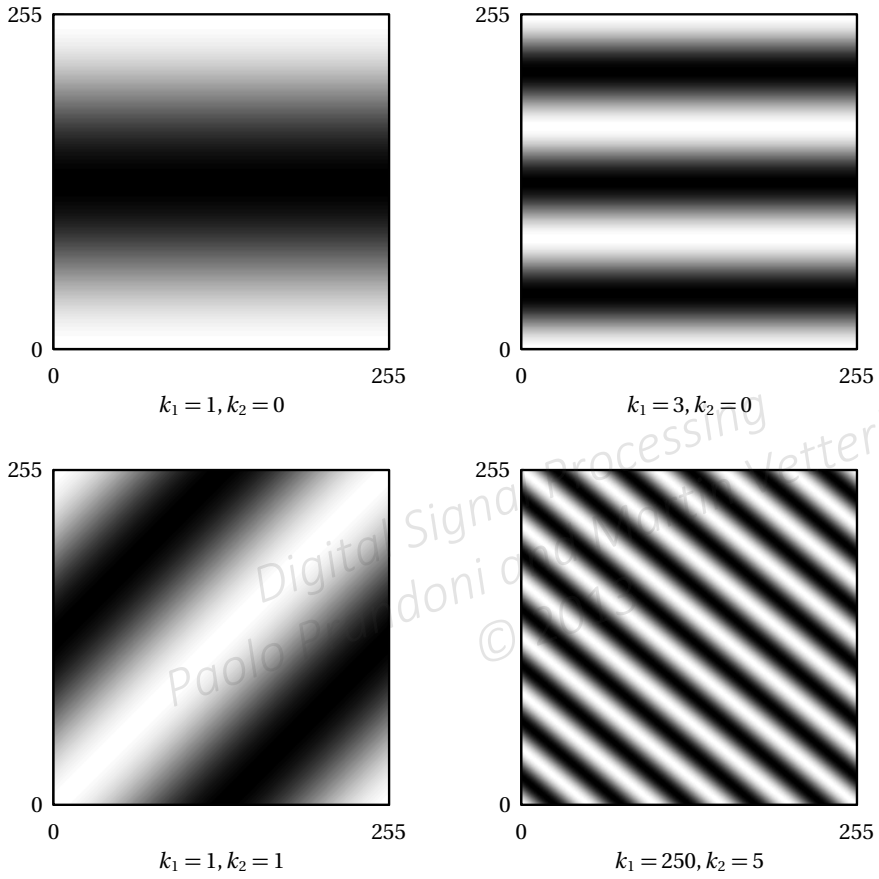


Figure 13.7: Some basis vectors for $\mathbb{C}^{256\times256}$ (real part).

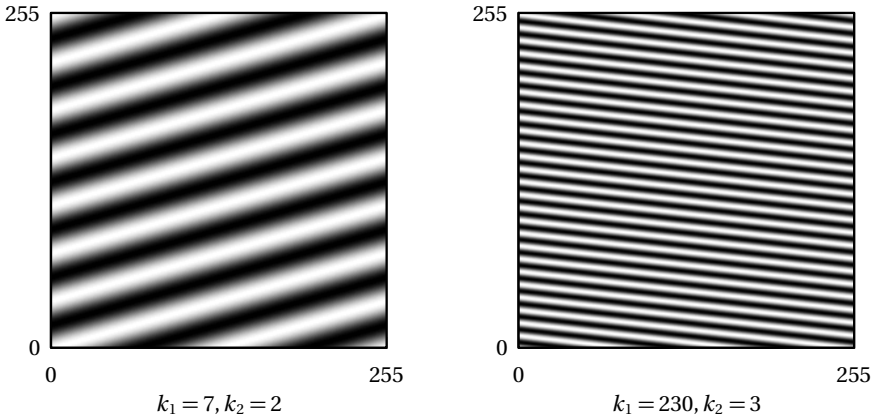


Figure 13.8: Some basis vectors for $\mathbb{C}^{256\times256}$ (real part).

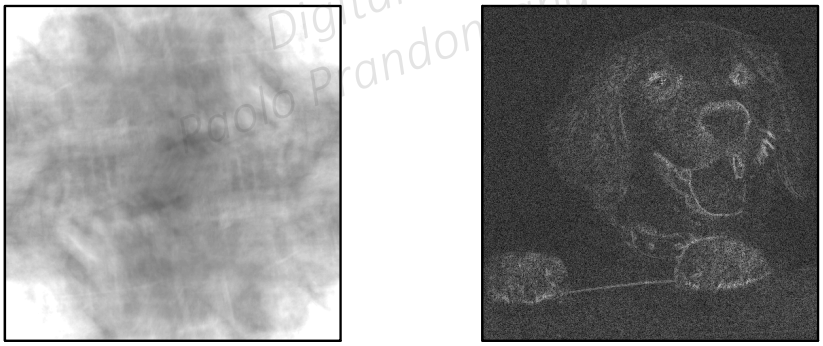


Figure 13.9: Image reconstruction from only magnitude and phase

13.3.3 2D-DTFT

For theoretical derivations involving infinite-size images, the DTFT can be extended to the two-dimensional domain as

$$X(\omega_1,\omega_2)=\sum_{n_1=-\infty}^{\infty}\sum_{n_2=-\infty}^{\infty}x[n_1,n_2]e^{-j(\omega_1 n_1+\omega_2 n_2)} \tag{13.12}$$

The above expression, when it exists, maps an infinite two-dimensional discrete signal to a complex function of two real variables, 2π-periodic in both arguments. It can be inverted via the formula

$$x[n_1,n_2]=\frac{1}{(2\pi)^2}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi}X(\omega_1,\omega_2)e^{j(\omega_1 n_1+\omega_2 n_2)}d\omega_1 d\omega_2. \tag{13.13}$$

13.4 Image Filtering

To begin, a cautionary note: an unfortunate reality in image processing is that very little that can be accomplished by means of linear operators only; a filtering operation is usually just the preparatory step that preludes to some nonlinear and very often adaptive treatment. The reason for this is actually rather easy to understand: an image is a finite-size signal that depicts widely different types of information in different areas. Think of a photo of a person in front of a car in front of a building: the variability in types of surface, edge curvature, textures and so on is extremely large. A filter, by definition, is supposed to be linear and *space-invariant* and therefore only very simple and very "mild" filters can be applied to the entirety of the data without negatively affecting one or the other of the several coexisting regions. As a consequence, the design of 2D image filters, although a theoretically rich and challenging subject, remains a much more specialized discipline than its one-dimensional counterpart.

13.4.1 Filter Types

Just as in the one-dimensional case, two-dimensional filters can have a finite or an infinite support, they can be causal or not, and they can be classified in terms of their frequency response. Generally speaking, by eliminating or attenuating the high frequency components, **lowpass** filters perform a smoothing operation; in the case of images, this correspond to *blurring* the visual content: edges lose their sharpness and small details become less visible. **Highpass** filters, by contrast, reduce the low-frequency components; in images, this means that uniform areas are eliminated while the fast-varying regions are preserved. Since edges are points of discontinuity in an image and therefore possess a high frequency content, highpass filtering is a key step in edge detection.

In image processing, in contrast to one-dimensional signal processing, causality is hardly ever a concern. Images are finite-size entities and, as such, we can rely on the fact that they will be available in full when processing begins; the convolution sum can therefore be computed in any chosen direction (row by row, or column by column, right to left or left to right).

In "everyday" image processing, IIR filters are not very common because of four main difficulties:

- **Linear phase.** In images linear phase is very important since, as we have seen, most of the visual information is contained in edges and a sharp edge require precise phase alignment. As for their 1D counterparts, IIR filter with exact linear phase do not exist and the design of approximately linear phase IIRs can be difficult.
- **Border effects.** In one dimension, a finite-length signal has only two boundary points, ie its beginning and its end; as a consequence, even for moderately long signals, border effects can be safely neglected since they affect only a minute percentage of the whole data set. For finite-size images, on the other hand, the number of boundary points is proportional to the square root of the total number of pixels; in this case, border effects become much more significant; one way to combat the problem is to use FIR filters with a small support so that only a few pixels around the border are affected.

- **Stability Issues.** Multidimensional CCDE's are more complex entities than their one-dimensional counterparts; multivariate polynomials do not factor into first-order terms so that a multidimensional z-transform representation does not lead to either a convenient stability test via root checking nor to simple inversion formulas via partial fraction expansion.

- **Computability.** An arbitrary multidimensional CCDE may not be computable: consider for instance the filter defined by

$$y[n_1,n_2] = a_0 y[n_1+1,n_2] + a_1 y[n_1,n_2-1] + \\ a_2 y[n_1-1,n_2] a_1 y[n_1,n_2+1] + x[n_1,n_2];$$

  each output sample depends on four neighboring output samples (see Figure 13.10) which, in turn, depend on the current output. There is no order in which the terms can be rearranged to provide a computable algorithm and therefore the system described by the above CCDE is not implementable in practice.

For these reasons, most filters used in practice are small-support FIRs while the design and implementation of IIRs remains the province of more specialized papers and applications.

13.4.2 Lowpass Filtering

Moving Average. The straightforward 2D extension of the moving average filter is the separable FIR

$$h[n_1,n_2] = \frac{n_1}{(N+1)^2}\text{rect}\left(\frac{n_1}{2N},\frac{n_2}{2N}\right)$$

which has a square support; again, in image processing causality is rarely a concern so that we can express the filter as centered around the origin. Each output pixel $y[n_1,n_2] = x[n_1,n_2] * h[n_1,n_2] =$ by averaging the the original image over a square of side $N+1$ pixels centered around (n_1,n_2). Examples of MA filtering are shown in Figure 13.11.

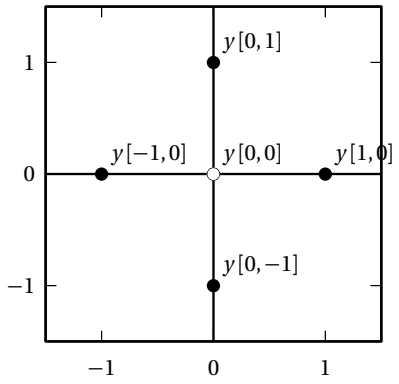


Figure 13.10: Noncomputable 2D IIR: computing the value in $(0,0)$ depends on four neighboring output values.

Figure 13.11: Moving average blurs with a 11×11 and a 51×51 FIR.

Gaussian Filter. A preferred lowpass alternative to the moving average is the separable Gaussian filter

$$h[n_1, n_2] = \frac{1}{2\pi\sigma^2} e^{-\frac{n_1^2+n_2^2}{2\sigma^2}}$$

which has a smoother frequency characteristic (the Fourier transform of an Gaussian function is itself a Gaussian). Although a Gaussian response is nonzero over the entire real plane, a common approach is use a square-support FIR approximation with a width of approximately 3σ from the mean, which ensures that over 99% of the total filter's energy is represented. For example, if $\sigma = 1$ we would use the 5×5 FIR with values

$$h[n_1, n_2] = \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \cdot \frac{1}{273}$$

where the taps have been normalizes so that $\sum_{n_1} \sum_{n_2} h[n_1, n_2] = 1$. Note how, in the case of FIR filters, we can specify the impulse response as a $N_1 \times N_2$ matrix with the assumption (unless otherwise noted) that the center value of the matrix coincides with the coordinate origin. Some examples of Gaussian blur are shown in Figures 13.12 and 13.13.

13.4.3 Higpass Filtering

Sobel Operators. Sobel filters are used to obtain an estimate of the first derivative of the image along n_1 and n_2. The *actual* first derivative, intended as the sampled first derivative of the underlying continuous-space 2D signal, could be computed exactly by applying the ideal differentiator derived in Section 9.7.1 to the rows and the columns of the image. More practically, however, the Sobel *horizontal* Sobel operator approximates the first deriva-

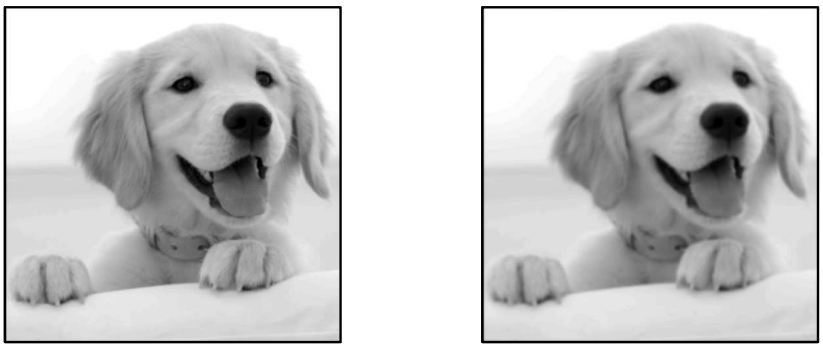


Figure 13.12: Gaussian blur: $\sigma = 1$ (5×5 FIR) and $\sigma = 1.8$ (9×9 FIR).

Figure 13.13: Gaussian blur: filter's impulse response for $\sigma = 5$ (29×29 FIR) and filtering result.

tive along n_1 via the impulse response

$$s_o[n_1,n_2]=\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}.$$

The response is separable and we can write the above matrix as

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

this shows the operator to be the cascade of a vertical three-point weighted average and a horizontal three-point truncation of the ideal differentiator's impulse response. The vertical averaging prior to differentiation ensures that the operator is less sensitive to noise while still responsive to magnitude jumps. Similarly, the *vertical* Sobel operator approximates the first derivative along n_2 via

$$s_v[n_1,n_2]=\begin{bmatrix}-1 & -2 & 1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix}.$$

Using the output of both filters, the magnitude of the gradient of an image $x[n_1,n_2]$ can be estimated via the nonlinear operator

$$|\nabla x|^2=|s_o*x|^2+|s_v*x|^2 \tag{13.14}$$

Since the gradient is large across jumps in signal magnitude, its local maxima can be used to discover edges in the image. An example of the gradient estimate obtained via the Sobel operator is shown on the left of Figure 13.14; a common approach to edge detection would begin by thresholding this image as

$$y[n_1,n_2]=\begin{cases}0 & \text{if } x[n_1,n_2]>T\\ 1 & \text{otherwise}\end{cases} \tag{13.15}$$

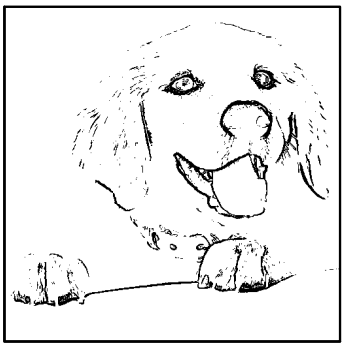


Figure 13.14: Sobel gradient estimation: gradient image and thresholded gradient ($T = 80$).

which yields the picture on the right.

Laplace Operator. The Laplacian, in the case of a functions of two continuous variables, is defined as the sum of all the second partial derivatives with respect to each free variable:

$$\Delta f(t_1, t_2) = \frac{\partial^2 f}{\partial t_1^2} + \frac{\partial^2 f}{\partial t_2^2}$$

In the discrete-space case, we can approximate the Laplacian operator as follows. First, in order to derive a compact approximation to the second partial derivative, consider the Taylor series expansion of a function of a real variable:

$$f(t+\tau) = \sum_{n=0}^{\infty} \frac{f^{(n)}(t)}{n!} \tau^n$$

If we write out the expansions for $f(t+\tau)$ and $f(t-\tau)$, stopping at the second order term, we have

$$\begin{aligned} f(x+\tau) &= f(t)+f'(t)\tau+\tfrac{1}{2}f''(t)\tau^2 \\ f(x-\tau) &= f(t)-f'(t)\tau+\tfrac{1}{2}f''(t)\tau^2 \end{aligned}$$

the sum of which yields the approximation

$$f''(t)=\frac{f(t-\tau)-2f(t)+f(t+\tau)}{\tau^2}.$$

On the discrete-space grid $\tau=1$ and we can therefore approximate the second derivative with the filter response $\begin{bmatrix}1 & -2 & 1\end{bmatrix}$. By summing the horizontal and vertical approximations we finally obtain the nonseparable Laplacian filter:

$$h[n_1,n_2]=\begin{bmatrix}0 & 1 & 0\\ 1 & -4 & 1\\ 0 & 1 & 0\end{bmatrix}.$$

A common variation, which improves the rotational symmetry of the operator, includes the approximation of the derivatives along the 45-degree diagonals:

$$h[n_1,n_2]=\begin{bmatrix}1 & 1 & 1\\ 1 & -8 & 1\\ 1 & 1 & 1\end{bmatrix}.$$

The Laplacian is another useful tool in edge detection; indeed, since edges correspond to sharp amplitude transition, the "curvature" of the image signal will exhibit sharp positive and negative peaks around the edges. An example of image processing using the Laplacian filter is shown in Figure 13.15; the first figure shows the absolute value of the Laplacian, while the second figure shows the image obtained by thresholding the first. The figure clearly shows that the Laplacian operator is particularly sensitive to noise, which results in a lot of disconnected pixels in the thresholded image. A way to combat this problem is to precede the Laplacian operator by a Gaussian blur.

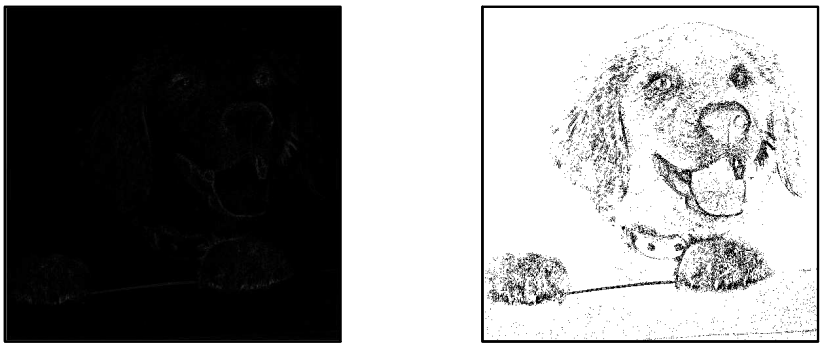


Figure 13.15: Laplacian operator: magnitude and thresholded magnitude ($T = 50$).

13.5 Image Compression

Images that make sense both visually and culturally (let's call them "natural images") are extremely redundant signals. Take for instance a 256×256 grayscale image, with 8 bits per pixel. In order to store the raw data contained in the image we need $B = 524,288$ bits; with these many bits per image, the number of *all possible* 256×256 images is a staggering $2^B \approx 10^{157,826}$. Compare this to the number of atoms in the universe (which is estimated at 10^{82}) and it's clear that the set of "reasonable" images must be an extraordinarily thin slice of the whole pie, so to speak. If the number of meaningful images is therefore $M \ll 2^B$, ultimately we should be able to encode each image with a number of bits much smaller than B; image compression algorithms try to achieve this data rate reduction in practical ways.

To pursue this line of thought further, assume we could enumerate exhaustively all M possible natural images (and let's assume just for the sake of argument that all such images have the same size). In this "encyclopedia" of pictures, each entry can be uniquely identified by its cardinal number in the series, i.e. by at most $B' = \log_2 M$. Two people needing to exchange

some visual content would need only to communicate the image's index to each other. Can we obtain even a rough estimate of M? One possible answer is to prosaically consider the total number of images currently available on the Internet, which (in 2012) is estimated at around 50 billion, i.e. less than 10^{10}. If we again neglect size, color encoding etc. (which do not change much the end result of this analysis) we can say that an exhaustive compression scheme by enumeration would yield an encoding scheme in which $B' \approx \log_2 10^{10} \approx 33$ bits[5], which is *four orders of magnitude* less than the raw encoding strategy.

Although this encoding minimizes the number of bits needed to identify an image, it is far from practical since it requires each user to possess an enormous amount of side information, namely, the very encyclopedia in which to look up each image. Therefore, practical compression schemes try to exploit the "physical" nature of the redundancy in image signals rather than the ensemble statistical properties of all possible image realizations. For example, most natural images exhibit patches of rather uniform intensity separated by edges; since edges are much more important semantically than the exact intensity value of a uniform patch, most of the bits used for encoding an images should be spent there. This prioritization of what to encode leads to *lossy* compression schemes in which a certain bit budget is strategically allocated to visually important features in the image; the process is fine-tuned with respect to the human visual system in order to minimize the unavoidable compression artifacts.

13.5.1 JPEG

JPEG is probably the most common image compression method in use today; the name is an acronym for "Joint Photographic Experts Group," a moniker for the committee that oversaw its formalization in 1994. Although more recent compression schemes can boast better performance, JPEG still offers very respectable results: a color image encoded at a cost of 2 bits per pixel (bpp), for instance, is virtually undistinguishable from the 24bpp original and a compression level of half a bit per pixel is generally adequate for

[5] and of course even less if, as seems likely, the occurrence probability of each image were far from uniform.

most images. Although JPEG does include a lossless encoding procedure, the standard operating mode is lossy and increasing compression levels are achieved by discarding more and more visual information in ways that are designed to minimize the associated visual impact.

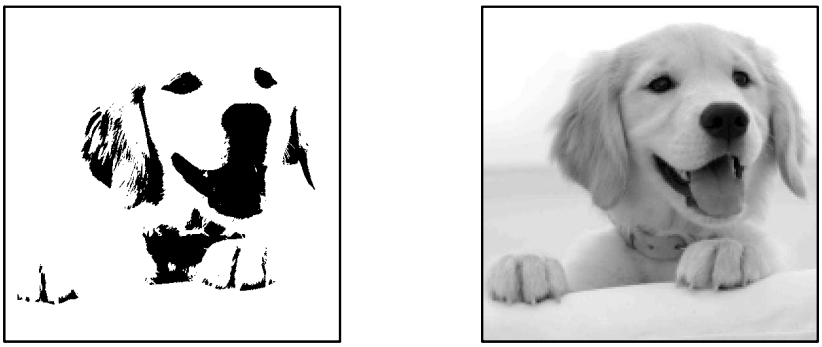
JPEG belongs to the class of *transform coders* in the sense that compression takes place after a change of basis for the 2D signal. The process can be very broadly summarized as follows:

1. the image is locally transformed using a DCT;
2. the DCT coefficients are thresholded and small coefficients are discarded (lossy compression step);
3. the remaining coefficients are efficiently encoded to further reduce the final size.

The compression scheme reflects a general paradigm in image processing where a linear step (the DCT) is followed by nonlinear manipulations. In the next paragraphs we will examine each step in more detail for the simple case of a grayscale image.

DCT. To understand the usefulness of a transform step in a lossy coder we need to consider the fact that the bulk of the achieved compression comes from re-quantizing the encoded data with a coarser quantizer, ie. by assigning fewer bits to each pixel. Now, suppose we want to re-quantize a grayscale image at just one bit per pixel; by using a uniform quantizer with two quantization levels on each pixel we can obtain a bilevel image such as the one shown in Figure 13.16-(a). On the other hand, consider the following strategy: divide the image into contiguous 3-by-3 blocks of 9 pixels each and encode the *average* value of the block with 8 bits; the resulting rate is slightly lower than one bit per pixel and the result is shown in Figure 13.16-(b). Since the average of each block is the normalized first Fourier coefficient, the strategy can be seen as a rudimentary transform coding.

Full-fledged transform coding perfects the concept by using a suitable change of basis for the data. Images are real-valued (integer-valued, actually, since we're considering digital images with a finite range of gray levels),



(a) (b)

Figure 13.16: Images coded at 1bpp: pixel-based one-bit quantization (a) and 3x3 block-based quantization.

so it makes sense to use a real-valued transform. In JPEG, the tool of choice is a two-dimensional DCT (Discrete Cosine Transform) which we have encountered in Example 4.3 for the one-dimensional case; the 2D extension to a square $N \times N$-point dataset is straightforward:

$$C[k_1,k_2] = \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} \cos\left[\frac{\pi}{N}\left(n_1+\frac{1}{2}\right)k_1\right] \cos\left[\frac{\pi}{N}\left(n_2+\frac{1}{2}\right)k_2\right]$$

Since images exhibit drastic variability, the transform is not applied to the whole data set; rather, the image is split in 8×8-pixel blocks and each block is transformed independently using an 8×8 DCT. Thanks to the manageable size of an 8×8 transform, we can actually easily plot the 64 DCT basis vectors in use, which are shown in Figure 13.17; perhaps more so here than for any other transform, the right way to look at the 2D-DCT is to focus on the fact that it is a correlation, i.e. a measure of similarity, between the image block and each basis vector. In the "recipe" for JPEG decoding, if the basis vectors are the ingredients, then the DCT coefficients represent the amounts to use: the first coefficient encodes the average grayscale value of the block, while

successive coefficients add higher-frequency details. Most images are rather smooth at a 8×8 scale and therefore the higher-order DCT coefficients are usually very small; this is the key fact that is exploited by the quantization step of the JPEG algorithm. On the other hand, if the quantization is too drastic, the boundaries between neighboring blocks become apparent and give images the typical "blockiness" of low-quality JPEG.

To sum up, the transform step modifies the original data set in three fundamental ways

- it creates a *hierarchical* representation of the data in which, starting from the average graylevel of a block, further details are added by successive coefficients;
- it decorrelates the information in neighboring pixels by finding structures and patterns that are encoded by the basis vectors across the entire block;
- it generally compacts the visual information of the block since most blocks will be smooth and only have a few significant coefficients.

Quantization. The second crucial step in JPEG compression is the quantization of the DCT coefficients; by using different bitrates for different coefficients, large savings can be obtained at a relatively modest price in terms of image distortion. The fundamental idea is to change the quantization step as a function of the visual significance of the associated DCT basis vector. To this end, quantization tables have been determined by means of psychovisual experiments: one such table, as described in the JPEG standard, is

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix};$$

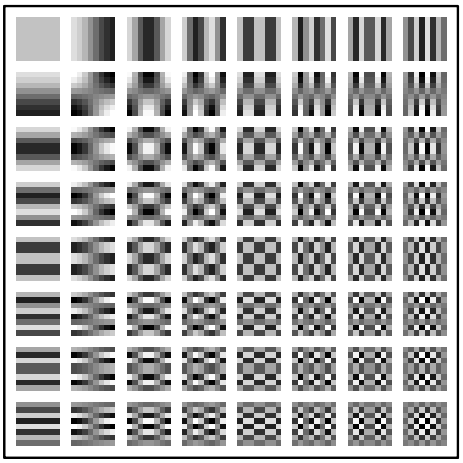


Figure 13.17: The DCT basis vectors for $R^{8\times 8}$

element m_{hk} in the above matrix represents the quantization step for DCT coefficient $C[h,k]$. Note that the levels are not uniform across the table, highlighting the relative visual importance of the different basis vectors. As an example, consider the detail in Figure 13.19, where the original image has been encoded at a rather low quality to enhance the issue; on the left we used the above table and on the right a uniform table designed to have the same bitrate. Clearly, a perceptually weighted allocation improves the visual quality, especially at low rates.

Note that quantization is performed as

$$\hat{C}[h,k] = \text{round}\left(\frac{C[h,k]}{m_{hk}}\right),$$

i.e. with a *dead zone* quantizer, which is a little different from the model studied in Chapter 10. An example is shown in Figure 13.18, where a normal quantization characteristic is shown on the left and a deadzone characteristic is shown on the right. Although the deadzone quantizer is not symmetric (we can see that, for 2 bits, only 3 levels are encoded) it has the advantage that all small values fall into the interval around the origin and are therefore

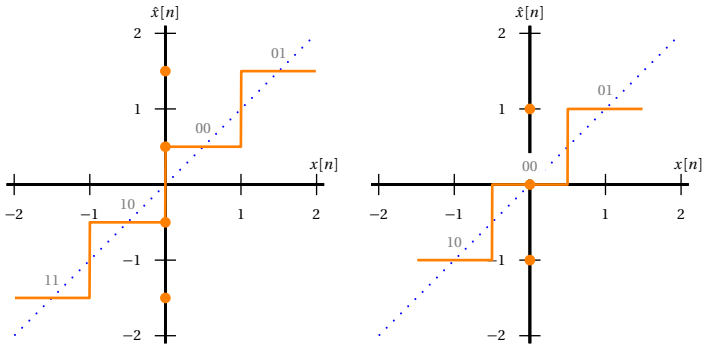


Figure 13.18: Standard and deadzone quantizers with unit step; 2-bit output (two's complement).

"killed" to a value of zero (hence the name). Having long runs of zero-valued coefficients is the key ingredient for the further bitrate reduction achieved by the successive step of entropy coding.

Different level of encoding quality can be attained by changing the quantization table; low quality level will specify very large step sizes for all but a few coefficients (thereby resulting in very few nonzero values and hence a very low bitrate) and, conversely, high quality levels can be achieved with smaller quantization steps. The quantization table is stored along with the data so that the JPEG decoder can reconstruct the image; this allows different implementations of the encoder to adopt different (and possibly quite sophisticated) strategies in defining the tables in use.

Entropy Coding. The final step in the JPEG algorithm produces a compact representation for the quantized DCT coefficients. An exhaustive treatment of entropy coding is clearly beyond the scope of this introduction, but we will illustrate the basic principles from an intuitive point of view. The two fundamental ingredients in obtaining a compact bitstream are runlength encoding followed by Huffman encoding.

The quantized DCT coefficients are "unwrapped" following the zigzag

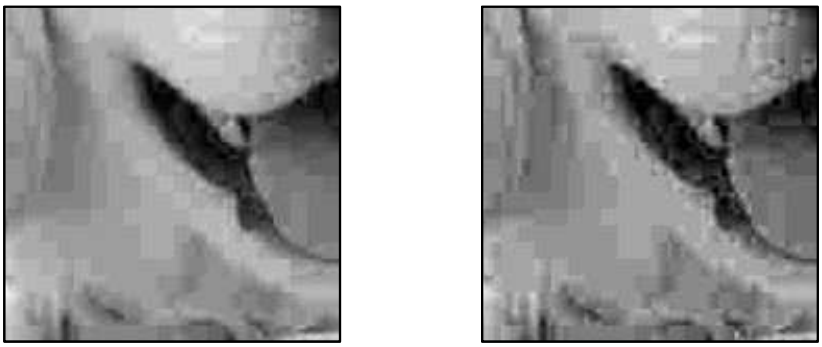


Figure 13.19: Detail of a 512x512 image encoded at 0.25 bpp using the standard quantization table (left) and a uniform quantization table (right).

scheme of Figure 13.20. Because of the previous quantization strategy, most of the coefficients will be zero and it is therefore advantageous to encode them by specifying the number of zero values preceding each nonzero element. Also, each nonzero coefficient is preceded by its size, namely the number of bits necessary to express its value. For instance, the quantized DCT block

$$\begin{bmatrix} 100 & -60 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 13 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

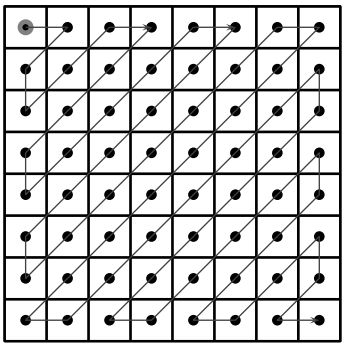


Figure 13.20: Zigzag scan.

would be encoded by the sequence[6]

$$[(0,7),\ 100],\ [(0,6),\ -60],\ [(4,3),\ 6],\ [(3,4),\ 13],\ [(8,1),\ -1],[(0,0)]$$

where the special runlength-size pair $(0,0)$ indicates the end of block, i.e. all remaining coefficients are zero. The above sequence is called the *intermediate symbol sequence* and it is composed of two types of elements, runlength-size pairs and amplitudes; both are designed to take values over a finite set and therefore form a finite, discrete-valued set of "symbols" which must be encoded in binary format.

In general terms, when encoding a high-level data set to a string of zeros and ones, it is essential that the binary sequence be easily and uniquely decodable, i.e. that it can be parsed and mapped back to the original data set. For example, the common ASCII encoding maps each letter in the English alphabet to an 8-bit block and a long text is converted to a very long binary sequence simply by concatenating the successive ASCII blocks; decoding exploits the knowledge that each symbol is 8-bit long and parses the bitstream simply by cutting it into 8-bit chunks.

The ASCII uniform mapping, while convenient, is certainly not optimal

[6] We're neglecting several technicalities here, especially with respect to the encoding method for the first coefficient. Please consult the literature for details.

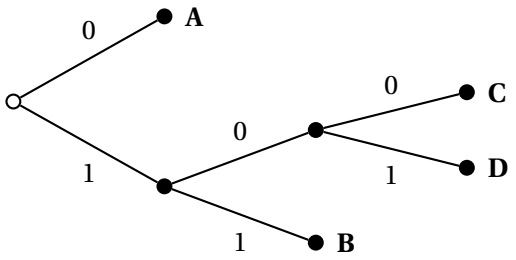


Figure 13.21: A prefix-free code for the alphabet {A, B, C, D}

in the sense that it allocates the same amount of space to all characters regardless of their frequency in a natural language. The fact that not all letters in the alphabet are equally probable is very well known and anyone who has played Scrabble is very familiar with the concept. Samuel Morse was equally aware of that fact and he designed his telegraph codes by associating dot-dash blocks of different lengths to different letters, with shorter blocks representing the most common characters; the letter "e" for instance, which is the most recurring letter in the English language, is encoded with just a single dot. Of course, when blocks different length are used, parsing a concatenated sequence becomes nontrivial. In Morse code, for instance, the dot and the dash symbols are actually supplemented by two "surreptitious" symbols, the short pause and the long pause, which mark the end of a block (i.e. a letter) and the end of a word respectively.

In binary sequences we cannot use pauses, but we can achieve perfect decodability for variable-length blocks by using so-called prefix-free codes. The idea is that no two blocks in the code can share the same prefix; therefore, we can parse the bitstream sequentially and know immediately when a block terminates. The best way to look at prefix-free codes is to represent it as in Figure 13.21, that is, as a binary tree in which each final symbol is placed on a leaf and in which the binary block associated to a symbol is obtained by traversing the tree from the root to the leaf and collecting bits on branches. Conversely, decoding a bitstream consists in using each incoming bit to steer left or right in a tree traversal until we reach a leaf, at which point the block is decoded and the traversal is restarted at the root. For instance,

the 15-bit sequence

001100110101100

would be decoded by the tree in Figure 13.21 to the 9-letter string

AABAABADC

which represents a saving of 3 bits with respect to a uniform encoding of 2 bits per symbol.

The remaining problem is how to design a prefix-free code tree so that the average length of the encoded message is minimized with respect to the probabilities of the symbols in the original data set. Formally, if we have an alphabet of symbols $\mathscr{A}$ and the probability of occurrence of each symbol is $p_A(a)$, a fundamental result in information theory tells us that a message of M symbols will require at least

$$L = -\sum_{n=0}^{M-1} p_A(x[i]) \log_2 p_A(x[i]) \quad \text{bits;}$$

L is called the *entropy* of the message. Unfortunately the proof of the above theorem is nonconstructive but, given a set of symbol probabilities, Huffman's remarkably simple and elegant algorithm can be used to build a prefix-free code that performs very close to the lower bound provided by the theoretical entropy. Given a list of N characters and relative probabilities, the algorithm proceeds by selecting at each step the two characters with lowest probability and by building a binary subtree; the two selected characters are replaced in the list by a "meta-character" that coincides with the subtree's root and with a probability equal to the sum of the leaves. After $N-1$ iterations, a prefix-free code tree is obtained. For a simple example, consider a four-letter alphabet {A, B, C, D} with probabilities:

$$p(A) = 0.38 \quad p(B) = 0.32$$
$$p(C) = 0.1 \quad p(D) = 0.2$$

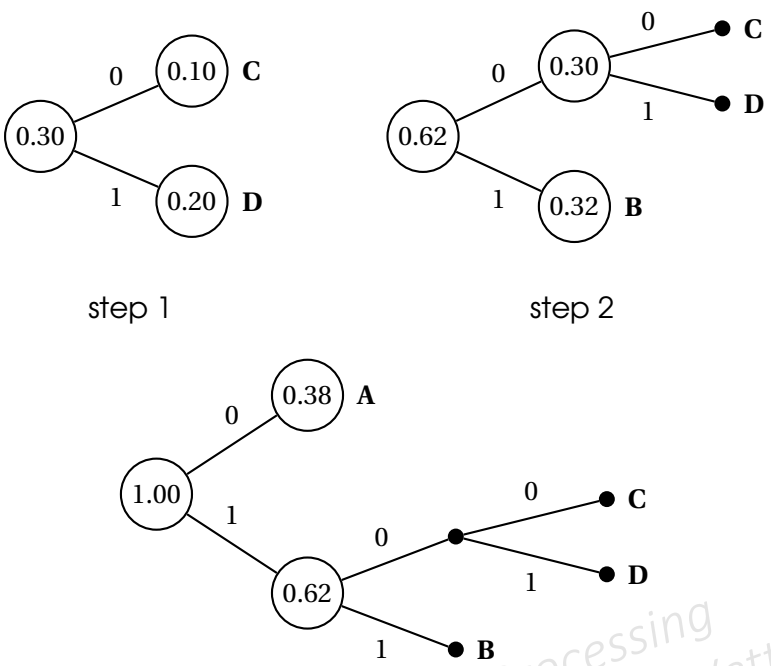


Figure 13.22: Building a Huffman code

The iterations leading to a Huffman code for the alphabet are shown in Figure 13.22.

In JPEG, a Huffman code is used to encode the runlength-size pairs, while the amplitudes are simply encoded in two's complement format. As in the case of the quantization tables, a JPEG encoder may use a default Huffman table, obtained by studying the ensemble probabilities of the runlength-size pairs for a large set of images; alternatively, the encoder may choose to build a custom Huffman table from the frequency distribution of symbols in the specific image and send the table along with the encoded data; the tradeoff is between the gain achieved by a perfectly matched code and the cost of the necessary side information use to add the code tables to the image data.