Lecture 2: Image Processing Review, Neighbors, Connected Components, and Distance

©Bryan S. Morse, Brigham Young University, 1998–2000
Last modified on January 6, 2000 at 3:00 PM

Reading

SH&B, Chapter 2

2.1 Review of CS 450

2.1.1 Image Basics

Image Domains

An image (picture) can be thought of as being a function of two spatial dimensions:

$$f(x, y) \tag{2.1}$$

For monochromatic images, the value of the function is the amount of light at that point.

Sometimes, we can go to even higher dimensions with various imaging modalities. Medical CAT and MRI scanners produce images that are functions of three spatial dimensions:

$$f(x, y, z) \tag{2.2}$$

An image may also be of the form

$$f(x, y, t) \tag{2.3}$$

(x and y are spatial dimensions; t is time.) Since the image is of some quantity that varies over two spatial dimensions and also over time, this is a video signal, animation, or other time-varying picture sequence.

Be careful—although both volumes and time-varying sequencies are three-parameter types of images, they *are not* the same!

For this course, we'll generally stick to static, two-dimensional images.

The Varying Quantities

The values in an image can be of many types.

Some of these quantities can be scalars:

- Monochromatic images have a single light intensity value at each point.

Sometimes, these scalars don't correspond to quantities such as light or sound:

- In X-ray imaging, the value at each point corresponds to the attenuation of the X-ray beam at that position (i.e., not the radiation that gets through but the amount that *doesn't* get through).
- In one form of MR imaging, the value at each point indicates the number of single-proton atoms (i.e., hydrogen) in that area.
- Range images encode at each point in an image the distance to the nearest object at that point, not it's intensity. (Nearer objects are brighter, farther objects are darker, etc.)

Some signals don't have scalar quantities but vector quantities. In other words, multiple values at each point.

- Color images are usually stored as their red, green, and blue components at each point. These can be thought of as a 3-dimensional vector at each point of the image (a 2-dimensional space). Each color is sometimes called a *channel*.
- Satellite imaging often involves not only visible light but other forms as well (e.g., thermal imaging). LANDSAT images have seven distinct channels.

Sampling and Quantization

The spacing of discrete values in the domain of an image is called the *sampling* of that image. This is usually described in terms of some *sampling rate*–how many samples are taken per unit of each dimension. Examples include "dots per inch", etc.

The spacing of discrete values in the range of an image is called the *quantization* of that image. Quantization is usually thought of as the number of bits per pixel. Examples include "black and white images" (1 bit per pixel), "24-bit color images", etc.

Sampling and quantization are independent, and each plays a significant role in the resulting signal.

Resolution

Sampling and quantization alone, though, don't tell the whole story. Each discrete sample is usually the result of some averaging of the values around that sample. (We just can't make physical devices with infinitely small sampling areas.) The combination of the sampling and the averaging area for each sample determines the *resolution* of the digital signal.

For example, a digital monitor with 5000x5000 pixels and 24-bit color may sound great, but look before you buy. If the pixels are 0.1 mm apart but each pixel has a 10 mm spread, would you buy it?

Note: This differs from the definition of resolution given in your textbook, which defines resolution simply as the sampling rate. This is a common misconception.

Resolution is the ability to discern fine detail in the image, and while the sampling rate plays a factor, it is not the only factor.

2.1.2 The Delta Function

The *Dirac delta function* is defined as

$$\delta(x,y) = \begin{cases} \infty & \text{if } x=0 \text{ and } y=0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \delta(x,y)\ dx\ \ dy = 1$$

For discrete images, we use a discrete version of this function known as the *Kroenecker delta function*:

$$\delta(x,y) = \begin{cases} 1 & \text{if } x=0 \text{ and } y=0 \\ 0 & \text{otherwise} \end{cases}$$

and (as you'd expect)

$$\sum_{-\infty}^{\infty}\sum_{-\infty}^{\infty} \delta(x,y)\ dx\ \ dy = 1$$

One important property of the delta function is the *sifting property*:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\ \ \delta(x-a,y-b)\ \ dx\ \ dy = f(a,b)$$

2.1.3 Convolution

One of the most useful operations in image processing is *convolution*:

$$g(x,y) = f(x,y) * h(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(a,b)\;\; h(x-a, y-b)\;\; da\;\; db$$

Remember that convolution was useful for describing the operation of a *linear* and *shift-invariant* system. If $h(x,y)$ is the system's response to a delta function (impulse), the output of the system for *any* function $f(x,y)$ is $f(x,y) * h(x,y)$. Many useful systems can be designed by simply designing the desired convolution *kernel*.

2.1.4 The Fourier Transform

Another way of characterizing the operation of a linear, shift-invariant system is the *Fourier Transform*.

Remember that any image can be derived as the weighted sum of a number of sinusoidal images of different frequencies:

$$[a\cos(2\pi ux) + b\sin(2\pi ux)]\,[a\cos(2\pi vx) + b\sin(2\pi vx)]$$

where u is the frequency in the x direction and v is the frequency in the v direction.

For mathematical convenience and for more compact notation, we often write these using complex arithmetic by putting the cosine portion of these images as the real part of a complex number and the sine portion of these images as the imaginary part:

$$e^{i(2\pi ux)} = \cos(2\pi ux) + i\sin(2\pi ux)$$

To get the weights we use the **Fourier Transform**, denoted as $\mathcal{F}$:

$$F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-i2\pi(ux+vx)}\;\; dxdy$$

And to recombine the weighted sinusoids we use the **Inverse Fourier Transform**, denoted $\mathcal{F}^{-1}$:

$$f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{i2\pi(ux+vx)}\;\; dudv$$

What's unique about these sinusoidal images is that each goes through the system unchanged other than amplification. This amplification differs according to frequency. So, we can describe the operation of a system by measuring how each frequency goes through the system ($H(u,v)$)—a quantity called the system's *transfer function*. We can describe what happens to a particular image going through that system by decomposing any image into its weights for each frequency ($F(u,v)$), multiplying each component by its relative amplification ($F(u,v)H(u,v)$), and recombining the weighted, multiplied components.

So, the Fourier Transform of the output is the Fourier Transform of the input with each frequency component amplified differently. If $F(u,v)$ is the Fourier Transform of the input $f(x,y)$ and $G(u,v)$ is the Fourier Transform of the output $g(x,y)$,

$$G(u,v) = F(u,v)H(u,v)$$

2.1.5 The Convolution Theorem

The *convolution theorem* states that

$$g(x,y) = f(x,y) * h(x,y)$$

implies

$$G(u,v) = F(u,v)H(u,v)$$

and similarly (though often overlooked):

$$g(x,y) = f(x,y)h(x,y)$$

implies

$$G(u,v) = F(u,v) * H(u,v)$$

Notice that this implies that the transfer function $H(u,v)$ is the Fourier transform of the impulse response $h(x,y)$.

2.1.6 Linear Systems

To summarize, the relationship between a linear shift-invariant system, its input $f(x, y)$, the Fourier transform of its input $F(u, v)$, its output $g(x, y)$, and the transform of its output $G(u, v)$ can be summarized as follows:

1. The output $g(x, y)$ is the convolution of the input $f(x, y)$ and the impulse respones $h(x, y)$.
2. The transform $G(u, v)$ of the output is the product of the transform $F(u, v)$ of the input and the transfer function $H(u, v)$.
3. The transfer function is the Fourier transform of the impulse response.
4. The Convolution Theorem states that convolution in one domain is multiplication in the other domain, and vice versa.
5. It doesn't matter in which domain you choose to model or implement the operation of the system—mathematically, it is the same.

2.1.7 Sampling

Remember that a digital image is made up of samples from a continuous field of light (or some other quantity). If undersampled, artifacts can be produced. Shannon's sampling theorem states that if an image is sampled at less than twice the frequency of the highest frequency component in the continuous source image, *aliasing* results.

2.1.8 Color Images

In CS 450, you covered *color spaces* and *color models*. One of the key ideas from this is that rather than describing color as simply (red,green,blue) components, we can also describe it in a variety of ways as an intensity component and two *chromaticity* components. This is useful for vision because for some application we may wish to operate on (analyze) the intensity or hue components independently.

2.1.9 Histograms

As you learned in CS 450, a *histogram* of an image can be a useful tool in adjusting intensity levels. It can also be a useful tool in analyzing images, as we'll see later in Section 5.1 of your text.

2.1.10 Noise

Remember also from CS 450 that images usually have noise. We usually model this noise as

$$g(x, y) = f(x, y) + \tilde{n}(x, y)$$

where the noise $\tilde{n}(x, y)$ is added to the "real" input $f(x, y)$. So, although we'd ideally like to be analyzing $f(x, y)$, all we really have to work with is the noise-added $g(x, y)$.

2.2 Playing on the Pixel Grid: Connectivity

Many of the simplest computer vision algorithms involve what I (and others) call "playing on the pixel grid". These are algorithms that essentially involve operations between neighboring pixels on a rectangular lattice. While these algorithms are usually simple, they are often very useful and can sometimes become more complex.

2.2.1 Neighborhoods and Connectivity

One simple relationship between pixels is connectivity—which pixels are "next to" which others? Can you "get to" one pixel from another? If so, how "far" is it?

Suppose that we consider as neighbors only the four pixels that share an edge (not a corner) with the pixel in question: `(x+1,y)`, `(x-1,y)`, `(x,y+1)`, and `(x,y-1)`. These are called "4-connected" neighbors for obvious reasons.

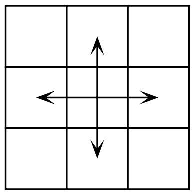

Figure 2.1: 4-connected neighbors.

Now consider the following:

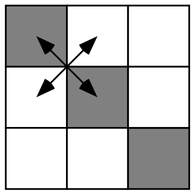

Figure 2.2: Paradox of 4-connected neighbors.

The black pixels on the diagonal in Fig. 2.2 are not 4-connected. However, they serve as an effective insulator between the two sets of white pixels, which are also not 4-connected across the black pixels. This creates undesirable topological anomalies.

An alternative is to consider a pixel as connected not just pixels on the same row or column, but also the diagonal pixels. The four 4-connected pixels plus the diagonal pixels are called “8-connected” neighbors, again for obvious reasons.

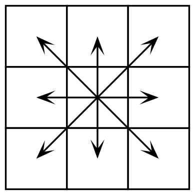

Figure 2.3: 8-connected neighbors.

But again, a topological anomaly occurs in the case shown in Figure 2.2. The black pixels on the diagonal are connected, but then again so are the white background pixels. Some pixels are connected across the links between other connected pixels!

The usual solution is to use 4-connectivity for the foreground with 8-connectivity for the background or to use 8-connectivity for the foreground with 4-connectivity for the background, as illustrated in Fig 2.4.

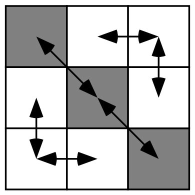

Figure 2.4: Solution to paradoxes of 4-connected and 8-connected neighbors: use different connectivity for the foreground and background.

Another form of connectivity is "mixed-connectivity" (Fig. 2.5, a form of 8-connectivity that considers diagonally-adjacent pixels to be connected if no shared 4-connected neighbor exists. (In other words, use 4-connectivity where possible and 8-connectivity where not.)

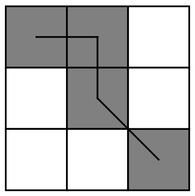

Figure 2.5: Mixed connectivity.

2.2.2 Properties of Connectivity

For simplicity, we will consider a pixel to be connected to itself (trivial connectivity). In this way, connectivity is reflexive.

It is pretty easy to see that connectivity is also symmetric: a pixel and its neighbor are mutually connected.

4-connectivity and 8-connectivity are also transitive: if pixel A is connected to pixel B, and pixel B is connected to pixel C, then there exists a connected path between pixels A and C.

A relation (such as connectivity) is called an equivalence relation if it is reflexive, symmetric, and transitive.

2.2.3 Connected Component Labeling

If one finds all equivalence classes of connected pixels in a binary image, this is called *connected component labeling*. The result of connected component labeling is another image in which everything in one connected region is labeled "1" (for example), everything in another connected region is labeled "2", etc.

Can you think of ways to do connected component labeling?

Here is one algorithm:

1. Scan through the image pixel by pixel across each row in order:
   - If the pixel has no connected neighbors with the same value that have already been labeled, create a new unique label and assign it to that pixel.
   - If the pixel has exactly one label among its connected neighbor with the same value that has already been labeled, give it that label.
   - If the pixel has two or more connected neighbors with the same value but different labels, choose one of the labels and remember that these labels are equivalent.
2. Resolve the equivalencies by making another pass through the image and labeling each pixel with a unique label for its equivalence class.

Algorithm 2.1: One algorithm for connected component labeling

A variation of this algorithm does not keep track of equivalence classes during the labeling process but instead makes multiple passes through the labeled image resolving the labels. It does so by updating each label that has a neighbor with a lower-valued label. Since this process may require multiple passes through the label image to resolve the equivalence classes, these passes usually alternate top-to-bottom, left-to-right and bottom-to-top, right-to-left to speed label propagation.

You will implement this algorithm (or a similar one of your choosing) as part of your second programming assignment.

2.3 Distances Between Pixels

It is often useful to describe the distance between two pixels (x_1, y_1) and (x_2, y_2).

- One obvious measure is the Euclidean (as the crow flies) distance
$$\left[(x_1 - x_2)^2 + (y_1 - y_2)^2\right]^{1/2}$$
.
- Another measure is the 4-connected distance D_4 (sometimes called *city-block distance*
$$|x_1 - x_2| + |y_1 - y_2|$$
.
- A third measure is the 8-connected distance D_8 (sometimes called *chessboard distance*
$$\max\left(|x_1 - x_2|, |y_1 - y_2|\right)$$
.

For those familiar with vector norms, these correspond to the L_2, L_1, and L_∞ norms.

2.3.1 Distance Maps

It is often useful to construct a *distance map* (sometimes called a *chamfer*) for a region of pixels. The idea is to label each point with the minimum distance from the pixel to the boundary of the region. Calculating this precisely for Euclidean distance can be computationally intensitive, but doing so for the city-block, chess-board, or similar measures can be done iteratively. (See Algorithm 2.1 on page 28 of your text.)

2.4 Other Topological Properties

2.4.1 Convex Hull

The *convex hull* of a region is the minimal convex region that entirely encompasses it.

2.4.2 Holes, Lakes, and Bays

One simple way of describing a shape is to consider the connected regions inside the convex hull but *not* in the shape of interest.

2.5 Edges and Boundaries

An *edge* is a pixel that has geometric properties indicative of a strong transition from one region to another. There are many ways to find edges, as we'll talk about later, but for now simply think of it as a strong transition.

Sometimes, we want to consider edges not as *at* pixels but as *separating* pixels. These are called *crack* edges.

Another concept is of a border. Once a region is identified, its border is all pixels in the region that are adjacent to pixels outside the region. One would hope that the set of boundary pixels and edge pixels are the same, but this is rarely so simple.

New Vocabulary

- 4-connected neighbor
- 8-connected neighbor
- mixed-connected neighbor
- Connected Component Labeling
- Distance metrics (Euclidean, city block, chessboard)
- Distance map
- Convex hull
- Edge
- Crack edge
- Border

Lecture 3: Data Structures for Image Analysis

Last modified on Monday, January 10, 2000 at 9:30 PM.

Reading

SH&B, Chapter 3

3.1 Introduction

If you're going to analyze the content of an image, you first have to develop ways of representing image information.

3.2 Image Maps

The simplest way to store information about an image is on a per-pixel basis. The idea is to build a one-to-one mapping between information and pixels. Of course, the easiest way to do this is with other images of the same size. So, while you may have an original image that you're trying to analyze, you may also have any number of other images that store information about the corresponding pixel in the original image. Such information may include

- region labels (which object does this picture belong to)
- local geometric information (derivatives, etc.)
- distances (the distance maps we saw in the last lecture)
- etc.

3.3 Chains

One step up from image maps is to store information on a per-region basis. The most basic thing to store about a region is where it is. One could store the location of each pixel in the region, but one can be more efficient by simply storing the border. By traversing the border in a pre-defined direction around the region, one can build a *chain*.

Instead of encoding the actual pixel locations, we really need to only encode their relative relationships: a *chain code*. For 4-connected borders, we can do this with only two bits per pixel. For 8-connected borders, we need three bits per pixel. An example of a chain code is shown in Figure 3.1 of your text.

Later on, we'll talk about other ways of representing chains and using these encodings to extract shape object information.

3.4 Run-Length Encoding

In CS 450, you probably talked about run-length encoding as a way of compressing image content. We can also use run-length encoding to store regions (or images, or image maps). An example run-length encoding is given in Figure 3.2 of your text.

3.5 Hierarchical Image Structures

Another way of storing image information is hierarchically. There are many ways to do this, but we'll talk about two methods here: pyramids and trees.

3.5.1 Pyramids

An image pyramid is a hierarchy of successively lower-resolution images based on the original image. Each stage in building the pyramid involves the following two steps:

1. Blur the image by some resolution-reducing kernel (low-pass filtering).

2. Because the image has now been low-pass filtered, one can reduce the sampling rate accordingly.

Notice that there are no constraints on the type of low-pass filtering done or the reduction in sampling rate other than the contraint imposed by the sampling theorem.

The most common way of building a pyramid is to do a 2-to-1 reduction in the sampling rate. This makes things convenient for storing the image, mapping lower-resolution pixels to higher-resolution pixels, etc. The simplest way to blur the image so as to reduce the resolution by a factor of 2 is with a 2×2 uniform kernel. Done this way, each pixel in the pyramid is simply the average of the corresponding 2×2 region in the next lower-resolution image.

The problem with this type of reduction is that the "footprints" of the lower-resolution pixels don't overlap in the higher-resolution image. This means that this upper levels of the pyramid can be extremely sensitive to single-pixel shifts in the original image. Ideally, we want to build pyramidal representations that are invariant to such translation.

It's usually better then to build pyramids by using footprints that overlap spatially in the original image. One can use small triangles of finite extent, Gaussians, etc.

Pyramids can be used to greatly speed up analysis algorithms. The provide a "divide and conquer" approach to vision algorithms. We'll see various examples of such pyramidal algorithms as the semester progresses.

3.5.2 Trees

Another way of representing images it to store large, coherent regions using a single piece of data. One example of this is the *quadtree*. A quadtree is built by breaking the image into four equal-size pieces. If any one of these pieces is homogeneous (in whatever property you're using the tree to represent), don't subdivide it any further. If the region isn't homogeneous, split it into four equal-size subregions and repeat the process. The resulting representation is a tree of degree 4 where the root of the tree is the entire image, the four child nodes are the four initial subregions, their children (if any) are their subregions, etc. The leaves of the tree correspond to homogeneous regions. An example of a quadtree is shown in Figure 3.6 of your text.

Quadtrees have been used for compression, rapid searching, or other applications where it is useful to stop processing homogeneous regions.

Notice that although built differently (bottom-up vs. top-down), reduce-by-two pyramids and quadtrees are almost the same. One just uses images and the other uses trees. Pyramids have the advantage that it is easy to do spatial operations at any level of the hierarchy; trees have the advantage that they don't store redundant information for homogeneous regions.

3.6 Relation Graphs

Once you've segmented an image into regions that (you hope) correspond to objects, you may want to know the spatial relationships between the regions. By recording which regions are next to which other regions, you can build a graph that describes these relationships. Such a graph is called a *region adjacency graph*. An example such a graph is in Figure 3.3 of your text. Notice that nodes of degree 1 are inside the region that they're adjacent to. (It's common to use a node to represent everything outside the image.)

3.7 Co-occurrence Matrices

Suppose that you want to record how often certain transitions occur as you go from one pixel to another. Define a spatial relationship r such as "to the left of", "above", etc. The co-occurrence matrix C_r for this relationship r counts the number of times that a pixel with value i occurs with relationship r with a pixel with value j. Co-occurrence matrices are mainly used to describe region texture (and we'll come back to them then), but they can also be used on image maps to measure how often pixels with certain labels occur with certain relationships to other labels.

Vocabulary

- image map
- chain
- run length encoding
- pyramid
- quadtree
- region adjacency graphs

Lecture 4: Thresholding

Reading

SH&B, Section 5.1

4.1 Introduction

Segmentation involves separating an image into regions (or their contours) corresponding to objects. We usually try to segment regions by identifying common properties. Or, similarly, we identify contours by identifying *differences* between regions (edges).

The simplest property that pixels in a region can share is intensity. So, a natural way to segment such regions is through *thresholding*, the separation of light and dark regions.

Thresholding creates binary images from grey-level ones by turning all pixels below some threshold to zero and all pixels about that threshold to one. (What you want to do with pixels at the threshold doesn't matter, as long as you're consistent.)

If $g(x, y)$ is a thresholded version of $f(x, y)$ at some global threshold T,

$$g(x,y) = \begin{cases} 1 & \text{if } f(x,y) \geq T \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

4.2 Problems with Thresholding

The major problem with thresholding is that we consider only the intensity, not any relationships between the pixels. There is no guarantee that the pixels identified by the thresholding process are contiguous.

We can easily include extraneous pixels that aren't part of the desired region, and we can just as easily miss isolated pixels within the region (especially near the boundaries of the region). These effects get worse as the noise gets worse, simply because it's more likely that a pixels intensity doesn't represent the normal intensity in the region.

When we use thresholding, we typically have to play with it, sometimes losing too much of the region and sometimes getting too many extraneous background pixels. (Shadows of objects in the image are also a real pain—not just where they fall across another object but where they mistakenly get included as part of a dark object on a light background.)

4.3 Local Thresholding

Another problem with global thresholding is that changes in illumination across the scene may cause some parts to be brighter (in the light) and some parts darker (in shadow) in ways that have nothing to do with the objects in the image.

We can deal, at least in part, with such uneven illumination by determining thresholds locally. That is, instead of having a single global threshold, we allow the threshold itself to smoothly vary across the image.

4.4 Automated Methods for Finding Thresholds

To set a global threshold or to adapt a local threshold to an area, we usually look at the histogram to see if we can find two or more distinct modes—one for the foreground and one for the background.

Recall that a histogram is a probability distribution:

$$p(g) = n_g/n \tag{4.2}$$

That is, the number of pixels n_g having greyscale intensity g as a fraction of the total number of pixels n.

Here are five different ways to look at the problem:

4.4.1 Known Distribution

If you know that the object you're looking for is brighter than the background and occupies a certain fraction $1/p$ of the image, you can set the threshold by simply finding the intensity level such that the desired percentage of the image pixels are below this value. This is easily extracted from the cumulative histogram:

$$c(g) = \sum_{0}^{g} p(g) \tag{4.3}$$

Simply set the threshold T such that $c(T) = 1/p$. (Or, if you're looking for a dark object on a light background, $c(T) = 1 - 1/p$.)

4.4.2 Finding Peaks and Valleys

One extremely simple way to find a suitable threshold is to find each of the modes (local maxima) and then find the valley (minimum) between them.

While this method appears simple, there are two main problems with it:

1. The histogram may be noisy, thus causing many local minima and maxima. To get around this, the histogram is usually smoothed before trying to find separate modes.
2. The sum of two separate distributions, each with their own mode, may not produce a distribution with two distinct modes. (See the plots on the right side of Figure 5.4 of your text.)

4.4.3 Clustering (K-Means Variation)

Another way to look at the problem is that we have two groups of pixels, one with one range of values and one with another. What makes thresholding difficult is that these ranges usually overlap. What we want to do is to minimize the error of classifying a background pixel as a foreground one or vice versa. To do this, we try to minimize the area under the histogram for one region that lies on the other region's side of the threshold. The problem is that we don't have the histograms for each region, only the histogram for the combined regions. (If we had the regions, why would we need to do segmentation?)

Understand that the place of minimum overlap (the place where the misclassified areas of the distributions are equal) is *not* is not necessarily where the valley occurs in the combined histogram. This occurs, for example, when one cluster has a wide distribution and the other a narrow one. (See the plots in Figure 5.4 of your text.)

One way that we can try to do this is to consider the values in the two regions as two *clusters*. Section 5.1.2 of your book describes a method for finding a threshold by clustering the histogram. The idea is to pick a threshold such that each pixel on each side of the threshold is closer in intensity to the mean of all pixels on that side of the threshold than the mean of all pixels on the other side of the threshold.

In other words, let $\mu_B(T)$ be the mean of all pixels less than the threshold and $\mu_O(T)$ be the mean of all pixels greater than the threshold. We want to find a threshold such that the following holds:

$$\forall g \geq T : |g - \mu_B(T)| > |g - \mu_O(T)|$$

and

$$\forall g < T : |g - \mu_B(T)| < |g - \mu_O(T)|$$

Algorithm 5.2 in your text shows an algorithm for finding such clustering. This is a variation of the *k-means clustering algorithm* used in pattern recognition and discussed in CS 521. The basic idea is to start by estimating $\mu_B(T)$ as the average of the four corner pixels (assumed to be background) and $\mu_O(T)$ as the average of everything else. Set the threshold to be halfway between $\mu_B(T)$ and $\mu_O(T)$ (thus separating the pixels according to how close their intensities are to $\mu_B(T)$ and $\mu_O(T)$ respectively). Now, update the estimates of $\mu_B(T)$ and $\mu_O(T)$ respectively by actually calculating the means of the pixels on each side of the threshold. This process repeats until the algorithm converges.

This method works well if the spreads of the distributions are approximately equal, but it does not handle well the case where the distributions have differing variances.

4.4.4 Clustering (The Otsu Method)

Another way of accomplishing similar results is to set the threshold so as to try to make each cluster as tight as possible, thus (hopefully!) minimizing their overlap. Obviously, we can't change the distributions, but we can adjust where we separate them (the threshold). As we adjust the threshold one way, we increase the spread of one and decrease the spread of the other. The goal then is to select the threshold that minimizes the combined spread.

We can define the *within-class* variance as the weighted sum of the variances of each cluster:

$$\sigma^2_{\text{Within}}(T) = n_B(T)\sigma^2_B(T) + n_O(T)\sigma^2_O(T) \tag{4.4}$$

where

$$\begin{aligned}
n_B(T) &= \sum_{i=0}^{T-1} p(i) && (4.5)\\
n_O(T) &= \sum_{i=T}^{N-1} p(i) && (4.6)\\
\sigma^2_B(T) &= \text{the variance of the pixels in the background (below threshold)} && (4.7)\\
\sigma^2_O(T) &= \text{the variance of the pixels in the foreground (above threshold)} && (4.8)\\
& && (4.9)
\end{aligned}$$

and $[0, N-1]$ is the range of intensity levels.

Computing this within-class variance for each of the two classes for each possible threshold involves a lot of computation, but there's an easier way.

If you subtract the within-class variance from the total variance of the combined distribution, you get something called the between-class variance:

$$\begin{aligned}
\sigma^2_{\text{Between}}(T) &= \sigma^2 - \sigma^2_{\text{Within}}(T)\\
&= n_B(T)\left[\mu_B(T) - \mu\right]^2 + n_O(T)\left[\mu_O(T) - \mu\right]^2
\end{aligned} \tag{4.10}$$

where σ^2 is the combined variance and μ is the combined mean. Notice that the between-class variance is simply the weighted variance of the cluster means themselves around the overall mean. Substituting $\mu = n_B(T)\mu_B(T) + n_O(T)\mu_O(T)$ and simplifying, we get

$$\sigma^2_{\text{Between}}(T) = n_B(T)n_O(T)[\mu_B(T) - \mu_O(T)]^2 \tag{4.11}$$

So, for each potential threshold T we

1. Separate the pixels into two clusters according to the threshold.
2. Find the mean of each cluster.
3. Square the difference between the means.
4. Multiply by the number of pixels in one cluster times the number in the other.

This depends only on the difference between the means of the two clusters, thus avoiding having to calculate differences between individual intensities and the cluster means. The optimal threshold is the one that maximizes the between-class variance (or, conversely, minimizes the within-class variance).

This still sounds like a lot of work, since we have to do this for each possible threshold, but it turns out that the computations aren't independent as we change from one threshold to another. We can update $n_B(T)$, $n_O(T)$, and the respective cluster means $\mu_B(T)$ and $\mu_O(T)$ as pixels move from one cluster to the other as T increases. Using simple recurrence relations we can update the between-class variance as we successively test each threshold:

$$\begin{aligned} n_B(T+1) &= n_B(T) + n_T \\ n_O(T+1) &= n_O(T) - n_T \\ \mu_B(T+1) &= \frac{\mu_B(T)n_B(T) + n_T T}{n_B(T+1)} \\ \mu_O(T+1) &= \frac{\mu_O(T)n_O(T) - n_T T}{n_O(T+1)} \end{aligned}$$

This method is sometimes called the *Otsu* method, after its first publisher.

4.4.5 Mixture Modeling

Another way to minimize the classification error in the threshold is to suppose that each group is Gaussian-distributed. Each of the distributions has a mean (μ_B and μ_O respectively) and a standard deviation (σ_B and σ_O respectively) *independent of the threshold we choose*:

$$h_{\text{model}}(g) = n_B \; e^{-(g-\mu_B)^2/2\sigma_B^2} + n_O \; e^{-(g-\mu_O)^2/2\sigma_O^2}$$

Whereas the Otsu method separated the two clusters according to the threshold and tried to optimize some statistical measure, *mixture modeling* assumes that there already exists two distributions and we must find them. Once we know the parameters of the distributions, it's easy to determine the best threshold.

Unfortunately, we have six unknown parameters (n_B, n_O, μ_B, μ_O, σ_B, and σ_O), so we need to make some estimates of these quantities.

If the two distributions are reasonably well separated (some overlap but not too much), we can choose an arbitrary threshold T and assume that the mean and standard deviation of each group approximates the mean and standard deviation of the two underlying populations. We can then measure how well a mix of the two distributions approximates the overall distribution:

$$F = \sum_{0}^{N-1} \left[h_{\text{model}}(g) - h_{\text{image}}(g) \right]^2$$

Choosing the optimal threshold thus becomes a matter of finding the one that causes the mixture of the two estimated Gaussian distributions to best approximate the actual histogram (minimizes F). Unfortunately, the solution space is too large to search exhaustively, so most methods use some form of gradient descent method. Such gradient-descent methods depend heavily on the accuracy of the initial estimate, but the Otsu method or similar clustering methods can usually provide reasonable initial estimates.

Mixture modeling also extends to models with more than two underlying distributions (more than two types of regions). For example, segmenting CT images into grey matter, white matter, and cerebral spinal fluid (CSF).

4.5 Multispectral Thresholding

Section 5.1.3 of your text describes a technique for segmenting images with multiple components (color images, Landsat images, or MRI images with T1, T2, and proton-density bands). It works by estimating the optimal threshold in one channel and then segmenting the overall image based on that threshold. We then subdivide each of these regions independently using properties of the second channel. We repeat it again for the third channel, and so on, running through all channels repeatedly until each region in the image exhibits a distribution indicative of a coherent region (a single mode).

4.6 Thresholding Along Boundaries

If we want our thresholding method to give stay fairly true to the boundaries of the object, we can first apply some boundary-finding method (such as edge detection techniques that we'll cover later) and then sample the pixels only where the boundary probability is high.

Thus, our threshold method based on pixels near boundaries will cause separations of the pixels in ways that tend to preserve the boundaries. Other scattered distributions within the object or the background are of no relevance.

However, if the characteristics change along the boundary, we're still in trouble. And, of course, there's still no guarantee that we'll not have extraneous pixels or holes.

Vocabulary

- Thresholding
- Local Thresholding
- Clustering
- Otsu Method
- Mixture Modeling

Lecture 5: Binary Morphology

Contents

Reading

SH&B, 11.1–11.3
Castleman, 18.7.1–18.7.4

5.1 What is Mathematical Morphology?

"Morphology" can literally be taken to mean "doing things to shapes". "Mathematical morphology" then, by extension, means using mathematical principals to do things to shapes.

5.2 Image Regions as Sets

The basis of mathematical morphology is the description of image regions as sets. For a binary image, we can consider the "on" (1) pixels to all comprise a set of values from the "universe" of pixels in the image. Throughout our discussion of mathematical morphology (or just "morphology"), when we refer to an image A, we mean the set of "on" (1) pixels in that image.

The "off" (0) pixels are thus the set compliment of the set of on pixels. By A^c, we mean the compliment of A, or the off (0) pixels. We can now use standard set notation to describe image operations:

A	the image
A^c	the compliment of the image (inverse)
$A \cup B$	the union of images A and B
$A \cap B$	the intersection of images A and B
$A - B = A \cap B^c$	the difference between A and B (the pixels in A that aren't in B)
$\#A$	the cardinality of A (area of the object(s))

We'll also need to introduce the notion of a translated image set. The image A translated by movement vector t is

$$A_t = \{c \mid c = a + t \text{ for some } a \in A\} \tag{5.1}$$

Now, we have the necessary notation to introduce most of the rest of mathematical morphology. The first two basic operations are dilation and erosion.

5.3 Basic Binary Operations

5.3.1 Dilation

Dilation (sometimes called "Minkowsky addition") is defined as follows.

$$A \oplus B = \{c \mid c = a + b \text{ for some } a \in A \text{ and } b \in B\} \tag{5.2}$$

One way to think of this is to take copies of A and translate them by movement vectors defined by each of the pixels in B. (This means we have to define an origin for B.) If we union these copies together, we get $A \oplus B$. This can be written as

$$A \oplus B = \bigcup_{t \in B} A_t \tag{5.3}$$

Alternatively, we can take copies of B and translate them by movement vectors defined by each of the pixels in A—dilation is commutative. An interpretation of this latter way of thinking about it is to put a copy of B at each pixel in A. If we "stamp" a copy of B at each pixel in A and union all of the copies, we get $A \oplus B$. This can be written this way:

$$A \oplus B = \bigcup_{t \in A} B_t \tag{5.4}$$

In this way, dilation works much like convolution: slide a kernel to each position in the image and at each position "apply" (union) the kernel. In the case of morphology, we call this kernel B the *structuring element*.

Dilation is also associative as well as commutative:

$$(A \oplus B) \oplus C = A \oplus (B \oplus C)) \tag{5.5}$$

This follows naturally from the way we are unioning multiple translated sets together.

This associativity is useful for decomposing a single large dilation into multiple smaller ones. Suppose, for example, that you have available a board that does 3×3 dilation in hardware, but you want to perform dilation by a 7×9 structuring element. Does there exist a set of 3×3 element that when dilated by each other in succession gives the 7×9 one? If so, you can write the larger dilation as the result of successively dilating by the smaller elements.

5.3.2 Erosion

Erosion (sometimes called "Minkowsky subtraction") is defined as follows.

$$A \ominus B = \{x \mid x + b \in A \text{ for every } b \in B\} \tag{5.6}$$

One way to think of this is to take copies of A and again translate them by movement vectors defined by each of the pixels in B. However, this time we move them in the opposite direction ($-b$) and *intersect* the copies together. This can be written as

$$A \ominus B = \bigcap_{t \in B} A_{-t} \tag{5.7}$$

An interpretation of this latter way of thinking about it is to again put a copy of B at each pixel in A. If count only those copies whose translated structuring elements lie entirely in A and mark the pixels in A that these copies were translated to, we get $A \ominus B$.

Unlike dilation, erosion is not commutative. (Much like how addition is commutative while subtraction is not.)

Also unlike dilation, erosion is not associative:

$$(A \ominus B) \ominus C \stackrel{?}{=} A \ominus (B \ominus C) \tag{5.8}$$

5.3.3 Duality of Dilation and Erosion

When one operation is the *dual* of the other, it means that one can be written in terms of the other.

This does *not*, however, mean that they are opposites. Unlike arithmetic addition and subtraction,

$$(A \oplus B) \ominus B \overset{?}{=} A \tag{5.9}$$

Dilation and erosion are related as follows. If $\breve{B}$ denotes the reflection of B,

$$\begin{array}{rcl} (A \oplus B)^c & = & A^c \ominus \breve{B} \\ (A \ominus B)^c & = & A^c \oplus \breve{B} \end{array} \tag{5.10}$$

In other words, dilating the "foreground" is the same as eroding the "background", but the structuring element reflects between the two. Likewise, eroding the foreground is the same as dilating the background.

So, strictly speaking we don't really need *both* dilate and erode: with one or the other, and with set complement and reflection of the structuring element, we can achieve the same functionality. Hence, dilation and erosion are duals.

This duality can also be used to derive an equivalent to associativity for erosion:

$$\begin{array}{rcl} (A \ominus B) \ominus C & = & ((A \ominus B)^c \oplus \breve{C})^c \\ & = & ((A^c \oplus \breve{B}) \oplus \breve{C})^c \\ & = & (A^c \oplus (\breve{B} \oplus \breve{C}))^c \\ & = & (A \ominus \breve{(\breve{B} \oplus \breve{C})} \\ & = & (A \ominus (B \oplus C) \end{array} \tag{5.11}$$

Thus, we can combine the effects of eroding by first B then C into a single erosion by the *dilation* of B by C. Again making the analogy to normal arithmetic, this really isn't all that different from how $(A - B) - C = A - (B + C)$.

5.4 Some Examples of Using Dilation and Erosion

Suppose that you wanted to find all pixels lying on the boundary of an object. We could perform the following operations:

$$\text{Bound}_{\text{ext}}(A) = (A \oplus B) - A \tag{5.12}$$

where B was a 3×3 structuring element containing all 1s. This would give us all background pixels that bordered the object.

Or, if we wanted all foreground pixels that bordered the background, we could use

$$\text{Bound}_{\text{int}}(A) = A - (A \ominus B) \tag{5.13}$$

You could also extend this to give you minimum (8-connected) distance from an object:

$$\text{Dist}_i = (A \oplus_i B) - (A \oplus_{i-1} B) \tag{5.14}$$

where $\oplus_i$ denotes i applications of the dilation operator.

Do you see how you can create 4-connected equivalents of these?

5.5 Proving Properties of Mathematical Morphology

To construct proofs using mathematical morphology, you employ algebraic proofs using the properties shown here and other properties from set algebra. The example in Eq. 5.11 is one such type of proof. In some of your homework problems, you will use similar techniques to prove other useful properties.

5.6 Hit-and-Miss

When eroding, the “0”s in the structuring element act like “don’t care” conditions—they don’t really require that the image be on or off at that point, only that the remaining “1” pixels fit inside the object. In other words, it finds places that “look like this” (for the 1s), but has no way to say “but doesn’t look like this” (for the 0s).

We can can combine erosion and dilation to produce an operator that acts like this: the “hit and miss” operator. The operator takes two elements: one that must “hit” and one that must “miss”.

The operator is defined as follows. If the structuring elements J (hit) and K (miss) are applied to the image A:

$$A \otimes (J, K) = (A \ominus J) \cap (A^c \ominus K) \tag{5.15}$$

In other words, the structuring element J must fit inside the object and the element K must fit *outside* the object at that position.

This gives us a form of binary template matching. For example, the following structuring elements give an “upper right” corner detector:

0	0	0
1	1	0
0	1	0

J

0	1	1
0	0	1
0	0	0

K

The J element finds the points with connected left and lower neighbors, and the H element finds the points *without* upper, upper right, and right neighbors.

In some cases, we’ll simply use a single structuring element B, with the assumption that $J = B$ and $K = B^c$. I.e.,

$$A \otimes B = A \otimes (B, B^c) = (A \ominus B) \cap (A^c \ominus B^c)$$

Notice that this doesn’t, however, allow for a third case: “don’t care” pixels—ones that could be either inside or outside the shape. Some authors will for this reason write the two operators as a single one using 1s for the hits, 0s for the misses, and “x”s for the don’t-care positions. In the previous example, this would be written as

x	0	0
1	1	0
x	1	x

This form is perhaps more useful for visualizing what the structuring element is designed to find. Remember, though, that you still have to apply two different operators, one for hit (the 1s) and one for miss (the 0s), when implementing hit-and-miss.

5.7 Opening and Closing

5.7.1 Opening

An *opening* is an erosion followed by a dilation with the same structuring element:

$$A \circ B = (A \ominus B) \oplus B \tag{5.16}$$

Remember that erosion finds all the places where the structuring element fits inside the image, but it only marks these positions at the origin of the element. By following an erosion by a dilation, we “fill back in” the full structuring element at places where the element fits inside the object. So, an opening can be consider to be the union of all translated copies of the structuring element that can fit inside the object.

Openings can be used to remove small objects, protrusions from objects, and connections between objects. So, for example, opening by a 5×5 square element will remove *all* things that are less than 5 pixels high or less than 5 pixels tall.

5.7.2 Closing

Closing works in an opposite fashion from opening:

$$A \bullet B = (A \oplus B) \ominus B \tag{5.17}$$

Whereas opening removes all pixels where the structuring element won't fit inside the image foreground, closing *fills in* all places where the structuring element will not fit in the image *backgound*.

Remember, though, duality doesn't imply inverse operations: an opening following a closing will *not* restore the original image.

5.7.3 Properties of Opening and Closing

Duality

Closing is the dual of opening:

$$(A \circ B)^c = A^c \bullet \breve{B} \tag{5.18}$$

Just as erosion can be implemented using dilation (and vice versa), opening can be implemented using closing (and vice versa)

Idempotence

An important property of opening and closing operations is that they are *idempotent*: if you apply one more than once, nothing changes after the first application. So,

$$A \circ B \circ B = A \circ B \tag{5.19}$$

and

$$A \bullet B \bullet B = A \bullet B \tag{5.20}$$

5.7.4 Applications of Opening and Closing

Opening and closing are the basic workhorses of morphological noise removal. Opening removes small objects, and closing removes small holes.

Finding Things

You can also use opening and closing to find specific shapes in an image. While these may not be "noise", they may just be other things that you don't care about. (In this sense, perhaps noise is just things you don't care about?) Opening can be used to only find things into which a specific structuring element can fit.

Subdivision into Parts

You can also extend this idea to part subdivision. Suppose that you know that a particular object has specific parts with specific shapes. Openings can be used to separate those parts by shaping the structuring elements to "fit" into those parts. They don't have to exactly match the parts—it's sufficient to only fit into some parts while not fitting into others.

Vocabulary

- Mathematical Morphology
- Dilation
- Erosion

- Dual
- Hit-and-Miss
- Opening
- Closing
- Idempotence

Lecture 6: Morphology: thickening, thinning, hulls, skeletons, connectivity, distance maps conditional dilation

Contents

Reading

SH&B, 11.5.1–11.5.3

We'll cover the remainder of Chapter 11 (from 11.5.3 on) later in the course.

6.1 Thickening and Thinning

We can use hit-and-miss operators to find specific points and either add them or remove them from the object. If we find such pixels and add them to the object, that constitutes a *thickening*:

$$A \odot (J, K) = A \cup (A \otimes (J, K)) \tag{6.1}$$

Similarly, if we find such pixels and remove them from the object, that constitutes a *thinning*:

$$A \oslash (J, K) = A - (A \otimes (J, K)) \tag{6.2}$$

6.2 Recursive Morphology

Recursive morphological operators are those that are defined in terms of themselves and are applied recursively or iteratively to the image. Examples of recursive morphological processes are distance maps, convex hull, and thinned skeletons.

6.2.1 Distance Maps

We discussed distance maps earlier in Lecture 2. Distance maps can be calculated using morphological operations. The set of all pixels with distance i from the object can be calculated as follows:

$$\text{Dist}_i = (A \oplus_i B) - (A \oplus_{i-1} B)$$

where $\oplus_i$ denotes i applications of the dilation operator.

6.2.2 Convex Hull

The convex hull of an object is the smallest convex shape that entirely contains that object. We can calculate the convex hull of an image object by repeatedly finding and thickening convexities.

6.2.3 Skeletons

A skeleton of an object is a single-pixel thin set of pixels produced by repeatedly thinning the object. We can't do this using erosion alone, because we have to be able to detect when deletion of a pixel would make the object disconnected. So, we have to use hit-and-miss operators (thinning.

6.3 Conditional Dilation

Conditional dilation essentially involves dilation of an image followed by intersection with some other "condition" image. In other words, new pixels are added to the original

1. the pixel is in the dilation of the original, and
2. the pixel is in the condition image.

In this way, the condition image acts like a mask on the dilation. This is written as

$$A \oplus |_I B = (A \oplus B) \cap I \tag{6.3}$$

Here is an example of how we might use conditional dilation. Suppose that after applying certain morphological operations to find specific points in an image we wanted to find the original connected components within which those points were found. In essence, we want to "grow" those points out to fill the original connected component. One obvious way would be to combine connected-component labeling with morphological operations. However, you can do it with *conditional dilation*.

Let I be the original image and A the morphologically-reduced image that you want to "grow back". Let B be a 3×3 structuring element containing all 1s. Let $J_0 = A$ and $J_i = (J_{i-1} \oplus B) \cap I$. So, at each iteration I, J_i "grows" out one pixel *but only along the original image points*. Eventually, this will converge to $J_i = J_{i-1}$, and the algorithm stops. This is illustrated in Haralick, Figure 5.26.

Vocabulary

- Thickening
- Thinning

- Convex Hull
- Skeleton
- Conditional Dilation

Lecture 7: Shape Description (Contours)

Last modified on January 21, 2000 at 2:20 PM

Contents

Reading

SH&B, 6.1–6.2
Castleman 18.8.2; 19.2.1.2; 19.3.4-19.3.4.2.
These notes.

7.1 Introduction

Normally, shape description comes after we study segmentation. (First find the objects, then describe them.) However, we're going to study shape description first so that you can use it, along with the simple thresholding segmentation technique you've already learned, in your first programming assignment.

In this and the following lecture, we'll cover ways of representing and describing shapes by their *contours* or *borders*. In Lectures 9 and 10, we'll cover ways to represent and describe shapes by their *regions*.

First, we'll discuss how to encode the boundary (representation), and then we'll discuss how to describe each in ways that distill certain properties from this boundary (descriptors).

7.2 Region Identification

Section 6.1 of your text describes an algorithm that you may recognize as the connected-components algorithm we discussed in class in Lecture 2.

7.3 Invariance

One of the most useful properties of a descriptor is that it remain the same for a given object under a variety of transformations. This property is called *invariance*.

Specifically, invariance is defined as follows. A property P is invariant to transformation T iff measurement of the property P commutes with the transformation T:

$$P(T(I)) = T(P(I))$$

where I denotes our image. In other words, if we transform the image before measuring the property, we get the same result (perhaps similarly transformed) as if we measured the property on the image itself.

Simply put, invariance means that visual properties stay consistent under specific transformations: dogs don't turn into cats just because they're rotated, translated, darkened, lightened, resized, etc.

7.4 Encoding Boundary Curves Sequentially

In Lecture 3, we talked briefly about *chain codes* as a data structure. We begin by briefly reviewing this idea and then extend it to more robust variations.

7.4.1 Basic Chain Codes

Remember how in CS 450 we compressed a signal by only encoding the difference between successive values, not the full values themselves? Well, we can do the same thing with boundaries. Once we have identified the boundary pixels of a region, we can encode them by only storing the relative difference as we go from pixel to pixel clockwise around the contour.

If the contour is 4-connected, we only need four values (two bits) to encode the direction to the next pixel. If the contour is 8-connected, we need eight numbers (three bits).

By encoding relative, rather than absolute position of the contour, the representation is translation invariant. We can match boundaries by comparing their chain codes, but with the following problems:

- We have to have the same starting point. If the starting points are different, we have to rotate the code to different starting points.
- Small variations in the contour give different codes. Matching chain code for slightly noisy versions of the same object can be very difficult.
- The representation is not rotationally invariant.

7.4.2 Differential Chain Codes

We can deal (somewhat) with the rotational variance of chain codes by encoding not the relative direction, but *differences* in the successive directions: *differential chain codes*. This can be computed by subtracting each element of the chain code from the previous one and taking the result modulo n, where n is the connectivity.

This differencing allows us to rotate the object in 90-degree increments and still compare the objects, but it doesn't get around the inherent sensitivity of chain codes to rotation on the discrete pixel grid.

7.4.3 Shape Numbers

Remember how we discussed the effect changing the starting point has on chain codes and differential chain codes? Moving the starting point rotates the code. If we rotate an n-element code so that it has the smallest value when viewed as an n-digit integer, that would be invariant to the selection of the starting point.

Such a normalized differential chain code is called a *shape number*.

As with chain codes, shape numbers are very sensitive to rotation, to resizing, and to discrete pixel artifacts. We can try to normalize away these effects by resampling the grid along some principal axis of the shape. This resampling (perhaps at some angle and sampling rate different from the original grid) can account for rotation, resizing, and minor pixel artifacts.

7.4.4 Smoothing Chain Codes

Chain codes, their derivatives, and shape numbers are sensitive to the pixel grid. One way to deal with this is to first smooth the shape, then record the representation. While this loses information and does not permit reconstruction of the object, it helps remove "noise" in the representation when comparing two objects.

7.4.5 Resampling Chain Codes

To deal with this sensitivity to the pixel grid, we can also resample the boundary onto a coarser grid and then compute the chain codes of this coarser representation. This also smooths out small variations but can help compensate for differences in chain-code length due to the pixel grid. It can, however lose significant structure for small objects.

7.4.6 ψ-s Curves

Another method, similar to chain codes, is to store the tangent vector at every boundary point. In fact, we don't really care about the magnitude of the tangent, only the direction. So, we can encode the angle of the tangent ψ as a function of arc length s around the contour: $\psi(s)$ This representation is called a ψ-s curve.

Since we're encoding the angle of the tangent vector, we can choose some arbitrary reference frame (zero-degree tangent). If we let the tangent at the starting point have an angle of zero, the rest of the curve is described relative to it. Thus, our ψ-s curve always starts and ends at zero.

The ψ-s representation has the following advantages:

- Like chain codes, this representation is translationally invariant.
- Unlike chain codes, which are limited to 90- or 45-degree directions, we can compute tangent vectors to accuracy limited only by our segmentation method.
- It is (nearly) rotationally invariant.
- If we normalize by the length of the contour (i.e., the start is $s = 0$ and the end is $s = 1$ regardless of the length), it is scale invariant.

We are still limited when comparing curves with different starting points, but if we rotate the s axis of the ψ-s representation and shift the ψ axis so that the starting point is at zero, we can compare such curves.

So, an algorithm for contour matching based on ψ-s curves might proceed as follows:

1. Calculate the ψ-s curve for each contour and normalize by the length.

2. For each possible starting point on the second curve, compare the first curve to the second curve starting at that start point.

3. Select the largest (best) of these matches: the strength of this best match is the overall match between the curves, and the starting point that gives this best match tells you the relative rotation.

Notice that if we use summation of pointwise multiplication (i.e., inner product) as the method of comparison, step two is merely our old friend *correlation*.

7.4.7 R-S Curves

Another approach, similar to ψ-s curves is to store the distance r from each point s to the (arbitrarily chosen) origin of the object. By tracing r as a function of arclength s, one gets an r-s curve. Like ψ-s curves, r-s curves can be compared in a rotationally-invariant way using correlation.

7.5 Polygonal Approximation

Another method for compacting the boundary information is to fit line segments or other primitives to the boundary. We then need only to store the parameters of these primitives instead of discrete points. This is useful because it reduces the effects of discrete pixelization of the contour.

7.5.1 Merging Methods

Merging methods add successive pixels to a line segment if each new pixel that's added doesn't cause the segment to deviate from a straight line too much.

For example, choose one point as a starting point. For each new point that we add, let a line go from the start point to the new point. Then, compute the squared error or other such measure for every point along the segment/line. If the error exceeds some threshold, keep the line from the start to the previous point and start a new line. (See Figure 6.12 of your text for an example.)

If we have thick boundaries rather than single-pixel thick ones, we can still use a similar approach called *tunneling*. Imagine that we're trying to lay straight rods along a curved tunnel, and that we want to use as few as possible. We can start at one point and lay as long a straight rod as possible. Eventually, the curvature of the "tunnel" won't let us go any further, so we lay another rod and another until we reach the end.

7.5.2 Splitting Methods

Splitting methods work by first drawing a line from one point on the boundary to another. Then, we compute the perpendicular distance from each point along the segment to the line. If this exceeds some threshold, we break the line at the point of greatest error. We then repeat the process recursively for each of the two new lines until we don't need to break any more. See Figure 6.13 of your text for an example. This is sometimes known as the "fit and split" algorithm.

For a closed contour, we can find the two points that lie farthest apart and fit two lines between them, one for one side and one for the other. Then, we can apply the recursive splitting procedure to each side.

7.6 Other Geometric Representations

Many other pieces of geometric information can be used to classify shapes.

7.6.1 Boundary Length

The simplest form of descriptor is the length of the boundary itself. While this clearly doesn't allow you to reconstruct the shape, or even to say that much about it, it can be used as a quick test to eliminate candidates for matching.

7.6.2 Curvature

If one differentiates the ψ-s curve, one gets a function that describes the curvature at every point on the contour. Unlike the ψ-s curve, this representation is rotationally invariant and can be used for matching without requiring shifted comparison (correlation).

7.6.3 Bending Energy

If you integrate the squared curvature along the entire contour, you get a single descriptor known as *bending energy*. While not as descriptive as the curvature function itself, it is also far less verbose, thus making it a good feature for shape matching.

7.6.4 Signatures

In general, a *signature* is any 1-D function representing 2-D areas or boundaries. Chain codes, differential chain codes, ψ-s curves, and r-s curves are all signatures. Many other forms of signatures have been proposed in the computer vision literature.

Section 6.2.2 of your text describes one of these other forms of signature: the distance to the opposing side as a function of arclength s as one goes around the contour. See Figure 6.9 of your text for an example.

7.6.5 Chord Distribution

Section 6.2.2 also contains a description of a descriptor known as *chord distribution*. The basic idea is to calculate the lengths of all chords in the shape (all pair-wise distances between boundary points) and to build a histogram of their lengths and orientations. The "lengths" histogram is invariant to rotation and scales linearly with the size of the object. The "angles" histogram is invariant to object size and shifts relative to object rotation.

7.6.6 Convex Hulls

Yet another way of describing an object is to determine the *convex hull* of the object. As we discussed when covering Mathematical Morphology, a convex hull is a minimal convex shape entirely bounding an object. Imagine a rubber band snapped around the shape and you have the convex hull. We can look at where the convex hull touches the contour and use it to divide the boundary into segments. These points form extremal points of the object and can be used to get a rough idea of its shape.

7.6.7 Points of Extreme Curvature

Another useful method for breaking a boundary into segments is to identify the points of maximum positive or minimum negative curvature. These extremes are where the object has exterior or interior corners and make useful key points for analysis. Studies of human vision seem to indicate that such points also play a role in the way people subdivide objects into parts.

7.7 Fourier Descriptors

7.7.1 Definition

Suppose that the boundary of a particular shape has N pixels numbered from 0 to $N-1$. The k-th pixel along the contour has position (x_k, y_k). So, we can describe the contour as two parametric equations:

$$x(k) = x_k \tag{7.1}$$

$$y(k) = y_k \tag{7.2}$$

Let's suppose that we take the Fourier Transform of each function. We end up with two frequency spectra:

$$a_x(\nu) = \mathcal{F}(x(k)) \tag{7.3}$$
$$a_y(\nu) = \mathcal{F}(y(k)) \tag{7.4}$$

For a finite number of discrete contour pixels we simply use the Discrete Fourier Transform. Remember that the DFT treats the signal as periodic. That's no problem here—the contour itself is periodic, isn't it?

These two spectra are called *Fourier descriptors*.

You can take things one step further by considering the (x, y) coordinates of the point not as Cartesian coordinates but as those in the complex plane:

$$s(k) = x(k) + iy(k) \tag{7.5}$$

Thus, you get a single Fourier descriptor which is the transform of the complex function $s(k)$. In a sense, though, this is unnecessary:

$$\begin{aligned} a(\nu) &= \mathcal{F}(s(k)) && (7.6)\\ &= \mathcal{F}(x(k) + iy(k)) && (7.7)\\ &= \mathcal{F}(x(k)) + i\mathcal{F}(y(k)) && (7.8)\\ &= a_x(\nu) + ia_y(\nu) && (7.9)\\ &= [\Re(a_x(\nu)) - \Im(a_y(\nu))] + i[\Im(a_x(\nu)) + \Re(a_y(\nu))] && (7.10) \end{aligned}$$

Combining things into a single descriptor instead of two is useful, but it's really only folding one descriptor onto another using complex notation.

7.7.2 Properties

Let's consider what these spectra mean. First, suppose that the spectra have high-frequency content. That implies rapid change in the x or y coordinate at some point as you proceed around the contour, just like high frequencies normally mean some rapid change in the signal. What would such a contour look like? (Bumpy)

Now, suppose that the signal has little high-frequency content. This implies little change in the x or y coordinates as you proceed around the contour. What would this shape look like? (Smooth)

What happens if we low-pass filter a Fourier descriptor? Isn't this just the same as smoothing the contour? In this way, the low-frequency components of the Fourier descriptor capture the general shape properties of the object, and the high-frequency components capture the finer detail. Just as with filtering any other signal, though, it does so without regard to spatial position. There are times (as we'll see later) where this may be important.

Suppose that instead of using all k components of $a(\nu)$, we only used the first $m < k$ of them. What if we reconstruct the shape from such a truncated descriptor? Don't we just get successively refined approximations to the shape as we add more terms?

In this way, Fourier descriptors act much like moments: lower order terms/moments give approximate shape, and adding additional terms refines that shape.

7.7.3 Response to Transformations

We can use all of the properties of the Fourier transform to describe the properties of Fourier descriptors:

Translation. If we translate the object, we're really just adding some constant to all of the values of $x(k)$ and $y(k)$. So, we only change the zero-frequency component. In fact, what do you think the zero-frequency component tells us about the shape? (Mean position only—nothing about the shape.) So, except for the zero-frequency component, Fourier Descriptors are translation invariant.

Rotation. Remember from complex analysis that rotation in the complex plane by angle θ is multiplication by $e^{i\theta}$. So, rotation *about the origin of the coordinate system* only multiplies the Fourier descriptors by $e^{i\theta}$.

I emphasize that this is rotation about the origin of the coordinate system, not rotation about the center of the positioned shape. If we throw out the zero-frequency component (the position), the results of rotation are the same regardless of position.

Scaling Suppose that we resize the object. That's equivalent to simply multiplying $x(k)$ and $y(k)$ by some constant. As you are well-acquainted by now, that's just multiplication of the Fourier descriptor by the same constant. (Again, we ignore the value of the zero-frequency component.)

Start Point What if we change the starting point for the contour? Isn't this simply translation of the one-dimensional signal $s(k)$ along the k dimension? Remember from our discussion of the Fourier Transform that translation in the spatial domain (in this case, k) is a phase-shift in the transform. So, the magnitude part of $a(\nu)$ is invariant to the start point, and the phase part shifts accordingly.

7.7.4 Higher-Dimensional Descriptors

Fourier descriptors have also been used to describe higher-dimensional shapes. What happens is that the domain of the function increases from a one-dimensional k for two-dimensional images to an $N-1$-dimensional one for N-dimensional images.

For three-dimensional images, spherical harmonics have been used as higher-dimensional basis functions.

7.8 Shape Invariants

Our discussion so far has focused on comparing 2-dimensional shapes. One area of significant research involves comparing 2-dimensional *projections* of 3-dimensional shapes. In other words, the question isn't "do these 2-dimensional shapes match?" but "could these 2-dimensional shapes be different projections of the same 3-dimensional shape?" The key to this research is the notion of *projective invariants*—measurable properties that don't change when projected.

Nearly all forms of invariants, whether projective or not, involve *ratios*. By constructing ratios, even if one quantity changes under a tranformation, as long as another quantity changes proportionally under the same transformation, their ratio stays the same.

7.8.1 Cross Ratio

One example is the *cross-ratio*. This invariant relies on the property that a line in 3-space projects to a line on a plane, so long as the entire line doesn't project to a single point. (Geometers refer to this as a *generic* property: one that holds even under slight perturbations. A line projecting to a point is not generic—slight perturbations make it go away.) Four co-linear points in 3-space project to four co-linear points on the projected image. The cross-ratio of the distances between these points a, b, c, and d forms an invariant:

$$\frac{(a-c)(b-d)}{(a-d)(b-c)}$$

See Figure 6.19 of your text for a graphical depiction of this.

7.8.2 Systems of Lines

A set of four coplanar lines meeting in a point also forms an invariant, described in Eq. 6.26 in your text. Can you think of objects you may want to recognize that involve four lines meeting in a point?

7.8.3 Conics

Invariants can also be constructed for conic sections (circles, ellipses, parabolas, etc.) By combining straight lines and conics, may man-made objects can be modeled. These objects can be recognized regardless of viewpoint (so long as the curves can be seen or partially seen). Figure 6.21 of your text shows a good example.

7.9 Conclusion

Notice that as we progress from one representation to another, we’re basically trying to do three things:

1. distill more and more shape information from the representation

Lecture 9: Shape Description (Regions)

Contents

Reading

SH&B, 6.3.1-6.3.3
Castleman 19.1-19.2.1.1; 19.3.1-19.3.3.3

9.1 What Are Descriptors?

In general, descriptors are some set of numbers that are produced to describe a given shape. The shape may not be entirely reconstructable from the descriptors, but the descriptors for different shapes should be different enough that the shapes can be discriminated.

We saw some simple examples of descriptors in our discussion of contour representations. These included the contour length (perimeter) and the bending energy.

What qualifies as a good descriptor? In general, the better the descriptor is, the greater the difference in the descriptors of significantly different shapes and the lesser the difference for similar shapes. What then qualifies similarity of shape? Well, nobody's really been able to answer that one yet. If we could quantify similarity of shape, we'd have the perfect descriptor. Indeed, that's what descriptors are: attempts to quantify shape in ways that agree with human intuition (or task-specific requirements).

Regions can either describe boundary-based properties of an object or they can describe region-based properties. In this lecture, we focus on region-based descriptors.

9.2 Some Simple Shape Descriptors

A few simple descriptors are

Area. The number of pixels in the shape. Your text describes algorithms for calculating the area from quadtree or chain-coding representations.

Perimeter. The number of pixels in the boundary of the shape.

(Non-)Compactness or (Non-)Circularity. How closely-packed the shape is (not): perimeter^2/area. The most compact shape is a circle (4π). All other shapes have a compactness larger than 4π.

Eccentricity. The ratio of the length of the longest chord of the shape to the longest chord perpendicular to it. (This is one way to define it—there are others.)

Elongation. The ratio of the height and width of a rotated minimal bounding box. In other words, rotate a rectangle so that it is the smallest rectangle in which the shape fits. Then compare its height to its width.

Rectangularity. How rectangular a shape is (how much it fills its minimal bounding box): area of object/area of bounding box. This value has a value of 1 for a rectangle and can, in the limit, approach 0 (picture a thin X).

Orientation. The overall direction of the shape. (I'll come back and define this more precisely later.)

9.3 Practical Considerations for Measuring Properties

When measuring the perimeter of an object, do we follow the pixels on the inside of the object's boundary, the exterior pixels just outside that boundary, or the "cracks" between the interior and exterior pixels? Any of the three methods can be used, and the differences go away as the object size increases—just be consistent.

Also, when measuring perimeter, does one simply count pixels? That can artificially bias diagonal edges high (if counting 4-connected) or low (if counting 8-connected). We can do what we discussed earlier for distance measures: count 1 pixel for 4-connected neighbors and $\sqrt{2}$ for 8-connected ones.

9.4 Topological Descriptors

If we stretch a shape like we would a cutout from a rubber sheet, there are certain shapes we can make and others we can't. *Topology* refers to properties of the shape that don't change, so long as you aren't allowed to tear or join parts of the shape.

A useful topological descriptor is the *Euler number* E: the number of connected components C minus the number of holes H:

$$E = C - H$$

(You should know how to use connected-component labeling to determine the Euler number.) While the Euler number may seem like a simple descriptor (it is), it can be useful for separating simple shapes.

9.5 Convex Hull: Bays

Although not strictly a topological property, we can also describe shape properties by measuring the number or size of concavities in the shape. We can do this first finding the convex hull of the shape and then subtracting the shape itself. What are left are either holes ("lakes") or concavities ("bays"). This might be useful in trying to distinguish the letter "O" from "C", etc. One can then recursively identify concavities within each "bay", etc. The resulting structure is a *concavity tree*.

9.6 Extremal Points

Another way to describe a shape is to find its *extremal points*.

The simplest form of extremal representation is the *bounding box*: the smallest rectangle that completely contains the object.

A more powerful way of using extremal points is to find the eight points defined by: top left, top right, left top, left bottom, bottom right, bottom left, right top, and right bottom. (Obviously, some of these points may be the same.)

By connecting opposing pairs of extremal points (top left to bottom right, etc.) we can create four axis that describe the shape. These axis can themselves be used as descriptors, or we can use combinations of them.

For example, the longest axis and its opposite (though not necessarily orthogonal axis) can be labeled the *major and minor axes*. The ratio of these can be used to define an *aspect ratio* or *eccentricity* for the object. The direction of the major axis can be used as an (approximate) orientation for the object.

How do extremal points respond to transformations?

9.7 Profiles

A useful region-based signature is the *profile* or *projection*. The *vertical profile* is the number of pixels in the region in each column. The *horizontal profile* is the number of pixels in the region in each row. One can also define *diagonal profiles*, which count the number of pixels on each diagonal.

Profiles have been used in character recognition applications, where an L and a T have very different horizontal profiles (but identical vertical profiles), or where A and H have different vertical profiles (but identical horizontal profiles).

Profiles are useful as shape signatures, or they may be used separate different regions. A technique known as "signature parsing" uses vertical profiles to separate horizontally-separated regions, then individual horizontal profiles to recursively separate vertically-separated regions, etc. By alternating this process and recursively splitting each region, one "parse" binary objects that are organized horizontally and vertically, like text on a page.

9.8 Moments

Another way to describe shape uses statistical properties called *moments*.

9.8.1 What Statistical Moments Are

For a discrete one-dimensional function $f(x)$, we can compute the mean value of the function using

$$\mu = \frac{\sum_{x=1}^{N} x f(x)}{\sum_{x=1}^{N} f(x)} \tag{9.1}$$

We can also describe the variance by

$$\sigma^2 = \frac{\sum_{x=1}^{N} (x-\mu)^2 f(x)}{\sum_{x=1}^{N} f(x)} \tag{9.2}$$

A third statistical property, called *skew*, describes how symmetric the function is:

$$\text{skew} = \frac{\sum_{x=1}^{N} (x-\mu)^3 f(x)}{\sum_{x=1}^{N} f(x)} \tag{9.3}$$

All of these are examples of moments of a function.

One can define moments about some arbitrary point, usually either about zero or about the mean. The n-th moment about zero, denoted as m_n, is

$$m_n = \frac{\sum_{x=1}^{N} x^n f(x)}{\sum_{x=1}^{N} f(x)} \tag{9.4}$$

The zeroeth moment, m_0, is always equal to 1. The mean μ is the first moment about zero:

$$\mu = m_1 \tag{9.5}$$

The n-th moment about the mean, denoted as μ_n and called the n-th *central moment* is

$$\mu_n = \frac{\sum_{x=1}^{N} (x-\mu)^n f(x)}{\sum_{x=1}^{N} f(x)} \tag{9.6}$$

The zeroeth central moment μ_0 is, again, equal to 1. The first central moment μ_1 is always 0. (Do you see why?) The second central moment μ_2 is the *variance*:

$$\sigma^2 = \mu_2 \tag{9.7}$$

The third central moment μ_3 is the *skew*:

$$\text{skew} = \mu_3 \tag{9.8}$$

The fourth central moment μ_4 is the *kirtosis*:

$$\text{kirtosis} = \mu_4 \tag{9.9}$$

If we have an infinite number of central moments, we can completely describe the shape of the function. (We do, of course, also need the mean itself to position the function).

9.8.2 Moments of Two-Dimensional Functions

Suppose that we have a discrete function not over one variable but of two. We can extend the concept of moments by defining the ijth moment about zero as

$$m_{ij} = \frac{\sum_{x=1}^{N} \sum_{y=1}^{N} x^i y^j f(x,y)}{\sum_{x=1}^{N} \sum_{y=1}^{N} f(x,y)} \tag{9.10}$$

Again, $m_{00} = 1$. The m_{10} is the x component μ_x of the mean and m_{01} is the y component μ_y of the mean.

We define the *central moments* as

$$\mu_{ij} = \frac{\sum_{x=1}^{N} \sum_{y=1}^{N} (x-\mu_x)^i (y-\mu_y)^j f(x,y)}{\sum_{x=1}^{N} \sum_{y=1}^{N} f(x,y)} \tag{9.11}$$

Similar to before, $\mu_{10} = \mu_{01} = 0$.

We can use these moments to provide useful descriptors of shape. Suppose that for a binary shape we let the pixels outside the shape have value 0 and the pixels inside the shape value 1. The moments μ_{20} and μ_{02} are thus the variances of x and y respectively. The moment μ_{11} is the *covariance* between x and y, something you may have already seen in other courses. You can use the covariance to determine the orientation of the shape. The covariance matrix C is

$$C = \begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix} \tag{9.12}$$

By finding the eigenvalues and eigenvectors of C, we can determine the eccentricity of the shape (how elongated it is) by looking at the ratio of the eigenvalues, and we can determine the direction of elongation by using the direction of the eigenvector with whose corresponding eigenvalue has the largest absolute value. The orientation can also be calculated using

$$\theta = \frac{1}{2}\tan^{-1}\frac{2\mu_{11}}{\mu_{20}-\mu_{02}} \tag{9.13}$$

(Note: be careful when implementing this—there is a 180-degree ambiguity in arctangents that turns into a 90-degree ambiguity when you take half the angle. If you're coding this in C or Java, use `atan2` instead of `atan`.

There are many other useful ways that we can combine moments to describe a shape. Even more fundamentally, though, we can completely reconstruct the shape if we have enough moments. But what if we don't use enough? Well, we can construct approximations to the shape that share these same properties. I could, for example, make a circle with the same mean as the shape, an ellipse with the same mean and covariance, etc.

In a sense, the sequence of moments is analogous to the components of a Fourier sequence—the first few terms give the general shape, and the later terms fill in finer detail.

Moments have been shown to be a very useful set of descriptors for matching.

9.8.3 Profile Moments

A number of satistical moments for 2-d regions can be calculated from the profiles of those regions. Let's define the following profiles:

$$\begin{aligned} P_v &= \text{vertical profile of } f(x,y) \\ P_h &= \text{horizontal profile of } f(x,y) \\ P_d &= \text{diagonal } (45 \text{ degree) profile of } f(x,y) \\ P_e &= \text{diagonal } (-45 \text{ degree) profile of } f(x,y) \end{aligned}$$

$$\begin{aligned}\mu_{10} &= \mu_v \\ \mu_{01} &= \mu_h\end{aligned}$$

$$\begin{aligned}\mu_{20} &= \mu_{vv} \\ \mu_{02} &= \mu_{hh} \\ \mu_{11} &= \frac{1}{2}\left(\mu_{dd} - \mu_{hh} - \mu_{vv}\right)\end{aligned}$$

These can then be used to derive the object orientation. This way of computing orientation can be much more efficient than calculating the 2-dimensional moments directly.

9.8.4 Response to Transformations

Finally, let's consider how moments respond to transformations.

Translation. If we translate the object, we only change the mean, not the variance or higher-order moments. So, none of the central moments affected by translation.

Rotation. If we rotate the shape we change the relative variances and higher-order moments, but certain quantities such as the eigenvalues of the covariance matrix are invariant to rotation.

Scaling Resizing the object by a factor of S is the same as scaling the x and y coordinates by S. This merely scales $x - \mu_x$ and $y - \mu_y$ by S as well. Hence, the n-th moments scale by the corresponding power of S^n. Ratios of same-order moments, such as the ratio of the eigenvalues of the covariance matrix, stay the same under scaling, as do area-normalized second-order moments.

By combining moments, we can thus produce *invariant moments*, ones that are invariant to rotation, translation, and scale. The methods for doing so for second-order moments are straightforward. The methods for doing so for higher-order moments is more complicated but also possible.

Vocabulary

- Invariance
- Euler number
- Concavity Tree
- Compactness
- Eccentricity
- Orientation
- Profile
- Statistical Moments
- Central Moment
- Variance
- Skew

- Kirtosis
- Covariance
- Covariance Matrix

Lecture 10: Shape Description (Regions, cont'd)

Contents

Reading

SH&B, 6.3.4–6.4
Castleman, 19.3.4.3

10.1 Distance Transforms (or Distance Maps)

Recall from our earlier discussions that a *distance transform* or *distance map* assigns to each point $\overline{x}$ in the image a minimal distance $D(\overline{x})$ from some locus of points (usually object boundaries). If we build a distance transform from the boundary of the object, we end up with zero at the boundary, increasing values just inside it, yet higher values farther inside, etc. Similarly, we get distances increasing as we go outward from the boundary. (A large number of applications use a positive distance map outside the object and a negative one inside, or vice versa.)

10.1.1 Using Distance Maps

Distance maps have several applications:

- Interpolation

  Powerful techniques have been developed for interpolating between two shapes by interpolating between their (positive outside, negative inside) distance maps. By using the 0-set of the map as the boundary at each step in the iteration, one can find intermediate boundaries between the two original ones. This is called interpolation by *implicit functions*.

- Shape description

  Distance functions can also be used to help create representations for shape. Since shape and spatial relationships (distance) are so integrally related, this would seem to be an obvious application. One specific such application is the *medial axis*, a shape representation that we'll spend the rest of this lecture discussing.

10.2 Why Medial?

All of the methods we've discussed so far can be used to differentiate between shapes, but they don't describe the shape well in any way that people readily understand.

The boundary- and region-based representations we've seen so far give ready access to certain parameters of the shape but not to others. Suppose we wanted to know how wide the object is, or how many branches it has, etc. These are things not readily available from the boundary description. In a sense these are *structural properties* of the shape that go beyond simple boundaries.

To infer larger structure from a shape requires that we be able to group together parts of the boundaries that aren't necessarily adjacent. For example, there is an immediacy to the way we see the two sides of an object. If we look at telephone poll, for example, we can make immediate judgments of how thick or thin it is. We do so by pairing up the two opposide sides and comparing them, even without having to trace completely around the object to ensure that they are indeed part of the same contour.

Once we pair sides in this fashion, it's reasonable to start talking about object middles. These middles give structure to the shape, allowing us to identify trunks, arms and branches, narrowing or widening sections, etc. Such middle-related structures are called *medial* representations.

10.3 Blum's Medial Axis

10.3.1 Definition

The most common form of medial representation is Blum's *medial axis*. (In later years, Harry Blum wrote that he preferred to call it the *symmetric axis*, but others continue to call it the medial axis as part of the larger class of medial representations.)

Blum's medial axis can be defined in a few different intuitive ways, all equivalent:

Grassfire Analogy. Consider a fire started at the boundary that spreads with uniform speed and burns everything in its path. As the fire spreads into the middle, each part eventually meets other parts of the fire started at other parts of the boundary. When these two wavefronts meet, they quench each other because there is nothing left for either to burn. Since the fire spreads at uniform speed, these ”quench points” must be equidistant from two different parts of the boundary. The locus of these quench points is the medial axis.

Maximal Discs. Suppose that at each point of the boundary you placed a circle on the inside of the boundary such that it was tangent to it. If you start with a very small circle and expand it outwards, it will eventually ”bump into” other parts of the boundary. When you've grown it as large as possible, you'll have a circle that has the following properties:

- It is entirely within the object, and
- It is tangent to the object at more than one point

The locus of centers of all such circles is the medial axis.

Ridges in a Distance Transform. As we progress from away from a particular point on the boundary, we find increasing values in the distance map until we reach a point as which we are now closer to some other part of the boundary than the place that we left. So, the distance increases as you progress inwards until it forms a *ridge*, a locus of points with values higher than those just off the ridge on either side. The locus of these ridges in the distance transform is the medial axis.

There are, of course, various mathematical definitions in the literature as well. They come in different forms, but they typically correspond to one of these three intuitive definitions.

All three of these definitions can be used to construct morphological approaches to finding medial axis.

- If we strip away boundary pixels, we eventually get down to single-pixel thin curves that cannot be safely removed—these approximate the medial axis (grassfire or distance transform definitions).

- If we find explictly maximal disks by using morphological openings, we can also find the skeleton. In this case, though, we may find that because of discrete pixel locations we end up with a disconnected set of points. This is sometimes called the *morphological skeleton*. Like the medial axis it can be used to reconstruct an object, but it loses important shape information.

10.3.2 Medial Hierarchies

The medial axis is a continuous structure that may branch in places. Each branch corresponds to a different piece of the object. It these branches that give the structural decomposition of the medial axis by forming a hierarchy—each bump on an arm of the shape is a branch of a branch.

10.3.3 Radius Values

At each point of the medial axis, we also have a corresponding r value: the radius of the maximal disc, the distance from the grassfire, or the distance from the distance map.

If we let s be arc length along any given segment of the axis, we have a one-dimensional function $r(s)$ along the segment.

If $\frac{dr}{ds}$ is positive, this part of the object widens as we progress along the axis segment; if it's negative the part narrows.

10.3.4 Types of Points

The easiest definition for which to consider the various properties of the medial axis is the maximal-disk one. Consider one such disk. As mentioned earlier, it touches the boundary in more than one place. These points can come in different flavors:

Two disjoint sets of points. Most of the axis points fall into this category.

Three disjoint sets of points. These are branch points.

Exactly one contiguous set of points. These are end points.

Notice that its possible for the medial disc to touch the boundary in a set of contiguous points rather than a single one. In other words, the boundary is locally circular.

These sets of points don't really change the properties of the resulting axis—it's the number of disjoint sets of points, not the actual number of points themselves, that are important.

10.3.5 Sensitivity to Noise

Sounds great so far, right? Well, the medial axis has one problem—it is extremely sensitive to noise in the boundary. Since the axis has a branch for each structural piece of the object, it has to spawn a new branch for each little bump and wiggle.

The length of the new branch isn't necessarily reflective of the significance of the variation. It must extend all the way out to the base of the bump itself.

10.3.6 Building and Pruning Hierarchies

Because of this sensitivity to noise, many researchers have studied ways to prune the noisy tree-like structure produced by the medial axis. Ways to do this include

- Smoothing the boundary before computing the axis,
- Pairing the boundary points to the corresponding axis segments and pruning based on length of the boundary (Ogniewicz, 1992).

By ordering the segments according to the order in which they would be pruned, you can organize them into a top-down hierarchy.

Pruning away segments or arranging them in order of significance can (to a limited degree) let you compare similar but slightly different objects.

10.4 Other Medial Representations

Let's consider the maximal disc in Blum's definition. As already discussed, it touches the boundary in at least two places.

- The center of the circle lies on the Blum symmetric axis.
- The midpoint of the chord between the two boundary points lies on Brady's *Smooth Locus of Symmetries* or SLS.
- The midpoint of the chord between the two boundary points lies on Leyton's *Process-Induced Symmetric Axis* of PISA.

Brady's SLS definition seems to be more intuitive than Blum's because the axis really does lie directly between the related points. It is more complex to compute, however.

Leyton's PISA definition isn't quite as intuitive, but the resulting representation seems to describe the way you have to push in or push out the boundary to produce the shape (hence the "process-induced" in the name).

Another method, which approaches the Blum definition when taken to the continuous limit, is the *Voronoi Skeleton*. I'll come back to this later.

10.5 Computational Methods

One can use the definitions of the axis to precisely determine the medial axis of a continuous curve. There are, however, two problems:

1. Computational expense.
2. The discrete pixel grid.

For this reason, various algorithms have been produced that produce medial axes that are approximations to Blum's.

10.5.1 Thinning

One way to find the middle of something is through the *thinning* process we discussed earlier. Suppose that we successively peel away thin layers from the object, but only if they don't get rid of single-pixel thick parts of the remaining object. This thinning method is analogous to the grassfire definition of the medial axis. What is left from this thinning is a structure called a *skeleton*, similar in principle to the medial axis.

10.5.2 Distance Maps

Another way to compute medial axes is to use the distance-map definition. We can explicitly compute a distance map and then analyze it to find ridges. This requires two steps:

1. Generate the distance map, and
2. Find its ridges.

Finding Ridges

We won't talk right now about finding ridges in functions, but we'll come back to it later when we discuss geometric methods for local analysis of functions (images).

10.5.3 Voronoi Methods

If we distribute discrete points along the boundary and compute the Voronoi diagram for the points (the locus of all points that are no nearer to one point than another), contained within the Vornoi diagram is a skeletal structure for the shape. If we take this to the limit by adding more and more points along a continous contour, this Voronoi skeleton approaches the Blum medial axis. [Make sure to convince yourself why.]

We don't have to go to this limit, though, to get a usable skeletal structure.

10.6 Higher Dimensions

The medial structure of a three-dimensional shape isn't composed of one-dimensional curve segments but two-dimensional manifolds within the three-dimensional space. These *medial manifolds* also have useful applications for three-dimensional vision.

Vocabulary

- Distance
- Distance Transform
- Skeleton
- Medial Axis
- Voronoi Diagram

Lecture 11: Differential Geometry

©Bryan S. Morse, Brigham Young University, 1998–2000
Last modified on February 28, 2000 at 8:45 PM

Contents

Reading

These notes.

11.1 Introduction

11.1.1 What Differential Geometry Is

Differential Geometry is very much what the name implies: geometry done using differential calculus. In other words, shape description through derivatives.

11.1.2 Assumptions of Differentiability

The first major assumption made in differential geometry is, of course, that the curve, surface, etc. is everywhere differentiable—there are no sharp corners in a differential-geometry world. (Of course, once one accepts the notion of scaled measurements, there are no infinitely-precise corners anyway.)

11.1.3 Normals and Tangent Planes

The first obvious derivative quantity is the curve or surface *tangent*. For curves, this is a vector; for surfaces, this is a *tangent plane*. For differential geometry of 3D shapes, there is no "global" coordinate system. All measurements are made relative to the local tangent plane or normal.

For images, though, we're going to use a coordinate system that defines the image intensity as the "up" direction.

Be aware that differential geometry as a means for analyzing a function (i.e., an image) is quite different from differential geometry on general surfaces in 3D. The concepts are similar, but the means of calculation are different.

11.1.4 Disclaimer

If a true differential geometer were to read these notes, he would probably cringe. This is a *highly* condensed and simplified version of differential geometry. As such, it contains no discussion of forms (other than the Second Fundamental Form), covectors, contraction, etc. It also does not attempt to address non-Euclidean aspects of differential geometry such as the bracketing, the Levi-Civita tensor, etc.

11.1.5 Notation

For this lecture, we'll use the notation common in most of the literature on the subject. This notation uses subscripts to denote derivatives in the following fashion.

If we let $L(x, y)$ be our image's luminance function (pixel values), the derivative with respect to x is L_x, and the derivative with respect to y is L_y. In fact, we can denote the derivative of L with respect to *any* direction (unit vector) $\bar{v}$ as L_v.

By extension, the second derivative with respect to x is L_{xx}, the second derivative with respect to y is L_{yy}, and the mixed partial (the derivative of L_x with respect to h is L_{xy}. In general, the derivative with respect to vector $\bar{u}$ *as it changes in direction* $\bar{v}$ is L_{uv}.

11.2 First Derivatives

11.2.1 The Gradient

The primary first-order differential quantity for an image as the *gradient*. The gradient of a function f is defined as

$$\nabla L(x, y) = \begin{bmatrix} L_x(x, y) \\ L_y(x, y) \end{bmatrix}$$

Notice that the gradient is a 2-D vector quantity. It has both direction and magnitude *which vary at each point*. For simplicity, we usually drop the pixel locations and simply write

$$\nabla L = \begin{bmatrix} L_x \\ L_y \end{bmatrix}$$

The gradient at a point has the following properties:

- The gradient direction at a point is the direction of steepest ascent at that point.
- The gradient magnitude is the steepness of that ascent.
- The gradient direction is the normal to the level curve at that point.

- The gradient defines the tangent plane at that point.
- The gradient can be used to calculate the first derivative *in any direction*:

$$L_v = \bar{v} \cdot \nabla L$$

The gradient can thus be used as a *universal first-derivative calculator*. It includes all first-derivative information there is at a point.

11.2.2 First-order Gauge Coordinates

One of the fundamental concepts of differential geometry is that we can describe local surface properties not with respect to some global coordinate system but with respect to a coordinate system *dictated by the local (image) surface itself*. This concept is known as *gauge coordinates*, a coordinate system that the surface "carries along" with itself wherever it goes. Many useful image properties can likewise be most easily described using gauge coordinates. (More on this later.)

The gradient direction is one of those intrinsic image directions, independent of our choice of spatial coordinate axis. (The numbers we use to express the gradient as a vector in terms of our selected coordinate system will, of course, change as we change coordinates, but the gradient direction itself will not.)

The gradient direction and its perpendicular constitute *first-order gauge coordinates*, which are best understood in terms of the image's level curves. An *isophote* is a curve of constant intensity. The normal to the isophote curve is the gradient direction. Naturally, the tangent to the isophote curve is the gradient perpendicular.

We denote the first-order gauge coordinates as directions v and w where

$$\begin{aligned} w &= \frac{\nabla L}{\|\nabla L\|} \\ v &= w_{\perp} \end{aligned}$$

Local properties of isophotes (image level curves) are best described in terms of these gauge coordinates.

11.3 Second Derivatives

11.3.1 The Hessian

For second-order geometry, the equivalent of the gradient is the matrix of second derivatives or *Hessian*:

$$H = \begin{bmatrix} L_{xx} & L_{xy} \\ L_{yx} & L_{yy} \end{bmatrix}$$

Since $L_{xy} = L_{yx}$, this matrix is symmetric.

We can use the Hessian to calculate second derivatives in this way:

$$L_{uv} = \bar{u}^T H \bar{v}$$

or if we use the same vector on both sides of the matrix:

$$L_{vv} = \bar{v}^T H \bar{v}$$

In other words, it's a sort of "universal second derivative calculator"

Here's an example. The unit vector in the x direction is $[1, 0]^T$. We get the second derivative in this direction by calculating

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} L_{xx} & L_{xy} \\ L_{yx} & L_{yy} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This is simply L_{xx}. We can do this with any unit vector in any direction.

The Hessian matrix, being real and symmetric, has a few interesting properties:

1. Its determinant is equal to the product of its eigenvalues and is invariant to our selection of x and y.
2. Its trace (sum of the diagonal elements) is also invariant to our selection of x and y.
3. Proper selection of x and y can produce a diagonal matrix—these are the eigenvectors of H, and the diagonal elements are the eigenvalues.

11.3.2 Principal Curvatures and Directions

The eigenvalues and eigenvectors of the Hessian have geometric meaning:

- The first eigenvector (the one whose corresponding eigenvalue has the largest absolute value) is the direction of greatest curvature (second derivative).
- The second eigenvector (the one whose corresponding eigenvalue has the smallest absolute value) is the direction of least curvature.
- The corresponding eigenvalues are the respective amounts of these curvatures.

The eigenvalues of H are called *principal directions* and are directions of pure curvature (no mixed partial derivative). They are always orthogonal.

The eigenvalues of H are called *principal curvatures* and are invariant under rotation. They are denoted κ_1 and κ_2 and are always real valued.

11.3.3 Second-order Gauge Coordinates

Just as we used the gradient direction and perpendicular as *first-order gauge coordinates*, we can use the principal directions as *second-order gauge coordinates*. These second-order gauge coordinates, like their first-order counterparts, are intrinsic to the image surface. They are the most useful (and most natural) coordinate system for describing second-order properties.

We denote these second-order gauge directions (the principal curvature directions) as p and q. So, $L_{pp} = \kappa_1$, $L_{qq} = \kappa_2$, and by convention $|L_{pp}| \geq |L_{qq}|$.

11.4 Using Principal Curvatures and Directions

11.4.1 Gaussian Curvature

We can write the determinant of H as the product of κ_1 and κ_2. This quantity is the *Gaussian Curvature* and is denoted as K.

Geometrically, it too has a meaning. Suppose that you took a local patch around a point on the surface and "squashed" it flat. To do so, you'd have to tear the surface, because the patch has more area than its perimeter would normally contain for a flat region. The Gaussian curvature is the amount of this "extra stuff".

In second-order gauge coordinates, the Gaussian curvature is $L_{pp}L_{qq}$.

11.4.2 Mean Curvature

The mean curvature is the average of κ_1 and κ_2 and is denoted as H. It is also equal to the half the trace of H, which we earlier said was invariant to our selection of x and y.

In second-order gauge coordinates, the mean curvature is $(L_{pp} + L_{qq})/2$.

11.4.3 Laplacian

The *Laplacian* that you learned about in CS 450 can be written as $L_{xx} + L_{yy}$. This is simply twice the mean curvature and is also invariant to rotation.

11.4.4 Deviation From Flatness

The *deviation from flatness* is $L_{pp}^2 + L_{qq}^2$ and is another useful way of measuring local "unflatness".

11.4.5 Patch Classification

Surface regions can be classified according to their mean and Gaussian curvatures.

- Elliptic patches: $K > 0$. The curvature in any direction is positive.
  - Convex: $H > 0$
  - Concave: $H < 0$. The curvature in any direction is negative.
- Hyperbolic patches: $K < 0$. The curvatures in some directions are positive and others are negative.

11.4.6 Parabolic Points

We've covered the cases for $K > 0$ and $K < 0$. If K exactly equals 0, one or both of the principal curvatures is 0. Such points form curves on the surface known as *parabolic curves*. These curves naturally lie on the boundaries between elliptic and hyperbolic regions.

11.4.7 Umbilics

An *umbilic* is a point whose principal curvatures are equal: $\kappa_1 = \kappa_2$. At such a point, the eigenvectors are not uniquely defined (the eigenspace of H is the entire tangent plane). In other words, such a point has the same curvature *in every direction* (locally spherical). Since the principal directions are not definted at an umbilic, we generally choose them to vary smoothly through the umbilic.

11.4.8 Minimal Points

Hyperbolic regions are saddle-shaped. One special point that can occur in such regions is the cousin of an umbilic: a point with equal-magnitude but opposite-sign principal curvatures: $\kappa_1 = -\kappa_2$. Such a point is known as a *minimal point*. (The name comes from *minimal surfaces*, the soap-bubble surfaces that form across holes by minimizing the surface area across the film.)

11.4.9 Shape Classification

Jan Koenderink has proposed a "shape classification" space where each point lives in a Cartesian $\kappa_1 \times \kappa_2$ plane. However, this space can best be thought of in its polar form:

$$\begin{aligned} S &= \tan^{-1}\left(\frac{\kappa_2}{\kappa_1}\right) \\ C &= \sqrt{\kappa_1^2 + \kappa_2^2} \end{aligned}$$

where S defines the shape angle, and C defines the degree of curvature. (Observe that C is the square-root of the deviation from flatness.) Points with the same S value but differing C values can be thought of as being the same shape but only "more stretched".

Quadrant I of this space is convex, Quadrant III is concave, and Quadrants II and IV are hyperbolic. The Cartesian axes of the space are parabolic points. At 45 and -135 degrees lie the umbilics; at 135 and -45 degrees lie the minimal points.

11.5 Other Concepts

11.5.1 Geodesics

A *geodesic* is a path of shortest distance between two points. On a plane, geodesics are straight lines ("the shortest path between two points is a straight line"). On a surface, though, this isn't necessarily true. In fact, there may be multiple geodesics between two points. "Great circles" on (flattened) earth maps are an example of geodesics.

11.5.2 Genericity

Points with special properties are usually only important if they are predictable and stable instead of coincidental. Properties that occur by coincidence go away with a small perturbation of the geometry. Consider, for example, two lines intersecting in three-space—their intersection goes away if either line is perturbed by a small amount. Such situations are called *nongeneric*.

Consider, however, the intersection of two lines in a two-dimensional plane. If either line moves a small amount, the point of intersection may move, but it doesn't go away. Such situations are called *generic*.

11.6 Applications (Examples)

11.6.1 Corners and Isophote Curvature

Corners are places where the isophote has high curvature. The isophote curvature can be written as

$$\kappa = -\frac{L_{vv}}{L_w}$$

Homework exercise: Using what we've talked about so far, express this in terms of L_x, L_y, L_{xx}, L_{yy}, and L_{xy}.

11.6.2 Ridges

Ridges are an intuitively simple concept, but their mathematical definition is not well agreed upon. One definition is that a ridge is a maximum of image intensity in the direction of maximal curvature. This may be written as two constraints:

1. $L_p = 0$
2. $L_{pp} < 0$

This is simply the classic maximum test from calculus.

Another is that a ridge is a maximum of isophote curvature (i.e., a ridge is a connected locus of isophote corners). This can be defined as

$$L_{vvv} = 0$$

11.7 Conclusion

The whole aim of this discussion of differential geometry on image surfaces is to introduce the notion of gauge coordinates as a way of measuring *invariants*, quantities that do not change with arbitrary selection of spatial coordinate systems. The following quantities are invariant to rotation:

- gradient magnitude L_w (L_v is also invariant, but it's always zero)
- the first principal curvature L_{pp}
- the second principal curvature L_{qq}

The following quantities can be computed from these and are thus also invariant:

- isophote curvature
- the Laplacian
- Gaussian curvature
- mean curvature
- deviation from flatness
- all special points such as umbilics, minimal points, ridges, corners, etc.

Vocabulary

- Gradient
- Hessian
- Gauge coordinates
- First-order gauge coordinates
- Second-order gauge coordinates
- Principal directions
- Principal curvatures
- Gaussian curvature
- Mean curvature
- Deviation from flatness
- Laplacian
- Elliptical regions
- Hyperbolic regions
- Parabolic point/curve
- Umbilic
- Minimal points
- Shape classification
- Isophote
- Isophote curvature
- Ridge
- Generic geometric properties

Lecture 12: Local Image Preprocessing (Smoothing)

Reading

SH&B, 4.3.1

12.1 Introduction

The material in Sections 4.1–4.2 should be material you've already covered in CS 450. In this lecture, we'll start covering section 4.3, which discusses both smoothing and edge detection. We'll cover smoothing only a little (since it was discussed somewhat in CS 450) and focus primarily on edge detection.

12.2 Smoothing

In CS 450, you learned that smoothing generally involves averaging. In particular, you should have learned two linear techniques:

- averaging multiple frames
- neighborhood averaging

and one non-linear technique:

- median filtering

Averaging multiple frame is an effective technique for diminishing noise, but only if you have multiple images available for the same scene.

Neighborhood averaging, though it has the advantage of working with only a single image, has the drawback that it also blurs the image. Several non-linear variations have been used that try to remove noise while still preserving sharp edges. Median filtering is one of those techniques.

Your book also describes three other non-linear techniques:

- averaging with limited data validity
- averaging according to inverse gradient
- averaging with rotating mask

In each of these three techniques, the idea is the same: examine the local neighborhood to try to determine which neighbors are part of the same region, and average only with those. This attempts to avoid averaging across object regions, thus (hopefully) avoiding blurring.[1] Notice that this introduces vision into image preprocessing: if you can accurately determine which pixels are part of the same regions, you can better smooth them to remove noise while simultaneously preserving sharpness. There are many other variations on this idea, all with the same basic motivation.

[1] If you'll pardon the analogy, a colleague of mine used to refer to this as "racist pixels": each pixel wants to be like its neighbors, but only those that are sufficiently like it already.

Vocabulary

- Non-linear smoothing
- averaging with limited data validity
- averaging according to inverse gradient
- averaging with rotating mask

Lecture 13: Edge Detection

Last modified on February 12, 2000 at 10:00 AM

Contents

Reading

SH&B, 4.3.2–4.3.3
Castleman 18.4–18.5.1

13.1 Introduction

Remember back in Lecture 2 that we defined an *edge* as a place of local transition from one object to another. They aren't complete borders, just locally-identifiable *probable* transitions.

In this lecture and the next, we'll discuss ways for detecting edges locally. To do this, we will exploit local neighborhood operations as covered in CS 450. If you need to brush up on the basics of convolution, now would be a good time to do so.

13.2 First-Derivative Methods

Most edge detectors are based in some way on measuring the intensity gradient at a point in the image.

Recall from our discussion of vector calculus and differential geometry that the gradient operator ∇ is

$$\nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \tag{13.1}$$

When we apply this vector operator to a function, we get

$$\nabla f = \begin{bmatrix} \frac{\partial}{\partial x} f \\ \\ \frac{\partial}{\partial y} f \end{bmatrix} \tag{13.2}$$

As with any vector, we can compute its magnitude $\|\nabla f\|$ and orientation $\phi\left(\nabla f\right)$.

- The gradient magnitude gives the *amount* of the difference between pixels in the neighborhood (the *strength* of the edge).
- The gradient orientation gives the *direction* of the greatest change, which presumably is the direction across the edge (the edge normal).

Many algorithms use only the gradient magnitude, but keep in mind that the gradient orientation often carries just as much information.

Most edge-detecting operators can be thought of as gradient-calculators. Because the gradient is a continuous-function concept and we have discrete functions (images), we have to approximate it. Since derivatives are linear and shift-invariant, gradient calculation is most often done using convolution. Numerous kernels have been proposed for finding edges, and we'll cover some of those here.

13.2.1 Roberts Kernels

Since we're looking for differences between adjacent pixels, one way to find edges is to explictly use a $\{+1, -1\}$ operator that calculates $I(\overline{x}_i) - I(\overline{x}_j)$ for two pixels i and j in a neighborhood. Mathematically, these are called *forward differences*:

$$\frac{\partial I}{\partial x} \approx I(x+1, y) - I(x, y)$$

The Roberts kernels attempt to implement this using the following kernels:

+1	
	-1

g_1

	+1
-1	

g_2

While these aren't specifically derivatives with respect to x and y, they *are* derivatives with respect to the two diagonal directions. These can be thought of as components of the gradient in such a coordinate system. So, we can calculate the gradient magnitude by calculating the length of the gradient vector:

$$g = \sqrt{(g_1 * f)^2 + (g_2 * f)^2}$$

The Roberts kernels are in practice too small to reliably find edges in the presence of noise.

13.2.2 Kirsch Compass Kernels

Another approach is to apply rotated versions of these around a neighborhood and "search" for edges of different orientations:

3	3	3
3	0	3
-5	-5	-5

3	3	3
-5	0	3
-5	-5	3

-5	3	3
-5	0	3
-5	3	3

$\ldots$

Notice that these do not explicitly calculate the gradient—instead, they calculate first-derivatives in specific directions. By taking the result that produces the maximum first derivative, we can approximate the gradient magnitude. However, the orientation is limited to these specific directions. These kernels make up the *Kirsch compass kernels* (so called because they cover a number of explicit directions).

13.2.3 Prewitt Kernels

The Prewitt kernels (named after Judy Prewitt) are based on the idea of the *central difference*:

$$\frac{df}{dx} \approx [f(x+1) - f(x-1)]/2$$

or for two-dimensional images:

$$\frac{\partial I}{\partial x} \approx [I(x+1, y) - I(x-1, y)]/2$$

This corresponds to the following convolution kernel:

-1	0	1

(To get the precise gradient magnitude, we must then divide by two.) By rotating this 90 degrees, we get $\partial I/\partial y$.

These kernels are, however, sensitive to noise. Remember from CS 450 (or from another signal- or image-processing course) that we can reduce some of the effects of noise by *averaging*. This is done in the Prewitt kernels by averaging in y when calculating $\partial I/\partial x$ and by averaging in x when calculating $\partial I/\partial y$. These produce the following kernels:

-1	0	1
-1	0	1
-1	0	1

$\partial/\partial x$

-1	-1	-1
0	0	0
1	1	1

$\partial/\partial y$

(Of course, now we have to divide by six instead of two.) Together, these give us the components of the gradient vector.

13.2.4 Sobel Kernels

The Sobel kernels (named after Irwin Sobel, now currently working for HP Labs) also rely on central differences, but give greater weight to the central pixels when averaging:

-1	0	1
-2	0	2
-1	0	1

$\partial/\partial x$

-1	-2	-1
0	0	0
1	2	1

$\partial/\partial y$

(Of course, now we have to divide by eight instead of six.) Together, these also give us the components of the gradient vector.

The Sobel kernels can also be thought of as 3×3 approximations to first-derivative-of-Gaussian kernels. That is, it is equivalent to first blurring the image using a 3×3 approximation to the Gaussian and then calculating first derivatives. This is because convolution (and derivatives) are commutative and associative:

$$\frac{\partial}{\partial x}(I * G) = I * \frac{\partial}{\partial x} G$$

This is a very useful principle to remember.

13.2.5 Edge Extraction

Most edges are not, however, sharp dropoffs. They're often gradual transitions from one intensity to another. What usually happens in this case is that you get a rising gradient magnitude, a peak of the gradient magnitude, and then a falling gradient magnitude. Extracting the ideal edge is thus a matter of finding this curve with optimal gradient magnitude. Later, we'll discuss ways of finding these optimal curves of gradient magnitude.

13.3 Second-Derivative Methods

There are other operators that try to find these peaks in gradient magnitude directly.

Let's first consider the one-dimensional case. Finding the ideal edge is equivalent to finding the point where the derivative is a maximum (for a rising edge with positive slope) or a minimum (for a falling edge with negative slope).

How do we find maxima or minima of one-dimensional functions? We differentiate them and find places where the derivative is zero. Differentiating the first derivative (gradient magnitude) gives us the second derivative. Finding optimal edges (maxima of gradient magnitude) is thus equivalent to finding *places where the second derivative is zero*.

When we apply differential operators to images, the zeroes rarely fall exactly on a pixel. Typically, they fall between pixels. We can isolate these zeroes by finding *zero crossings*: places where one pixel is positive and a neighbor is negative (or vice versa).

One nice property of zero crossings is that they provide closed paths (except, of course, where the path extends outside the image border). There are, however, two typical problems with zero-crossing methods:

1. They produce two-pixel thick edges (the positive pixel on one side and the negative one on the other, and
2. They can be extremely sensitive to noise.

For two dimensions, there is a single measure, similar to the gradient magnitude that measures second derivatives. Consider the dot product of ∇ with itself:

$$\nabla \cdot \nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \tag{13.3}$$

$$\tag{13.4}$$

$$= \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{13.5}$$

We usually write $\nabla \cdot \nabla$ as ∇^2. It has special name and is called the *Laplacian* operator.

When we apply it to a function, we get

$$\nabla^2 f = \left(\begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial}{\partial x} \\ \\ \frac{\partial}{\partial y} \end{bmatrix} \right) f \tag{13.6}$$

$$= \frac{\partial^2}{\partial x^2} f + \frac{\partial^2}{\partial y^2} f \tag{13.7}$$

One interesting property of the Laplacian is that it is rotationally invariant—it doesn't depend on what directions you choose, so long as they are orthogonal. The sum of the second derivatives in *any* two orthogonal directions is the same.

We can find edges by looking for zero-crossings in the Laplacian of the image. This is called the *Marr-Hildreth* operator.

Let's now turn our attention then to neighborhood operators for measuring the Laplacian.

13.3.1 Laplacian Operators

One way to measure the second drivative in one dimension is to extend the forward-differencing first-derivative operator to the forward difference of two forward differences:

$$\begin{aligned} \tfrac{d^2 f}{dx^2} &\approx [f(x+1) - f(x)] - [f(x) - f(x-1)] \\ &= f(x+1) - 2f(x) + f(x-1) \end{aligned} \tag{13.8}$$

We can thus use a simple $[1, -2, 1]$ neighborhood operator to approximate the second derivative in one dimension. The Laplacian is one of these in the x direction added to one of these in the y direction:

0	0	0
1	-2	1
0	0	0

+

0	1	0
0	-2	0
0	1	0

=

0	1	0
1	-4	1
0	1	0

Other variations include the following:

$$\begin{array}{|c|c|c|}\hline 0.5 & 0.0 & 0.5 \\ \hline 1.0 & -4.0 & 1.0 \\ \hline 0.5 & 0.0 & 0.5 \\ \hline \end{array} + \begin{array}{|c|c|c|}\hline 0.5 & 1.0 & 0.5 \\ \hline 0.0 & -4.0 & 0.0 \\ \hline 0.5 & 1.0 & 0.5 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & -8 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

or

$$\begin{array}{|c|c|c|}\hline 1 & -2 & 1 \\ \hline 1 & -2 & 1 \\ \hline 1 & -2 & 1 \\ \hline \end{array} + \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline -2 & -2 & -2 \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 2 & -1 & 2 \\ \hline -1 & -4 & -1 \\ \hline 2 & -1 & 2 \\ \hline \end{array}$$

As mentioned earlier, zero crossings of this operator can be used to find edges, but it is susceptible to noise. This sensitivity comes not just from the sensitivity of the zero-crossings, but also of the differentiation. In general, the higher the derivative, the more sensitive the operator.

13.3.2 Laplacian of Gaussian Operators

One way to deal with the sensitivity of the Laplacian operator is to do what you'd do with any noise: blur it. This is the same philosophy we saw earlier with the Sobel kernels for first derivatives.

If we first blur an image with a Gaussian smoothing operator and then apply the Laplacian operator, we can get a more robust edge-finder.

But why go to all the trouble? Convolution is associative and commutative, so we can combine the Gaussian smoothing operator with the Laplacian operator (by convolving them one with another) to form a single edge-finding operator.

But this is still too much work. Instead of approximating the Laplacian operator with forward differencing and then applying it to a Gaussian, we can simply differentiate the Gaussian

$$G(x,y) = e^{-r^2/2\sigma^2} \tag{13.9}$$

(where $r^2 = x^2 + y^2$) to get a closed-form solution for the Laplacian of Gaussian:

$$\nabla^2 G(x,y) = \left(\frac{r^2 - \sigma^2}{\sigma^4}\right) e^{-r^2/2\sigma^2} \tag{13.10}$$

We can then approximate this over a discrete spatial neighborhood to give us a convolution kernel.

But wait! There's an even better way. We can not only calculate the closed-form solution for a Laplacian of Gaussian, we can compute its Fourier Transform. We know that the Fourier Transform of a Gaussian is another Gaussian, and we also know that we can differentiate using a ramp function ($2\pi i\nu_x$ or $2\pi i\nu_y$) in the frequency domain. Multiply together the spectrum of the image, the Fourier Transform of a Gaussian, and two differentiating ramps in the one direction and you have a second-derivative of Gaussian in one direction. Do the same thing in the other direction, add them together, and you have the Laplacian of Gaussian of the image. (See, that Fourier stuff from CS 450 comes in handy after all.)

13.3.3 Difference of Gaussians Operator

It can be shown that the Laplacian of a Gaussian is the derivative with respect to $2\sigma^2$ of a Gaussian. That is, it is the limit of one Gaussian minus a just smaller Gaussian. For this reason, we can approximate it as the difference of two Gaussians. This is called the *Difference-of-Gaussians* or DoG operator.

Also appearing in the literature is the *Difference-of-LowPass filters* or DoLP operator.

13.4 Laplacians and Gradients

Zero-crossings of Laplacian operators only tell you where the edge is (actually where the edge lies *between*), not how strong the edge is. To get the strength of the edge, and to pick which of the two candidate pixels is the better approximation, we can look at the gradient magnitude at the position of the Laplacian zero crossings.

13.5 Combined Detection

If one operator works well, why not use several? This seems like a reasonable thing to do, but the way to do it is nontrivial. We won't go into it in class, but understand that it is trying to combine information from multiple orthogonal operators in a vector space and then project the results to the ”edge subspace”.

13.6 The Effect of Window Size

The results you get with a 3×3, 5×5, or 7×7 operator will often be different. What is the correct window size? Well, there isn't one standard one that will work under every circumstance. Sometimes you have to play with it for your application. Smart vision programs will often use multiple windows and combine the results in some way. We'll talk about this notion more in the next lecture.

Vocabulary

- Gradient
- Gradient magnitude
- Gradient orientation
- Roberts edge detector
- Prewitt edge detector
- Sobel edge detector
- Kirsch compass gradient kernels
- Laplacian
- Laplacian-of-Gaussian
- Difference-of-Gaussians
- Marr-Hilbert edge detector

Lecture 14: Edge Detection (cont'd)

Contents

Reading

SH&B, 4.3.4–4.3.5

14.1 Second-Derivative Zero Crossings and Gradient Magnitude

One caution should be added to our discussion of zero-crossings of second derivatives (e.g., the Laplacian) as an edge detector. Remember that the idea is to use zeroes of second derivatives to indicate maxima of gradient magnitude. The problem is that this doesn't include any notion of *how strong* that maximum is. In other words, even a small blip in the gradient magnitude is labelled an edge. We can remedy this by combining second- and first-order derivatives. In other words, an edge can be defined as places where

$$\nabla^2 f = 0 \text{ and } \nabla f > T$$

Of course, now we lose one of the greatest virtues of zero crossings: their continuity. One way to deal with such breaks in continuity is to slowly drop the threshold in places where there is clearly a break. Your text refers to this as *hysteresis thesholding*. In other contexts, it is sometimes known as *edge relaxation*.

14.2 Scale Space

Section 4.3.4 briefly introduces the notion of a *scale space*. This is the space of measurements that you get after blurring an image by successively larger and larger Gaussian kernels. The essential idea of scale spaces is this:

1. It is impossible to make an infinitely small measurement—all measurements of the physical world involve an *aperture*.
2. The size of the aperture affects the measurement.
3. In a sense "information" exists at different aperture scales.
4. To extract information at different scales, you must measure at those scales (or simulate such).
5. *Smaller ("more precise") apertures don't necessarily produce more information than larger ones.*

In a sense, what we've been doing all along by blurring images to smooth away noise is to make measurements at large scales (with larger apertures). By doing so, we get above the scale of noise and try to make measurements at scales more natural for the objects in the image. Of course, if the objects are small relative to the noise (or the noise is large relative to the objects of interest), we can't separate them.

Moral: Choose the right scale for the information you're trying to extract.

We will come back and talk about this idea more in Lecture 22. For now, it is sufficient to think only about some edges being more accurately detected at scales larger than single-pixel changes. This idea, plus another important one from differential geometry, is illustrated in the following edge detection algorithm.

14.3 Canny Edge Detector

14.3.1 Basic Definition

One problem with Laplacian zero-crossings as an edge detector is that it is simply adding the principal curvatures together—it doesn't really determine a maximum of gradient magnitude.

The *Canny Edge Detector* defines edges as *zero-crossings of second derivatives in the direction of greatest first derivative*.

Let's examine each of the parts of this definition:

The Direction of Greatest First Derivative This is simply the gradient direction (w in first-order gauge coordinates).

The Second Derivative in The Direction of ... We can compute this using the matrix of second derivatives (Hessian):

$$L_u = u^T H u$$

Zero Crossings of... As with the Marr-Hildreth edge detector, we'll use positive-negative transitions to "trap" zeroes.

Thus, in gauge coordinates, the Canny detector is

$$L_{ww} = 0$$

In terms of x and y derivatives, this can be expanded to

$$L_{ww} = \frac{1}{L_x^2 + L_y^2} \begin{bmatrix} L_x & L_y \end{bmatrix} \begin{bmatrix} L_{xx} & L_{xy} \\ L_{yx} & L_{yy} \end{bmatrix} \begin{bmatrix} L_x \\ L_y \end{bmatrix} = 0 \tag{14.1}$$

Zero-crossings in this measure give connected edges much like the Laplacian operator but more accurately localize the edge.

14.3.2 Scale and the Canny Operator

The Canny algorithm also makes use of multiple scales to best detect the edges. In particular, it uses Gaussian blurring kernels of varying sizes $\{\sigma_k\}$ and selects the one that is most stable. The selection of the best scale at which to make the measurement is the subject of much research, including current research.

Your text writes this as

$$\frac{\partial}{\partial \overline{n}}(G * g) = 0$$

where g denotes the image, $\overline{n}$ denotes the edge normal (the gradient direction), and G denotes a Gaussian blurring kernel. Remember, though, this can be rewritten as

$$\frac{\partial G}{\partial \overline{n}} * g = 0$$

This is a nice, compact notation, but remember that $\overline{n}$ is also determined from the blurred image $G * g$. This notation also leaves out the details in Eq. 14.1.

If we use Eq. 14.1 to determine the edges, we can use Gaussian blurring by measuring each of the five derivatives in Eq. 14.1 using derivatives-of-Gaussians of the appropriate standard deviation.

14.3.3 The Algorithm

Your text describes the algorithm in this way [my comments added]:

1. Repeat steps (2) till (6) for ascending values of the standard deviation σ.
2. Convolve an image g with a Gaussian of scale σ.
   [Remember that this step is not explicitly necessary—simply convolve with derivatives of Gaussians when performing differentiation.]
3. Estimate local edge normal directions $\overline{n}$ using equation (4.61) for each pixel in the image.
   [Remember: $\overline{n}$ is the gradient direction.]
4. Find the location of the edges using equation (4.63) (non-maximal suppression).
   [i.e., find the zero crossings]
5. Compute the magnitude of the edge using equation (4.64).
   [i.e., compute the gradient magnitude as well]
6. Threshold edges in the image with hysteresis to eliminate spurious responses.
   [Threshold the gradient magnitude as discussed in Section 14.1 earlier in this lecture.]
7. Aggregate the final information about edges at multiple scale using the 'feature synthesis' approach.
   [Select the "best" scale.]

Vocabulary

- Scale space
- Canny edge detector

Lecture 15: Segmentation (Edge Based, Hough Transform)

Contents

Reading

SH&B, 5.2.6
Castelman, 18.5.2.3

15.1 Introduction

Today we'll start talking about how to group isolated edge points into image structures. We'll come back later to talk about edge thresholding, edge relaxation, edge linking, and optimal edge determination using graph searching. All of these require some continuous path of edge pixels, nearby edge pixels, or edge-cost information.

The technique we'll be talking about today, called the *Hough Transform*, doesn't require connected or even nearby edge points.

15.2 Hough Transform: Basic Algorithms

15.2.1 Parametric Representations of Primitives

Let's first talk about finding edges that we know are straight lines. Consider a single isolated edge point (x, y)—there could be an infinite number of lines that could pass through this point.

Each of these lines can be characterized as the solution to some particular equation. The simplest form in which to express a line is the *slope-intercept* form:

$$y = mx + b \tag{15.1}$$

where m is the slope of the line and b is the y-intercept (the y value of the line when it crosses the y axis). Any line can be characterized by these two parameters m and b.

We can characterize each of the possible lines that pass through point (x, y) as having coordinates (m, b) in some slope-intercept space. In fact, for all the lines that pass through a given point, there is a unique value of b for m:

$$b = y - mx \tag{15.2}$$

The set of (m, b) values corresponding to the lines passing through point (x, y) form a line in (m, b) space.

Every point in image space (x, y) corresponds to a line in parameter space (m, b) and each point in (m, b) space corresponds to a line in image space (x, y).

15.2.2 Accumulators

The Hough transform works by letting each feature point (x, y) vote in (m, b) space for each possible line passing through it. These votes are totalled in an *accumulator*.

Suppose that a particular (m, b) has one vote—this means that there is a feature point through which this line passes. What if it has two votes? That would mean that two feature points lie on that line. If a position (m, b) in the accumulator has n votes, this means that n feature points lie on that line.

15.2.3 The Hough Transform Algorithm

The algorithm for the Hough transform can be expressed as follows:

```
1.    Find all of the desired feature points in the image.
2.    For each feature point
3.        For each possibility $i$ in the accumulator that passes
          through the feature point
4.            Increment that position in the accumulator
5.    Find local maxima in the accumulator.
6.    If desired, map each maxima in the accumulator
      back to image space.
```

Algorithm 15.1: Hough transform algorithm

For finding lines, each feature point casts a line of votes in the accumulator.

15.2.4 A Better Way of Expressing Lines

The slope-intercept form of a line has a problem with vertical lines: both m and b are infinite.

Another way of expressing a line is in (ρ, θ) form:

$$x \cos\theta + y \sin\theta = \rho \tag{15.3}$$

One way of interpreting this is to drop a perpendicular from the origin to the line. θ is the angle that the perpendicular makes with the x-axis and ρ is the length of the perpendicular. θ is bounded by $[0, 2\pi]$ and ρ is bounded by the diagonal of the image.

Instead of making lines in the accumulator, each feature point votes for a sinusoid of points in the accumulator. Where these sinusoids cross, there are higher accumulator values. Finding maxima in the accumulator still equates to finding the lines.

15.2.5 Circles

We can extend the Hough transform to other shapes that can be expressed parametrically. For example, a circle of fixed radius can be described fully by the location of its center (x, y).

Think of each feature (edge) point on the circle as saying, ”if I’m on the circle, the center must be in one of these places”. It turns out that the locus of these votes is itself a circle.

But what about circles of unknown size? In this case, we need a third parameter: the radius of the circle. So, we can parameterize circles of arbitrary size by (x, y, r). Instead of casting votes in a circular pattern into a two-dimensional accumulator, we cast votes in circles of successively larger size in a three-dimensional accumulator.

15.2.6 More Complicated Shapes

We can extend the Hough transform to find shapes of arbitrary complexity, so long as we can describe the shape with some fixed number of parameters. The number of parameters required to describe the shape dictates the dimensionality of the accumulator.

15.3 Implementation Details and Variations

15.3.1 Different Orientations

For the line-matching Hough Transform, the orientation of the line is one of the parameters. For the circular form of the transform, orientation is unnecessary. For other methods, you can choose whether or not to use orientation as one of your parameters.

If you don’t use an orientation parameter, you will only be able to find matches in a specific orientation.

If you do include an orientation parameter, you’ll have a larger-dimensional accumulator, but you’ll be able to find matches in various orientations (as many as you want to discretely parameterize).

15.3.2 Using Gradient Orientation

The basic Hough transform can be made more efficient by not allowing feature (edge) points to vote for all possible parametric representations but instead voting only for *those whose representation is consistent with the edge orientation*. For example, instead of casting votes for all lines through a feature point, we may choose to vote only for those with orientation θ in the range $\phi - a \leq \theta \leq \phi + a$ where ϕ is the gradient orientation. We can take this idea further by weighting the strength of the vote by some function of $|\theta - \phi|$.

15.3.3 Discrete Accumulator

This discussion is all well and good for continuous parameterizations, but we need to discretize the parameter space. The granularity with which we discretize the accumulator for the parameter space determines how accurate we’ll be able to position the sought-after target.

15.3.4 Smoothing the Accumulator

Because of noise, discretization of the image and accumulator, and factors inherent in the application itself, we sometimes want to allow a little tolerance in the fitting of lines or other shapes to the edge pixels. We can do this by allowing a feature point to vote note just for a sharp line or curve in the accumulator, but to cast fuzzy votes for nearby possibilities. In essence, this votes not just for all lines or other shapes that pass through the feature point but also for those that pass close by.

This casting of votes for nearby possibilities can be done directly during the voting phase of the transform or afterwards through a post-processing blurring of the accumulator. The way in which you blur the accumulator depends on what nearby possibilities you want to allow.

15.3.5 Finding Relative Maxima in the Accumulator

If you're just looking for one thing, simply pick the largest value in the accumulator and map it back to the image space. If you're looking for other shapes, you need to find all relative maxima.

As a further test, you may want to threshold the maxima that you find by the number of points that voted for it. Relative maxima with few votes probably aren't real matches.

You'll find that this part is the real trick of getting the Hough transform to work.

15.3.6 Grey-level Voting

The discussion so far has been of binary feature points casting binary votes. A more effective way of doing this is to use the full range of grey-level feature detection (i.e., gradient magnitude) and cast votes proportional to the strength of the feature.

15.4 Comparison to Template Matching

Of course, one could also use template matching (correlation) to find lines, circles, etc. However, for an $N \times N$ image and and $M \times M$ template, the computational complexity is $O(N^2M^2)$.

The Hough Transform, by matching only image edge points to target contour points, requires much less computation. In particular, the number of edge points goes up only linearly with N, not by N^2. Likewise with the number of target contour points. Thus, the complexity of the Hough Transform is only $O(NM)$.

15.5 The Generalized Hough Transform

Some shapes may not be easily expressed using a small set of parameters. In this case, we must explicitly list the points on the shape.

Suppose that we make a table that contains all of the edge pixels for our target shape. We can store for each of the pixels its position relative to some reference point for the shape. We can then feature point "think" as follows: "if I'm pixel i on the boundary, the reference point must be at ref$[i]$."

This is called the *Generalized Hough Transform* and can be expressed as follows:

```
1.  Find all of the desired feature points in the image.
2.  For each feature point
3.      For each pixel $i$ on the target's boundary
4.          Get the relative position of the reference point from $i$
5.          Add this offset to the pisition of $i$
6.          Increment that position in the accumulator
7.  Find local maxima in the accumulator.
8.  If desired, map each maxima in the accumulator back to image
    space using the target boundary table
```

Algorithm 15.2: Generalized Hough transform algorithm

We can make this more efficient by incorporating the idea from Section 15.3.2. We can build a table that records for each point with edge tangent orientation θ the direction α and distance r to the reference point. Thus, when we find a point with edge tangent orientation θ, we have to vote only in the direction of r, α. Of course, depending on the complexity of the shape, there may be multiple such points with tangent orientation θ. We thus build our table to store (and vote for!) all such points:

$$\begin{array}{c}\theta_1 : (r_1^1, \alpha_1^1), (r_1^2, \alpha_1^2), \ldots \\ \theta_2 : (r_2^1, \alpha_2^1), (r_2^2, \alpha_2^2), \ldots \\ \theta_3 : (r_3^1, \alpha_3^1), (r_3^2, \alpha_3^2), \ldots \\ \vdots \end{array}$$

This is called an *R-table*.

15.6 Refining the Accumulator

In the Hough transform, feature points vote for *all* possibilities through that point. This can cause unwanted clutter in the accumulator and false matches.

A way to deal with this is to do a second voting phase. In this phase, each point examines the locus of points in the accumulator that it voted for and finds the maximum. It then casts a single vote into a second accumulator *only for this maximum*. In other words, it sees where there is agreement with its neighbors and then goes along with the crowd. Gerig (1987) has shown that this removes this unnecessary clutter and leadsto only a few isolated points in the accumulator.

As another approach, suppose that after we find the global maximum we remove all votes cast by feature points on or near the corresponding match in the image space. We can then continue the process by looking for the next largest global maximum, remembering it, and removing the votes from its points.

Vocabulary

- Hough Transform
- Accumulator
- Generalized Hough Transform

Lecture 16: Segmentation (Edge Based, cont'd)

Contents

Reading

SH&B, 5.2.1–5.2.3

16.1 Introduction

In earlier lectures, we discussed how to identify possible edge pixels. However, these tests are made one-by-one and don't necessarily produce closed object boundaries because of noise, intensity variation, etc. As we discussed in the last lecture, there are many techniques for extracting closed contours given isolated potential edge pixels.

Fitting approaches, such as the Hough transform, don't need to connect the edge pixels because they try to find the best fit of a *known* shape to the edge data. Such approaches are useful for many applications where the general shape (though not its exact position and other parameters) are known ahead of time. In many applications, you don't know the general shape ahead of time, and you thus need to find connected sets of edge pixels explicitly.

In this lecture, we'll start talking about how to link sets of possible edge pixels to find connected boundaries. (In other words, we'll start playing connect-the-dots.)

16.2 Gradient-Magnitude Thresholding

As we have mentioned previously, a common step in edge detection is to follow the gradient operator by a magnitude thresholding operation.

$$E(x,y) = \begin{cases} 1 & \text{if } \|\nabla f(x,y)\| > T \text{ for some threshold } T \\ 0 & \text{otherwise} \end{cases} \tag{16.1}$$

We'll call $\{(x,y) : E(x,y) = 1\}$ the set of *edge pixels*.

Thresholding the gradient magnitude leaves only "strong" edges, but it makes no guarantees of continuity. Your text discusses two general techniques for getting around this problem.

- thresholding with hysteresis
- edge relaxation

Both of these really fall in to the broader category of "relaxation algorithms": ones in which you make an initial tentative labeling or decision then "relax" the categorization *based on what you've already tentatively decided*. The goal is to come up with a globally consistent decision or set of labels.

16.3 Thresholding with Hysteresis

Hysteresis generally refers to the way certain systems tend to stay consistent under gradually changing conditions. They may change from one state to another, but only after a large amount of change–they tend not to fluctuate mildly under small changes.

Thresholding with hysteresis uses two thresholds:

$$\begin{array}{rl} \|\nabla f(x,y)\| \geq t_1 & \text{definitely an edge} \\ t_0 \geq \|\nabla f(x,y)\| < t_1 & \text{maybe an edge, depends on context} \\ \|\nabla f(x,y)\| < t_0 & \text{definitely not an edge} \end{array}$$

The idea is to first identify all definite edge pixels. Then, add all "maybe" pixels (those greater than t_0) *only if they are next to an already-labeled edge pixel*. This process is repeated until it converges.

16.4 Local Processing: Edge Relaxation

A process similar to thresholding with hysteresis is *edge relaxation*.

The idea of edge relaxation is not not simply add pixels if they are next to other edge pixels but to consider the context as well.

Let's consider this question of whether or not a pixel between two sets of edge pixels is itself an edge pixel. One way of determining this is to look at the magnitude of the intervening pixel: if it is relatively high, but less than the threshold used to determine that its neighbors are edge pixels, it's probably an edge. Of course, we can also check the similarity of the gradient magnitude and gradient orientation, just like we did with edge linking.

We can also use this not just to interpolate between edge pixels, but to extrapolate from them as well. Suppose that we have two adjacent edge pixels followed my a slightly sub-threshold one (with similar gradient magnitude and orientation). Again, it's likely that this is really an edge pixel.

We can add these possible pixels to the set of edge pixels and repeat the process. Supposing that these are now really edge pixels, there may be other near-misses that we might want to allow as edge pixels.

In a sense, we are successively relaxing the criteria used to determine edge pixels, taking into account not just the properties of the pixel in question but of its neighbors as well. This process is called *edge relaxation*.

In general, the term *relaxation* applies to any technique such as this that iteratively re-evaluates pixel classification.

16.5 Local Processing: Edge Linking

One way to find connected sets of edge pixels, without having to explicitly first identify which are or are not edges is to trace from pixel to pixel through possible edge points, considering as you go the context along the path.

We can link adjacent edge pixels by examining each pixel-neighbor pair and seeing if they have similar properties:

1. Similar gradient magnitude: $|\ \|\nabla f(x,y)\| - \|\nabla f(x',y')\|\ | \leq T$ for some magnitude difference threshold T.
2. Similar gradient orientation $|\ \phi(\nabla f(x,y)) - \phi(\nabla f(x',y'))\ | \leq A$ for some angular threshold A.

Once the links are established, we take sets of linked pixels and use them as borders.

Notice that unless you constrain the linked pixels in some sense (for example, by scanning along horizontal or vertical lines), these can create clusters of linked pixels rather than long single-pixel thick chains.

Edge linking is usually followed by postprocessing to find sets of linked pixels that are separated by small gaps–these can the be filled in.

As we'll see later, this edge-linking idea can be extended further by considering the set of possible edges as a graph and turning it into a minimum-cost graph searching problem.

16.6 Boundaries of Already-Segmented Regions

In some cases, we may already have an image segmented into regions for which we want to calculate boundaries. In this case, we can simply generate the boundaries by tracing around the region contours. We can do this two ways:

- Trace once around each contour in the image. When we finish tracing one contour, scan the image until we run into another.
- Make one pass through the image, using data structures to keep track of each contour and adding pixels to the appropriate contour as encountered.

The first method is far more common, because you can simply trace each border one by one.

Your text gives a number of algorithms for tracing borders, and they're fairly easy to understand. Think of them as walking around the object with your hand on one wall. The algorithms for tracing these borders are pretty much the same as you'd use if you were in a dark room tracing along a wall and around corners.

These algorithms include tracing either the inner border or the outer border. However, tracing the inner borders means that the borders of two adjacent regions do not share the same pixels (they are adjacent). Likewise, tracing the outer borders means that the border of one object is *inside* the border of an adjacent object. To get around these problems, a hybrid method known as *extended borders* has been developed.

Extended borders basically use inner borders for the upper and left sides of the object and outer borders for the lower and right sides. By doing this, one object's border is the same as the shared border with the adjacent object. (It also has the nice property of being closer in perimeter to the actual shape, thus avoiding the inner-perimeter/outer-perimeter ambiguity we talked about earlier.) The algorithm for extended border extraction is similar to those for inner or outer borders, but you basically keep track at each stage of where the object is relative to you (above or below, left or right).

Vocabulary

- Edge thresholding
- Thresholding with hysteresis
- Edge relaxation
- Edge linking
- Relaxation algorithms

Lecture 17: Segmentation (Edge Based, cont'd)

Contents

Reading

SH&B, 5.2.4–5.2.5

17.1 Global Processing: Graph Searching

Suppose that instead of finding clusters of linked edge pixels we want to find single-pixel thick curves for object boundaries.

One way to do this is to consider the image as being a graph where the vertices of the graph are the pixel corners and the edges of the graph are the pixel cracks. We can then use gradient or other edge-detecting operators to assign costs to each edge. Edge-like cracks would have low costs; cracks that do not look like edges would have high cost. The problem of finding an optimal boundary through as set of possible pixels thus becomes a simple minimal-cost path, graph-searching problem. (And of course, you should all know how to find minimum-cost paths through graphs.)

There are four basic subparts to this problem:

1. How to contruct the graph,
2. How to assign costs,
3. How to choose starting and ending points, and
4. How to find the minimum-cost path.

Let's take a look at an example before we discuss each of these four in more detail.

17.1.1 Example: Top-to-Bottom Scan

Graph Formulation

As an example, let's consider the case where we know that the boundary runs from the top to the bottom of some subset of the image without looping backwards. We can thus build a starting node representing the top of the image. The edges proceeding out from this node are all vertical cracks on the top row of the image. For each of these edges, successors include the corresponding vertical crack on next row, the horizontal crack to the left, and the horizontal crack to the right. Each of these has successors that go down vertically or sideways horizontally. Eventually, we get to the bottom row of the image where all paths lead to the end node.

Cost Function

In this simple case, we can let the cost of the edge along the crack between pixels p and q be

$$c(p,q) = I_{\text{max}} - |I(p) - I(q)| \tag{17.1}$$

where f_{max} is the maximum pixel value. Remember that we're trying to trying to trace between the inside (one intensity) and the outside (presumably another intensity) of an object.

Starting/Ending Points

In this case, selecting the starting and ending nodes is easy—they're simply the top and bottom of the image.

Graph Searching

The following is an algorithm for finding the minimum cost path.

Let $g(n)$ be the cost from the start point to point n.
Let $h(n)$ be an estimate of the cost from n to the end point.
Let $r(n) = g(n) + h(n)$

1. Mark the start point s as open and set $g(s) = 0$.
2. In no node is open, exit with a failure; otherwise continue.
3. Mark closed the open node n whose estimate $r(n)$ is the smallest.
4. If n is the end point, the least-cost path has been found—the least-cost path can be found by following the pointers back from n to s.
5. Expand node n by generating all of its successors.
6. If a successor n_i is not marked, set
   $$g(n_i) = g(n) + c(n, n_i)$$
   mark it as open, and direct pointers from it back to n.
7. If a successor is already marked, update the cost:
   $$g(n_i) = \min(g(n_i, g(n) + c(n, n_i))$$
   and set the pointer for n_i to n. Mark as open those successors that were so updated.
8. Go back to step 2.

Algorithm 17.1: Simple graph-searching algorithm

(We'll work through this in class to show how it works).

17.2 Graph Formulation

Suppose that we know roughly where the boundary lies or some other information that constrains the location of the boundary. One such case is the example we just used.

The example is obviously a simple case, but we can do similar things for more complicated cases. If we know that the boundary goes around the object with a monotomically increasing radial angle (i.e., it may move towards or away from the center as it goes around, but it never doubles back as one sweeps around), we can construct a constraining band around the potential boundary. We can also construct our graph so that it at each stage it sweeps radially around the image, but can vary in radial distance.

Both of these formulations give us discrete multistage graphs, either as we move from row to row or radially around the boundary. Sometimes, though, we may want to allow the graph to curve and wind, passing through possibly *any* pixel. In such a formulation, we can allow unoriented edges along any pixel crack in the image. In other words, an $N \times N$ image would have $N - 1 \times N - 1$ nodes and $N - 1 \times N - 1$ edges. Our search could be between any two vertices and could trace through any of the edges. This is obviously a huge graph and a compute-intensity search.

17.3 Cost Functions

The cost function in the example uses only a simple 2-element mask. Better cost functions can be constructed by using

- better gradient operators,
- gradient orientation,
- other edge-determining operators such as Laplacian zero-crossings. (Remember that this gives you the position, not the magnitude of the edge—we can use it as a cost function by giving lower costs to pixels closer to the zero crossings.)

All of the above possible cost functions are based on the image data, not properties of the boundary curve itself. If we want smoother contours, we can add more cost for paths that would cause greater curvature.

Cost properties based on image data are called *external cost* terms; cost properties based on the boundary curve are called *internal cost* terms. Notice that internal cost for an edge is not fixed—it depends on the curve preceding the edge.

17.4 Selecting Start/End Points

There are three common ways to select starting and ending points:

- Use some known special points.
- Find ”landmark” points—ones that have certain special geometric properties.
- Ask the user.

One important property of the starting and ending points is that thier selection shouldn’t dramatically change the result. While this sometimes happens, it is not a good thing.

17.5 Graph Searching

Finding minimum-cost paths is a very old and well-studied topic in Computer Science. Common ways include:

- Exhausting searching. (Yuck!)
- Greedy algorithms
- Expansion-style algorithms
- Expansion with heuristic algorithms for pruning
- Dynamic programming

Notice that all of these involve optimization. This is a recurring throughout segmentation and other parts of Computer Vision: set up the solution as the optimization of some criteria-evaluation function.

17.5.1 Greedy Algorithms

Greedy algorithms attempt to get the most at each step. Simple pixel-by-pixel linking by taking the best successive edge would be one such greedy algorithm.

17.5.2 Expansion Algorithms

The algorithm discussed earlier in the example (and in your book) is an expansion-based one. The idea is to grow the search outwards looking for optimal additions to each path. When we finally get a path that leads to the end point, it is guaranteed to be optimal. One such method is *Dijsktra's algorithm*.

17.5.3 Expansion with Heuristics for Pruning

If we can estimate using some type of heuristics (guesses) what the remainder of a past might cost, we can prune unproductive paths. These might miss optimal paths through indiscriminate pruning, but it has been shown that if the heuristic is truly a lower bound on the remaining cost (though not necessarily exact), this finds the optimal path. An example of this is the one shown in Algorithm 17.1. The classic paper on using heuristic graph-searching methods for boundary finding is Martelli, 1976.

17.5.4 Dynamic Programming

Suppose that the graph has several discrete stages through which the paths must go. It can be shown that we can find the optimal path by finding the optimal path from the start to each node in the first stage, from each node in the first stage to the second, and so forth. The optimal path from the start to the second stage is the one whose sum from the start to the first stage plus the first to the second stage is minimal. This type of algorithm is called *dynamic programming*.

17.6 Global vs. Local Methods

This lecture highlights one recurring theme in Computer Vision: local processing vs. global processing. Sometimes we can use strictly-local processing, but we may miss more general properties in the image. We may also use strictly-global methods, but while they might optimize some global criteria they may not seem like ideal solutions when viewed locally. This balance between both local and global optimization is one we'll see frequently in this course.

Vocabulary

- Graph-based contour following
- Cost function
- Greedy algorithm
- Heuristic graph searching
- Dynamic programming

Lecture 18: Segmentation (Region Based)

Last modified on March 3, 2000 at 6:00 PM

Contents

Reading

SH&B, 5.3
Castleman 18.8.1, 18.6

18.1 Introduction

We now turn from segmentation by finding boundaries (pixel differences) to segmentation by finding coherent regions (pixel similarities).

This approach has specific advantages over boundary based methods:

- It is guaranteed (by definition) to produce coherent regions. Linking edges, gaps produced by missing edge pixels, etc. are not an issue.
- It works from the inside out, instead of the outside in. The question of which object a pixel belongs to is immediate, not the result of point-in-contour tests.

However, it also has drawbacks:

- Decisions about region membership are often more difficult than applying edge detectors.

- It can't find objects that span multiple disconnected regions. (Whereas edge-based method can be designed to handle "gaps" produced by occlusion—the Hough transform is one example.)

The objectives of region-based approaches can be summarized as follows:

- Produce regions that are as large as possible (i.e., produce as few regions as possible).
- Produce coherent regions, but allow some flexibility for variation within the region.

Notice the inherent tradeoffs here. If we require that the pixels in a region be too similar, we get great coherency and probably won't span separate objects, but we *oversegment* the image into regions much smaller than the actual objects. If we allow more flexibility, we can produce larger regions that more likely fill entire objects, but they may cross multiple objects and "leak" across what should otherwise be boundaries. Remember: the real goal is to find regions that correspond to objects as a person sees them—not an easy goal.

18.2 Basic Idea of Region Growing

Suppose that we start with a single pixel p and wish to expand from that *seed pixel* to fill a coherent region. Let's define a similarity measure $S(i, j)$ such that it produces a high result if pixels i and j are similar and a low one otherwise. First, consider a pixel q adjacent to pixel p. We can add pixel q to pixel p's region iff $S(p, q) > T$ for some threshold T. We can then proceed to the other neighbors of p and do likewise.

Suppose that $S(p, q) > T$ and we added pixel q to pixel p's region. We can now similarly consider the neighbors of q and add them likewise if they are similar enough. If we continue this recursively, we have an algorithm analogous to a "flood fill" but which works not on binary data but on *similar* greyscale data.

Of course, we now have a few unanswered questions to address:

1. How do we define similarity measure S?
2. What threshold T do we use? Does it change or stay constant?
3. If we wish to add q's neighbor r, do we use $S(p, r)$, $S(q, r)$, or something else?

18.3 A Few Common Approaches

18.3.1 Similarity Measures

One obvious similarity measure is to compare individual pixel intensities. This can be sensitive to noise, though.

We can reduce our sensitivity to noise by comparing neighborhood characteristics between pixels. For example, we could compare the *average intensities* over a neighborhood around each pixel. Notice, though, that this is simply the same as applying an averaging kernel through convolution and then doing single-pixel comparison.

One can also gather other features from the neighborhood, such as texture (which we'll cover in a later lecture), gradient, or geometric properties.

18.3.2 Comparing to Original Seed Pixel

One approach is to always compare back to the seed point p by using $S(p, r)$ when considering adding pixel r to the growing region. This the advantage of using a single basis for comparison across all pixels in the region. However, it means that the region produced is very sensitive to the choice of seed pixel. Does it make sense that the region produced by growing pixel p is different than that produced by its neighbor q also in the same region?

18.3.3 Comparing to Neighbor in Region

One way of removing this effect is to compare pixel r to the neighboring pixel q already in the region of p: $S(q, r)$. In this way, each pixel that is already in the region can bring in neighbors who are like it.

The advantage of this method is that it produces transitive closures of similarity. If p is similar to q, and if q is similar to r, p and r end up in the same region.

Of course, this method can cause significant drift as one grows farther away from the original seed pixel. Indeed, the original seed is of no significance once one grows out more than one pixel. What started out finding green pixels ends up adding yellow ones if the transition is gradual enough.

18.3.4 Comparing to Region Statistics

A third approach is to compare candidate pixel r to the *entire* region already collected. Initially, this region consists of pixel p alone, so pixel p dominates. As the region grows, aggregate statistics are collected, and any new pixel r which may be added to the region is compared not to pixel p or to its already-included neighbor q, but to these aggregate statistics.

One simple such statistic is to keep an updated mean of the region pixels. As each new pixel is added, this mean is updated. Although gradual drift is still possible, the weight of all previous pixels in the region act as a damper on such drift. Some texts refers to this as *centroid* region growing.

18.3.5 Multiple Seeds

Another approach is to initialize the region with not only a single pixel but a small set of pixels to better describe the region statistics. With such initialization, not only a region mean is suggested but the *variance* as well. Candidate points can be compared to the region mean *with respect to the region variance*. This gives us at least some hope of producing identical regions under varying noise conditions.

Multiple seeds can be selected either through user interaction or by sampling a small area around the initial seed point. In other words, if we assume that no desired region is indeed smaller than, say, 5 pixels across, we can sample our initial statistics from a 5×5 area around the seed and grow outwards from there.

18.3.6 Cumulative Differences

Another approach is to use a similarity measure that involves not only comparison to a single pixel or to a region statistic, but by calculating cumulative differences as one follows a path from the seed to the candidate point.

In other words, if point q is a neighbor or seed p, and candidate point r is a neighbor of q, instead of using $S(p, r)$ or $S(q, r)$, we can use $S(p, q) + S(q, r)$. This is equivalent to finding the minimum-cost path from p to r and using this as the basis for the addition or rejection of point r.

18.3.7 Counterexamples

Yet another approach is to provide not only a seed pixel that *is* in the desired region but also counterexamples that *are not* in the region. This method allows us to use not only the similarity to the region but the dissimilarity to the exterior (the counterexample). The has the advantage of not requiring a predetermined threshold—the threshold is simply the value at which the candidate point becomes more similar to the background than to the foreground region. It does, however, require some prior knowledge to "train" the system in this way.

18.4 Multiple Regions

18.4.1 Selecting Seed Points

One way to select seed points is to do so interactively. A user can click a mouse inside a desired object and ideally fill the entire object.

But what about automatically segmenting an entire scene this way? Surely user-specified seed points are insufficient for this task. One way is to scatter seed points around the image and hope to fill all of the image with regions. If the current seed points are insufficient, one can include additional seeds by selecting points not already in segmentated regions.

18.4.2 Scanning

Multiple regions can also be identified by scanning the image in a regular fashion adding pixels to growing regions and spawning new regions as needed. We can then make additional passes through the image resolving these regions. Notice that this is basically the same connected-component labelling that we saw earlier, only with a similarity measure instead of binary values.

18.4.3 Region Merging

The limit of the multiple-seed approach is to let *every* pixel be a seed. If two adjacent pixels are similar, merge them into a single region. If two adjacent regions are collectively similar enough, merge them likewise. This collective similarity is usually based on comparing the statistics of each region. Eventually, this method will converge when no further such mergings are possible.

18.4.4 Split and Merge Algorithm

Pure merging methods are, however, computationally expensive because they start from such small initial regions (individual points). We can make this more efficient by recursively splitting the image into smaller and smaller regions until all individual regions are coherent, then recursively merging these to produce larger coherent regions.

First, we must split the image. Start by considering the entire image as one region.

1. If the entire region is coherent (i.e., if all pixels in the region have sufficient similarity), leave it unmodified.

2. If the region is not sufficiently coherent, split it into four quadrants and recursively apply these steps to each new region.

The "splitting" phase basically builds a quadtree like we discussed earlier in Lecture 3. Notice that each of the regions (squares) so produced is now coherent. However, several adjacent squares of varying sizes might have similar characteristics.

We can thus merge these squares into larger regions from the bottom up, much as we merged regions earlier. Since we are starting with regions (hopefully) larger than single pixels, this method is more efficient.

18.5 Hybrid Edge/Region Approaches

One can extend the power of both region- and boundary-based segmentation methods by combining the strengths of the two. For example, we can make region-joining decisions based not only on pixel or neighborhood similarity but also on already-extracted edges and completion of these edges.

Once we've identified two adjanct regions that are candidates for merging, we can examine the boundary between them. If the boundary is of sufficient strength (gradient magnitude, number of edge pixels according to some operator, etc.), we keep the regions separate. If the boundary between them is weak, we merge them.

Most such algorithms use a minimum fraction of edge pixels along the shared region boundary instead of just looking for holes in the boundaries. Thus, even if there is a small hole in the edge contour (noise, variation, etc.), the regions are still kept separate unless this hole or other holes form a sufficiently large fraction of the boundary.

18.6 Watersheds

18.6.1 Basic Definition

A *watershed region* or *catchment basin* is defined as the region over which all points flow "downhill" to a common point. The idea originates from geology but has been applied to vision as well. In geology, one might want to consider the local region where all rainwater flows to a single lake or river. This might not seem to be applicable to intensity-based regions (why do we care about following intensity changes to a common dark point?) but it makes sense if we apply them to *gradient magnitude* images.

18.6.2 Watersheds of Gradient Magnitude

We talked before about how we'd like our edges to be locii of "maximal" gradient magnitude. One way of defining these maximal curves (ridges) is as the boundaries of watershed regions—everything on one side "flows" downhill to one side and everything on the other flows to the other side. Thus, as you cross from one watershed region to another, you've had to cross over some local ridge curve. Unfortunately, this definition of a ridge isn't based on purely local properties but instead requires building the regions first. But for segmentation, isn't that what we're after?

So, a common technique for segmentation is to use *gradient watershed regions*. First build a gradient magnitude image, then find the watershed regions in this image.

18.6.3 Basic Implementation

One way to find watersheds is to think of filling them from the bottom up and finding where different pools merge.

Suppose that the greylevel range is [0,255]. Start first with the 0 pixels. Clearly, nothing can be less than these, so these form the basis for new watersheds. Then add all pixels with intensity 1. If they are next to existing watersheds (0-intensity pixels), add them to these regions. Otherwise, start a new region for each pixel that is not next to an existing region.

This process repeats for each intensity k up to the maximum (255 in this example). Each k-intensity pixel that is next to an already-labeled watershed region is added to that region. Each k-intensity pixel that is *not* next to an already-labeled watershed region starts a new region.

One must be careful though: there may be multiple k-intensity pixels together, and if any one of these is adjacent to an already-labeled region, they all should be added.

This basic algorithm can be made faster by maintaining histogram-like data structures that index all pixels by their intensity. This makes it very fast to find all pixels with a particular intensity k.

18.6.4 Tobogganing

Another approach is to "slide" down a watershed to the common minimum. This approach links each pixel with the smallest of its neighbors. That one is linked to the smallest of its neighbors, that one to the smallest of its neighbors, etc. Eventually you get to points that have no smaller neighbors (local minima), which form the basis for a watershed region. The closure of all points that eventually link to the same minimum defines the watershed. This approach has been dubbed "tobogganing" because of its similarity to riding a sled down a mountainside.

18.7 Postprocessing

The major problem with region-based segmentation is that they usually undersegment (too few, too large regions) or oversegment the image (too many, too little regions). This is especially true for watersheds—even a little noise can make an otherwise homogeneous region "crinkle" into lots of small watersheds causing significant oversegmentation. Because of this, most implementations of region-based segmentation involve some form of postprocessing step that attempts to either split undersegmented regions or merge oversegmented ones.

The problem with this is that *the segmentation depends on what you are trying to do with the image*. If you are segmenting a tiger or a zebra, is it the stripes or the entire body that you're after? The only way of segmenting the image in any meaningful way is the one that considers images at these different levels.

This idea is similar to the one we talked about earlier for scale—indeed, some approaches to this problem do use multiscale segmentation. The idea is to build a *hierarchy* of regions that (hopefully!) are isolated regions at one level and parts of the same object at others.

Vocabulary

- Region growing
- Region splitting
- Region Merging
- Split and Merge Algorithm
- Watershed regions
- Tobogganing

Lecture 20: Segmentation (Matching, Advanced)

Contents

Reading

SH&B, 5.4–5.5

20.1 Matching

So far, we have talked about ways of segmenting general images (edges or regions) and for finding configurations of edges in parametrically-defined shapes (Hough transform). We'll now talk about segmenting where you know roughly what you're looking for in both intensity and configuration.

20.1.1 Overall Strategy

The overall strategy of template matching is easy: for every possible position, rotation, or other geometric transformation, compare each pixel's neighborhood to a template. After computing the strength of the match for each possibility, select the largest one, the largest n, or all that exceed some threshold.

20.1.2 Comparing Neighorhoods to Templates

Suppose that you already have a template for what you're looking for. One way of finding matches is *correlation*: shifting the template to each point in the image, point-wise multiplying each pixel in the neighborhood with the template, and adding the results. Essentially, this is the dot product of each neighborhood with the template. Remember that the dot product of two vectors is essentially the projection of one onto the other—the more they match, the larger the dot product.

There are other ways of measuring similarity of individual neighborhoods to a template, also based on vector-similarity metrics. The most common vector (or matrix) similarity metric is the L-norm of their difference. The L-norm of order p for a vector $\bar{v}$ is

$$\|\bar{v}\|_p = \left(\sum_i |\bar{v}_i|^p \right)^{1/p}$$

The most common forms of L-norms are the L_1 norm (sum of absolute values), the L_2 norm (square root of the sum of squares—i.e., Euclidean distance), and the L_∞ norm (maximum absolute value).

The C_2 metric (Eq. 5.39) in your text is the reciprocal of the L_1 norm of the difference between each neighborhood and the template. The C_3 metric (Eq. 5.40) in your text is the reciprocal of the L_2 norm squared. The C_1 metric (Eq. 5.39) in your text is the reciprocal of the L_∞ norm.

Picking the neighborhood with the largest match criterion (C_1, C_2, or C_3) is the same as choosing the neighborhood with whose respective norm (L_∞, L_1, or L_2) of the difference between the neighborhood and the template is the smallest.

20.1.3 Flexible Templates

Sometimes, what we're looking for might not be exactly the same in every image. One way to deal with this is to break the template into pieces and try to match each piece as if it was its own template. Position the entire template over the neighborhood, then search around the normal position of each subtemplate for the best match. The best combined match for all subtemplates gives the match for the overall template.

20.1.4 Control Strategies

Obviously, checking all possible transformations is computationally prohibitive in many cases. There are a number of ways of avoiding this, usually by only checking possible cases and then refining those that are "promising".

Hierarchical Matching (Pyramids)

One way to reduce the computational complexity of matching is to use the *pyramid* structure introduced in Lecture 3. By matching a coarser template to a coarser level of the pyramid, fewer comparisons must be performed. Once the strength of each coarser-resolution match is calculated, only those that exceed some threshold need to be evaluated/compared for the next-finer resolution. This process proceeds until the finest resolution (or at least a sufficiently fine resolution for the current task) is reached.

Iterative Refinement

Another way to perform matching is to use gradient-descent minimization techniques to iteratively tweak the transformation parameters until the best match is found. Unfortunately, these techniques require starting "near" the right solution. Still, this means that we don't have to check *all* possible transformations, just a sufficient number to allow us to find the matches through further refinement. For example, we might test rotations in 5- or 10-degree increments only. For each initial rotation, we then further refine the match by iteratively minimizing the difference from the template.

Chamfer Matching

Other approaches use distance maps to guide edge-based matches to corresponding edges. Remember that a *chamfer* is a map of the distance from each point to the nearest boundary. We might characterize the degree of mismatch for two curves (boundaries or other curve-based representations) by creating a distance map for one and then integrating the distance map for the first curve over all positions in the second curve. We can then iteratively adjust the position/orientation of the second curve until this integrated distance is minimized. This approach is known as *chamfer matching* and can be a very powerful technique.

Chamfer matching has also been used to match skeletal structures such as the medial axis.

20.2 Advanced Boundary Tracking

20.2.1 Tracking Both Sides of a Thin Object

Earlier, we discussed how to formulate boundary tracking as a graph-searching problem. This approach only tracks one boundary at a time. For some applications, we may want to simultaneously track both sides of a long, thin object (blood vessel, road, etc.). Simply tracking each side independently doesn't work when the object branches, crosses other edges, etc.

We can extend tracking of one contour to tracking two by constructing a graph that represents simultaneous movement along both edges. This can be done by constructing a separate graph for tracking each edge then "folding" one orthogonal to the other (like folding a piece of paper in half so that the two halves are perpendicular to each other) to create the basis for a three-dimensional graph. As we move from along the length of the shape, we simultaneously move side-to-side in each of the two other dimensions of the graph, which each correspond to a different side of the object. Your text does a nice job explaining a cost function that combines the strength of the edge on each side.

20.2.2 Surface Tracking

The concepts of boundary tracking can also be extended to surfaces in 3-d. The general form of this problem is actually quite hard: there isn't a two-dimensional surface analog to a one-dimensional minimum-cost path from one point to another. One can calculate the cost for a surface by adding all of the costs of the individual nodes, but what defines the "start" and "end" of the surface? Some approaches have tried to use 1-d ribbons or similar structures to "wrap" the surface. Other approaches use constrained forms of the problem.

One such constrained approach is described in your text. The idea is to construct the surface-search so that it goes from one side of a 3-d volume to the opposite side. As you progress through each slide through the volume, you can move in each of the other two dimensions by some predefined maximum transition, but always progressing from one face of the volume towards the opposite face. The problem can thus be set up as a dynamic-programming problem and implemented as such.

When we talked about using a top-to-bottom search graph earlier, we also talked about how you could wrap a resampled search grid around an already-approximated boundary. We can also do the same for surfaces. If the object's surface can be "unrolled" to a simple function of two search variables, we can find it using this surface-tracking technique.

Vocabulary

- Chamfer matching
- Surface tracking

Lecture 21: Image Understanding

Contents

Reading

SH&B, Chapter 8 (primarily 8.1–8.3)

The material in Section 8.4 is covered in more detail in CS 521.

21.1 Introduction

So far, we've looked at low-level processing (clean-up, edge or other feature detection, etc.), segmenting the image into regions that hopefully correspond to objects, and representing those objects using various representations. *Image understanding* is the process of actually interpreting those regions/objects to figure out what's actually happening in the image. This may include figuring out what the objects are, their spatial relationship to each other, etc. It may also include ultimately making some decision for further action.

21.2 Control Strategies

21.2.1 Bottom-Up

The process as we've just described it is *bottom-up*: it starts from the raw image data, works upward to object shape representation, and from there to a higher-level analysis or decision.

21.2.2 Top-Down

Image understanding can also work from the top down. Such processing makes hypotheses about that is happening, then uses the image data to validate/reject those hypotheses. Most of these approaches are *model-based*. That is, they have an approximate model of what they think they're looking at, then try to fit that model to the data. In a sense, primitive-fitting approaches such as the Hough transform used this idea. This idea can be extended further to so-called *deformable models*, in which you can deform the model to better fit the data. The goodness of the match is the inverse of how much you have to work to deform the model to fit the data.

21.2.3 Hybrid Hierarchical Control

Obviously, these two paradigms aren't mutually exclusive: one might use bottom-up processing to bring the information to something higher than the pixel level, then switch to top-down processing using this intermediate representation to validate or reject hypotheses.

21.2.4 Active Vision

A third alternative, neither quite bottom-up or top-down, presents itself when the vision system is part of a larger system capable of acting so as to be able to influence the position/view/etc. Such *active vision* approaches more accurately model the way people interact with the world. The idea is to make a tentative hypothesis (using either top-down or bottom-up processing) then ask yourself, "based on what I already know (and suspect), what do I need to do to be able to acquire the information that will best help me analyze the image or otherwise accomplish my task?"

21.3 Active Contours (Snakes)

The earliest and best known active contour approach is *snakes*: deformable splines that are acted upon by image, internal, and user-defined "forces" and deform to minimize the "energy" they exert in resisting these forces.

Your text has a good description of snakes, and you should also read the original paper (Kass, Witken, and Terzopolous in *CVPR'87*).

Notice that the general form of a snake follows the idea introduced earlier when we discussed graph-based approaches:

1. Establish the problem as the minimization of some cost function.
2. Use established optimization techniques to find the optimal (minimum cost) solution.

In the case of a snake, the cost function is the "energy" exerted by the snake in resisting the forces put upon it. The original formulation of this "energy" was

$$E_{\text{snake}} = w_{\text{int}} E_{\text{int}} + w_{\text{image}} E_{\text{image}} + w_{\text{con}} E_{\text{con}} \tag{21.1}$$

where each term is as follows:

E_{int}	Internal Energy	Keeps the snake from bending too much
E_{image}	Image Energy	Guides the snake along important image features
E_{con}	Constraint Energy	Pushes or pulls the snake away from or towards user-defined positions

The total energy for the snake is the integral of the energy at each point:

$$E^*_{\text{snake}} = \int_0^1 E_{\text{snake}}(s) \; ds \tag{21.2}$$

21.3.1 Internal Energy

The *internal energy* term tries to keep the snake smooth. Such smoothness constraints are also a common theme in computer vision, occurring in such approaches as

- Bayesian reconstruction
- Shape from shading
- Stereo correspondence
- and many others ...

The internal energy term in general keeps the model relatively close to its original *a priori* shape. In this case, we explicitly assume that the contour we want is generally smooth but otherwise unconstrained. Other models might start with an approximation of a brain, a heart, a kidney, a lung, a house, a chair, a person, etc.orm this model—in all cases, the internal energy term constrains the deformation.

One has to be careful with the weighting given to internal energy terms, though: too much weight means the model stays too "rigid" and the system "sees what it wants to see", too little weight means the model is too flexible and can pretty much match up to anything.

In the original snakes implementation, they used two terms to define the internal energy: one to keep the snake from stretching or contracting along its length (elasticity) and another to keep the snake from bending (curvature):

$$E_{\text{int}}(s) = \alpha(s)\left\|\tfrac{d\bar{v}}{ds}(s)\right\|^2 + \beta(s)\left\|\tfrac{d^2\bar{v}}{ds^2}(s)\right\|^2$$

Notice that both $\alpha(s)$ and $\beta(s)$ are functions of the arc length along the snake. This means that we can (perhaps interactively), keep the snake more rigid in some segments and more flexible in others.

21.3.2 Image Energy

The *image energy* term is what drives the model towards matching the image. It is usually inversely based on image intensity (bright curve finding), gradient magnitude (edge finding), or similar image features. Make sure to note the inverse relationship: strong features are *low* energy, and weak features (or no features) are *high* energy.

An interesting image energy term used in the original snakes paper also tracked *line terminations*. These are useful in analyzing visual illusions such as the Kinisa and Ehringhaus illusions illustrated in your text. This line termination term is the same level-set curvature measure that we learned about when we studied differential geometry (and which you expanded in HW 5).

21.3.3 Constraint Energy

Some systems, including the original snakes implementation, allowed for user interaction to guide the snakes, not only in initial placement but also in their energy terms. Such *constraint energy* can be used to interactively guide the snakes towards or away from particular features.

The original snakes paper used "springs" (attraction to specified points) and "volcanos" (repulsion from specified points).

21.3.4 Implementation

The simplest form of optimization is *gradient-descent minimization*. The idea is to find the minimum of a function f by iteratively taking a step "downhill".

For example, let's consider a function of only one variable. If we have a starting guess at the value of the solution, we can look at the slope at that point and decide to increment our solution (negative slope) or decrement our solution (positive slope). Notice the negation there: if the slope is positive, downhill is *backwards*; and if the slope is negative, downhill is *forwards*. We can thus implement gradient-descent minimization as

$$x_{t+1} = x_t - \gamma \frac{df}{dx}(x_t)$$

where γ controls the size of the step at each iteration.

For functions of two dimensions, the fastest direction downhill is the opposite of the fastest direction uphill (the gradient). This is basically just the same as doing gradient-descent minimization in each variable at the same time:

$$x_{t+1} = x_t - \gamma \frac{df}{dx}(x_t)$$

and

$$y_{t+1} = y_t - \gamma \frac{df}{dy}(y_t)$$

Or, more generally for any function of a vector $\overline{x}$:

$$\overline{x}_{t+1} = \overline{x}_t - \gamma \nabla f(\overline{x}_t)$$

Implementing such minimization in this form is actually quite simple:

```
grad = CalculateGradient(f,x);
while (magnitude(grad) > convergence_threshold) {
    x -= gamma * grad;
    grad = CalculateGradient(f,x);
}
```

The difficult part isn't implementing the minimization, it's differentiating the function you're trying to minimize.

In the case of Eq. 21.2, we can approximate it using a number of discrete points on the snake $\bar{v}_i = (x_i, y_i)$:

$$E^*_{\text{snake}} \approx \sum_1^n E_{\text{snake}}(\bar{v}_i)$$

what's nice about this is that the derivative of a sum is the sum of the derivatives, so

$$\begin{aligned} \nabla E^*_{\text{snake}} &\approx \nabla \left[\sum_1^n E_{\text{snake}}(\bar{v}_i) \right] \\ &= \sum_1^n \nabla E_{\text{snake}}(\bar{v}_i) \end{aligned}$$

We can thus think of the problem as iteratively adjusting each of the points $\bar{v}_i$ using its own gradient-descent minimization:

$$\bar{v}_i \leftarrow \bar{v}_i - \nabla E_{\text{snake}}(\bar{v}_i)$$

Using Eq. 21.1, we get

$$\begin{aligned} \nabla E_{\text{snake}}(\bar{v}_i) &= \nabla \left[w_{\text{int}} E_{\text{int}}(\bar{v}_i) + w_{\text{image}} E_{\text{image}}(\bar{v}_i) + w_{\text{con}} E_{\text{con}}(\bar{v}_i) \right] \\ &= w_{\text{int}} \nabla E_{\text{int}}(\bar{v}_i) + w_{\text{image}} \nabla E_{\text{image}}(\bar{v}_i) + w_{\text{con}} \nabla E_{\text{con}}(\bar{v}_i) \end{aligned}$$

Notice that $w_{\text{image}} \nabla E_{\text{image}}(\bar{v}_i) + w_{\text{con}} \nabla E_{\text{con}}(\bar{v}_i)$ depends only on the image, not on the relationship of the snake to any other part of itself, so we can precalculate this for every point in the image. Simply calculate $w_{\text{image}} E_{\text{image}} + w_{\text{con}} E_{\text{con}}$ everywhere, then measure its derivatives locally. Let's call this ∇E_{ext}:

$$\nabla E_{\text{ext}} = w_{\text{image}} \nabla E_{\text{image}} + w_{\text{con}} \nabla E_{\text{con}}$$

Substituting, our minimization now becomes

$$\bar{v}_i \leftarrow \bar{v}_i - \gamma \left[w_{\text{int}} \nabla E_{\text{int}}(\bar{v}_i) + \nabla E_{\text{ext}}(\bar{v}_i) \right]$$

So, the only thing left to do is to solve for the gradient of the internal energy. Unfortunately, this is rather complicated since it's a function of the spline itself, not the image. Fortunately, it simplifies considerably if $\alpha(s)$ and $\beta(s)$ are constant—Kass, Witkin, and Terzopolous published the following:

$$\begin{aligned} \nabla E_{\text{int}}(s) &= \nabla \left[\alpha \left\| \tfrac{d\bar{v}}{ds}(s) \right\|^2 + \beta \left\| \tfrac{d^2\bar{v}}{ds^2}(s) \right\|^2 \right] \\ &= \left[\alpha \nabla \left\| \tfrac{d\bar{v}}{ds}(s) \right\|^2 + \beta \nabla \left\| \tfrac{d^2\bar{v}}{ds^2}(s) \right\|^2 \right] \\ &= \alpha \frac{\partial^2 \bar{v}}{\partial s^2} + \beta \frac{\partial^4 \bar{v}}{\partial s^4} \end{aligned}$$

These can be approximated using finite differences—the second derivative w.r.t. s can be calculated using three adjacent points on the snake, and the fourth derivative w.r.t. s can be calculated using five adjacent points. It also helps to separate the x and y components.

Putting it all together:

$$\bar{v}_i \leftarrow \bar{v}_i - \gamma \left\{ w_{\text{int}} \left[\alpha \frac{\partial^2 \bar{v}}{\partial s^2}(\bar{v}_i) + \beta \frac{\partial^4 \bar{v}}{\partial s^4}(\bar{v}_i) \right] + \nabla E_{\text{ext}}(\bar{v}_i) \right\}$$

or more simply:

$$x_i \leftarrow x_i - \gamma \left\{ w_{\text{int}} \left[\alpha \frac{\partial^2 x}{\partial s^2}(\bar{v}_i) + \beta \frac{\partial^4 x}{\partial s^4}(\bar{v}_i) \right] + \frac{\partial}{\partial x} E_{\text{ext}}(\bar{v}_i) \right\}$$

and

$$y_i \leftarrow y_i - \gamma \left\{ w_{\text{int}} \left[\alpha \frac{\partial^2 y}{\partial s^2}(\bar{v}_i) + \beta \frac{\partial^4 y}{\partial s^4}(\bar{v}_i) \right] + \frac{\partial}{\partial y} E_{\text{ext}}(\bar{v}_i) \right\}$$

Got all that? Before you start the iteration, precalculate $E_{\text{ext}}(\bar{v}_i)$ and calculate the derivatives of this w.r.t. x and y separately. When you're ready to start the iteration, calculate at each point $\frac{\partial^2 x}{\partial s^2}(\bar{v}_i)$ and $\frac{\partial^2 y}{\partial s^2}(\bar{v}_i)$ using three adjacent points and $\frac{\partial^4 x}{\partial s^4}(\bar{v}_i)$ and $\frac{\partial^4 y}{\partial s^4}(\bar{v}_i)$ using five adjacent points. Then calculate the incremental change in the x and y components of each point v_i. You then have to *recalculate* the derivatives of the internal energy on each iteration. For the external energy term, you can simply read from the precalculated images for the derivatives of $E_{\text{ext}}(\bar{v}_i)$ using the new positions of each v_i.

21.4 Point-Density Models

Point density models can be thought of as deformable models in which the deformation "energy" is based on statistical properties of a large number of training examples. This has the powerful advantage of *allowing deformation where the objects themselves normally differ while remaining more rigid where the objects themselves are normally consistent*.

First identify a number of key *landmark* points for the object. These need not be on the contour but can be *any* identifiable points.

Now, gather a collection of sample images with varying shapes that you want to recognize. (For example, if you're building a system to recognize and analyze brains, get a collection of sample brains; if you're building a system to recognize different kinds of fish, get yourself samples of each kind of fish; etc.) For each image in your training set, find the landmarks and store their locations. Now, you need to register these images by transforming (translating, rotating) the landmark points so that they are all registered relative to a common *mean shape*.

We can now measure the covariance matrix for all of the landmarks across all of the training shapes. This tells us not only the consistency of each landmark but *the way each landmark tends to move as the others move*.

The covariance matrix can be further analyzed by computing its eigenvalues and eigenvectors (called *principal component analysis*). These eigenvectors are called the *modes of variation* of the shapes, and the eigenvalues tell you the rigidity or flexibility along these modes. Notice that a mode is *not* a single direction—it's an overall change in all of the landmarks relative to each other.

Point-density or landmark-based models, though they can be computationally expensive, are among the most powerful models for shape description and deformation currently in use.

Vocabulary

- Bottom-up processing
- Top-down processing
- Active vision
- Active contours / Snakes
- Gradient-descent minimization

- Point-density models
- Landmarks

Lecture 22: Texture

Contents

Reading

SH&B, Chapter 14
Castleman, 19.4

22.1 Introduction

In previous lectures we've talked about how to describe the shape of a region. In this lecture, we'll be talking about how to describe the content of the region itself.

22.2 Intensity Descriptors

The easiest descriptors for the contents of a region describe the general intensity properties: average grey-level, medial grey-level, minimum and maximum grey-level, etc.

These descriptors are, however, subject to intensity gain or other parameters of the imaging system.

22.3 Overview of Texture

Texture is one of those words that we all know but have a hard time defining. When we see two different textures, we can clearly recognize their similarities or differences, but we may have a hard time verbalizing them.

There are three main ways textures are used:

1. To discriminate between different (already segmented) regions or to classify them,
2. To produce descriptions so that we can reproduce textures, and
3. To segment an image based on textures.

In this lecture we'll mainly discuss ways of describing textures, each of which can be applied to all three of these tasks.

There are three common ways of analyzing texture:

1. Statistical Approaches
2. Structural Approaches
3. Spectral Approaches

22.4 Statistical Approaches

Since textures may be random, but with certain consistent properties, one obvious way to describe such textures is through their statistical properties.

22.4.1 Moments of Intensity

We discussed earlier the concept of *statistical moments* and used them to describe shape.

These can also be used to describe the texture in a region. Suppose that we construct the histogram of the intensities in a region. We can then compute moments of the 1-D histogram:

- The first moment is the mean intensity, which we just discussed.
- The second central moment is the variance, which describes how similar the intensities are within the region.
- The third central moment, skew, described how symmetric the intensity distribution is about the mean.
- The fourth central moment, kirtosis, describes how flat the distribution is.

The moments beyond this are harder to describe intuitively, but they can also describe the texture.

22.4.2 Grey-level Co-occurrence

Another statistical way to describe shape is by statistically sampling the way certain grey-levels occur in relation to other grey-levels.

For a position operator p, we can define a matrix P_{ij} that counts the number of times a pixel with grey-level i occurs at position p from a pixel with grey-level j.

For example, if we have three distinct grey-levels 0, 1, and 2, and the position operator p is "lower right", the counts matrix P of the image

$$\begin{array}{ccccc} 0 & 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 2 & 1 & 0 & 0 \\ 1 & 1 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{array} \tag{22.1}$$

is

$$P = \begin{bmatrix} 4 & 2 & 1 \\ 2 & 3 & 2 \\ 0 & 2 & 0 \end{bmatrix} \tag{22.2}$$

If we normalize the matrix P by the total number of pixels so that each element is between 0 and 1, we get a *grey-level co-occurrence matrix* C.

Note: Different authors define the co-occurence matrix a little differently in two ways:

- by defining the relationship operator p by an angle θ and distance d, and
- by ignoring the direction of the position operator and considering only the (bidirectional) relative relationship.

This second way of defining the co-occurrence matrix makes all such matrices symmetric. So, if $P_{\text{left}} = P_{\text{right}}^T$, $P_{\text{horizontal}} = P_{\text{left}} + P_{\text{right}}$.

We can get various descriptors from C by measuring various properties, including the following:

1. the maximum element of C
$$\max_{ij}(c_{ij}) \tag{22.3}$$

2. the element difference moment of order k
$$\sum_i \sum_j c_{ij}(i-j)^k \tag{22.4}$$

3. the inverse element difference moment of order k
$$\sum_i \sum_j c_{ij}/(i-j)^k \tag{22.5}$$

4. entropy
$$-\sum_i \sum_j c_{ij} \log c_{ij} \tag{22.6}$$

5. uniformity
$$\sum_i \sum_j c_{ij}^2 \tag{22.7}$$

Other similar measures are described in your text.

22.5 Structural Approaches

A second way of defining the texture in a region is to define a grammar for the way that the pattern of the texture produces structure.

Because as CS students you should be familiar with grammars by now, we won't go into great detail here. The basic scheme is to build a grammar for the texture and then parse the texture to see if it matches the grammar. The idea can be extended by defining *texture primitives*, simple patterns from which more complicated ones can be built. The parse tree for a the pattern in a particular region can be used as a descriptor.

22.6 Spectral Approaches

22.6.1 Collapsed Frequency Domains

A third way to analyze texture is in the frequency domain. If textures are periodic patterns, doesn't it make sense to analyze them with periodic functions?

The entire frequency domain is, however, as much information as the image itself. We can condense the information by collapsing a particular frequency across all orientations (by integrating around circles of fixed distance from the frequency origin) or by collapsing all frequencies in a particular orientation (by integrating along each of a unique orientation though the origin). If we express the frequency-domain coordinates in polar coordinates, these are

$$S(r) = \sum_{\theta=0}^{\pi} S(r,\theta) \tag{22.8}$$

and

$$S(\theta) = \sum_{r=0}^{N/2} S(r,\theta) \tag{22.9}$$

$S(r)$ tells us the distribution of high and low frequencies across all angles. $S(\theta)$ tells us the distribution of frequency content in specific directions. These two one-dimensional descriptors can be useful for discriminating textures.

22.6.2 Local Frequency Content

As we discussed earlier, the frequency domain contains information from all parts of the image. This makes the previous method useful for global texture analysis, but not local. If a region has already been segmented, you could pad the region with its average intensity to create a rectangular image. This doesn't, however, provide a useful way of using texture to do the segmentation.

We can define local frequency content by using some form of co-joint spatial-frequency representation. As we discussed, though, this only partially localizes the position or frequency of the information—you can't do both perfectly.

A simple way to do this would be to examine the $N \times N$ neighborhood around a point and to compute the Fourier Transform of that $N \times N$ subimage. As you move from one textured region to another, the frequency content of the window changes. Differences in the frequency content of each window could then be used as a means of segmentation.

Of course, one still needs to distill descriptors from the frequency content of each window. One such descriptor (Coggins, 1985) is to compute the total energy (squared frequency content) of the window. If you exclude the zero-frequency term, this is invariant to the average intensity. If you normalize by the zero-frequency term, it is invariant to intensity gain as well. There are, of course, other descriptors you could use as well.

22.7 Moments

We talked earlier about how moments of an intensity histogram can be used to describe a region.

We can go one step further by describing the combination of both intensity and pattern of intensity by computing moments of the two-dimensional grey-level function itself. Remember that we said that if you had enough moments, you could reconstruct the function itself? Well, this means that we can reconstruct an entire image from its moments, much like we could reconstruct the image from the transforms we've discussed. As we've done before, though, we don't really need all of the moments to be able to get good matching criteria.

However, moments themselves aren't invariant to all of the transformations we've discussed. Certain combinations of the moments can be constructed so as to be invariant to rotation, translation, scaling, and mirroring. These combinations can be used as descriptors for matching images.

Vocabulary

- Texture
- Grey-level Co-occurrence Matrix
- Spectral Energy