Lecture 7: Shape Description (Contours)

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

Contents

7.1 Introduction . . . 2
7.2 Region Identification . . . 2
7.3 Invariance . . . 2
7.4 Encoding Boundary Curves Sequentially . . . 2
7.4.1 Basic Chain Codes . . . 2
7.4.2 Differential Chain Codes . . . 3
7.4.3 Shape Numbers . . . 3
7.4.4 Smoothing Chain Codes . . . 3
7.4.5 Resampling Chain Codes . . . 3
7.4.6 ψ-s Curves . . . 3
7.4.7 R-S Curves . . . 4
7.5 Polygonal Approximation . . . 4
7.5.1 Merging Methods . . . 4
7.5.2 Splitting Methods . . . 4
7.6 Other Geometric Representations . . . 4
7.6.1 Boundary Length . . . 4
7.6.2 Curvature . . . 5
7.6.3 Bending Energy . . . 5
7.6.4 Signatures . . . 5
7.6.5 Chord Distribution . . . 5
7.6.6 Convex Hulls . . . 5
7.6.7 Points of Extreme Curvature . . . 5
7.7 Fourier Descriptors . . . 5
7.7.1 Definition . . . 5
7.7.2 Properties . . . 6
7.7.3 Response to Transformations . . . 6
7.7.4 Higher-Dimensional Descriptors . . . 7
7.8 Shape Invariants . . . 7
7.8.1 Cross Ratio . . . 7
7.8.2 Systems of Lines . . . 7
7.8.3 Conics . . . 7
7.9 Conclusion . . . 8

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