Chapter 2

Pixel-based image processing

We begin our tour of computer vision by considering some basic operations that can be performed on an image. These techniques will enable us to achieve some interesting results without requiring much mathematical background. They will also serve as a foundation upon which we can build in later chapters when we look at more sophisticated methods.

2.1 What is an image?

An image is simply a two-dimensional array of values, much like a matrix. In the case of a grayscale image, the values are scalars indicating the intensity of each pixel, while for a color image the values are triples containing the values of the three color channels: red, green, and blue. Usually there are eight bits per channel, leading to images with one byte per pixel (grayscale images) or three bytes per pixel (color images). Larger values indicate more light intensity, so for an 8-bit grayscale image, 0 represents black and 255 represents white. Using hexadecimal notation, these are 0x00 and 0xFF, respectively. For an RGB color image, 0x000000 is black and 0xFFFFFF is white. Some specialized applications such as medical imaging require more quantization levels (e.g., 12 or 16 bits) to increase the dynamic range that can be captured, but the same techniques can be applied with only slight modification. We adopt the convention in this book that images are accessed by a pair of coordinates (x, y) with the positive x axis pointing to the right and the positive y axis pointing down, so that x specifies the column and y specifies the row, and $(0, 0)$ indicates the top-left pixel.

Figure 2.1 shows an 8-bit-per-pixel grayscale image of some objects on a conveyor belt. Such images are common in manufacturing environments, where machine vision techniques play an important role in ensuring that the parts being manufactured are without defects, are sorted into the proper bins, and so on. For such applications, we might want to isolate the objects from the background and compute properties of the objects for the purpose of classifying and manipulating them. This image will serve as a motivation for this chapter, enabling us to explore many techniques that are useful for such problems.

2.2 Histograms

Let L represent the number of *gray levels* (i.e., the number of possible intensity values) of a grayscale image. That is, each pixel is assigned a value $0, \ldots, L-1$. For an 8-bit image $L = 256$, so that each pixel's value is between 0 and 255, inclusive. An important concept is the gray-level **histogram**, which is a one-dimensional array that stores for each gray level the number of pixels having that value:

$$h[k] = n_k, \quad k = 0, \ldots, L-1,$$

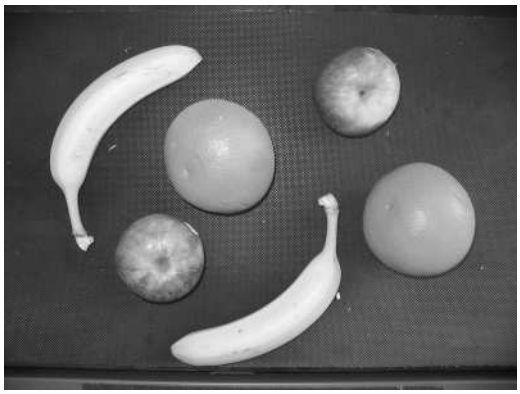

Figure 2.1: An 8-bit grayscale image of several types of fruit on a dark background (conveyor belt).

where n_k is the number of pixels in the image with gray level k. Represented mathematically, n_k is the cardinality of the set of pixels whose intensity is k, i.e., $n_k = |\{p : I(p) = k\}|$, where $p = (x, y)$ is a pixel in the image I. The histogram can be thought of as a summary of the image, in which all spatial information has been discarded. The computation of the histogram is straightforward: We simply visit all the pixels and keep track of the count of pixels for each possible value.

```
ComputeHistogram(I)
1  for k ← 0 to L − 1 do
2        h[k] ← 0
3  for (x, y) ∈ I do
4        h[I(x, y)] ← h[I(x, y)] + 1
5  return h
```

In this code the image I is treated both as a set, so that $(x, y) \in I$ means a pixel in the image, and as a function, so that $I(x, y)$ yields the gray level of pixel (x, y). The arrow pointing to the left indicates assignment, and the bracket operator is used to access a particular value within the array h.

Once the histogram has been found, the **normalized histogram** $h_n[k] = h[k]/n,\ k = 0, \ldots, L-1$ is computed by simply dividing each value in the histogram by n, where n is the total number of pixels in the image. The normalized histogram is the probability density function (PDF) capturing the probability that any pixel drawn at random from the image has a particular gray level.[1] Note that while $h[k]$ is an integer, $h_n[k]$ is a floating point value.

```
ComputeNormalizedHistogram(I)
1  h ← ComputeHistogram(I)
2  n ← width ∗ height
3  for k ← 0 to L − 1 do
4        h[k] ← h[k]/ n
5  return h
```

This code, like the regular histogram, requires just a single pass through the image. The variable n holds the number of pixels in the image, which is the image width times the image height. The normalized histogram of the fruit image is given in Figure 2.2.

The histogram is related to the contrast in an image: A flat histogram indicates that the gray levels are equally distributed throughout the image, thus maximizing the options available; while a

[1] Technically, since the image is discrete, h_n is a probability *mass* function (PMF), but we will refer to it as a PDF to simplify the discussion since the distinction is not important for our purposes.

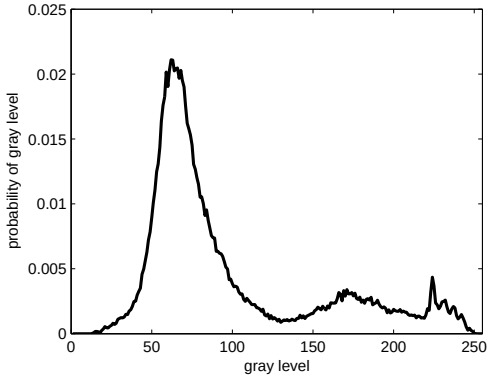

Figure 2.2: The normalized gray-level histogram of the image in Figure 2.1.

peaked histogram indicates an image whose values are all nearly the same. Given a low-contrast image, a common technique for increasing contrast is to more evenly distribute the pixel values of an image across the range of allowable values. This approach is known as **histogram equalization**. The algorithm first converts the PDF (captured by the normalized histogram) to a cumulative distribution function (CDF) by computing a *running sum* of the histogram:

$$c_n[k] = \sum_{\ell=0}^{k} h_n[\ell], \quad k = 0, \ldots, L-1. \tag{2.1}$$

The running sum can be computed efficiently, of course, by setting $c_n[0] = h_n[0]$ and $c_n[k] = c_n[k-1] + h_n[k], k = 0, \ldots, L-1$. Once the CDF has been computed, a pixel with gray level k is transformed to $k' = (L-1)c_n[k]$. Note that $c_n[L-1] = 1$, so the output $0 \leq k' \leq L-1$ as desired.

HistogramEqualize(I)
1 $h_n \leftarrow$ ComputeNormalizedHistogram(I)
2 $c_n[0] \leftarrow h_n[0]$
3 **for** $k \leftarrow 1$ **to** 255 **do**
4 $\quad c_n[k] \leftarrow c_n[k-1] + h_n[k]$
5 **for** $(x, y) \in I$ **do**
6 $\quad I(x, y) \leftarrow$ Round$((L-1) * c_n[I(x, y)])$
7 **return** I

Why does such a simple algorithm work? In other words, what is Line 6 (which is the heart of the program) doing? To answer this question, consider the example shown in Figure 2.3, in which we assume that the gray levels are continuous for simplicity. The desired PDF $p'(k)$, which is the normalized histogram of the gray levels of the output image, should be flat, i.e., $\int_{a'}^{a'+\delta} p'(k)dk$ should be constant for any gray level a' and any given constant δ. Since the algorithm uses the scaled CDF $q(k)$ of the original histogram to transform gray levels, this transformation is visualized in the lower-left plot of the figure, with a gray level k along the horizontal axis transforming to a new gray level $c = q(k)$ along the vertical axis. Since q is the scaled integral of p, we see that the area under the PDF for any interval corresponding to an output interval of δ is $\int_{q^{-1}(a')}^{q^{-1}(a'+\delta)} p(k)dk = q(q^{-1}(a'+\delta)) - q(q^{-1}(a')) = a' + \delta - a' = \delta$. In other words, equally spaced intervals of width δ along the axis of the new PDF capture equal numbers of pixels in the original PDF. The CDF thus provides a simple means of ensuring that an equal number of pixels contribute to an equally spaced interval in the output, if the variable k were continuous. Of course in practice the algorithm only produces an approximately flat output because of discretization effects. Figure 2.4 shows the results of histogram equalization.

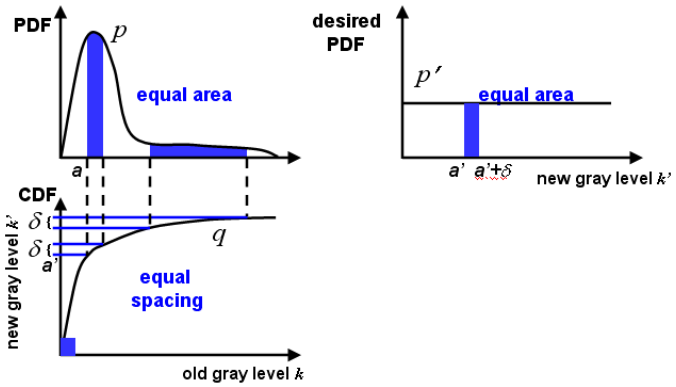

Figure 2.3: Why histogram equalization works. In this example, the histogram of the original image is heavily weighted toward darker pixels. If we let the CDF be the mapping from the old gray-level value to the new one, the new PDF is flat and therefore weights all gray levels equally. This is because any interval of width δ in the new histogram captures the same number (δ) of pixels in the original image. Note that discretization effects have been ignored for this illustration.

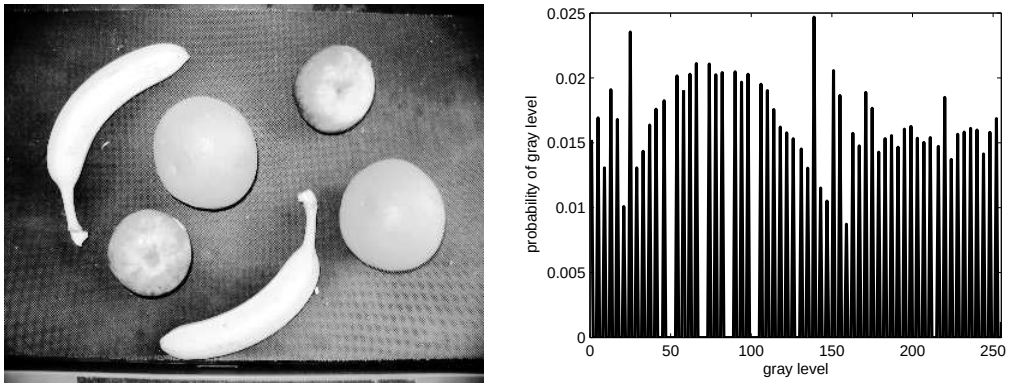

Figure 2.4: LEFT: The result of histogram equalization applied to the image in Figure 2.1. The increase in contrast is noticeable. RIGHT: The normalized histogram of the result is much flatter than the original histogram, but it is not completely flat due to discretization effects.

2.3 Thresholding a grayscale image

Often one wishes to separate the foreground from the background in a grayscale image. A common example is separating parts on a conveyor belt from the conveyor belt itself. This is a simple form of segmentation, and it also appears in the context of background subtraction in which motion information is thresholded to detect moving foreground objects.

Thresholding an image involves simply setting all the pixels whose values are above the threshold to 1, while setting all pixels whose values are less than or equal to the threshold to 0. The result is a binary image that separates the foreground from the background. We will adopt the convention that 0 (which we shall call `OFF`) indicates the background, while 1 (which we shall call `ON`) indicates the foreground. Do not be confused, though, because sometimes the foreground will be displayed as black on a white background (as on paper, for example), while at other times you will see a white foreground displayed on a black background (on a computer monitor). Also, it is common implementation practice to store a 1 in every bit for a foreground pixel, so that 0xFF instead of 0x01 is stored; but this

implementation detail will not affect our discussion.

2.3.1 Ridler-Calvard algorithm

The difficult part of thresholding is determining the correct threshold. Recalling the histogram shown in Figure 2.2, we see that a good threshold is one which separates two modes of the histogram, i.e., the threshold should lie in a valley between two hills (approximately $k = 130$ in this case). A simple iterative algorithm to achieve this is the Ridler-Calvard algorithm. Let t be a threshold, and let $\mu_{\blacktriangleleft}$ be the mean gray level of all the pixels whose gray level is less than or equal to t, while $\mu_{\triangleright}$ is the mean gray level of all the pixels whose gray level is greater than t. (Assuming the background is darker than the foreground, then $\mu_{\blacktriangleleft}$ is the mean of the background pixels, and $\mu_{\triangleright}$ is the mean of the foreground pixels.) It is easy to show that $\mu_{\blacktriangleleft} = m_1[t]/m_0[t]$, where $m_0[k] = \sum_{\ell=0}^{k} h[\ell]$ is the *zeroth moment* of the histogram h from gray level 0 to k, and $m_1[k] = \sum_{\ell=0}^{k} kh[\ell]$ is the *first moment*. We will cover moments in more detail later in the chapter. For now, note that m_0 is just the cumulative normalized histogram c_n, scaled so that $m_0[L-1]$ is the number of pixels in the image. Since m_0 and m_1 are running sums, the zeroth moment of the pixels from gray level $k+1$ to $L-1$ is simply $m_0[L-1] - m_0[k]$, and the first moment is just $m_1[L-1] - m_1[k]$. Putting these together yields a simple expression for the mean of the second set of pixels: $\mu_{\triangleright} = (m_1[L-1] - m_1[t])/(m_0[L-1] - m_0[t])$.

The Ridler-Calvard algorithm iteratively computes the two means based on the threshold, then sets the threshold t to the average of the two means. Although iterative algorithms usually require a good starting point, this algorithm is powerful because in practice any initial value will converge to the same solution.

```
RIDLER-CALVARD(I)
 1  h ← COMPUTEHISTOGRAM(I)
 2  m0[0] ← h[k]
 3  m1[0] ← k * h[k]
 4  for k ← 1 to L − 1 do
 5        m0[k] ← m0[k − 1] + h[k]
 6        m1[k] ← m1[k − 1] + k * h[k]
 7  t ← L/2        ; reasonable initial guess, but not important
 8  repeat
 9        μ◀ ← m1[t]/m0[t]
10        μ▷ ← (m1[L − 1] − m1[t])/(m0[L − 1] − m0[t])
11        t ← ROUND ((1/2)(μ◀ + μ▷))
12  until t does not change
13  return t
```

This classic algorithm requires one pass through the image to compute the histogram, then one pass through the histogram to compute m_0 and m_1, followed by a small number of iterations (usually less than 5) consisting only of constant-time operations. Note in Line 1 that the normalized histogram could be used instead of the regular histogram since the divisions in Lines 9 and 10 cancel the scaling factor. But since the exact same t value will result either way, we might as well use the regular histogram to save the expense of having to normalize the histogram.

The algorithm is based on the assumption that the foreground and background gray level intensities are distributed as Gaussians with equivalent standard deviations. In such a case, the optimal decision boundary occurs where the two distributions cross:

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(t-\mu_{\blacktriangleleft})^2}{2\sigma^2}\right\} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(t-\mu_{\triangleright})^2}{2\sigma^2}\right\}, \tag{2.2}$$

where $\mu_{\blacktriangleleft}$ and $\mu_{\triangleright}$ are the mean gray levels of the two groups of pixels. Solving this equation for t yields

$$t = \frac{(\mu_{\blacktriangleleft} + \mu_{\triangleright})}{2}, \tag{2.3}$$

which appears in Line 11 of the algorithm. This derivation illustrates an important point in the design and analysis of image processing algorithms, namely, that algorithms often make statistical assumptions whether they are explicitly acknowledged or not. Therefore, specifying the assumptions explicitly enables the algorithm to be separated from the model, thus often yielding new insight into the problem. In this case the analysis reveals that Ridler-Calvard assumes that the foreground and background variances are identical.

2.3.2 Otsu algorithm

Ridler-Calvard assumes that the background and foreground regions have the same variance in intensity. If we relax this assumption and instead allow the two regions to have different variances, we arrive at the Otsu algorithm. The goal of Otsu is to find the threshold t that minimizes the *within-class variance*, which is defined as the weighted sum of the variances of the two groups of pixels:

$$\sigma_w^2(t) = p_{\blacktriangleleft}(t)\sigma_{\blacktriangleleft}^2(t) + p_{\triangleright}(t)\sigma_{\triangleright}^2(t), \tag{2.4}$$

where t is the unknown threshold, $p_{\blacktriangleleft}(t) = \sum_{\ell=0}^{t} h_n[\ell] = m_0[t]/m_0[L-1]$ is the proportion of pixels whose gray level is less than or equal to t, $\mu_{\blacktriangleleft} = m_1[t]/m_0[t]$ is their mean intensity, and $\sigma_{\blacktriangleleft}^2(t) = \sum_{\ell=0}^{t}(h_n[\ell] - \mu_{\blacktriangleleft})^2$ is their variance in intensity. The other variables represent analogous quantities for the remaining pixels. It is easy to show that the sum of the within-class variance and the between-class variance is the total variance of all the pixel intensities: $\sigma_w^2(t) + \sigma_b^2(t) = \sigma^2$, where the *between-class variance* is defined as:

$$\sigma_b^2(t) = p_{\blacktriangleleft}(t)\left(\mu_{\blacktriangleleft}(t) - \mu\right)^2 + p_{\triangleright}(t)\left(\mu_{\triangleright}(t) - \mu\right)^2, \tag{2.5}$$

where $\mu = m_1[L-1]/m_0[L-1]$ is the mean gray level of all the pixels. Since the total variance σ^2 does not depend on the threshold t, minimizing σ_w^2 is the same as maximizing σ_b^2. The advantage of the latter is that it is dependent only upon first-order properties (means) rather than second-order properties (variances), thus making it easier to compute.

From the above, we see that for a given value of t, $p_{\blacktriangleleft}(t) = m_0[t]/m_0[L-1]$ while $p_{\triangleright}(t) = (m_0[L-1] - m_0[t])/m_0[L-1]$, because $p_{\blacktriangleleft}(t) + p_{\triangleright}(t) = 1$. Substituting these values, along with $\mu_{\blacktriangleleft}(t) = m_1[t]/m_0[t]$, and $\mu_{\triangleright}(t) = (\mu - m_1[t])/(m_0[L-1] - m_0[t])$ into the equation above and simplifying yields

$$\sigma_b^2(t) = \frac{(m_1[t] - \mu m_0[t])^2}{m_0[t](m_0[L-1] - m_0[t])}. \tag{2.6}$$

The Otsu algorithm iterates through all possible thresholds t to find the one that maximizes this quantity.

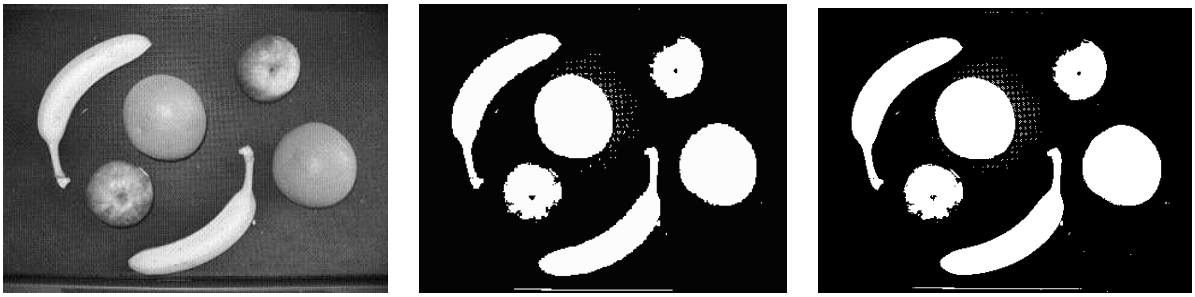

Figure 2.5: From left to right: Input image, output of Rider-Calvard, and output of Otsu. The outputs are almost indistinguishable.

```
OTSU(I)
 1  h ← COMPUTEHISTOGRAM(I)
 2  m_0[0] ← h[k]
 3  m_1[0] ← k * h[k]
 4  for k ← 1 to L − 1 do
 5        m_0[k] ← m_0[k − 1] + h[k]
 6        m_1[k] ← m_1[k − 1] + k * h[k]
 7  μ ← m_1[L − 1]/m_0[L − 1]
 8  σ̂_b^2 ← 0
 9  for k ← 0 to L − 1 do
10        σ_b^2 ← (m_1[k] − μm_0[k])^2 / (m_0[k] * (m_0[L − 1] − m_0[k]))
11        if σ_b^2 > σ̂_b^2 then
12              σ̂_b^2 ← σ_b^2
13              t ← k
14  return t
```

The Otsu algorithm begins with the same precomputation as Ridler-Calvard, and it also can be performed with either the standard histogram or the normalized histogram, since the division in Lines 7 and 10 cancel the normalization. The difference between the two algorithms is that Otsu performs an exhaustive search over all possible $L = 256$ thresholds rather than iteratively converging on a soution from a starting point. Otsu requires one pass through the image, then two passes through the histogram. The algorithm illustrates the principle that a more general model offers more degrees of freedom but also requires more computation to search for the correct solution (but in this case the extra work is almost negligible). Figure 2.5 shows a comparison of the outputs of Ridler-Calvard and Otsu on the fruit image.

The careful reader may notice that the description here is the original formulation of the Otsu algorithm, which does not require recursive relations. Often the algorithm is described using recursive relations, but we have avoided them here because they complicate the algorithm for no reason.

2.4 Morphological operations

Morphological processing refers to changing the form or structure of a region in an image. We will look only at binary images, though the concepts presented here can be extended to gray-level images.

2.4.1 Erosion and dilation

The two fundamental morphological operations are erosion and dilation. Let A be a binary image, and let B be a binary mask. Typically A is much larger than B, because A is the size of the original image, while B is on the order of 3×3 or 5×5. The pixel values in each are either `ON` (stored by the computer as 1) or `OFF` (stored as 0). Now suppose we overlay B so that the center of B is on top of some pixel (x, y) in A. We set the value at the corresponding output binary image $C(x, y)$ to `ON` if there is an `ON` pixel in A under *all* of the `ON` pixels in B; otherwise we set the output pixel to `OFF`. If we repeat this procedure by sliding B across A both horizontally and vertically, computing an output for each pixel in A, the result will be a binary output image C that is the **erosion** of A by the **structuring element** B. Ignoring border effects, C will be the same size as A.

Now suppose that, instead of computing the value in the manner just described, we set the value of the output pixel to `ON` if there is an `ON` pixel in A under *at least one* of the `ON` pixels in $\hat{B}$, where $\hat{B}$ refers to a binary mask created by flipping B horizontally and vertically. Then the output is called the **dilation** of A by B.

Dilation and erosion are duals of each other with respect to complementation and reflection (flipping). Using the notation $A \ominus B$ to refer to erosion and $A \oplus B$ to refer to dilation, we have $(\overline{A \ominus B}) = \overline{A} \oplus \hat{B}$, where the overbar indicates binary complementation. The code for both erosion and dilation is straightforward:

```
ERODE(I, B)
1  for (x, y) ∈ I do
2        all ← ON
3        for (x', y') ∈ B do
4              if B(x', y') AND NOT I(x + x' − ⌊w_B/2⌋, y + y' − ⌊h_B/2⌋)
5              then all ← OFF
6        C(x, y) ← all
7  return C
```

```
DILATE(I, B)
1  for (x, y) ∈ I do
2        any ← OFF
3        for (x', y') ∈ B do
4              if B(x', y') AND I(x − x' + ⌊w_B/2⌋, y − y' + ⌊h_B/2⌋)
5              then any ← ON
6        C(x, y) ← any
7  return C
```

In this code, note that $x' = 0, \ldots, w_B - 1$, and $y' = 0, \ldots, h_B - 1$, where w_B and h_B are the width and height of B, respectively. If the size of B is odd (as is common), then the floor of the half-width and half-height simplify to $\lfloor \frac{w_B}{2} \rfloor = \frac{w_B - 1}{2}$ and $\lfloor \frac{h_B}{2} \rfloor = \frac{h_B - 1}{2}$. The flipping of B in DILATE has been accomplished by changing the signs in Line 4.

While the structuring element is allowed to be any arbitrary binary pattern, it is often a 3×3 matrix of all ones, B_8, or a 3×3 cross of ones, B_4. In such a case the symmetry of B allows one to ignore the flipping in the formulas above because $B = \hat{B}$, and the structure of B allows the code to be greatly simplified. In the case of B_4, for example, we can replace the code in the outer **for** loop (Lines 2–6) with one line, leading to the following:

```
ERODE_B4(I)
1  for (x, y) ∈ I do
2        C(x, y) ← I(x, y) AND I(x − 1, y) AND I(x + 1, y) AND I(x, y − 1) AND I(x, y + 1)
3  return I
```

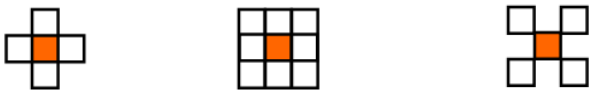

Figure 2.6: Neighborhoods: $\mathcal{N}_4$, $\mathcal{N}_8$, and $\mathcal{N}_D$.

```
DILATE_B4(I)
1  for (x, y) ∈ I do
2        C(x, y) ← I(x, y) OR I(x − 1, y) OR I(x + 1, y) OR I(x, y − 1) OR I(x, y + 1)
3  return I
```

This is the first algorithm we have considered that uses neighbors of pixels to compute a result. A pixel q is a **neighbor** of pixel p if q is in the **neighborhood** of p: $q \in \mathcal{N}(p)$, where $\mathcal{N}$ is the neighborhood function. The most common neighborhoods used in image processing are shown in Figure 2.6:

- the 4-neighborhood, denoted by $\mathcal{N}_4$, which is a set consisting of the four pixels to the left, right, above, and below the pixel; and

- the 8-neighborhood, denoted by $\mathcal{N}_8$, which is $\mathcal{N}_4 \bigcup \mathcal{N}_D$, where $\mathcal{N}_D$ are the four pixels diagonal from the pixel.

Note that in the structuring element B_4, the pixels that are `ON` are the central pixel and its 4-neighbors, while the pixels in B_8 that are `ON` are the central pixel and its 8-neighbors.

Any algorithm that uses neighbors must decide what to do when those neighbors do not exist. For example, when the structuring element is centered near the boundary of the image, some of its elements will be extend past the image A and be out of bounds. There is no agreed-upon solution for this problem, but common ways to handle out-of-bounds pixels include:

- *Zero pad.* Keep the structuring element (kernel) in bounds at all times and set the output pixels near the boundary to zero (or some other arbitrary value). This is the fastest and simplest solution and works fine if you do not care what happens near the border.

- *Resize the kernel.* Near the border, shrink the kernel so that it does not extend past the image border. For example, if you have a 3×3 kernel of all ones, you might want to use a 2×3 kernel of all ones near the left and right border, a 3×2 kernel near the top and bottom borders, and 2×2 kernels (with the center placed appropriately) near the four corners.

- *Hallucinate values outside the image.* The most common approaches are replicate (out of bounds pixels are assigned the value of the nearest pixel in the image), reflect (image values are mirror-reflected about the image border), and wrap (image values are extended in a period wrap, which is what the discrete Fourier transform does implicitly).

2.4.2 Opening and closing

Two additional morphological operations are opening and closing. **Opening** is just erosion followed by dilation: $A \circ B = (A \ominus B) \oplus B$, while **closing** is dilation followed by erosion: $A \bullet B = (A \oplus B) \ominus B$. Opening and closing are also duals of each other: $(\overline{A \bullet B}) = (\overline{A} \circ \hat{B})$. Repeated applications of opening or closing do nothing: $(A \bullet B) \bullet B = A \bullet B$, and $(A \circ B) \circ B = A \circ B$. An example of applying various morphological operations to a binary image (obtained by thresholding a background subtraction result) is shown in Figure 2.7.

image erode dilate open close

Figure 2.7: A binary image and the result of morphological operations: Erode, dilate, open, and close.

2.5 Finding regions

We have seen that thresholding is able to produce a binary image, and that morphological operations are useful for cleaning up noise in the result. Another important step is to be able to find a **region** in the binary image, which is a set of connected pixels. Pixels are said to be **connected** (or contiguous) if there exists a path between them, where a **path** is defined a sequence of pixels $p_0, p_1, \ldots, p_{n-1}$ such that p_{i-1} and p_i are adjacent for all $i = 1, \ldots, n-1$. Two pixels in a binary image are said to be **adjacent** if they have the same value (0 or 1) and if they are neighbors of each other. Thus, the type of neighborhood determines the type of adjacency. Not surprisingly, the two most common adjacencies are 4-adjacency and 8-adjacency:

- Two pixels p and q in an image I are *4-adjacent* if $I(p) = I(q)$ and $q \in \mathcal{N}_4(p)$;
- Two pixels p and q in an image I are *8-adjacent* if $I(p) = I(q)$ and $q \in \mathcal{N}_8(p)$.

There is also something called m-adjacency which we will define later in this section. For all adjacencies, it is assumed that the neighborhood relations are symmetric, so that $q \in \mathcal{N}(p)$ if and only if $p \in \mathcal{N}(q)$ for any neighborhood $\mathcal{N}$.

Although the definitions above are given in terms of a binary image, they can be easily generalized to a grayscale or color image. The simplest generalization is to consider two pixels adjacent if they have the *exact same* gray level or color. This is the scenario that we shall consider in this chapter. Other definitions, in which pixels must have *similar* gray levels or colors, introduce significant complications which we shall consider when we look at the topic of segmentation in Chapter 7.

2.5.1 Floodfill

Floodfill is the problem of coloring all the pixels that are connected to a *seed pixel*. Several algorithms exist for performing floodfill. The most compact to describe is based upon repeated conditional dilations, but it is terribly inefficient and only applies to binary images. Another approach that is often taught is one that uses recursive calls, but this algorithm is never used in practice because recursive calls not only incur function overhead (thus decreasing computational efficiency) but, even more importantly, they often cause the stack to be overrun, thus causing the program to crash. This is true even on modern computers because it is not uncommon for floodfilled regions to contain tens of thousands of pixels. Therefore, we will not take the time to describe these approaches further.

The most computationally efficient approach, and yet still simple to describe, overcomes these problems by using a stack of pixels called the *frontier*. The algorithm takes a seed pixel $p = (x, y)$, a new color, and an image, and it colors the pixels in place. By "color," we mean a red-green-blue triplet for an RGB image, or a graylevel $0 \leq k \leq L-1$ for a grayscale image. In the initialization, the original color of the seed pixel p is grabbed, the seed pixel is colored with the new color, and the coordinates of p are pushed onto the frontier. Then the algorithm repeatedly pops a frontier pixel off the stack and expands all the adjacent pixels (the neighbors that still have the original color), where expansion involves setting the pixel to the new color and pushing its coordinates onto the frontier. The algorithm terminates when the frontier is empty.

```
FLOODFILL(I, p, new-color)
 1  orig-color ← I(p)
 2  frontier.push(p)
 3  I(p) ← new-color
 4  while NOT frontier.isEmpty() do
 5        p ← frontier.pop()
 6        for q ∈ N(p) do
 7              if I(q) == orig-color
 8              then frontier.push(q)
 9                   I(q) ← new-color
10  return I
```

In this code $\mathcal{N}(q)$ is the set of neighbors of q, and typically either 4-neighbors or 8-neighbors are used. The double equal signs in Line 7 test for equality. The algorithm performs a depth-first search, since the frontier stack supports only LIFO (last-in-first-out) operations. If the stack is replaced by a queue, then the FIFO (first-in-first-out) operations will cause a breadth-first search instead, but the output will be the same either way, so we use a stack because its memory management is simpler.

Variations on the algorithm are easy to obtain by making minor modifications to this basic pseudocode. One common variation is to leave the original input image intact and instead to change an output image. The algorithm below implements this variation, setting each pixel $O(x, y)$ in the output if the pixel $I(x, y)$ in the input would have been changed by the previous algorithm. This version of the algorithm will be used in the next section on connected components, as well as in Chapter 7 on segmentation. It does not return a value, since it operates on the output image that is passed into the procedure.

```
FLOODFILLSEPARATEOUTPUT(I, O, p, new-color)
1  orig-color ← I(p)
2  frontier.push(p)
3  O(p) ← new-color
4  while NOT frontier.isEmpty() do
5        p ← frontier.pop()
6        for q ∈ N(p) do
7              if I(q) == orig-color
8              then frontier.push(q)
9                   O(q) ← new-color
```

2.5.2 Connected components

Recall that two pixels are said to be **connected** if there is a path between them consisting of pixels all having the same value. A **connected component** is defined as a maximal set of pixels that are all connected with one another. It is often useful to be able to find all the connected components in an image, for example to separate the various foreground objects in a binary image. Given a binary image with `ON` pixels signifying foreground and `OFF` pixels indicating background, the result of a connected components algorithm is a two-dimensional array (the same size as the image) in which each element has been assigned an integer label indicating the region to which its pixel belongs. That is, all the pixels in one contiguous foreground region are assigned one label, while all the pixels in a different contiguous foreground region are assigned a different label, all the pixels in a contiguous background region are assigned yet another label, and so forth. Thus, connected components is a partitioning problem, because it assigns the image pixels to a relatively small number of discrete groups.

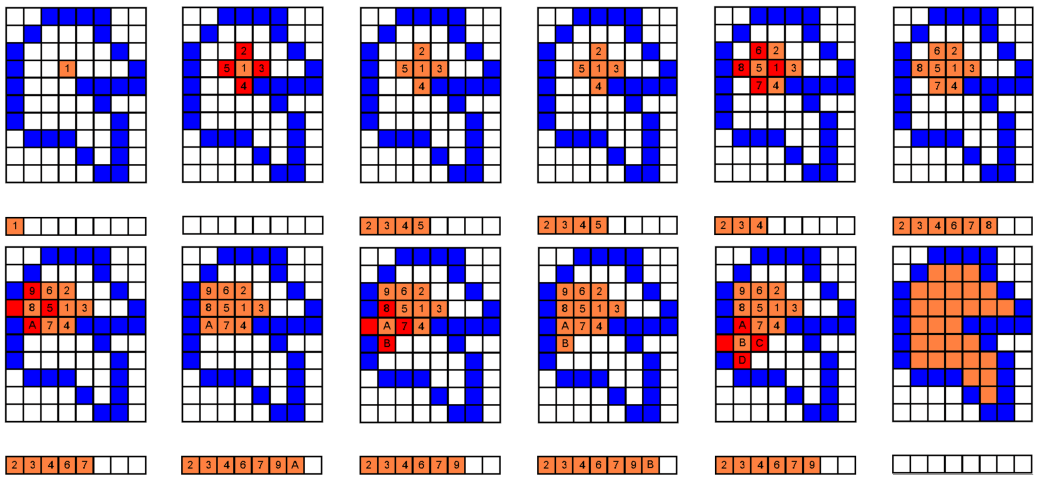

Figure 2.8: Step-by-step illustration of the 4-neighbor FLOODFILL algorithm on a small image. The frontier is shown below the image. Starting from the seed pixel labeled 1, the interior region of white pixels is changed to orange by the algorithm, while red is used to indicate the pixels being considered in the current expansion. The labels are artificially introduced to aid in associating pixels in the image with those in the frontier.

One way to implement connected components is by repeated applications of floodfill, starting from a new unlabeled pixel as the seed point for each iteration. Initially the output array L is created to be the same size as the input image, all elements in this output array are unlabeled, and a global label is set to zero. Then the image is scanned, and whenever a pixel is encountered that has not yet been labeled, floodfill is applied to the image with that pixel as the seed pixel, filling the elements in the output array with the global label. The global label is then incremented, and the scan is continued. This relatively simple procedure labels each pixel with the value of its contiguous region. One advantage of this algorithm is that the regions are labeled with consecutive labels of $0, 1, 2, \ldots$ so that the number of regions found is the global label minus one.

CONNECTEDCOMPONENTSBYREPEATEDFLOODFILL(I)

```
1  for (x, y) ∈ L do
2        L(x, y) ← UNLABELED
3  next-label ← 0
4  for (x, y) ∈ L do
5        if L(x, y) == UNLABELED then
6              FLOODFILLSEPARATEOUTPUT(I, L, (x, y), next-label)
7              next-label ← next-label +1
8  return L
```

A more common approach, sometimes known as the *classic connected components algorithm*, involves scanning the image twice. In the first pass, the image is scanned from left-to-right and from top-to-bottom, and all the pixels are labeled with preliminary labels based on a subset of their neighbors. For 4-neighbor connectedness, the algorithm compares a pixel with its two neighbors above and to the left; for 8-neighbor connectedness, the pixel is also compared with the two neighbors diagonally above-left and above-right (see Figure 2.9). While performing the preliminary labeling, an equivalence table is built to keep track of which preliminary labels need to be merged. In the second pass, the label of each pixel is set to the equivalence of its preliminary label, using the equivalence table. This approach is also known as a **union-find** algorithm because it performs the two operations of finding regions and

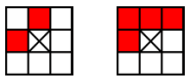

Figure 2.9: Masks for the 4-neighbor and 8-neighbor versions of the classic union-find connected components algorithm. The red pixels are neighbors of the central pixel that are examined by the algorithm.

merging them. It is the first algorithm we have considered where the order in which the pixels are processed matters.

```
ConnectedComponentsByUnionFind(I)
 1  ; first pass
 2  for y ← 0 to height − 1 do
 3        for x ← 0 to width − 1 do
 4              v ← I(x, y)
 5              if v == I(x − 1, y) and v == I(x, y − 1) then
 6                    L(x, y) ← L(x − 1, y)
 7                    SetEquivalence(L(x − 1, y), L(x, y − 1))
 8              elseif v == I(x − 1, y) then
 9                    L(x, y) ← L(x − 1, y)
10              elseif v == I(x, y − 1) then
11                    L(x, y) ← L(x, y − 1)
12              else
13                    L(x, y) ← next-label
14                    next-label ← next-label +1
15 ; second pass
16 for (x, y) ∈ L do
17       L(x, y) ← GetEquivalentLabel(L(x, y))
18 return L
```

An example of this 4-connected version of the algorithm at work can be seen in Figure 2.10. The output on a real image is displayed in Figure 2.11. To extend the code to 8-neighbors, simply insert two additional tests comparing the pixel with its neighbors $I(x-1, y-1)$ and $I(x+1, y-1)$. Equivalences are set between any of the four neighboring pixels (left, above, above-left, and above-right) with the same value as the pixel. Note that out-of-bounds accessing has been ignored; to turn this pseudocode into executable code, bounds checking must be added in the **if** and **elseif** clauses, so that the top-left pixel $(0, 0)$ in the image falls through to the **else** clause, and all remaining pixels along the top row and left column are only compared with existing pixels.

The algorithm relies on two helper functions. The first function, SetEquivalence, sets the equivalence between two labels, storing the equivalence in a one-dimensional array of integers, *equiv*. The array is initialized with its own indices, i.e., $equiv[i] \leftarrow i$ for all i. The convention is adopted that $equiv[i] \leq i$, to avoid creating cycles in the data structure. (It is assumed that the array grows dynamically in size or is created large enough to hold the total number of labels encountered.)

```
SetEquivalence(a, b)
1  a′ ← GetEquivalentLabel(a)
2  b′ ← GetEquivalentLabel(b)
3  if a′ > b′ then
4        equiv[a′] ← b′
5  else
6        equiv[b′] ← a′
```

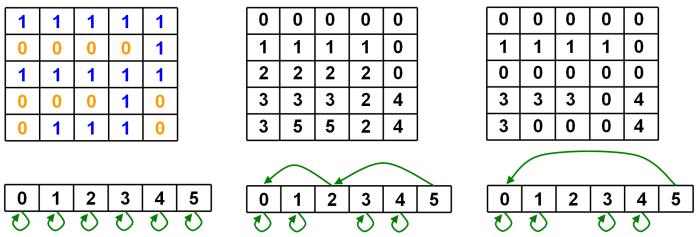

Figure 2.10: Classic union-find connected components algorithm on an example binary image. From left to right: The input image, the labels after the first pass, and the labels after the second pass. Below the image is the equivalence table, with green arrows pointing from a label to its equivalent label. Notice that the final image contains gaps, for example no pixel is labeled 2.

The second helper function, GetEquivalentLabel, returns an equivalent label simply by accessing the array, using recursion to ensure that the smallest possible label has been found. While getting an equivalent label, the array is updated with the smallest possible equivalent label. An alternative is to traverse the equivalence table once between the two passes, after which GetEquivalentLabel(a) can simply call **return** $equiv[a]$ without having to resort to recursion.

GetEquivalentLabel(a)
1 **if** $a =$ GetEquivalentLabel(a) **then**
2 **return** a
3 **else**
4 $equiv[a] \leftarrow$ GetEquivalentLabel($equiv[a]$)
5 **return** $equiv[a]$

Both algorithms for connected components are linear in the number of pixels. To be more precise, the union-find algorithm applied to an image with n pixels is $O(n\alpha(n))$, where $\alpha(n)$ is the inverse Ackerman function that grows so extremely slowly that $\alpha(n) \leq 4$ for any conceivable image (one with fewer than a googol of pixels, for example). The four-neighbor floodfill version requires touching most pixels seven times (to set the output to UNLABELED in the initialization, to check whether the pixel has been labeled, and 5 times during the floodfill to set the pixel and check its label from the four directions); pixels along the border of two regions may require slightly more. The union-find algorithm involves touching each pixel just four times (the first pass, the second pass, and the check from the pixels to its right and below). Thus, in practice the union-find algorithm is usually slightly more efficient in run time despite the additional computation required by the equivalence table. However, one drawback of union-find is that it leaves gaps in the labels. That is, the final result might have (as in the example of Figure 2.10) a region 1 and a region 3, but no region of pixels labeled 2. This inconvenience can be removed by another pass through the equivalence table to produce a new equivalence table in which the base labels are sequential.

With either algorithm, it is easy to compute properties of the regions found such as area, moments, and bounding box. All of these quantities can be updated during the connected components algorithm with appropriate calculations each time an output pixel is set, with minimal overhead.

2.5.3 Wall follow

Given a contiguous region of pixels found by a floodfill or connected components algorithm, it is oftentimes useful to find its *boundary*. We distinguish between the *exterior boundary*, which is the smallest set of connected pixels that encloses all the pixels in the region, and the *interior boundary*, which is the smallest set of connected pixels that encloses all the holes (if any) inside the region. If the region contains no holes, then the interior boundary is null. We refer to the union of the exterior and interior

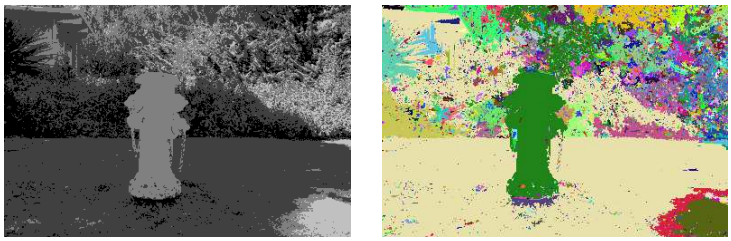

Figure 2.11: Because the connected components algorithm assumes neighboring pixels have the exact same value, it works best on images with a small number of values. Shown here are an input image quantized to four gray levels (left) and the result of connected components (right), pseudocolored for display.

boundaries as the *complete boundary*. In each case, we use *inner boundary* to refer to the pixels in the region that are adjacent to some pixel not in the region, while the *outer boundary* refers to pixels out of the region that are adjacent to some pixel in the region.

Let R be a region represented as a binary image, that is $R(p) = \texttt{ON}$ if p is in the region, and $R(p) = \texttt{OFF}$ otherwise, where $p = (x, y)$ is a pixel. The inner complete boundary of the region can be computed easily enough by simply computing the difference between the region itself and an eroded version of the region:

$$R \,\&\, \overline{(R \ominus B)}, \tag{2.7}$$

where the ampersand & indicates the logical `AND` operator, and the over line indicates the binary complement. See Figure ??. That is, each pixel in the output is `ON` if and only if the corresponding pixel in R is `ON` but in the eroded version is `OFF`. Although the outer complete boundary can be computed in a similar manner using dilation, $(R \oplus B)$ & $\overline{R}$, its usefulness is limited because outer boundaries cannot easily be represented for any region that touches the image border. Either way, the choice of the structuring element B will affect the result: B_4 will produce a 4-connected boundary, whereas B_8 will produce an 8-connected boundary.

While morphology provides a conveniently compact description of the boundary, procedures based on morphology — such as the one in (2.7) — only return the *set* of pixels on the boundary. However, for some applications it is necessary to compute the boundary as a path, i.e., as a *sequence* of pixels. One common use of such a sequence is to calculate the perimeter of a region, which we shall explore in a moment. A simple procedure for computing the boundary of a region as a path is the **wall following algorithm**. The wall following algorithm derives its name from the analogy of a person desiring to traverse the edges of a room blindfolded. By holding out his left arm stiff to the side and his right arm stiff in front, the person continually walks straight until either contact with the left wall has been lost (in which case the person turns left) or the person detects a wall in front (in which case the person turns right). In a similar manner, the wall following algorithm traverses the boundary of a region by examining pixels in front and to the left, turning appropriately based upon the values of the pixels. The algorithm below computes the clockwise inner boundary – other variations can be easily obtained from this one.

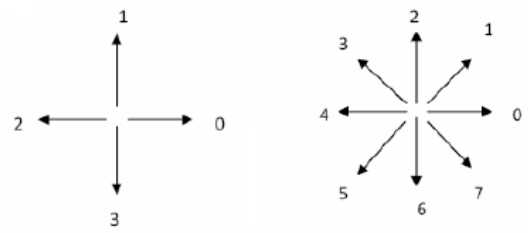

Figure 2.12: Freeman chain code directions for 4- and 8-neighbors.

```
WallFollow(I)
 1  p ← p_0 ← FindBoundaryPixel(I)
 2  dir ← 0
 3  while Front(I, p, dir) == ON do
 4        dir ← dir_0 ← TurnRight(dir)
 5  repeat
 6        boundary-path .Push(p)
 7        if Left(I, p, dir) == ON then
 8              dir ← TurnLeft(dir)
 9              p ← MoveForward(p, dir)
10        elseif Front(I, p, dir) == OFF then
11              dir ← TurnRight(dir)
12        else
13              p ← MoveForward(p, dir)
14  until p == p_0 AND dir == dir_0
15  return boundary-path
```

We adopt the convention of the Freeman chain code directions, shown in Figure 2.12, in which the list of directions proceeds counterclockwise starting from the right (positive x axis). For 4-neighbor connectedness, *dir* takes on values in the set $\{0, 1, 2, 3\}$, while for 8-neighbor connectedness, *dir* takes on values of $\{0, 1, \ldots, 7\}$. Therefore, in the code TurnLeft and TurnRight return the next and previous direction in the list, respectively, using modulo arithmetic: TurnLeft(*dir*) returns $(dir - 1) \bmod z$, while TurnRight(*dir*) returns $(dir + 1) \bmod z$, assuming z-neighbor connectedness. MoveForward computes the pixel attained after moving forward according to the current direction and the current pixel. Recalling that the y axis points down, this means MoveForward$(p, 0)$ returns $p + (1, 0)$, and MoveForward$(p, 1)$ returns $p + (0, -1)$ if $z = 4$ or $p + (1, -1)$ if $z = 8$. The function Left returns the pixel immediately to the left as one faces the current direction. For example, assuming $z = 4$, Left$(I, p, 0)$ returns $I(p + (0, -1))$, that is, the pixel directly above p in the fixed image coordinate system.

While the algorithm is very simple, careful attention must be paid to the starting condition. To ensure that FindBoundaryPixel returns one of the pixels on the exterior boundary of the region, it is best to start at the boundary of the image and search until a pixel in the region is found. In contrast, starting from an arbitrary pixel inside the image or region may yield an interior boundary pixel rather than an exterior boundary pixel, in which case the rest of the algorithm will mistakenly trace a hole rather than the exterior boundary. However, for the ending condition, it is necessary (as shown in the code) to test for pixel location *and* direction to handle the case of a single-pixel-thick bridge or protrusions.

Wall following is useful for several tasks. First, to compute the perimeter of a region, apply the 8-neighbor version of WallFollow, then compute the distance along the resulting path using the techniques described in the next section. (Note that the 4-neighbor version can be used as well, but in

the resulting sequence of pixels the distance between any two consecutive pixels will be 1. As a result, the computed perimeter will simply be the number of pixels remaining after subtracting the eroded image from the original region, using B_8 as the structuring element.) Secondly, to fill all the holes in a region, call WallFollow to find the exterior boundary, then call Floodfill to fill the interior.

The path computed by the 4-neighbor version of WallFollow contains pixels that are 8-neighbors of the background, while the path computed by the 8-neighbor version contains pixels that are 4-neighbors of the background. While consecutive pixels in the former path are 4-neighbors, in the latter path some of the consecutive pixels may be 4-neighbors while others are 8-neighbors. This blending of 4- and 8-neighbors leads to something called m-adjacency:

- Two pixels p and q in an image I are *m-adjacent* if $I(p) = I(q)$ and ($q \in \mathcal{N}_4(p)$ or ($q \in \mathcal{N}_D(p)$ and $\mathcal{N}_4(p) \bigcap \mathcal{N}_4(q) = \emptyset$)).

In other words, two pixels are m-adjacent if they are either 4-adjacent or they are 8-adjacent and there does not exist another pixel which is 4-adjacent to both of them. It can be shown that 8-neighbor WallFollow produces a path in which consecutive pixels are m-adjacent if we define the test $I(p) = I(q)$ to be false iff one pixel is on the path while the other is not.

2.6 Computing distance in a digital image

For many applications it is important to be able to compute the distance between two points in an image, for example to measure the perimeter of a region or the length of an object. We now discuss techniques for performing this task.

2.6.1 Distance functions

Let $p = (x_p, y_p)$ and $q = (x_q, y_q)$ be the coordinates of two pixels in an image. A function $d(p, q)$ of two vectors is a distance function (or **metric**) if it satisfies three properties:

- $d(p, q) \geq 0$ and $d(p, q) = 0$ iff $p = q$ (non-negativity)
- $d(p, q) = d(q, p)$ (symmetry)
- $d(p, q) \leq d(p, r) + d(r, q)$, $\forall r$ (triangle inequality)

A function satisfying only the first two conditions is called a **semi-metric**, an example being the quadratic function $\|p-q\|^2 = (x_p - x_q)^2 + (y_p - y_q)^2$. (There is also something called an **ultra-metric**, but we will not worry about that until Chapter 7.) Three common distance functions for pixels are the Euclidean distance, which is the square root of the quadratic function, and two others:

$$\begin{aligned} d_E(p, q) &= \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2} && \text{Euclidean} \\ d_4(p, q) &= |x_p - x_q| + |y_p - y_q| && \text{Manhattan, or city-block} \\ d_8(p, q) &= \max\{|x_p - x_q|, |y_p - y_q|\} && \text{chessboard} \end{aligned}$$

The Manhattan distance is known as d_4 because the pixels that are one unit of distance away are the 4-neighbors of the pixel. The chessboard distance is known as d_8 because the pixels that are one unit of distance away are the 8-neighbors of the pixel. Manhattan always overestimates Euclidean, while chessboard always underestimates it: $d_8(p, q) \leq d_E(p, q) \leq d_4(p, q)$. Moreover, chessboard is never more than 30% away from the Euclidean value: $0.7 d_E(p, q) < d_8(p, q) \leq d_E(p, q)$. To see these relations, note that for any two non-negative numbers $a, b \geq 0$,

$$\frac{1}{2}(a + b) \leq \frac{1}{\sqrt{2}}\sqrt{a^2 + b^2} \leq \max(a, b) \leq \sqrt{a^2 + b^2} \leq a + b. \tag{2.8}$$

The proof of these relations is straightforward and left as an exercise.

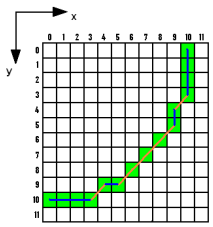

Figure 2.13: A sampled 90-degree sector of a circle with radius 10. The true arc length is $10(\pi/2) = 15.7$. The estimated path length according to the three formulas is 16.5 (Freeman), 15.2 (Pythagorean), and 15.7 (Kimura-Kikuchi-Yamasaki). Blue lines indicate isothetic moves, while orange lines indicate diagonal moves.

2.6.2 Path length

Now suppose that we wish to calculate the length of a specific path ϕ between pixels p and q, which as we saw earlier is defined as a sequence of pixels beginning with p and ending with q such that each successive pixel in the path is adjacent to the previous pixel. Let $n_{i,\phi}$ be the number of *isothetic* moves in the path, where an isothetic move is one which is horizontal or vertical (i.e., the two pixels are 4-neighbors of each other). Similarly, let $n_{d,\phi}$ be the number of *diagonal* moves in the path (i.e., the two pixels are D-neighbors of each other). Generally the path will use m-adjacency, but if 4-adjacency is used then $n_{d,\phi}$ is simply zero.

Even in a continuous space the length of a path (or curve) is not always well-defined (consider the well-known fractal question, "What is the length of the British coastline?"). In a discrete image it is also impossible to precisely define or solve the problem, but nevertheless one reasonable approach is to sum the Euclidean distances between consecutive pixels along the path. Since the Euclidean distance between two pixels that are 4-neighbors of each other is 1, and the Euclidean distance between two pixels that are D-neighbors of each other is $\sqrt{2}$, this is equivalent to measuring the length of the path ϕ as

$$\text{length}(\phi) = n_{i,\phi} + n_{d,\phi}\sqrt{2}, \qquad \text{(Freeman)} \tag{2.9}$$

which is known as the Freeman formula. An alternate approach is to rearrange the node pairs and use the Pythagorean theorem to estimate the length of the curve as the hypotenuse of the resulting right triangle:

$$\text{length}(\phi) = \sqrt{n_{d,\phi}^2 + (n_{i,\phi} + n_{d,\phi})^2}. \qquad \text{(Pythagorean)} \tag{2.10}$$

While the Freeman formula generally overestimates the length of a curve, the Pythagorean theorem usually underestimates it.

Insight into the problem is obtained by noticing that the previous two equations can be written as special cases of the more general formula:

$$\text{length}(\phi) = \sqrt{n_{d,\phi}^2 + (n_{d,\phi} + cn_{i,\phi})^2} + (1-c)n_{i,\phi}, \qquad \text{(Kimura-Kikuchi-Yamasaki)} \tag{2.11}$$

where $c = 0$ for the Freeman formula and $c = 1$ for the Pythagorean theorem. By setting c to $\frac{1}{2}$, we achieve a compromise between overestimation and underestimation. This approach, known as the Kimura-Kikuchi-Yamasaki method, tends to work well in practice. An example of the three measures applied to a sampled continuous curve is shown in Figure 2.13.

2.6.3 Chamfering

We have seen how to compute the distance between two pixels, as well as the distance between two pixels along a specific path. But what if we want to compute a large number of distances? Such a

problem arises, for example, when performing template matching using intensity edges, in which we need to compute distances between all the pixels in the image and a substantial subset (the intensity edges) of the pixels. For reasons of computational efficiency, it is not feasible to directly compute all of these distances directly. Instead, we will employ a computational trick that will enable us to efficiently compute a good approximation.

Let us define the (a, b) **chamfer distance** between pixel $p = (x_p, y_p)$ and pixel $q = (x_q, y_q)$ as $d_{a,b}(p, q) = \min_\phi\{an_{i,\phi} + bn_{d,\phi}\}$, where a and b are nonnegative values, the number of isothetic and diagonal moves in the path ϕ are given by $n_{i,\phi}$ and $n_{d,\phi}$, respectively, and the minimum is computed over all possible paths ϕ between the two pixels. We shall assume that $0 < a \leq b$, in which case the (a, b) chamfer distance $d_{a,b}$ is a metric. We shall also assume the *Montanari condition*, $b \leq 2a$, to ensure that diagonal moves are not ignored. Rather than searching over all possible paths, simple observation reveals that the shortest path always consists of a horizontal or vertical line segment, along with a diagonal line segment if the two pixels do not share the same column or row. That is, if we define $d_x = |x_p - x_q|$ and $d_y = |y_p - y_q|$, then the distance between p and q is simply given by $d_{a,b}(p, q) = an_i + bn_d$, where $n_d = \min(d_x, d_y)$ is the number of moves along the diagonal line segment and $n_i = \max(d_x, d_y) - n_d$ is the number of remaining isothetic moves. This single formula arises because the Montanari condition favors diagonal moves. If, on the other hand, the Montanari condition does not hold, then diagonal moves are ignored, in which case the distance is $d_{a,b}(p, q) = a(d_x + d_y)$, which is a scaled version of the Manhattan distance.

The Euclidean distance is usually considered the "correct" distance, while other distance functions are approximations to it, often referred to as *quasi-Euclidean*. It turns out that the (a, b) chamfer distance that best approximates Euclidean is the one with $a = 1$ and $b = \frac{1}{\sqrt{2}} + \sqrt{\sqrt{2} - 1} \approx 1.351$. A nearby integer ratio is 4/3, so if there is a need to avoid floating point computations, then $d_{3,4}$ yields a reasonable approximation to the Euclidean distance (scaled by the factor 3). It is easy to see that if $a = 1$ and $b = \infty$, the chamfer distance reduces to the special case of Manhattan because it computes the minimum number of isothetic moves between the pixels, or if $a = 1$ and $b = 1$, then the chamfer distance reduces to chessboard because it treats isothetic and diagonal moves equally. This relationship between Euclidean and quasi-Euclidean helps shed light on why this procedure is called *chamfering*. In woodworking, chamfering refers to the process of reducing the harsh 90-degree angles of a surface by introducing a beveled edge. In a similar manner, the chamfer distance in an image approximates the Euclidean distance by smoothing out the harsh corners of the Manhattan or chessboard distances by allowing appropriately weighted diagonal moves.

The algorithm to compute the (a, b) chamfer distance is straightforward. It involves two passes through the image, with the first pass scanning the image from left-to-right and top-to-bottom, then the second pass scanning in the reverse direction. For each pixel, four of its 8-neighbors are examined in one direction, then the other four in the reverse direction. One can think of the algorithm as casting shadows from the foreground pixels in the two diagonal directions (southeast in the first pass, and northwest in the second). After the first pass, the distances from every pixel to all of the pixels above and to the left have been computed, and after the second pass, the distances to all of the pixels have been computed.

```
CHAMFER(I; a, b)
 1  ; first pass
 2  for y ← 0 to height − 1 do
 3        for x ← 0 to width − 1 do
 4              if I(x, y) == ON then
 5                    δ(x, y) ← 0
 6              else
 7                    δ(x, y) ← MIN(∞, a + δ(x − 1, y), a + δ(x, y − 1), b + δ(x − 1, y − 1), b + δ(x + 1, y − 1))
 8  ; second pass
 9  for y ← height − 1 to 0 step -1 do
10        for x ← width − 1 to 0 step -1 do
11              if I(x, y) ≠ ON then
12                    δ(x, y) ← MIN(δ(x, y), a + δ(x + 1, y), a + δ(x, y + 1), b + δ(x + 1, y + 1), b + δ(x − 1, y + 1))
13  return δ
```

In the code $\delta(x, y) = \infty$ if x or y is out of bounds, and ∞ is meant to represent a large number that is greater than any possible distance in the image. The chamfering algorithm is unique in that the manner in which boundary pixels should be handled is specified precisely: Out of bounds pixels should simply be ignored, so that the minimum is computed over less than five values when the pixel is along the image border. Figure 2.14 shows an example of the chamfering algorithm applied to an image, while Figure 2.15 illustrates one use of the chamfer distance.

For the special case of Manhattan distance ($a = 1$ and $b = \infty$), the code simplifies to only require examining two of the 4-neighbors in each pass:

```
CHAMFERMANHATTAN(I)
 1  ; first pass
 2  for y ← 0 to height − 1 do
 3        for x ← 0 to width − 1 do
 4              if I(x, y) == ON then
 5                    δ(x, y) ← 0
 6              else
 7                    δ(x, y) ← MIN(∞, 1 + δ(x − 1, y), 1 + δ(x, y − 1))
 8  ; second pass
 9  for y ← height − 1 to 0 step -1 do
10        for x ← width − 1 to 0 step -1 do
11              if I(x, y) ≠ ON then
12                    δ(x, y) ← MIN(δ(x, y), 1 + δ(x + 1, y), 1 + δ(x, y + 1))
13  return δ
```

Since Manhattan always overestimates the Euclidean distance, while chessboard always underestimates the Euclidean distance, a combination of the two is sometimes used. One approach is to alternate the computations between the two distance metrics as the image is scanned; this requires two passes through the image, as usual. Another approach is to compute both d_4 and d_8 and then to combine the results: $\max\{d_8(p, q), \frac{2}{3}d_4(p, q)\}$, which requires four passes through the image. Of course, as mentioned earlier, a good approximation can also be obtained by simply computing $d_{3,4}$ and then dividing by three.

2.7 Moments

Suppose we have a region in an image found by thresholding or some other algorithm. Let us define the region by a non-negative *mass density function* $f(x, y) \geq 0$ defined over the image domain, where

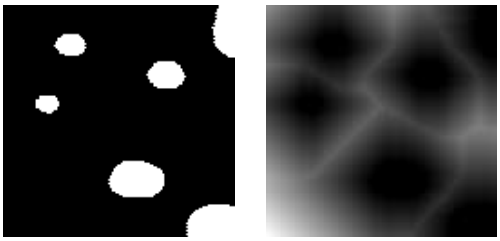

Figure 2.14: A binary image and its chamfer distance (brighter pixels indicate larger distances).

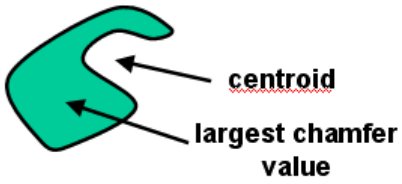

Figure 2.15: For a region with concavities, its centroid may not even lie within the region. Therefore, the location with maximum chamfer distance (computed on the inverted image, so that the distance to the background is computed) is often a better estimate of the "center" of a region, because it provides the center of the largest part of the region.

the function generally returns larger values inside the region than outside. We will focus our attention on the simplest (and most common) case of a binary region in which $f(x, y) = 1$ inside the region and $f(x, y) = 0$ outside the region, but the analysis of this section applies to any non-negative function. As another possibility, f could be a grayscale image obtained by applying the binary mask of the region to the original intensity image or the result of some probability map generation. Notice that we do *not* require $\sum_{x,y} f(x, y) = 1$.

Properties of the region can be determined by computing its moments. Given the function f and non-negative integers p and q, the pqth **moment** of a 2D region is defined as:

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y). \tag{2.12}$$

We say that the pqth moment is of order $p + q$. Thus, the zeroth-order moment is m_{00}, the first-order moments are m_{10} and m_{01}, and the second-order moments are m_{20}, m_{02}, and m_{11}. Computing these moments is easy, requiring a single pass through the image:

ComputeMoments(f)
1 $m_{00} \leftarrow m_{10} \leftarrow m_{01} \leftarrow m_{20} \leftarrow m_{02} \leftarrow m_{11} \leftarrow 0$
2 **for** $(x, y) \in f$ **do**
3 $\quad m_{00} \leftarrow m_{00} + f(x, y)$
4 $\quad m_{10} \leftarrow m_{10} + x * f(x, y)$
5 $\quad m_{01} \leftarrow m_{01} + y * f(x, y)$
6 $\quad m_{20} \leftarrow m_{20} + x * x * f(x, y)$
7 $\quad m_{02} \leftarrow m_{02} + y * y * f(x, y)$
8 $\quad m_{11} \leftarrow m_{11} + x * y * f(x, y)$
9 **return** $m_{00}, m_{10}, m_{01}, m_{20}, m_{02}, m_{11}$

The reader may recall the concept of moments from studying statics. The inertial properties of a solid planar body with a mass density function f can be calculated from the moments of f. In a

similar manner, the zeroth moment of an image region yields its area, the first order moments are used to compute its center of mass, and the second order moments are related to its moments of inertia. We define the centroid $(\bar{x}, \bar{y})$ of a region as simply the weighted average of the pixels, which can be easily computed from the moments:

$$(\bar{x}, \bar{y}) = \frac{1}{\sum_{x,y} f(x,y)} \left(\sum_{x,y} x f(x,y), \sum_{x,y} y f(x,y) \right) \tag{2.13}$$

$$= \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right). \tag{2.14}$$

Using our physical analogy, the centroid of the image region plays the same role as the center of mass of the rigid body specified by f. In the case of a binary function f, a cardboard cutout with the same shape as the region will remain (in the absence of external forces besides gravity) horizontal when suspended by a string attached at the center of mass.

The moments are affected by the location of the region in the image. To make the moments translation invariant, we define the pqth **central moment** as the pqth moment about the centroid:

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x,y).$$

It is easy to show that the central moments are functions of the regular moments:

$$\mu_{00} = m_{00} \tag{2.15}$$

$$\mu_{10} = 0 \tag{2.16}$$

$$\mu_{01} = 0 \tag{2.17}$$

$$\mu_{20} = m_{20} - \bar{x} m_{10} \tag{2.18}$$

$$\mu_{02} = m_{02} - \bar{y} m_{01} \tag{2.19}$$

$$\mu_{11} = m_{11} - \bar{y} m_{10} = m_{11} - \bar{x} m_{01}. \tag{2.20}$$

In other words, the central moments can be computed with just a single pass through the image:

ComputeCentralMoments(f)
1 $(m_{00}, m_{10}, m_{01}, m_{20}, m_{02}, m_{11}) \leftarrow$ ComputeMoments(f)
2 $\mu_{00} \leftarrow m_{00}$
3 $\bar{x} \leftarrow m_{10}/m_{00}$
4 $\bar{y} \leftarrow m_{01}/m_{00}$
5 $\mu_{20} \leftarrow m_{20} - \bar{x} * m_{10}$
6 $\mu_{02} \leftarrow m_{02} - \bar{y} * m_{01}$
7 $\mu_{11} \leftarrow m_{11} - \bar{y} * m_{10}$
8 **return** $\mu_{00}, \mu_{20}, \mu_{02}, \mu_{11}$

There is no need to compute μ_{10} or μ_{01} because they are always zero, and in Line 7 we could equivalently have used $\mu_{11} \leftarrow m_{11} - \bar{x} * m_{01}$.

Under a uniform scale change $x' = \alpha x, y' = \alpha y$, $\alpha \neq 0$, the central moments change according to $\mu'_{p,q} = \alpha^{p+q+2} \mu_{p,q}$. This result is easily shown using the continuous formulation of moments:

$$\begin{aligned} \mu'_{pq} &= \iint (x' - \bar{x}')^p (y' - \bar{y}')^q f(\frac{x'}{\alpha}, \frac{y'}{\alpha}) dx'\, dy' \\ &= \iint (\alpha x - \alpha \bar{x})^p (\alpha y - \alpha \bar{y})^q f(x,y) \alpha^2 dx\, dy \\ &= \alpha^{p+q+2} \iint (x - \bar{x})^p (y - \bar{y})^q f(x,y) dx\, dy \\ &= \alpha^{p+q+2} \mu_{pq} \end{aligned}$$

since, by change of variables, $dx' = \alpha\, dx$ and $dy' = \alpha\, dy$. Now for the zeroth moment ($p = q = 0$) we have $\mu'_{00} = \alpha^2 \mu_{00}$. Therefore, if we normalize a region's central moment by dividing by $\mu_{00}^{\frac{p+q+2}{2}}$ we obtain a quantity that does not change with scale:

$$\frac{\mu'_{pq}}{(\mu'_{00})^{\frac{p+q+2}{2}}} = \frac{\alpha^{p+q+2}\mu_{pq}}{(\alpha^2\mu_{00})^{\frac{p+q+2}{2}}} = \frac{\alpha^{p+q+2}\mu_{pq}}{\alpha^{p+q+2}\mu_{00}^{\frac{p+q+2}{2}}} = \frac{\mu_{pq}}{(\mu_{00})^{\frac{p+q+2}{2}}}. \tag{2.21}$$

This leads to the definition of the pqth **normalized central moment** as

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}}, \tag{2.22}$$

where $\gamma = \frac{p+q+2}{2}$ for $p + q \geq 2$. The case $p + q = 1$ is not included simply because $\mu_{10} = \mu_{01} = 0$, as shown earlier.

So far we have seen the central moments, which are invariant to translation; and the normalized central moments, which are invariant to translation and uniform scaling. A natural extension are the **Hu moments**, which are invariant to translation, rotation and uniform scale changes. These are given by:

$$\begin{aligned}
\phi_1 &= \eta_{20} + \eta_{02} \\
\phi_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
\phi_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
\phi_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
\phi_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
\phi_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
\phi_7 &= (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{12} - \eta_{30})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}$$

The first six values are also invariant to reflection, while ϕ_7 changes sign upon reflection, allowing to distinguish between mirror images.

Other invariants are the Legendre moments and the complex Zernike moments. While it is not possible to make the Legendre moments rotation invariant, the complex Zernike moments are generally superior to the Hu moments for rotation invariance.

2.8 Region Properties

Computing properties of an image region is important for classification, defect detection, and so forth. As we shall see, most of the important properties can be computed using the moments.

2.8.1 Area

The area of a region is given by its zeroth moment $m_{00} = \mu_{00}$. For a binary region, this is simply the number of pixels in the region. It may be worth noting that another way to estimate the area of a region, given the pixels defining the boundary (as in wall follow), is to use

$$\frac{1}{2}\sum_{i=0}^{n-1}(x_{i+1}y_i - x_i y_{i+1}), \tag{2.23}$$

where subscripts are computed modulo n, and n is the number of pixels in the boundary. This expression follows naturally from the definition of integration under the upperside of the curve subtracting the integration under the underside of the curve. One nice property of this equation is that it also holds for polygonal approximations of the region, by replacing the pixel coordinates with vertex coordinates and replacing the number of pixels with the number of vertices.

2.8.2 Orientation

Suppose we have a binary region containing pixels $\mathbf{x}_i = \begin{bmatrix} x_i & y_i \end{bmatrix}^T$, $i = 1, \ldots, n$, where vector notation is here used to represent the coordinates, and T refers to the transpose operation that makes $\mathbf{x}_i$ vertical. To capture relationships between the pixels, we calculate the covariance matrix of the region, which is defined as the mean of the outer products of the pixel coordinates after shifting the coordinate system to the region centroid:[2]

$$
\begin{aligned}
C &= \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T && (2.24)\\
&= \frac{1}{n}\sum_{i=1}^{n}\begin{bmatrix} x_i - \bar{x} \\ y_i - \bar{y} \end{bmatrix}\begin{bmatrix} x_i - \bar{x} & y_i - \bar{y} \end{bmatrix} && (2.25)\\
&= \frac{1}{n}\sum_{i=1}^{n}\begin{bmatrix} (x_i - \bar{x})^2 & (x_i - \bar{x})(y_i - \bar{y}) \\ (x_i - \bar{x})(y_i - \bar{y}) & (y_i - \bar{y})^2 \end{bmatrix} && (2.26)\\
&= \frac{1}{n}\begin{bmatrix} \sum_{i=1}^{n}(x_i - \bar{x})^2 & \sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \\ \sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) & \sum_{i=1}^{n}(y_i - \bar{y})^2 \end{bmatrix} && (2.27)\\
&= \frac{1}{\mu_{00}}\begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix}. && (2.28)
\end{aligned}
$$

Thus the covariance matrix of a binary 2D region is a simple function of the central moments:

$$C = \frac{1}{\mu_{00}}\begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix}. \tag{2.29}$$

In our physical analogy, the covariance matrix is the *inertia moment tensor* of the object and captures the moments of inertia about various axes. The diagonal elements are known as the *principal moments of inertia*, while the off-diagonal elements are the *products of inertia*. Just as the area captures the object's resistance to linear forces, these values capture the object's resistance to rotational forces about the axes. Note that this equation can be used to define a covariance matrix even for non-binary regions.

The orientation of the region is the angle of the major axis, where the major axis is defined as the axis about which the moment of inertia is minimized. Think of a baseball bat, where the principal axis is the axis of symmetry: It requires much less energy to rotate the bat about the axis of symmetry than it does to swing the bat, for example. It turns out that the angle θ of the major axis is given by a rather simple expression of the second-order central moments:

$$\tan 2\theta = \left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}}\right). \tag{2.30}$$

It is very important in this equation not to divide the numerator by the denominator. In other words, the angle θ must *not* be calculated as one half of the inverse tangent of the righthand side after performing ordinary division. The reason for this is that the inverse tangent function would in that case return an angle 2θ between $-\pi/2$ and $\pi/2$, which would then cause θ itself to be constrained to the range $-\pi/4$ to $\pi/4$, which of course does not represent the full range of possible line orientations. The solution to this problem is to treat the numerator and denominator as separate arguments to an inverse tangent function that computes the angle in the appropriate quadrant based on the individual signs of the numerator and denominator. Most programming languages have such a function, often called `atan2`. The right side of the equation indicates the rise over the run, so that the y argument to such a function

[2] Sometimes you will see the normalization of the covariance matrix to be $\frac{1}{n-1}$ instead of $\frac{1}{n}$. Such a definition is technically required to yield an unbiased estimate of the underlying distribution from which the sample points (x_i, y_i) are drawn. However, understanding this distinction is beyond the scope of this book. Thankfully, for any reasonably sized data set (e.g., $n \geq 100$), the difference between the two definitions is negligible.

is the numerator, while the x argument is the denominator. After calculating 2θ in such a manner, the desired angle θ is simply one half of the result. If all of the moments are computed using a standard image coordinate system with the positive x axis pointing right and the positive y axis pointing down, then the angle θ will be clockwise with respect to the positive x axis.

Another expression for the angle is given by

$$\tan\theta = \frac{\mu_{02} - \mu_{20} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2}}{2\mu_{11}} = \frac{2\mu_{11}}{\mu_{20} - \mu_{02} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2}}, \quad (2.31)$$

which comes from applying the double angle formula: $\tan 2\theta = \frac{2\tan\theta}{1-\tan^2\theta}$.

While proving Equation (2.30) is left as an exercise for the reader, some insight into the problem can be gained by examining the structure of the covariance matrix. Because C is real and symmetric, its eigenvalues (λ_1 and λ_2) are real and its eigenvectors ($\mathbf{e}_1$ and $\mathbf{e}_2$) are mutually orthogonal: $\mathbf{e}_1^T\mathbf{e}_2 = 0$. According to the definition of eigenvalues and eigenvectors, we have $C\mathbf{e}_i = \lambda_i\mathbf{e}_i$ for $i = 1, 2$. Stacking the eigenvectors into a matrix $P = [\,\mathbf{e}_1 \quad \mathbf{e}_2\,]$ and the eigenvalues into a diagonal matrix $\Lambda = diag(\lambda_1, \lambda_2)$ yields the equation $CP = P\Lambda$. Rearranging, we see that the covariance matrix has been factored into the product of a diagonal matrix and two orthogonal matrices: $C = P\Lambda P^T$, since $PP^T = I$ where I is the identity matrix.

The eigenvalues of C can be computed by solving the characteristic equation $\det(C - \lambda I) = 0$, leading to

$$\lambda_{\{1,2\}} = \frac{1}{2\mu_{00}}\left(\mu_{20} + \mu_{02} \pm \sqrt{(\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2}\right), \quad (2.32)$$

where λ_1 takes the plus sign and λ_2 takes the minus sign, so that $\lambda_1 \geq \lambda_2$. It is easy to show that the sum of the eigenvalues is the trace of the covariance matrix, $\lambda_1 + \lambda_2 = \mathrm{tr}(C) = (\mu_{20} + \mu_{02})/\mu_{00}$, and the product of the eigenvalues is its determinant, $\lambda_1\lambda_2 = det(C) = (\mu_{20}\mu_{02} - \mu_{11}^2)/\mu_{00}^2$. Note also that the eigenvalues are invariant to rotation, and their ratio is invariant to scale.

Since P is an orthogonal matrix, the vectors $\mathbf{e}_i$ form an orthonormal basis that spans the space. The matrix P is essentially a rotation of the coordinate axes (with a possible flipping if the determinant of P is -1). Thus P represents the axes of a coordinate system which, when the origin is placed at the centroid of the region, are aligned with the region. Adopting the convention that the eigenvalues are sorted in decreasing order, $\lambda_1 \geq \lambda_2$, the two columns of P represent the major and minor axis of the region, respectively. It can be shown (see the exercises) that the equation for the orientation above yields exactly the same orientation as that from computing the eigenvectors. In other words, $\mathbf{e}_1 = [\,\cos\theta \quad \sin\theta\,]^T$ and, from the property of orthogonality, $\mathbf{e}_2 = [\,-\sin\theta \quad \cos\theta\,]^T$.

One way of understanding the connection between the covariance matrix and the orientation of the region is to define the transformation $\mathbf{x}' = P^T(\mathbf{x} - \bar{\mathbf{x}})$. It is easy to show that the mean $\frac{1}{n}\sum_{i=1}^n \mathbf{x}_i' = 0$ and the resulting covariance matrix $\frac{1}{n}\sum_{i=1}^n \mathbf{x}_i'(\mathbf{x}_i')^T = \Lambda$. Since the covariance matrix is diagonal, the data are uncorrelated in the rotated coordinate system. Note that this idea of aligning coordinate axes with the data using eigenvalues and eigenvectors is the first step of *Principal Components Analysis (PCA)*. We will see more uses for PCA in later chapters.

2.8.3 Best-fitting ellipse

One reason for computing the moments is to approximate a region by a low-order parameteric model. This is a form of dimensionality reduction, where a small number of parameters are used to summarize an arbitrarily complicated region. Perhaps the simplest way to do this is to fit an axis-aligned non-isotropic 2D Gaussian to the region, setting the Gaussian mean as $(\bar{x}, \bar{y})$, and the variances in the two directions using the diagonal elements of C:

$$\sigma_x^2 = \mu_{20}/\mu_{00} \quad (2.33)$$

$$\sigma_y^2 = \mu_{02}/\mu_{00}. \quad (2.34)$$

The **level set** of a 2D function $G(x, y)$ is the set of points (x, y) such that $G(x, y) = h$ for some constant h, i.e., the set $\{(x, y) : G(x, y) = h\}$. Geometrically, the level set is the intersection of the function with the horizontal $z = h$ plane, and it can be thought of as a horizontal slice through the function, or equivalently a contour. With an appropriately defined value for h, the level set of the Gaussian will be an axis-aligned ellipse that defines an approximation to the region.

Since we know the orientation of the region, however, we can do even better by aligning the Gaussian with the region. This is easy to do using the complete matrix C to approximate the region using an ellipse obtained by slicing through the Gaussian represented by C. To see the precise connection between C and the ellipse, suppose that we have a binary region that is exactly defined as the set of pixels inside an ellipse centered at the origin:

$$\mathcal{R} = \{(x, y) : ax^2 + 2bxy + cy^2 \leq 1\} \tag{2.35}$$

with $ac \geq b^2$ to ensure that the equation describes an ellipse. It can be shown that the second-order central moments of the region are related to the coefficients of the ellipse in a simple way:

$$C = \frac{1}{\mu_{00}} \begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix} = \eta \begin{bmatrix} c & -b \\ -b & a \end{bmatrix}, \tag{2.36}$$

where $\eta = 4 \det(C)$ and $\det(C) = (\mu_{20}\mu_{02} - \mu_{11}^2)/\mu_{00}^2$ is the determinant of the covariance matrix. This relationship is reinforced by noting that the covariance matrix must be positive semidefinite, leading to $\det C \geq 0$, which leads to $\mu_{20}\mu_{02} \geq \mu_{11}^2$, which implies $ac \geq b^2$, which is exactly the requirement for the equation to be an ellipse.

There is also a simple relationship between the length of the ellipse axes and the eigenvalues of the covariance matrix:

$$\text{semi-major axis length} = 2\sqrt{\lambda_1} \tag{2.37}$$

$$\text{semi-minor axis length} = 2\sqrt{\lambda_2}, \tag{2.38}$$

where the semi-axis length is defined as the distance from the ellipse center to the intersection of the ellipse boundary with the major or minor axis, respectively. Since λ_i captures the variance along the axis, $\sqrt{\lambda_i}$ is the standard deviation, so that these expressions show that the ellipse corresponds to the level set that intersects the ellipse at $\pm 2\sigma_i$ in each direction.

Equations (2.37) and (2.38) are easy to derive by considering without loss of generality the simple case of an ellipse that is aligned with the coordinate axes ($b = 0$), so that $ax^2 + cy^2 = 1$ is the boundary of the ellipse. Such an ellipse crosses the x axis at $x = \pm\sqrt{\frac{1}{a}}$, and it crosses the y axis at $y = \pm\sqrt{\frac{1}{c}}$. Therefore the distance from the ellipse center to the intersection of the ellipse boundary with the x axis is given by $\sqrt{\frac{1}{a}}$. From (2.36), we know that $a = \frac{\mu_{02}}{\eta\mu_{00}}$. Noting that $\mu_{11} = 0$ since $b = 0$, we can substitute the definition of η from above, and we can also calculate $\lambda_1 = \mu_{20}/\mu_{00}$ from Equation (2.32). Putting these together yields the desired result:

$$\text{semi-major axis length} = \sqrt{\frac{1}{a}} = \sqrt{\frac{\eta\mu_{00}}{\mu_{02}}} = \sqrt{\frac{4\mu_{20}\mu_{02}\mu_{00}}{\mu_{00}^2\mu_{02}}} = \sqrt{\frac{4\mu_{20}}{\mu_{00}}} = 2\sqrt{\frac{\mu_{20}}{\mu_{00}}} = 2\sqrt{\lambda_1}. \tag{2.39}$$

Similar reasoning leads to the corresponding result for the length of the semi-minor axis. Since the eigenvalues and lengths are unaffected by rotation, the results hold even when $b \neq 0$.

Putting all of this together, we see that given any arbitrary binary region $\mathcal{R}$, we can compute its second-order central moments. We can then compute the orientation of the region either using Equation (2.30) or by calculating the eigenvalues and eigenvectors of the covariance matrix, then computing the angle of the eigenvector corresponding to the larger eigenvalue. The "best fitting ellipse" of the region (defined as the ellipse with the same second-order moments as the region) is given by $ax^2 + 2bxy + cy^2 = 1$, where the coefficients are determined by Equation (2.36). This ellipse is the level set of the non-isotropic Gaussian with standard deviations $\sqrt{\lambda_1}$ and $\sqrt{\lambda_2}$ oriented at an angle of θ, with the level corresponding to $\pm 2\sigma_i$ along the major and minor axes.

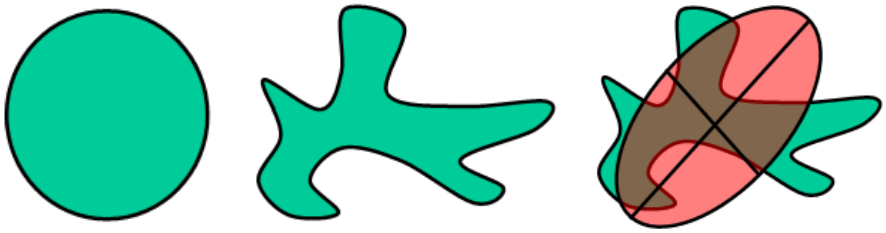

Figure 2.16: Left: A circle is the most compact shape, with a compactness of 1. Middle: A shape whose compactness is less than 1. Right: The eccentricity of the shape is computed as the eccentricity of the best fitting ellipse.

2.8.4 Compactness

Compactness is a measure of how close the pixels in the shape are to the center of the shape. Since the most compact shape is a circle, we define compactness as

$$\text{compactness} = \frac{4\pi(\text{area})}{(\text{perimeter})^2}, \tag{2.40}$$

so that a circle has a compactness of 1, since the area of a circle is πr^2 and its perimeter is $2\pi r$, where r is the radius. For all other shapes, their compactness according to this definition is less than 1 (between 0 and 1). The perimeter is typically computed by applying the Kimura-Kikuchi-Yamasaki distance to the boundary found by wall following. Note, however, that no matter how the perimeter is computed, discretization effects can cause the resulting compactness to be slightly greater than 1.

2.8.5 Eccentricity

The eccentricity of a region measures its elongatedness, that is, how far it is from being rotationally symmetric around its centroid. Since the eigenvalues capture the variance in the two principal directions, we define the eccentricity of a region as the difference between these variances, normalized by the larger variance. Units cause us to take the square root:

$$\text{eccentricity} = \sqrt{\frac{\lambda_1 - \lambda_2}{\lambda_1}}, \tag{2.41}$$

which ranges from 0 (when the region is a circle) to 1 (when the region is a straight line). Substituting Equation (2.32) yields the eccentricity in terms of the moments:

$$\text{eccentricity} = \sqrt{\frac{2\sqrt{(\mu_{20}-\mu_{02})^2 + 4\mu_{11}^2}}{\mu_{20}+\mu_{02}+\sqrt{(\mu_{20}-\mu_{02})^2 + 4\mu_{11}^2}}} = \sqrt{\frac{2\beta}{\text{tr(C)}+\beta}}, \tag{2.42}$$

where $\beta = \sqrt{\text{tr}^2(C) - 4\det(C)}$, and $\text{tr}^2(C)$ is the square of the trace of the covariance matrix.

There are several advantages to this definition of eccentricity. First, it matches the standard definition of the eccentricity of an ellipse, which is the ratio of the distance between the ellipse foci to the length of the major axis. Thus, the eccentricity of the region is equivalent to the eccentricity of the best fitting ellipse. Secondly, due to the normalization by λ_1, it is invariant to scale, which is a desirable property because it ties the eccentricity to the shape of the object without regard to its size in the image. Figure 2.16 shows an example of compactness and eccentricity.

Other definitions could be imagined for eccentricity, such as one minus the square root of the ratio of the two eigenvalues:

$$1 - \sqrt{\frac{\lambda_2}{\lambda_1}} = \sqrt{\frac{\mu_{20}+\mu_{02}-\sqrt{(\mu_{20}-\mu_{02})^2 + 4\mu_{11}^2}}{\mu_{20}+\mu_{02}+\sqrt{(\mu_{20}-\mu_{02})^2 + 4\mu_{11}^2}}}, \tag{2.43}$$

which also ranges from 0 (when the region is a circle) to 1 (when the region is a straight line). The primary drawback of this definition is that there is no straightforward relationship between a change in the axis lengths and the corresponding change in the value of the eccentricity. For example, a doubling of the ratio of the two axis lengths does not lead to a doubling of the eccentricity. This problem could be solved by removing the leading "one minus", but the resulting value would then decrease when the region became more elongated, and vice versa. An even worse measure is the difference between the two eigenvalues:

$$\lambda_1 - \lambda_2 = \frac{1}{\mu_{00}} \sqrt{(\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2}, \tag{2.44}$$

which ranges from 0 (when the region is a circle) to ∞ (when the region is a line). This definition suffers from two problems: (1) a doubling of the difference between the two principal axes leads to a quadrupling of the eccentricity, and (2) the eccentricity is dependent upon the scale of the region. Another definition that has been proposed by several authors is

$$\frac{(\mu_{20} - \mu_{02})^2 + 4\mu_{11}}{\mu_{00}}. \tag{2.45}$$

While this equation bears some resemblance to that proposed, it is fundamentally flawed because of the mismatch of units in the numerator, where one moment is added to the square of another.