we look in the mirror we say that we see an image of ourselves, because the person inside the mirror looks just like us. Because the image captured by a camera is usually a digitized version of some two-dimensional sensory input, it is appropriately called a **digital image**.

1.4.1 Accessing Image Data

At its most basic level, then, a digital image is simply a discrete two-dimensional array of values, much like a matrix. We use *width* to refer to the number of columns in the image, and *height* to refer to the number of rows, so that the dimensions of the image are *width* by *height*, represented as *width* × *height*, and the **aspect ratio** is width divided by height, or *width* / *height*. Each element of the array is known as a **pixel**, which is short for "picture element." Pixel values are accessed by a pair of coordinates (x, y), where x and y are nonnegative integers. For a grayscale image I, the value v of the pixel at coordinates (x, y) is given by

$$v = I(x, y) \tag{1.1}$$

Sometimes we will find it more convenient to represent pixel coordinates using a vector. According to the standard convention, each vector is vertically oriented, while its transpose is horizontally oriented:

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} = [x \ y]^\mathsf{T} = (x, y) \tag{1.2}$$

where the boldface indicates a vector, and the superscript $^\mathsf{T}$ indicates the transpose operator. We will use the vector and coordinate notation interchangeably, so that $I(x, y) = I(\mathbf{x})$. In the case of a color image, each pixel contains multiple values, which we represent as another vector, $\mathbf{v} = I(\mathbf{x})$, that contains the values of the different color channels, e.g., $\mathbf{v} = (v_{red}, v_{green}, v_{blue})$.

For accessing the pixels we adopt the convention that the positive x axis points to the right and the positive y axis points down, so that x specifies the column and y specifies the row, as depicted in Figure 1.5. We also assume zero-based indexing, so that the top-left pixel is at $(0, 0)$. Other conventions are possible, but this coordinate system has the advantage that it is closely tied to the way images are typically stored in memory, and, although in 2D this is a left-handed coordinate system, in 3D the right-hand rule causes the z axis to point toward the scene along the camera's optical axis, which is convenient when performing 3D reconstruction.

Despite the fact that an image is actually a 2D array, it is stored in memory as a 1D array. Sometimes images are stored in **column major order**, that is, the first column is stored, then the second column, then the third column, and so on until the last column. More commonly, however, they are stored in **row major order**, also known as raster scan order. Hearkening back to the days when images were displayed on a cathode ray tube (CRT) by an electron gun scanning the tube one row at a time, a **scanline** is one row of an image; **raster scan order** therefore refers to storing the first row, then the second row, then the third row, and so on until the final row. Since a CRT display always begins its scan at the top-left corner of the image and proceeds downward, this is the historical basis for setting the origin at the top-left pixel.

The elements of this 1D array have indices $0, 1, 2, \ldots, n - 1$, where $n = width \cdot height$ is the number of pixels in the image, and the dot $(\cdot)$ indicates ordinary multiplication. If we let i refer to the index of this 1D array, then the first pixel at $(x, y) = (0, 0)$ has the 1D index $i = 0$. Assuming row major order, the second pixel at $(1, 0)$ has the index $i = 1$, the third pixel at $(2, 0)$ has the index $i = 2$, and so on. If the pixels are stored contiguously, then the last pixel of the first row at $(width - 1, 0)$ has the index $i = width - 1$, while the first

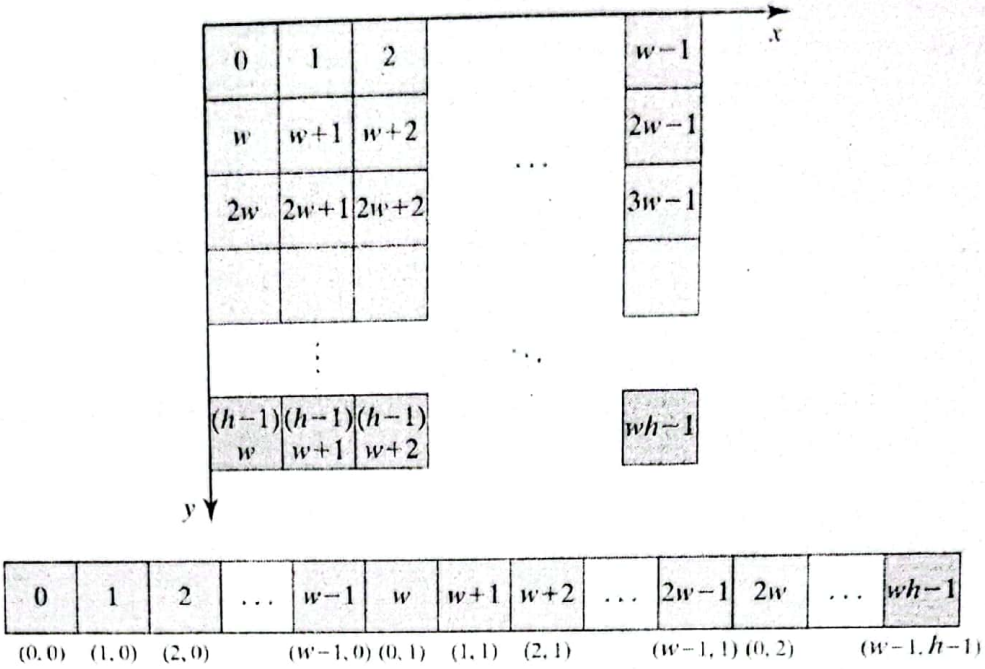


Figure 1.5: Top: Image as a 2D array, showing the 1D index of each pixel. Bottom: Internal representation of image as a 1D array using row major order.

pixel of the second row at (0, 1) has the index $i = width$. From this, it is easy to see that the 1D index can be obtained from the 2D coordinates as follows:

$$i = y \cdot width + x \tag{1.3}$$

and the inverse relationship is given by

$$x = \text{mod}(i, width) = i - y \cdot width \tag{1.4}$$

$$y = \lfloor i / width \rfloor \tag{1.5}$$

where mod (a, b) is the modulo operator that returns the remainder of a divided by b, and the floor operator $\lfloor c \rfloor$ returns the largest integer that is less than or equal to c.

1.4.2 Image Types

Several types of images exist. In a **grayscale image**, the value of each pixel is a scalar indicating the amount of light captured. These values are quantized into a finite number of discrete levels called **gray levels**. If b is the number of bits used to store each pixel value (called the **bit depth**), then 2^b is the number of gray levels, which we shall refer to as *ngray*. Usually there are eight bits (one byte) per pixel, so that $ngray = 2^b = 2^8 = 256$. Therefore, in an 8-bit grayscale image, a pixel whose value is 0 represents black, whereas a pixel whose value is 255 represents white. All the bits of a black pixel are 0, whereas all the bits of a white pixel are 1, so using hexadecimal notation these values are 00 and FF, respectively. Some specialized applications such as medical imaging require more quantization levels (e.g., 12 or 16 bits per pixel) to increase the dynamic range that can be captured, but we will generally assume 8 bits per pixel to simplify the presentation; the extension to larger bit depths is straightforward.

In an **RGB color image**, the pixel values are triples containing the amount of light captured in the three color channels: red, green, and blue. Color images, therefore, usually require 24 bits per pixel, or one byte for each of the three color channels. For an RGB color image, a black pixel has hexadecimal value 000000, while a white pixel has value FFFFFF. Although the bytes could be stored in the order of red-green-blue (*RGB*), with blue as the lowest-order byte, most frame buffers and frame grabbers adopt the reverse convention in which the order is blue-green-red (*BGR*), so that red is stored as 0000FF. The values for the different color channels are usually stored in an **interleaved** manner, that is, all three values for one pixel are stored before the three values of the next pixel, as in $B_0G_0R_0B_1G_1R_1B_2G_2R_2 \cdots B_{n-1}G_{n-1}R_{n-1}$, where the subscript is the pixel index. An alternate approach is to store the color channels in a **planar** manner, so that the red, green, and blue channels are stored as separate one-byte-per-pixel images, as in $B_0B_1B_2 \cdots B_{n-1}G_0G_1G_2 \cdots G_{n-1}R_0R_1R_2 \cdots R_{n-1}$. Either way, sometimes a fourth value is associated with each pixel, called the **alpha value** or the **opacity**, which is used for blending multiple images, as in $B_0G_0R_0A_0B_1G_1R_1A_1B_2G_2R_2A_2 \cdots B_{n-1}G_{n-1}R_{n-1}A_{n-1}$, in which case 32 bits are associated with each pixel; an alpha value of 00 indicates complete transparency, whereas an alpha value of FF indicates that the color is fully opaque.

Although grayscale and RGB color images are used for capture and display, the processing of images leads to several additional types. First, there is the **binary image**, which arises from applying a propositional test to each pixel. The most common test is that of thresholding, in which case each pixel in the output image receives the logical value ON or OFF (or equivalently TRUE or FALSE, respectively) depending upon whether the value of the input pixel is above or below a given threshold. These logical values can be stored using one bit per pixel, (0 for OFF or 1 for ON), or they can be stored using one byte per pixel, where their values are usually 0 (hexadecimal 00) or 255 (hexadecimal FF). Although this latter practice is somewhat wasteful, it is often more convenient for both display and processing. We adopt the convention that OFF is displayed as black, whereas ON is displayed as white, when the binary image is displayed as an image; we reverse this convention when graphically depicting algorithms, where OFF is displayed as white, and a color such as blue or orange is used for ON. This minor inconsistency arises naturally from the fact that, although black is the color of a blank screen, white is the color of a blank piece of paper.

Another type of image is the **real-valued image**, or **floating-point image**, in which each pixel contains a real number, at least conceptually. In practice, the number is stored in the computer as an IEEE single- or double-precision floating point number, in which case the number requires 32 or 64 bits, respectively, to be stored. A single-precision number can represent any integer in the range $[-2^{24}, 2^{24}]$ exactly, and it can represent any real number in the approximate range $[-10^{38}, 10^{38}]$ with an accuracy of about 10^{-7}. A double-precision number can represent any integer in the range $[-2^{53}, 2^{53}]$ exactly, and it can represent any real number in the approximate range $[-10^{308}, 10^{308}]$ with an accuracy of about 10^{-16}. Unlike *signal processing*, which often involves numerically delicate operations that require double-precision, for *image processing* it is difficult to find situations for which single-precision is not sufficient. In fact, an increasingly common format stores images using half-precision, which requires just 16 bits per pixel. A half-precision number can represent any integer in the range $[-2^{11}, 2^{11}]$ exactly, and it can represent any real number in the range $[-65535, 65535]$ with an accuracy of about 0.001. These numbers, which are summarized in Table 1.2, arise from the general rule that if e and s are the number of exponent and significand bits, respectively, then the range of exact integers is $[-2^{s+1}, 2^{s+1}]$, the entire range is $[-2^{2^e}, 2^{2^e}]$, and the accuracy is $s \log_{10} 2$. Floating-point images are useful not only to store the results of arithmetic operations, but also for high dynamic range images and radiance maps.

Some image processing algorithms output an **integer-valued image** in which the value of each pixel is an integer. Integer-valued images arise whenever it is necessary to store

	number of bits			range		
precision	sign	exponent	significand	total	integers	reals	accuracy
half	1	5	10	16	$[-2^{11}, 2^{11}]$	$[-10^4, 10^4]$	10^{-3}
single	1	8	23	32	$[-2^{24}, 2^{24}]$	$[-10^{38}, 10^{38}]$	10^{-7}
double	1	11	52	64	$[-2^{53}, 2^{53}]$	$[-10^{308}, 10^{308}]$	10^{-16}

TABLE 1.2: Half-, single-, and double-precision floating point representations.

negative numbers or somewhat arbitrarily large numbers. For example, to label each pixel with the region to which it belongs, we obviously cannot store the result in a grayscale image if there are more than 256 regions. Similarly, the subtraction of two images, which will in general contain negative numbers, cannot be stored in a grayscale image. Although in practice an integer-valued image uses a finite number of bits per pixel (usually 32 or 64), these values are large enough that the chance of overflowing the buffer is usually not a practical concern. A 32-bit integer, for example, can represent all integers between approximately -10^9 and 10^9, and a 64-bit integer can represent the integers from approximately -10^{19} to 10^{19}, both of which are extremely large ranges.

Finally, images can have multiple **channels**. We have already seen, for example, that an RGB color image is an 8-bit image with three channels. Similarly, after transforming from RGB color space to another color space, the result can be stored as a multichannel image, either real-valued or 8-bit. Another common multichannel image type is a **complex-valued image**, which arises from computing the Fourier transform of an image. A complex-valued image contains two floating-point values for each pixel, one for the real component and one for the imaginary component. Similarly, a multichannel integer-valued image might store the (x, y) coordinates of another pixel associated with each pixel, or a set of regions to which the pixel might belong.

These image types are summarized in Table 1.3. In this book we shall exercise care to maintain the distinction between the different types in order to support applications for which speed and memory considerations warrant this extra level of detail. Real-time applications tend to squeeze the result into as few bits as possible, so that grayscale and RGB color images are commonly used not only for capture and display, but also for holding results that may conceptually be considered integers or real values. The reason for this is that, although memory itself is cheap, processing time is greatly affected by the amount of memory used, due to the relatively high cost of cache misses and page swaps. Although the type of image should either be clear from the context or mentioned explicitly, when in doubt it will always be safe to assume (if computation is not an issue) the most general model, namely that of a multichannel floating-point image. Such a model is flexible enough to hold all of the image types mentioned (grayscale, RGB color, binary, integer, real, complex, and other color spaces), as well as any others that you will ever encounter.

	grayscale	RGB color	binary	integer-valued	real-valued	complex-valued
channels	1	3	1	1	1	2
bit depth	8	24	1	32/64	32/64	64/128
value range	$\{0, \ldots, 255\}$	$\{0, \ldots, 255\}^3$	$\{0, 1\}$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{R}^2$

TABLE 1.3: Common image types, shown with the number of channels, the most commonly encountered bit depth (number of bits per pixel), and the set of possible values. In the final three columns this set is conceptual only, since the integers $\mathbb{Z}$ and real numbers $\mathbb{R}$ are infinite sets.

1.4.3 Conceptualizing Images

We normally think of an image as a picture. That is, if we display the image so that the brightness of each tiny region on the screen or page is proportional to the value of a pixel, then the representation is easily interpreted by viewing it. There are several other ways to conceptualize an image, however, as shown in Figure 1.6, each of which provides additional insight into the algorithmic processing of images.

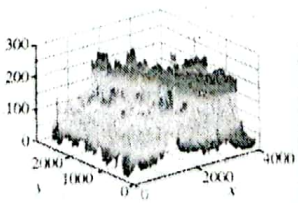
At its most basic level a digital image is stored in the computer as a discrete array of values, which can be visualized either by considering the raw pixel values themselves arranged in a 2D lattice, or equivalently as a height map, or 3D surface plot, where the height of each point is the value of the pixel. Alternatively, an image can be considered as a function that returns the value given the coordinates of a pixel. In this case, $I(x, y)$ means to evaluate the function at the position (x, y). If x and y are restricted to nonnegative integers in the domain of the image, then the function is equivalent to accessing a 2D array. However, if we expand the domain of each axis of the function to the entire set of real numbers, then it allows us to capture the values of the image even when accessed out of bounds. For example, $I(-1, -1)$ makes no sense when I is viewed as a 2D array, because $(-1, -1)$ would cause a memory access violation; but when viewed as a function, $I(-1, -1)$ yields a value that is computed from the nearby pixels, e.g., the value of the nearest pixel. Similarly, the parameters to $I(2.5, 3.5)$ would have to be rounded if the image were accessed as an array, but as a function we can define an appropriate interpolation function to compute values between pixels.

Another way to conceptualize an image is as a set of pixels. In its most general form, this set contains triplets of values capturing both the coordinates and values of the pixels. For example, the grayscale image

$$I = \begin{bmatrix} 3 & 8 & 0 \\ 2 & 9 & 4 \end{bmatrix} \tag{1.6}$$

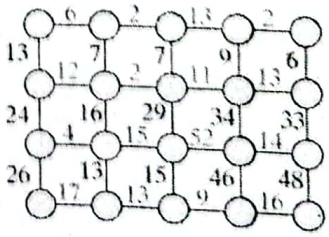
can be represented as $\{(0, 0, 3), (1, 0, 8), (2, 0, 0), (0, 1, 2), (1, 1, 9), (2, 1, 4)\}$. However, this representation is most commonly used for binary images, where the set is

Figure 1.6: Different ways to visualize an image: as a picture, as a height map, as an array of values, as a function, as a set, as a graph, and as a vector. The 5 × 4 array is a small portion of the image; the set contains the coordinates of all pixels in the array whose value is greater than 80; and the weights of the edges in the graph are the absolute differences between values in the array.



75	81	83	96	94
62	74	76	87	100
86	90	105	53	67
60	77	90	99	115

{(1, 0), (2, 0), (3, 0), (4, 0), (3, 1), (4, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3), (4, 3)}



75
81
83
96
94
62
74
76
87
...
115

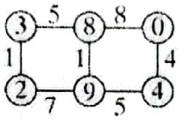
usually simplified to contain just the coordinates of those pixels whose value is on, that is, $\{(x, y) : I(x, y) = \text{ON}\}$. For example, the binary image

$$I = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{1.7}$$

can be represented as the set

$$\{(0, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (2, 2)\}. \tag{1.8}$$

An image can also be viewed as a graph, where each pixel of the image is a vertex in the graph, and each edge in the graph connects two pixels that are adjacent in the image. The weight associated with each edge is usually some measure of the similarity or dissimilarity in value between the two pixels. For example, the image in Equation (1.6) can be represented as a graph with 6 vertices and 7 edges, where the weights of the edges are given by the absolute difference between neighboring pixels:



Occasionally it is useful to view an image as a matrix. Since a matrix is a 2D array of values, this representation is easy to imagine. The only difficulty is that, for historical reasons, the conventions for matrices and images are different. Matrix entries are accessed using one-based indexing, so the top-left entry is at position (1, 1) rather than (0, 0). Also, matrices are indexed first by their row, then by their column, so the entry just to the right of the top-left entry is at position (1, 2), and an $m \times n$ matrix has m rows and n columns (as opposed to a $w \times h$ image, which has w columns and h rows). To avoid confusion, we will use boldface to indicate matrices, and we will access matrix entries using subscripts. Thus, if $\mathbf{A}$ is an $m \times n$ matrix whose $(i, j)^{\text{th}}$ entry is given by a_{ij}, we will write

$$\mathbf{A}_{\{m \times n\}} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \tag{1.9}$$

where the braces in the subscript of the matrix indicate its dimensions.

Finally, it is sometimes useful to view the image as a vector, which is obtained by either concatenating the columns of the image or by concatenating the rows and transposing the result. Adopting the latter approach, if we let $v_i = I(x, y)$ be the value of the pixel at (x, y) according to the 1D indexing of Equation (1.3), then the resulting vector is given by

$$\mathbf{v} = \begin{bmatrix} v_0 & v_1 & v_2 & \cdots & v_{n-1} \end{bmatrix}^{\mathsf{T}} \tag{1.10}$$

where n is the number of pixels in the image. This vector is a point in an n-dimensional space, so if we let each pixel take on a real value for simplicity, then $\mathbf{v} \in \mathbb{R}^n$. The vector notation allows us to imagine linear transformations of the image that involve multiplying the vector by an $m \times n$ matrix $\mathbf{T}$ on the left-hand side to produce a new vector $\mathbf{v}' = \mathbf{T}\mathbf{v}$:

$$\mathbf{v}'_{\{m \times 1\}} = \mathbf{T}_{\{m \times n\}} \mathbf{v}_{\{n \times 1\}} \tag{1.11}$$

If $\mathbf{T}$ is the $n \times n$ identity matrix $\mathbf{I}_{\{n \times n\}}$, then the input is unchanged: $\mathbf{v}' = \mathbf{v}$. More interestingly, $\mathbf{T}$ may be defined appropriately to translate or rotate the image, perform bilinear interpolation, downsample, upsample, crop, or extend past the borders as needed. Other linear operations, such as convolution and the Fourier transform, can also be represented in this way, as we shall see later in the book.

1.4.4 Mathematical Prerequisites and Notation

To successfully master the material in this book, it is necessary to be at least somewhat familiar with three areas of mathematical study. First, it is important to be comfortable with the basic concepts of linear algebra, such as matrices, vectors, matrix multiplication, and solving linear systems. Secondly, it is helpful to have some familiarity with probability and statistics, so that you know what is meant by joint probability, conditional probability, or a probability distribution function (PDF). Finally, the work will be easier if you already have been exposed to signal processing, so that discrete signals, convolution, and the Fourier transform are not entirely new concepts. Having said that, this book aims to ease the transition as much as possible by explaining concepts at an elementary level, so that having some deficiencies in these areas should not prevent anyone from progressing through the material and digesting most of it.

Because image processing and analysis are at the intersection of a number of different mathematical traditions, developing a clear and consistent notation is a challenge. The goal of this book has been to strike a balance between using notation that is internally consistent on the one hand, while at the same time maintaining consistency with existing conventions whenever possible. The result is the following set of notational conventions, which are used throughout the book almost everywhere. This list may not be interesting upon first reading, but you may find it helpful to refer to it from time to time as you progress through the book. On a few occasions these conventions are violated in order to adhere to existing widely established conventions, but the context should make the meaning clear wherever this occurs.

g, ψ	Lowercase Latin or Greek characters indicate scalars
g, ψ	Lowercase Latin or Greek characters also indicate functions of one variable
G, Ψ	Uppercase Latin or Greek characters indicate functions of more than one variable
$\mathcal{A}$	Uppercase calligraphic Latin characters indicate sets
$g(x)$	Either the 1D function g evaluated at x, or the function g itself
$G(x, y)$	Either the 2D function G evaluated at (x, y) or the function G itself
$g(\cdot)$	The function evaluated at some value, where the variable name is unimportant or obvious
$G(\cdot, \cdot)$	The function evaluated at some pair of values, where the variable names are unimportant or obvious
$g[x]$	Brackets indicate a discrete array indexed by nonnegative integers
$\dot{g}, \ddot{g}$	First and second derivatives of function
$a \doteq b$	The variable a is equal to b
$a \equiv b$	The variable a is defined to be equal to b
$\mathbf{g}, \boldsymbol{\psi}$	Boldface lowercase Latin or Greek characters indicate vectors
$\mathbf{G}, \boldsymbol{\Psi}$	Boldface uppercase Latin or Greek characters indicate matrices
$\mathbf{G} = [g_{ij}]$	The ij^{th} element of matrix $\mathbf{G}$ is given by g_{ij}
$\mathbf{g}^{\mathsf{T}}$	Transpose of vector $\mathbf{g}$
$\mathbf{FG}$	Matrix multiplication
$\cdot$	Central dot indicates ordinary multiplication (also used for divergence)

:	Colon means either a range, as in 1:10, or "such that," as in $\{x : x < 0\}$
$\circledast$	Asterisk with a circle indicates convolution
$\mathbb{R}$, $\mathbb{R}^n$, $\mathbb{R}^{m \times n}$	Set of real numbers, set of vectors of n real numbers, set of $m \times n$ real matrices
$\mathbb{Z}$, $\mathbb{Z}_{a,b}$	Set of integers, set of integers from a to b, inclusive
$O(\cdot)$	Big O notation for asymptotic running time of algorithms
*	Asterisk indicates ordinary multiplication (only used in pseudocode)
==	Long equal sign indicates test for equality (pseudocode)
$\leftarrow$	Assignment (pseudocode)
$\leftarrow_+$	Assignment with addition; same as += in C/C++/Java (pseudocode)
$\leftarrow_-$	Assignment with subtraction; same as -= in C/C++/Java (pseudocode)

1.4.5 Programming

It has been said that a person does not really know anything until he or she is able to write it down. In a similar way, a person does not really understand an algorithm until he or she is able to implement it. Therefore, the best way to learn image processing and analysis is by programming real algorithms on real images. To aid the reader in this endeavor, this book provides detailed pseudocode for many of the algorithms presented. Although the pseudocode may not be very interesting upon first reading, you will likely find it indispensable when you desire to acquire a deeper understanding of any given technique by implementing it yourself. The pseudocode has been written to balance between precision on the one hand and readability on the other. If you are proficient at a programming language, it should not be difficult to translate the pseudocode into actual working code.

By far the most common language used in learning image processing and analysis is MATLAB, or its open-source alternative, Octave. MATLAB has a clean syntax, is very easy to use, is interpreted rather than compiled, and comes with built-in visualization capabilities, an editor, and a debugger. In industry, however, the need for efficient computation requires the use of a lower-level language like C or C++, for which the most widely used library is OpenCV. OpenCV has extensive capabilities for loading and displaying images, connecting to cameras, and performing basic operations, as well as advanced algorithms like face detection and camera calibration. OpenCV also has bindings to other languages such as Python and Java for more rapid prototyping. Other libraries include CImg, vxl, ImageJ, and dozens of others. More information about these tools and libraries can easily be found by searching online.

1.5 Looking Forward

With these basics under our belt, we are now ready to begin tackling the topics of image processing and analysis. As we do so, one word of caution is in order. In other fields of study, we are accustomed to dealing with convergent problems. A **convergent problem** is one in which there is a single unique solution, and the more one studies the problem the more one learns about it. In contrast, as pointed out by a well-known economist [Schumacher, 1973], a **divergent problem** has no correct solution, and the more it is studied the more the answers seem to contradict one another. Image analysis, and to a lesser extent image processing, are full of divergent problems for which there is not a single unique solution but rather a variety of different solutions, each with its own merits and shortcomings. Therefore, do not be surprised if, when faced with a particular problem, you try the leading algorithms, only to discover that they fail miserably and that a completely different (and oftentimes far simpler) approach outperforms them all in the particular context in which you are working.

Image analysis is a young field, and the solutions are elusive. While progress will undoubtedly continue over the coming decades to produce practical systems that process imagery to provide useful information, this will happen by continually questioning existing techniques and exploring new ones. Therefore whether you are a student, researcher, or practitioner, put on your creativity cap and be ready to think outside the box and try new approaches. After all, image analysis is for the most part a bag of tricks, so feel free to select whatever tricks you find in the bag, as well as any new tricks you develop on your own, in order to solve the problems that you encounter.

1.6 Further Reading

This chapter has presented an overview of image processing and analysis, along with their relationship to machine and computer vision. A variety of alternative overviews of one or more of these fields can be found in various textbooks. Burger and Burge [2008] provide an easy-to-read introduction to the field of image processing, while Gonzalez and Woods [2008] present a more detailed treatment of the subject. For computer vision, Shapiro and Stockman [2001] provide an introduction, whereas Forsyth and Ponce [2012] cover the subject at an advanced level, and Szeliski [2010] provides a readable treatment with a helpful summary of the latest research. Machine vision is covered thoroughly by Davies [2005]. A combined treatment of the fields can be found in the introductory text of Umbaugh [2010] or the more comprehensive book of Sonka et al. [2008]. For more historical texts, the classic books of Rosenfeld and Kak [1982], Jain [1989], Pratt [1991], Jain et al. [1995], or Castleman [1995] on image processing; or the classic works of Marr [1982], Ballard and Brown [1982], Horn [1986], or Nalwa [1993] on computer vision can be consulted. For learning about 3D computer vision, Trucco and Verri [1998] provide an easy-to-read treatment, while Hartley and Zisserman [2003] is the definitive resource. A myriad of monographs or edited works on more specialized topics can also be found but are too numerous to list here.

The latest research can be found in a variety of conferences and journals. The leading conferences in image processing are International Conference on Image Processing (ICIP) and International Conference on Pattern Recognition (ICPR), while the leading journal is *IEEE Transactions on Image Processing*. The leading conferences in computer vision are Computer Vision and Pattern Recognition (CVPR), International Conference on Computer Vision (ICCV), and European Conference on Computer Vision (ECCV), while the leading journals are *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI) and *International Journal of Computer Vision* (IJCV). The leading venues for medical imaging research are *IEEE Transactions on Medical Imaging* and *Medical Image Analysis*.

PROBLEMS

1-1 Define image processing and image analysis.

1-2 Even though machine vision and computer vision are nearly synonymous, there are some subtle distinctions between them. List at least two of these differences.

1-3 Image analysis, as defined in this book, is very closely related to computer vision. What is the key difference?

1-4 Image processing, as defined in this book, produces an output image from an input image. What are the two primary purposes for such output images?

1-5 Another way to categorize the information in this book would be in terms of low-, mid-, and high-level vision. Explain how you would map image processing, image analysis, machine vision, and computer vision into these alternative categories.

1-6 List three basic image processing problems and three basic problems in image analysis.