IDENTIFICATION AND ANALYSIS OF CURVES IN DIGITAL IMAGES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Gideon Avrahami
August 1993

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Vaughan Pratt
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jean-Claude Latombe

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Brian Wandell

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

We address the problem of formally describing well-defined contours in clean images. As a case study, we consider this problem for characters from various fonts, which may only be available as ink-on-paper or photographic plates. Representing the contours of these characters as splines will enable us to incorporate these fonts in a Digital Printing System (DPS).

The process of capturing the images and making them usable in a Digital Printing System includes several steps. We begin with applying an Edge Detection algorithm that uses the assumptions about smoothness and cleanliness of the image to produce points at sub-pixel resolution. We then describe each contour by piecewise conics, by applying a segmentation procedure that optimizes with respect to the quality of the fit, tangent continuity at the segmentation points, and the number of segments. Finally, the result can be transformed to any desired DPS, such as the PostScript graphic engine.

This approach yields a system that compares favorably with the available tools for font reproduction, in terms of robustness, accuracy, speed and quality. It has been applied successfully to professional typographic sheets, and hand-drawn characters, in Latin and Asian alphabets.

Acknowledgements

First of all, I must thank my advisor, professor Vaughan Pratt, for his guidance, support and encouragement over the last four years.

The teachers, fellow students and friends at Stanford helped make my years here an educational, rewarding and pleasant experience. In particular, professor Chuck Bigelow, who taught me most of what I know about text, fonts, typography and calligraphy. Professors Brian Wandell and Jean-Claude Latombe served as my reading committee, and provided me with useful ideas and comments.

Sun Microsystems, Inc., provided hardware and financial support during the first years of my reserach.

And finally, my gratitude and love to all those who believed that I could do it, even when I myself wasn't so sure...

Introduction

In this work we address the problem of formally describing well-defined contours in clean images. Our main case study is the problem of incorporating characters of various fonts into a Digital Printing System (DPS).

Many fonts are available only in non-digital form: ink on paper, photographic plates, etc. In order to incorporate such fonts in a DPS, we would need to digitize the images, and convert them to a representation that can be handled easily by a DPS.

It is quite clear that a bitmap representation is far from adequate for practical purposes. Rather, a mathematical description of the outline of each character is the far more sensible representation. Such a description may not be more succinct than a small-scale bitmap of the character; but if the same font is to be used at many different scales, including very large ones, then the outline representation makes a lot of sense. The character can then be scaled by any desired factor, and the appropriate bitmap is created on demand. Rubinstein [R88] discusses this and other aspects of font representation in detail.

Figure 0.1 shows two characters from a font called *Diploma*. The font was found in a book of calligraphic styles; in order to use it in a DPS, the pages of the book were scanned, and the digital image was converted to outline form. It is in the resolution used here, each character occupies roughly 100 by 100 pixels, while the two outlines are described by 60 and 23 curves, respectively. When considering that the outline description can be used to generate a bitmap at any required scale, the advantages of this representation become apparent.

Font outlines are usually specified by a combination of straight lines and curves of a higher degree. Circular arcs are usually the first choice, but they have too few

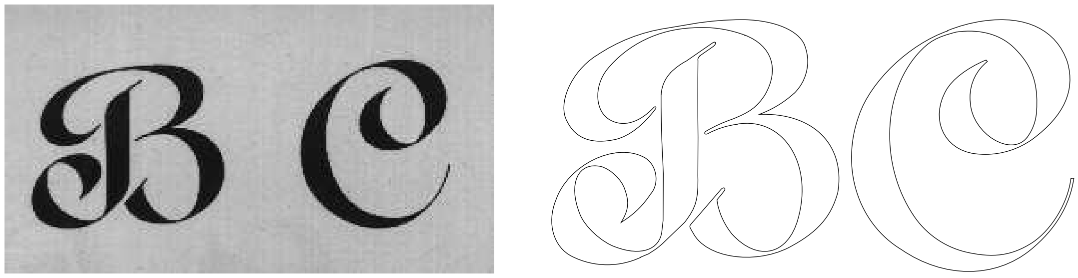

Figure 0.1: two characters from the *Diploma* font, as a scanned image and in outline form.

degrees of freedom and cannot even provide tangent continuity. The more common choice, in systems such as PostScript[1] [Ad86], is cubic splines. However, as noted by Pratt [P85], the increased complexity in the family of curves results in expensive rendering algorithms. Moreover, fitting cubics to a given set of data points — in our case, samples from the contour of a character — is a cumbersome process (Plass and Stone [PS83]).

We therefore follow Pratt's work in the use of conic sections, i.e., segments of ellipses, hyperbolas and parabolas, as well as circles and straight lines. Using conics enables us to fit the data by direct application of least squares methods, which are computationally simple and well-understood. Conics are also easy to analyze, making the discovery of curvature extrema quite simple. And unlike cubics, conic curves cannot have inflection points or self-loops, so the result of curve fitting need not be tested for such anomalies.

When creating a piecewise fit for each contour in the outlines, we shall be concerned with the smoothness of the final shape. In this work we require tangent continuity, except in places where we detect a corner. The early detection and interpretation of corners is an important part in the fitting process, since our goal is not only to obtain a good fit, but also one that approximates the intentions of a human designer.

Another requirement is that the description should use as few segments as possible.

[1]PostScript is a trademark of Adobe Systems Inc.

Again, this is not only out of desire to compress the data (though that is an important goal in itself) but also because of the assumption that this is the correct way to present the designs.

In the following chapters we describe a suite of algorithms that were implemented and applied to the task of capturing an image and making it available for use as a digital font.

Our edge detection procedure takes advantage of the inherent simplicity and smoothness in images of symbols. The result is an edge output with sub-pixel precision, effectively increasing the resolution of the scan by a factor of up to 5.

The curve fitting and contour segmentation algorithm creates a representation of each outline in the image in terms of conic sections. This representation is optimized with respect to the number of segments and the quality of the fit, and ensures tangent continuity, wherever the image itself has no obvious corners.

Finally, we describe the implementation of these algorithms and present some detailed examples of the use of the system. We analyze the performance on different sets of data, and suggest directions for further research.

Contents

Chapter 1

Sub-Pixel Edge Detection

Most often, the first task in analyzing a digital image is edge detection. General purpose edge detection has been covered by extensive research. The work of Canny [C83] and those who followed him (such as Deriche [D88]) give an answer to optimal edge detection on general images that come from various sources and contain all kind of noise. In general, these methods will produce a matrix of the same size as the input image, and denote which image pixels should be considered as parts of an edge. We may think of such an output as a list of edge points, located at the centers of selected pixels. In the frame of the image matrix, these points all have integer coordinates.

The subject of this chapter is to suggest a specialized method for increasing the accuracy of an edge detection algorithm by producing edge points with fractional coordinates. This can be thought of as increasing the resolution of the image. In human vision, this phenomenon is known as hyper-acuity: the ability to detect slight angles and non-linearities that improve on the physical resolution of visual cells in the eye. (See, for example, [W72]).

To do so, we must first think of a discrete image as samples of a continuous intensity function. In this function, a white region is a plateau at the maximum-height, a black region is a minimum-height valley, and the edges are the sloped walls around each plateau. The steepness of the walls reflects the gradual transition in the image from black to white.

The discrete samples will be mostly black and white, but some of them, taken at or near the edges, have intermediary values. To obtain sub-pixel locations for the edge, these gray values must be analyzed.

The analytic approach, shared by many vision researchers, calls for an attempt to reconstruct the intensity surface by appropriate convolution transforms; then, these methods report an edge at the locations where that surface has a big enough slope. For example, Nalwa [N87] reconstructs the intensity function on a grid with smaller spacing, so his sub-pixel resolution becomes the resolution of the new grid: typically, one half.

Such general methods must be prepared to handle images as diverse as any possible real-life scene. The intensity levels in any given feature and around it may be at any shade of gray. The steepness of the edge and the smoothness of the contours can vary from image to image and from feature to feature inside each image.

In our work, however, we expect to encounter a special class of input images. The images will be essentially black and white; the edges, which separate the black art from its white background, are well-defined and smooth. The transitions from black to white are sharp, and the edge contours are piecewise smooth curves, with only few isolated sharp corners.

Thus, we are led to a more "combinatorial" and less "analytic" approach. Instead of applying a convolution filter to the image, we proceed by following each edge as it winds around the image, and calculating edge points along the way. So instead of obtaining a region-oriented matrix, where ones denote edge points and zeros denote the absence of an edge, we produce a sequential list of edge points, each calculated with sub-pixel precision. Since the edge is produced by following the contour in, say, a clockwise direction, the list of edge points is already in a meaningful order (unlike a matrix output).

The tracking algorithm advances along the edge by applying the following simple rule: keep track of two neighboring pixels, a light and a dark, such that an edge must pass between the two. In order to do so, we only need to decide on a threshold level of gray. We then define any pixel whose gray level is higher than the threshold as "light," and any pixel with a lower level as "dark."

Clearly, using such a threshold is a valid way to locate contours in any image.

In arbitrary images, this method is too simplistic, and will result in ignoring some features and mishandling others. But for the high-contrast images with which we deal here, this criterion is good enough to capture all the important contours.

In the following sections, we shall first discuss the method for calculating a sub-pixel edge location between such a pair of light and dark neighbors. Then we shall discuss the accuracy of the method, and conclude with some notes on the edge-tracking mechanism.

1.1 Gray Level And Edge Distance

We begin by providing an estimate of the distance of an edge from the center of a pixel, based on the gray level of the pixel.

Assume initially that the pixel has a circular shape, the edge is a straight line, and the image is all black on one side of the edge, and all white on the other. (See Figure 1.1). The gray level of the pixel will be the fraction of the circle covered by white.

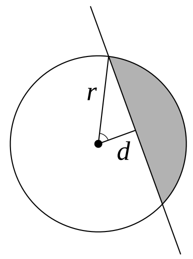

Figure 1.1: A straight edge cutting through a circular pixel.

The area of the region between the straight line and the circle's arc is given by the following formula, where r is the radius, and d is the distance from the edge to the center of the pixel:

$$\frac{\arccos(\frac{d}{r}) - \frac{d}{r}\sqrt{1-(\frac{d}{r})^2}}{4}.$$

If the blackened portion of the pixel contains the center, this formula describes the white portion. As we note below, this yields an expression for the gray level that is symmetrical around the case $d = 0$.

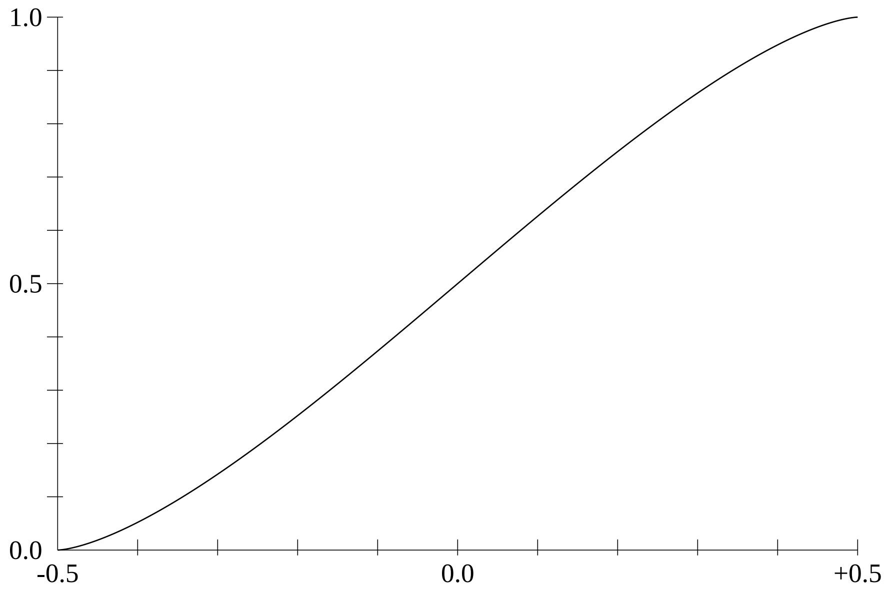

Figure 1.2: The function $G(d)$.

The area of the circle is, of course, πr^2. We will assume here that the diameter of the circle is the same as the inter-pixel distance, which is simply 1 in the units of image coordinates. (A pixel's effective diameter may be larger than 1; but as we shall discuss in the next section, this turns out not to be a crucial problem.)

So we assume $r = \frac{1}{2}$, and the area of the circle is $\frac{\pi}{4}$. The fraction of the darkened area in Figure 1.1 then becomes:

$$A(d) = \frac{\arccos(2d) - 2d\sqrt{1-4d^2}}{\pi}.$$

The gray level of the pixel is then the complement of this: $G(d) = 1 - A(d)$. Indeed, when $d = 0$, exactly half the circle is black, and the other half white, so $G(0) = \frac{1}{2}$. And when $d = r = \frac{1}{2}$, the line is just tangent to the circle and the black region does not intersect the circle, so $G(\frac{1}{2}) = 1$.

As noted above, when the roles of black and white in Figure 1.1 are reversed, the gray level is $A(d)$. To avoid confusion, we may denote the distance in this case by negative values of d; we can then see that $G(-\frac{1}{2}) = A(\frac{1}{2}) = 0$. The function $G(d)$ can be seen in Figure 1.2. The function is mostly linear in the middle of the range, but is gets flatter near the ends.

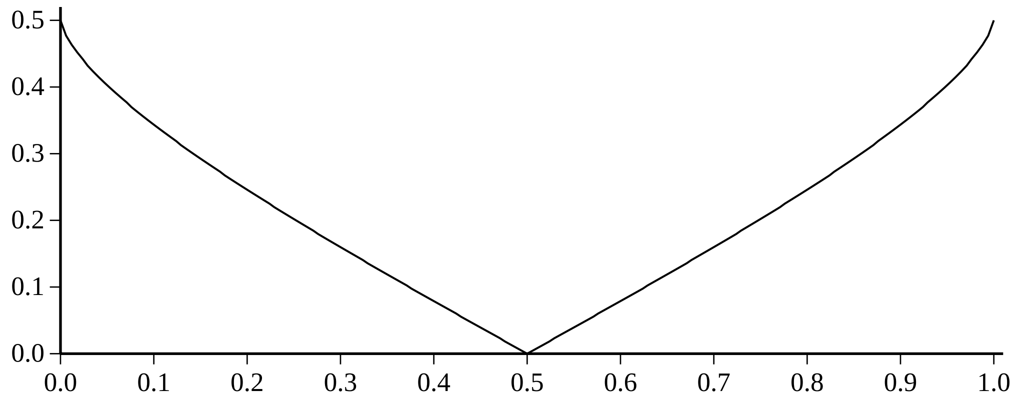

Figure 1.3: The inverse of $G(d)$.

Now we need to find the inverse of $G(d)$ (or more conveniently, the absolute value of this inverse — the negative distances were just a convenience, after all.) While this can't be done analytically, this inverse can be easily tabulated numerically. The size of the table depends on the range of gray values that we encounter in the images, and that is simply the integers from 0 to 255. (Figure 1.3.)

(However, the dynamic range of every image may be different — that is, the minimum and maximum values are not necessarily 0 and 255. Therefore, the inverse function's domain will have to be shifted a little, depending on the image. This also will be addressed later in this section.)

It is important to remember that this function was developed from very simple assumptions about the shape of the pixel and the way the intensity function is integrated over the pixel's area. In particular, we didn't assume any convolution function that might make the area near the center weigh more than the areas further away.

However, before studying more complex models, let us look at some experimental data. Figure 1.4 shows a few hundred samples taken from a scanned image of a straight line. The image was actually a black magnetic tape glued on white paper;

thus, the edge had excellent contrast, and consisted of a very straight line. Several images were produced, by scanning this tape at various angles.

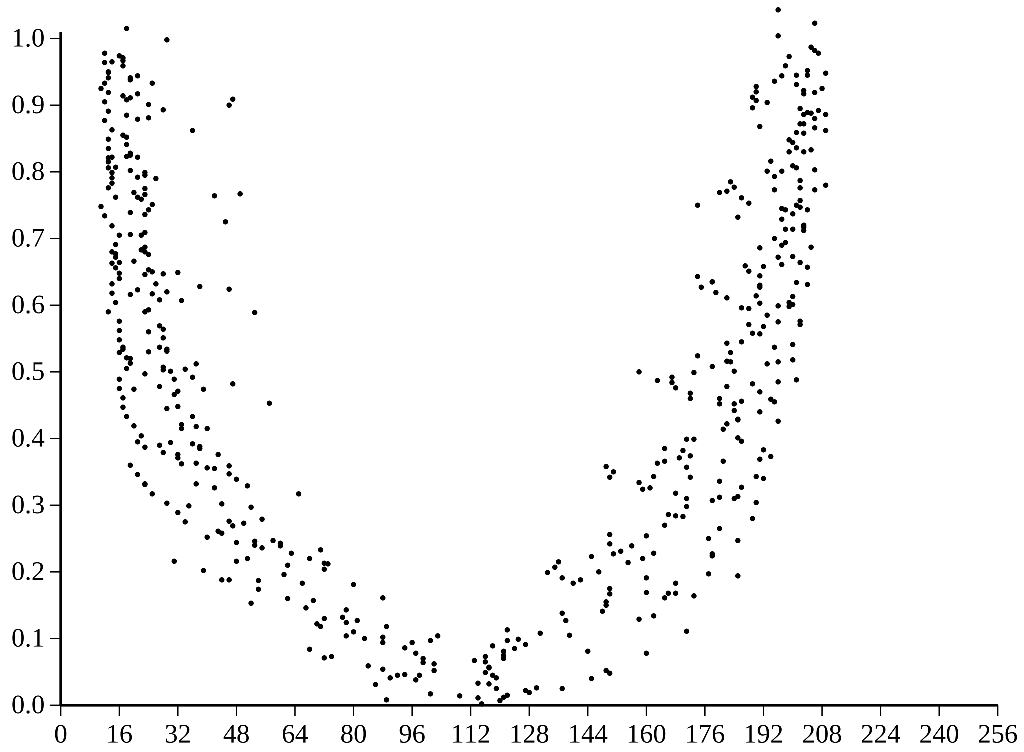

Figure 1.4: Gray levels, and distances from a straight edge.

To produce this data, a basic edge tracking algorithm was applied to the image, producing a list of integer edge points. Then, the least squares straight line was calculated for these points. The result was an accurate algebraic formula that describes the straight line edge segment for each subset of points. The subsets varied in size, having usually between 80 and 200 points each.

This allowed us to calculate, for every pixel, an accurate value of its distance from the edge — that is, from the least-squares straight line that describes that edge. The result is shown in Figure 1.4, where each pixel is plotted by its gray level and calculated distance from the edge line.

This experiment was repeated with straight lines at various angles, with the same result. Two things are noted right away: first, the noise in the samples creates an error of more than 16 out of 256 gray levels, reducing the effective number of gray level bits from 8 to only 4 or 3. Secondly, the underlying function appears to have an almost singular behavior at both ends of the gray range — i.e., at gray values that correspond to the lightest white and darkest black in the image. Such gray values can correspond to distances of more than half a pixel, since indeed they represent parts of the image that are quite far from any edge.

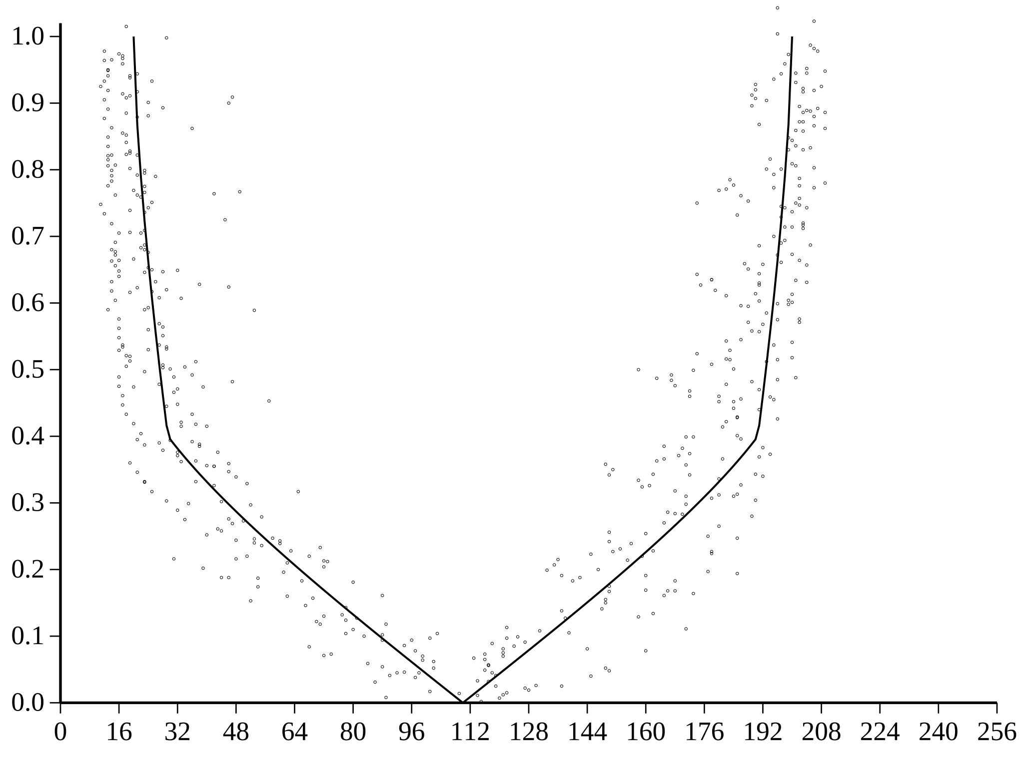

Figure 1.5: The constructed gray-to-distance function.

Combining the experimental data with the basic theory, we can now construct a function that will give us a good estimate of the appropriate distance for each gray level. (Figure 1.5). The function is kept in a look-up table and combines an almost-linear through most of the dynamic range (gray levels 32 to 192, in the figure) with a linear correction that approximates the near-singularities at both ends of that range. These corrections allow the values of the function extend up to a distance of 1.0 instead of just 0.5. In practice, this tends to have little effect, since not many pixels near the edge will have these extreme values.

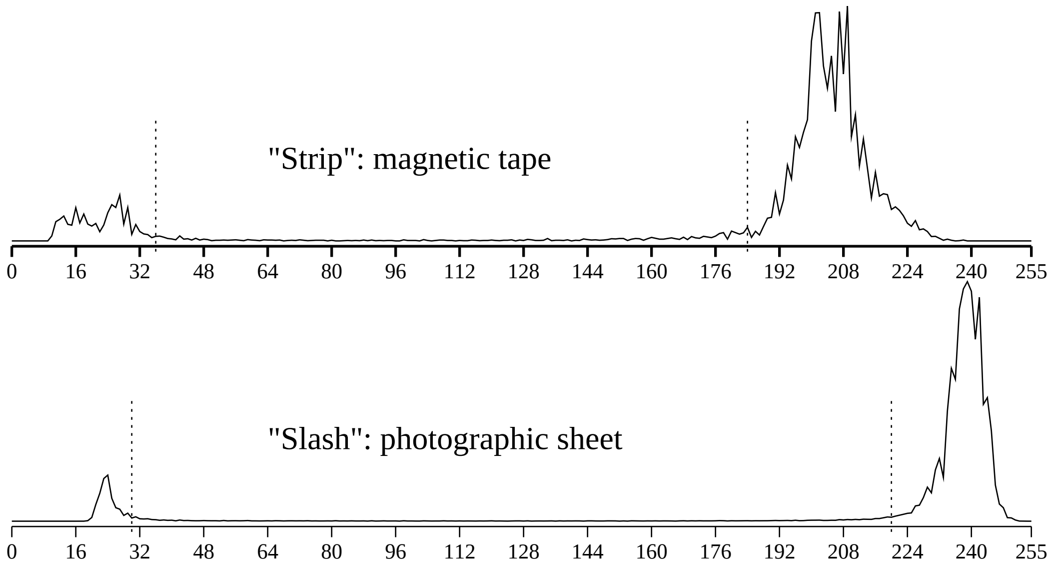

Figure 1.6: Gray level histograms for two types of images.

There is one more adjustment that needs to be made, usually for each image or class of images, namely, the dynamic range. As mentioned above, the minimum and maximum gray levels that occur in the image (we can think of them as the "real" values of black and white) are usually not 0 and 255, but closer to the middle. They depend on the exact quality and shade of the white paper and black ink, as well as on the scanner itself.

We solve this problem by analyzing the histogram of a sample image and looking for typical positions at the clusters of light and dark values. A good choice seems to be about halfway down the last slope as we move towards the middle of the histogram. (See Figure 1.6.) This choice needs only to be made once, at the start of work on a set of images scanned from a new source.

The difference in edge statistics can be seen in Figure 1.7, which presents the same kind of data as Figure 1.4, but from a different source. This time it was a photographic plate with samples of a professional font, and the edge was sampled at the long straight line in the contour of the "slash" symbol (/).

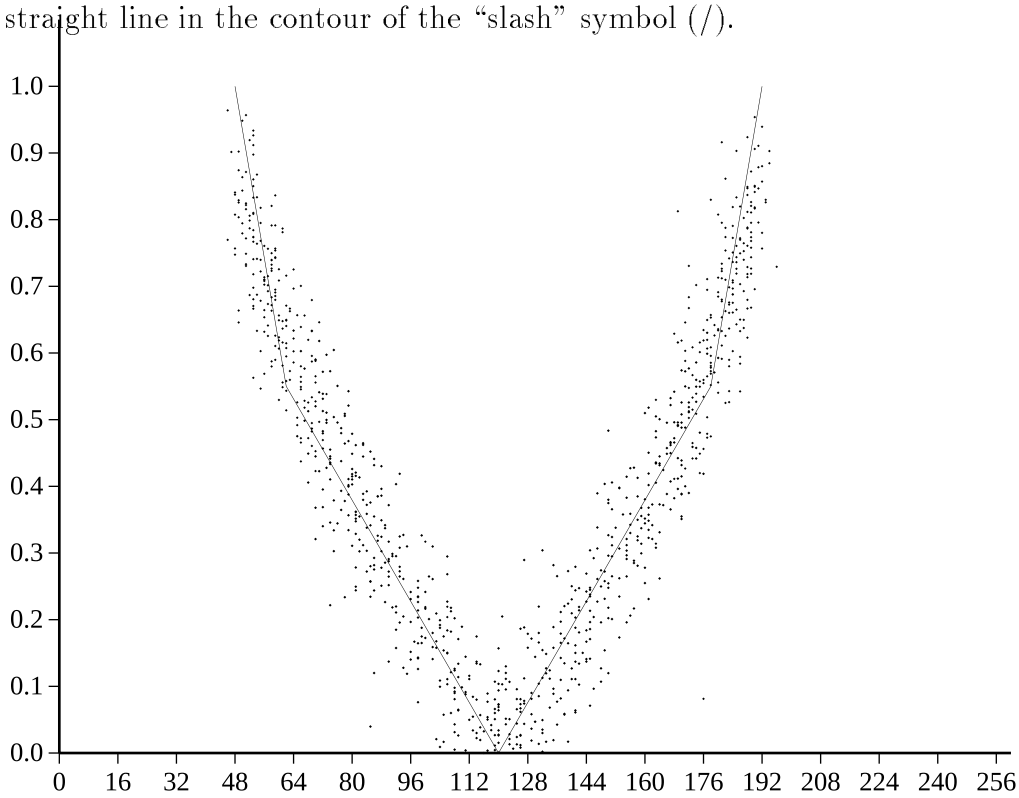

Figure 1.7: Distance/gray function in a photographic image.

In this set of samples, the dynamic range is roughly [48..192], compared to [16..200] in the "magnetic tape" example of Figure 1.4. The distance-to-gray function has the same behavior, parametrized by the new range.

1.2 The Edge Detection Algorithm

Let us see now how the algorithm calculates a sub-pixel value for an edge point. Recall that the algorithm will have a pair of neighboring pixels, one of which is lighter than the halfway gray, and one darker. Having two such neighbors is not only good for tracking the edge, but also enables us to calculate the edge point!

The inverse function of the previous section allows us to estimate the distance of the edge from the center of a given pixel — but clearly not the direction. So in order to determine a location, we need at least two pixels and two distance estimates. This is exactly what we get when we have a pair of pixels on both sides of the edge.

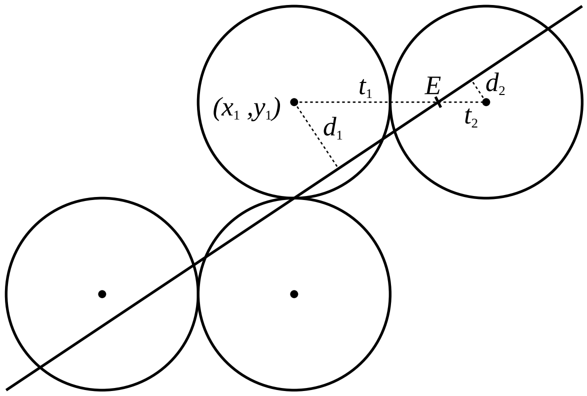

Figure 1.8: Calculating the location of the edge point E.

Figure 1.8 describes this situation. The edge is assumed to pass somewhere between the two pixels — this is an easy assumption, since one of them is darker than the halfway value, and the other lighter. Applying the gray-to-distance function to the gray levels of these pixels, we obtain the two distances, d_1 and d_2. Now we can calculate the location of the point E, which is the intersection of the edge and the line that connects the centers of the two pixels.

By triangle similarity, the distance of E from the center of the left pixel is:

$$t_1 = \frac{d_1}{d_1 + d_2}.$$

The coordinates of the edge point E are $(x_1 + t_1, y_1)$. Of course, if the pair of pixels is vertical rather then horizontal, the fraction t_1 is added to (or subtracted from) the y coordinate.

An important result of the equation above is that t_1 is unchanged when the distances d_1 and d_2 are replaced by $k \cdot d_1$ and $k \cdot d_2$, for any constant k. This means that the distance function need not be accurate up to a multiplicative constant. In particular, we need not have assumed that pixels have a diameter of 1! If we had assumed a radius of 1.5 or 2 and obtained a stretched version of the distance function, the calculation of edge points would not have been affected.

We now return to the detection algorithm. After calculating the point E and reporting it, the algorithm advances in a clockwise direction around the black portion of the image. Depending on the values of the pixels that lie ahead of the current pair, a next pair is selected within one pixel from the current pair, with one dark and one light neighbor. At most four neighbors need to be tested for this step; if the edge is bending sharply, one pixel from the current pair may be used again.

The algorithm is called from within a program that scans the image for a high contrast pair; as the contour is tracked, the coordinates of every pixel examined by the algorithm are marked in a special bit-matrix. Also, the coordinates of every new pair of pixels are compared to those of the original high contrast pair that started the contour. When this comparison succeeds, the algorithm returns to the calling program, having reported a full contour.

The program then begins to scan the image, row by row, searching for high contrast pairs (either vertical or horizontal) — but only if they are not already marked in the bit matrix. When the image is exhausted, every pair of neighbors has been tested, and so we know that every contour was tracked and reported.

1.3 Accuracy Estimates

Naturally, the first test for this algorithm would be its performance on the images of the perfect straight edges. After all, the gray-to-distance function is based on those images and the assumption that the edges there are, indeed, straight lines.

Table 1.1 summarizes the results of these tests. The same images, of a straight edge at various angles, were run again through the edge detection algorithm. This time, however, the algorithm used the sub-pixel calculation based on the gray-to-distance function. The bottom line of the table shows that 96% of the computed edge points were within one fifth of a pixel from the straight line. Also, all but a few isolated cases had an error of less than 0.3. For a visual example of this result, see

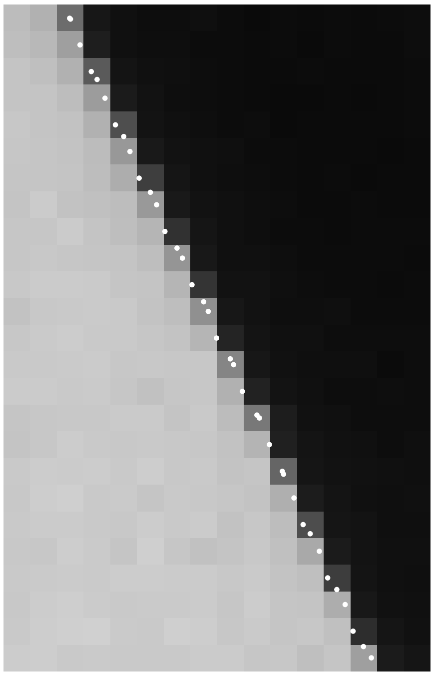

Figure 1.9: Sub-pixel edge points on a straight line edge.

Figure 1.9.

Another source of error that we can analyze is the fact that the edge is not really a straight line. The edge is, in general, curved, and so its intersection with a pixel-sized circle in the image is better modeled by a circular arc than by a straight line.

Consider Figure 1.10: calculating d as a function of the amount of gray is like assuming that the edge is the straight (dashed) line; whereas the true, curved, edge is lying at a slightly different distance, d', from the center of the pixel.

Table 1.2 summarizes the results of a numerical test to compare d to d' when the curvature of the edge is close to a pixel size. We assumed that the edge is a circle with a small radius, between 1 and 5. For each such radius, we calculated the exact area trapped between the circular pixel and the arc of the edge, when the arc was at distance d from the center of the pixel (as in Figure 1.10). This calculated area yields a gray value, which we run through the inverse function to obtain the value of d.

The table shows the differences between d and d' for each value of the radius and the distance. As we can see, the error is hardly noticeable when the edge's curvature

image	part	number of points	Fraction of points with error: at most 0.1	at most 0.2	at most 0.3
strip2	0	194	0.675	0.938	1.000
	1	84	0.655	0.976	1.000
	2	80	0.713	0.963	1.000
	3	84	0.679	0.952	1.000
	all	442	0.679	0.952	1.000
strip6	0	144	0.674	0.965	1.000
	1	81	0.877	1.000	1.000
	2	83	0.880	1.000	1.000
	all	308	0.782	0.984	1.000
strip7	0	151	0.563	0.874	0.980
	1	152	0.789	0.967	1.000
	all	303	0.677	0.921	0.990
strip8	0	162	0.889	1.000	1.000
	1	157	0.955	1.000	1.000
	all	319	0.922	1.000	1.000
stripE	0	79	0.405	0.937	0.987
	1	53	0.830	1.000	1.000
	all	132	0.576	0.962	0.992
combined		1504	0.742	0.963	0.997

Table 1.1: Error distribution in the straight line images.

is 3 or more. For smaller radii, the error is larger, but remains well below 0.1. We may conclude that the curvature of the edge is not going to produce large errors in our sub-pixel calculations.

As an example, consider Figure 1.11, showing some of the edge points detected in an image of a letter. In this region, the contour has a small radius curvature — around 5 or less. But the resulting series of edge points is smooth and follows the curve very nicely.

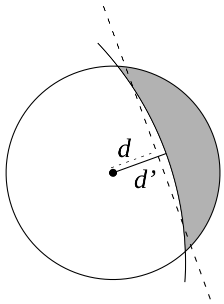

Figure 1.10: The straight line approximation for a curved edge

gray level radius	0.5	0.4	0.3	0.2	0.1
1.0	0.0710	0.0648	0.0617	0.0444	0.0314
1.5	0.0467	0.0435	0.0415	0.0328	0.0156
2.0	0.0348	0.0328	0.0285	0.0260	0.0137
3.0	0.0231	0.0221	0.0195	0.0154	0.0076
5.0	0.0138	0.0130	0.0119	0.0094	0.0048

Table 1.2: The distance between the straight line and circular arc that correspond to the same gray level.

1.4 Summary

The edge detection program detects and follows every contour in the given image, producing lists of edge points at sub-pixel resolution. The average accuracy can be seen to be on the order of 0.2 pixel, while the maximum errors are rarely over 0.3. This amounts to an effective 4 or 5-fold increase in the image's resolution.

We also note that this increase is achieved not by costly transforms applied to the whole image, but by simple calculations of a few arithmetic and look-up operations at every point of the output.

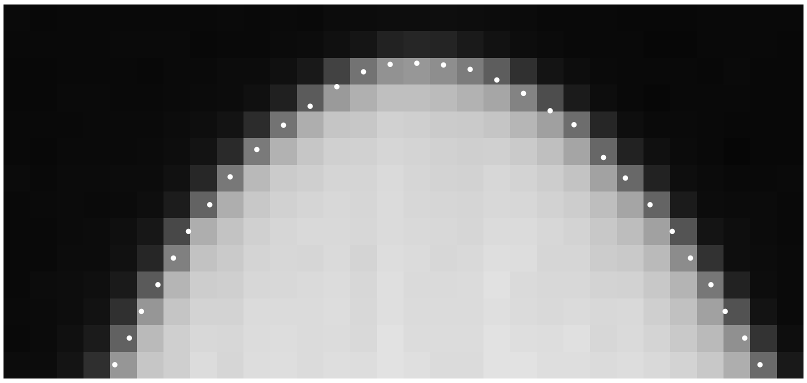

Figure 1.11: Edge points in the letter y.

Chapter 2

Curve Fitting and Segmentation

In this chapter we describe the process of curve fitting and segmentation. The input is a list of points, as produced by the edge detection procedure of the previous chapter.

We shall assume throughout this chapter that the list of input points follows the contour in a reasonable manner: the points are close to one another, at distances of roughly one pixel size; and they form a monotone sequence around the contour. This is a natural consequence of the edge detection procedure, but clearly, it is not always the case in general curve fitting problems.

We take the following approach: first, we use a rough polygonal fit (i.e., a sequence of straight lines) to detect sharp corners in the contour. These corners will serve as primary segmentation points, and there will be no attempt to fit single curves across them. Moreover, we will not even attempt to maintain tangent continuity at these corners.

In the second step, we work on regions between successive corners, producing a sequence of conics to fit every such region. A major part of the process is the decision on segmentation, i.e., the points along the contour where one conic ends and another starts. This decision must take into account the quality of the fit, and the smoothness of the join between the different parts.

Particular care must be taken in cases where the shape does not have any natural corners. In these cases, we must use a "global" segmentation algorithm, without knowing in advance any of the segmentation points.

In the following sections, we describe in detail each step in the fitting and segmentation process.

2.1 Corners

We consider every part of the curve where the radius of curvature is smaller than some small parameter to be a corner. For example, in Figure 2.1, which shows the outline of the letter 'k' in a Baskerville italic font, the P arrow points to a place where a long arc joins the main stem at a sharp angle. Fitting a circle or a conic to that small region would require a short arc, whose radius of curvature would be as small as 3 or 5 pixels.

A fit like that is possible, of course, but it may not be the most sensible way to describe such a corner. Aesthetically, we may wish to accentuate, rather than smooth out, the corners in the image: it is usually the case that the typographic designer *intended* that there be a sharp corner. And if that was the case, we realize that very sharp corners will get softened and smoothed along the way from the original picture to the digital data: the scanner and the edge detector algorithm have some implicit low pass filtering built into them, and even the ink and the printing processes are limited in the sharpness they can create.

Therefore, we make a conscious decision to sharpen these corners explicitly. When the estimate of the local curvature is smaller than some minimal parameter (3-5 pixels), we treat that part as an outright corner. This means that we allow tangent continuity to be violated in that region, and describe the corner by the intersection of two distinct curves.

It should be noted that a conic fit can always be applied even to the sharpest corners. The best fit will usually be a hyperbola that is almost identical with its asymptotes. In a sense, such a hyperbola is essentially a pair of straight lines, and the corner is implicit in the behavior of the hyperbola.

Instead, our method is to detect the corners right away and use them, explicitly as starting points for the segmentation process. One advantage is that this early step will save a lot of work during the segmentation process; and another is that the structure of the segmentation will be closer to that of the original, human design. After all, people prefer to view corners as the local intersection of two straight lines, rather than as a very sharp hyperbola!

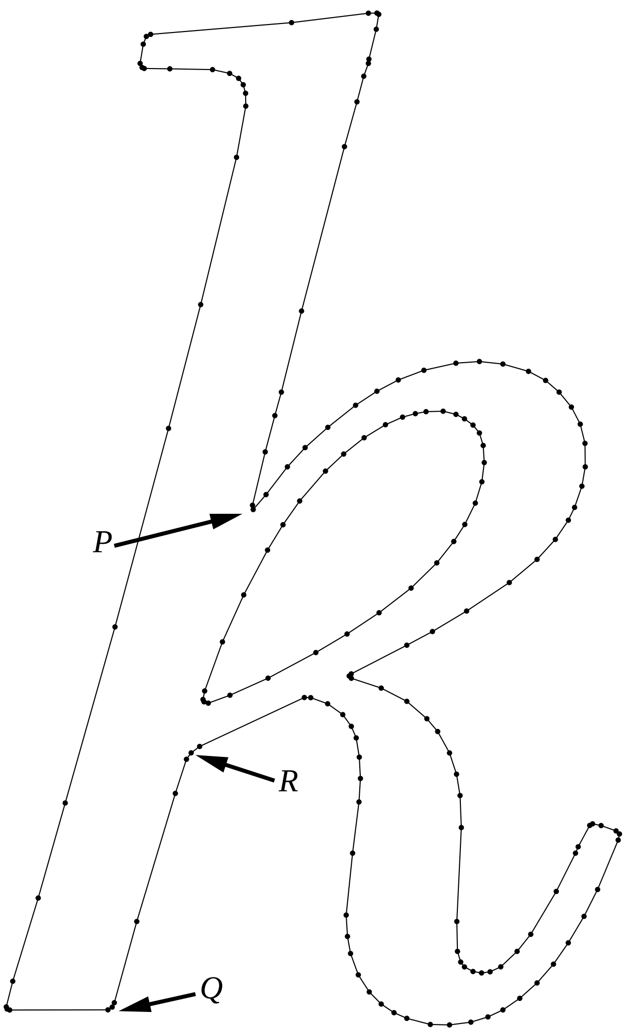

Figure 2.1: A rough polygon fitted to the letter 'k'.

We do give up some of the quality of the fit near the corner, in that the sharp corner can often stray away from the detected edge points. But once again, it should be noted that both the scanner and the edge detection algorithm are liable to smooth that region somewhat. In that sense, assuming that the corner is sharper than the data provides is consistent with the fact that the data in such a corner is less reliable.

2.2 Polygonal Fit and Detecting Corners

We start by applying a simple algorithm that fits a sequence of straight lines to the contour. We do not require a particularly good fit at this stage, since all we need here is to get a first approximation to the shape of the contour, and an idea of where the greatest curvatures occur.

Assume that the contour is given by a list of N edge points, $\{p_i : i = 0, \ldots, N-1\}$. The object is to produce a list of M indices, $\{i_j : j = 0, \ldots, M-1\}$, such that the approximating polygon has the points $\{p_{i_j}\}$ as its vertices. Clearly, the idea is to have M much smaller than N. At every stage, the algorithm has fixed the last vertex of the polygon at p_i, and it advances j to find the farthest place for the next vertex:

> $i \leftarrow 0$
> **repeat**
> $j \leftarrow i + 1$
>   **while** the line from p_i to p_j approximates the points
> $p_i, p_{i+1}, \ldots, p_j$ with an error no greater than ϵ, **do**
> $j \leftarrow j + 1 \pmod N$
>   **end**
>   report the index $j-1$, so p_{j-1} is the next polygon vertex
> $i \leftarrow j - 1$
> **until** j wraps around at 0.

Figure 2.2 illustrates the selection of the best index j for a given index i. Note also that some care must be taken at the wrap-around, when the points included in one line segment span both the start and the end of the list. But in this application, we are not interested in an optimal polygon, only in one that is good enough. (Compare, however, Pavlidis' "split and merge" paradigm for an optimal polygonal fit [HS92]).

Given the rough polygonal fit, we have an approximation to the curvature of every part of the polygon. First of all, we can look at the angle formed between successive

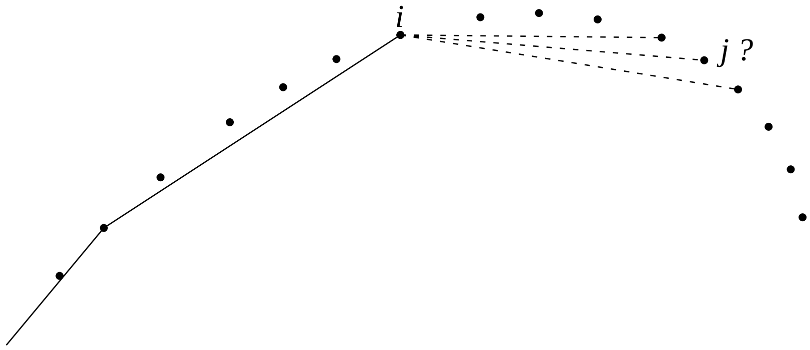

Figure 2.2: The polygonal fit algorithm.

sides of the polygon; if this angle exceeds some minimum (say 60°), we declare it to be a corner. We can also look at three consecutive sides, where the middle one is shorter than some minimum (say, 5 pixels) and examine the combined angle (see Figure 2.3). This time, the angle $\theta = \theta_1 + \theta_2$ must be greater than 60°, but at the same time both θ_1 and θ_2 must be greater than some minimum, say 20°.

Given a situation like this, a good way to estimate the curvature is to look at the radius of the circle that would be tangent to all three segments (see Figure 2.3). This radius turns out to be

$$R = \frac{l}{\tan(\frac{\theta_1}{2}) + \tan(\frac{\theta_2}{2})}$$

To see why this is true, note that that the two angles that are marked next to the radius are equal to $\frac{\theta_1}{2}$ and $\frac{\theta_2}{2}$, respectively.

Figure 2.4 shows three places where a corner is detected in the 'k' shape of Figure 2.1 — as well as one place where the radius of curvature is small, but not enough to make it as a corner. Here is how the formula above applies to each of these potential corners:

	angles		length	radius
Top right	59.9	49.4	2.3	2.2
Top right	24.6	51.8	4.8	6.8
Bottom left	37.9	59.3	3.2	3.5
Bottom center	31.7	29.0	6.0	11.1

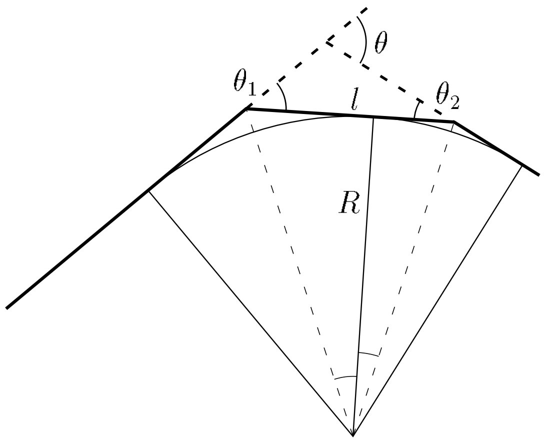

Figure 2.3: Curvature estimation by the polygonal fit.

Note also that on the left side, there are two consecutive angles of 59.3 and 24.6; but the distance between them is 10.1, which is clearly too large. The radius formula also yields R =12.8.

These heuristics are good enough to distinguish the obvious corners from the merely high curvature locations. Looking again at Figure 2.1, we can see that the Q and R arrows point to locations that are somewhat ambiguous. The algorithm will treat Q as a corner, but still let R be fitted by a flat conic.

2.3 Fitting and Segmentation

We now look at a set of edge points between two successive corners. The task is to provide a list of conic curves that satisfy the following criteria:

- There is a small number of *segmentation points,* including the first and last points in the set.

- Each curve is a good fit to the subset of points beginning at one segmentation point and ending at the next one.

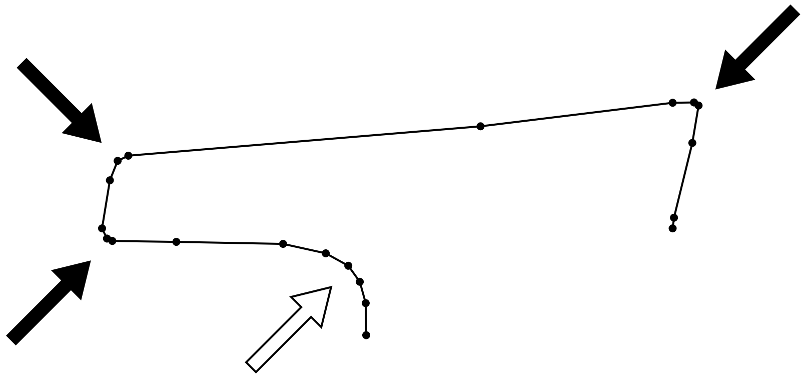

Figure 2.4: Potential corners at the top of the letter 'k'.

- The two curves that start and end in each segmentation point are well joined together, so that the piecewise curve has tangent continuity.

Note that we require a "small" number of curves, and that each curve must be a "good" fit. Both of these expressions refer to settable parameters of the segmentation algorithm. The ways to set these parameters, and their good ranges, will be discussed later in this section.

Formally, we can say that given a set of points $\{p_i : i = 1, \ldots, N\}$ we look for a number M, a list of indices $\{k_j : j = 0, \ldots, M\}$ and conic curves $\{f_j : j = 1, \ldots, M\}$ such that:

- The list starts and ends at 1 and N, and is monotonic:

$$k_0 = 1 < k_1 < \ldots < k_i < k_{i+1} < \ldots < k_M = N.$$

- The function f_j is a good fit for the points $\{p_i : i = k_{j-1}, \ldots, k_j\}$, where the goodness of fit may be measured by any reasonable norm.

- Finally, for every pair of consecutive curves, f_{j-1} and f_j, there is an intersection q_j that is near p_{k_j} and such that the tangents to both curves at that point are equal.

In order not to be confused with the indices, consider the following example: let N be 100, and the set of points be the subset $\{p_1 = 101, p_2 = 102, \ldots, p_{100} = 200\}$, taken from a larger set of points. A segmentation might then break this region into $M = 3$ parts, with segmentation indices $k_1 = 30$ and $k_2 = 70$. Each of the curves in this segmentation is responsible for its own sub-region: f_1 is a good fit for the points from $p_1 = 1$ to $p_{30} = 130$, f_2 is a good fit between p_{30} and $p_{70} = 170$, etc. And the intersection between f_1 and f_2, which we denote q_1, is going to be close, but not necessarily equal to, $p_{k_1} = p_{30} = 130$.

Note that the list of intersections $\{q_j : j = 1, \ldots, M\}$ is the true list of segmentation points, even though neither of these points need be part of the original set $\{p_i\}$. This lack of total dependence on the original data set is a difference between our method and that of Pavlidis [P83].

First, let us describe a procedure that tries to solve a slightly easier problem. The first allowance we make is that we limit ourselves to a small set of potential segmentation indices, from which the final M segmentation points will be chosen. Secondly, we will not require exact equality of tangents at the intersections, but only a certain goodness of fit. Specifically, we will make sure that consecutive curves f_{j-1} and f_j intersect close enough to p_{k_j}, and that their tangents at the intersection are close to each other, up to a small angle.

The output of this procedure will be then perturbed, ensuring that the tangents at the intersection points are identical, and not just close. This procedure will be described below.

Our first concession is to limit the segmentation process to choosing only from some subset of the points $\{p_i\}$. We do that by refining the polygon of the first phase. We take all the points that were selected as polygon vertices, and add the midpoints of each edge.

Specifically, if the polygon vertices in the region under discussion are the points whose indices are $\{m_j : j = 0, \ldots, N_p\}$, then we add to this list all the midpoints of the polygon sides, i.e., indices of the form $(m_{j-1} + m_j)/2$ (with the appropriate rounding, and care taken if the indices wrap around at 0). The points in this new list will serve as candidates for segmentation; the idea is that this list is rich enough so that the segmentation it produces is close to optimal, but small enough to make the segmentation algorithm tractable.

Now we calculate the best conic fit for every sub-range of points, i.e., for every subset that starts at one candidate point and ends at another. The fitting is done incrementally, which reduces the amount of work considerably. What's more, once a subset of points is known to yield only bad fits, we do not extend this set any further.

Having started with N_p polygonal sides, the refined list has $N_r = 2N_p + 1$ candidate points. Therefore, there are $N_r^2/2$ possible sub-ranges. However, under most circumstances, only one third or even less of these sub-ranges need be calculated, since it becomes evident early on that the larger sets cannot be fitted with an acceptably small error.

2.4 Least Squares Conics

It should be noted here that the problem of fitting conic sections to a set of data points is not an obvious one. The natural approach is to start with the algebraic description of all conics, which is given by:

$$ax^2 + 2bxy + cy^2 + dx + ey + f = 0 \tag{2.1}$$

The data is given as a set of N sample points (x_i, y_i). We evaluate the six basis functions, $x^2, 2xy, y^2, x, y$ and 1, in every point, obtaining the matrix

$$A = \begin{pmatrix} x_1^2 & 2x_1y_1 & y_1^2 & x_1 & y_1 & 1 \\ x_2^2 & 2x_2y_2 & y_2^2 & x_2 & y_2 & 1 \\ & & \cdots & & & \\ x_i^2 & 2x_iy_i & y_i^2 & x_i & y_i & 1 \\ & & \cdots & & & \\ x_N^2 & 2x_Ny_N & y_N^2 & x_N & y_N & 1 \end{pmatrix}$$

From this data we create the 6×6 matrix $M = A^TA$. The (i, j)-th entry in M is

$$M_{ij} = \sum_{k=1}^{N} A_{ik}A_{jk}$$

More explicitly, the upper triangle of the symmetric matrix M is:

$$\begin{pmatrix} \sum x_i^4 & 2\sum x_i^3 y_i & \sum x_i^2 y_i^2 & \sum x_i^3 & \sum x_i^2 y_i & \sum x_i^2 \\ & 4\sum x_i^2 y_i^2 & 2\sum x_i y_i^3 & 2\sum x_i^2 y_i & 2\sum x_i y_i^2 & 2\sum x_i y_i \\ & & \sum y_i^4 & \sum x_i y_i^2 & \sum y_i^3 & \sum y_i^2 \\ & & & \sum x_i^2 & \sum x_i y_i & \sum x_i \\ & & & & \sum y_i^2 & \sum y_i \\ & & & & & \sum 1 = N \end{pmatrix}$$

The least squares fit is obtained by finding an appropriate eigenvector of this matrix (see, for example, Bookstein [B79]).

Note that, in general, the matrix $A^T A$ is a source for instabilities, as it may contain some very large numbers alongside smaller ones. We shall address this problem below by centering and scaling the data.

In [P87], Pratt suggested applying the Cholesky decomposition procedure to M, but rather not in a straightforward order: instead, the largest element on the diagonal is selected after every step. This imposes a data-dependent permutation on the order of the basis function. Without such a permutation, the Cholesky procedure solves the least-squares problem under the implicit constraint $F = 1$ in Equation 2.1; but the permutation procedure means that any coefficient can be set to a constant, depending on the data: It is simply the one corresponding to the last remaining diagonal element.

As noted above, the elements of M are of vastly different orders of magnitude. Suppose that the typical size of the coordinates x_i and y_i is in the range of 100 to 1000, and that N is also in the hundreds. (This is reasonable if the points describe an edge of an image whose size is one or two inches, scanned at 400 dots per inch). In this case, then the top left 3×3 submatrix of M will contain numbers as large as 10^9 or 10^{13}, while the lower right submatrix will be only in the range of 10^3 or 10^4 — and of course, $M_{66} = N$ is even smaller.

Therefore, for most sets of data, Pratt's method tends to proceed from the second-order elements ($x^2, 2xy$ and y^2) to the first order (x and y) and the last diagonal element almost always corresponds to the constant basis function 1. So, in effect, the normalization constraint is almost always $F = 1$!

To avoid this, we tried the following correction: before applying Pratt's algorithm (Cholesky with Permutation), we translate and scale the data so that the new set of points have zero mean and unit variance in the "larger" dimension.

The transformation can be described as follows. To achieve zero mean, define the *center of gravity* (x_c, y_c) as

$$x_c = \frac{\sum x_i}{n}, \qquad y_c = \frac{\sum y_i}{n}. \tag{2.2}$$

Now, replace the data points (x_i, y_i) by their shifted counterparts:

$$x_i \rightarrow \quad x'_i = x_i - x_c, \qquad y_i \rightarrow \quad y'_i = y_i - y_c$$

Since the shifted samples have zero mean, their standard deviation of in the X and Y directions are given by:

$$\sigma_x = (\frac{\sum x'^2_i}{n})^{\frac{1}{2}}, \qquad \sigma_y = (\frac{\sum y'^2_i}{n})^{\frac{1}{2}} \tag{2.3}$$

Now we select a new *scale factor*, s, designed to make the standard deviation of the scaled data equal to 1. In order to avoid unnecessary distortion, both dimensions are scaled by the same factor. So we set

$$s = 1/\max\{\sigma_x, \sigma_y\}, \tag{2.4}$$

and the transformation is

$$x_i \rightarrow \quad x''_i = s \cdot x'_i = s(x_i - x_c), \qquad y_i \rightarrow \quad y''_i = s \cdot y'_i = s(y_i - y_c).$$

If we now construct the matrix M based on the shifted and scaled data, all the entries turn out to be of the same order of magnitude. In particular, the simple sums $M_{46} = M_{64} = \sum x''_i$ and $M_{45} = M_{54} = \sum y''_i$ are both zero. Also, depending on whether the scale factor s is set to $1/\sigma_x$ or $1/\sigma_y$, either the entries $M_{16} = M_{61} = M_{44} = \sum x''^2_i$ or $M_{36} = M_{63} = M_{55} = \sum y''^2_i$ will be equal to N.

The effect of translating and scaling is that of moving the data to the vicinity of the unit box in the plane. This means that the expressions involving x and y, as well as those involving x^2, $2xy$ and y^2, are all roughly of unit size; and so, all the entries in the matrix M have the same order of magnitude — close to the number N of data points.

As a result, the process suggested by Pratt — selecting the largest element on the diagonal of the matrix after every step — produces a permutation that really depends on the data at hand. Testing the modified procedure shows that, indeed, permutation without scaling cannot avoid the singularities associated with the constraint $F = 1$ (i.e., avoiding curves that pass near the origin), but after scaling these singularities disappear.

An important aspect of least squares fitting is the ease of adding and subtracting points from the data set. Note that the entries in the matrix M are, originally, just sums of small polynomials evaluated at each data point. Therefore, in order to add or remove a point to the data, we need only to evaluate the appropriate expression, and then add it to or subtract it from each entry. For example, $M_{12} = 2\sum x_i^3 y_i$, so when adding a point (x, y) we modify the matrix by setting M_{11} to $M_{11} + x^4$, M_{12} to $M_{12} + 2\sum x^3 y$, M_{13} to $M_{13} + x^2 y^2$, etc.

If the data is shifted and scaled, the entries are more complex, but they still only depend on sums of small polynomials. For example,

$$\begin{aligned} M_{12} &= 2\sum x_i''^3 y_i = 2s^4 \sum (x_i - x_c)^3 (y_i - y_c) = \\ &= \sum x_i^3 y_i - 3x_c \sum x_i^2 y_i + 3x_c^2 \sum x_i y_i - 3x_c^3 \sum y_i - y_c \sum x_i^3 + 3y_c \sum x_i^2 \\ &\qquad -3x_c^2 y_c \sum x_i + 3x_c^3 y_c \end{aligned}$$

So, in order to add or subtract a point to the data, we first update the values of N, $\sum x_i$, $\sum y_i$, $\sum x_i^2$ ... and all the sums of the polynomials in x and y up to degree four, terminating with $\sum x_i^2 y_i^2$, $\sum x_i y_i^3$ and $\sum y_i^4$.

Then we update the values of x_c and y_c according to Equation 2.2, and then update the standard deviations (Equation 2.3) and the scale factor (Equation 2.4). Then we can use all the updated sums to construct the new entries in the shifted and scaled matrix M.

Note that we need here to keep track of 14 different sums, since there are two polynomials of degree 1 (x and y), three of degree two (x^2, xy and y^2) four of degree of degree 3 (x^3, $x^2 y$, xy^2 and y^3) and five of degree 4 (x^4, $x^3 y$, $x^2 y^2$, xy^3 and y^4).

Also, the formula for some of the matrix entries such as M_{11} above can require several multiplications. In all the recomputation of the symmetric 6×6 matrix requires about 60 multiplications, even before applying the decomposition procedure.

In conclusion, the shift-and-scale transformations allow us to perform Pratt's method for least squares conic fit without worrying about the singularities, such as curves passing through the origin or straight lines. The cost for adding or subtracting a point is constant, so incremental fitting of subsets of points can be achieved rather easily.

2.5 Optimal Segmentation

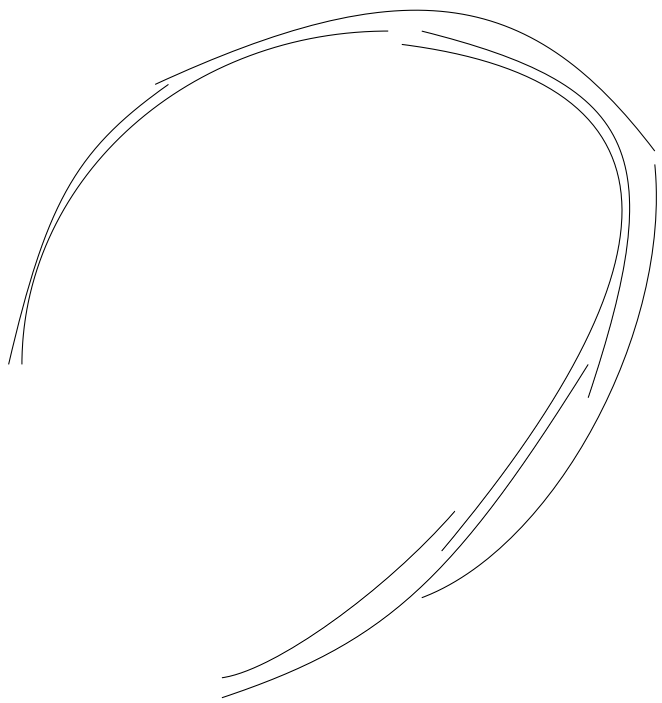

Figure 2.5: An illustration of the selection process.

After calculating the least squares fit for all the sub-ranges, we have a large set of curves, with each one starting and ending at (or near) one of the candidate points.

The situation looks somewhat like Figure 2.5. Of course, the curves are all going to be good fits to the data points, and so they would be much closer together.

Given this set of curves, the problem of segmentation now becomes one of selection. We want to choose a subset of these curves, such that the first one starts in the beginning, the second starts where the second ends, and so on until the last curve reaches the end of the data. Such a set will cover all the edge points, and the locations where curves start and end would be the segmentation points.

The quality of such a selection depends on three things: the quality of the fit for each curve; how well two successive curves join at their common end; and the total number of curves.

In regard to the quality of the join, consider Figure 2.6. This is a closeup look at the join of two curves. The curves, denoted by thick lines, were obtained by fitting two sets of data points, whose intersection is the common end point. (The data points are not shown in the figure). But since the fit is not perfect, the curves may not intersect each other at (or near) the common point. And their tangents near that end are close, but not identical.

We calculate a score for how well these two curves are joined by considering both the distance between their endpoints, and the angle α between the tangents. The formula that is used here is

$$score = dist + 10.0 \cdot \sin(\alpha)$$

The extra weight on the sine of the angle is tantamount to considering the displacement error over a distance of ten pixels; it was selected empirically, and seems to address well the importance of tangent continuity in obtaining a smooth-looking contour.

Optimizing the selection of curves can be formulated as a problem of finding the least expensive path in a graph. In this formulation, the nodes of the graph are the curves, and the cost of going from one node to another is the quality of the join.

Now we can apply Floyd's method for all shortest paths in a graph (an instance of dynamic programming) to obtain the segmentation. Note that by adding a suitable fixed constant to every curve, the algorithm will also minimize the number of segments

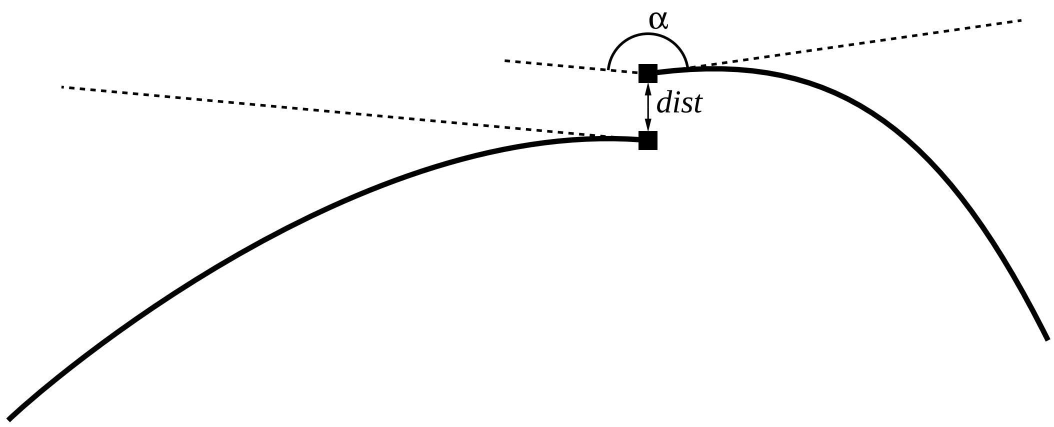

Figure 2.6: Evaluating the join of two curves

— subject, of course, the quaility of the overall solution.

The application of Floyd's method (see Even [E79]) to the problem at hand is as follows:

The partial information during the execution is kept in an array of back-pointers: the back-pointer from any point P (among the segmentation candidates) points to a previous point M, such that the optimal partial path, starting at the source and leading to P, has as its last edge a conic from M to P.

Given: A sorted array of N *midpoints*
    and the set F of all partial curves
Find a path from *source* to *sink*
Initialize the *back-pointers* array
for $j = 1$ **to** N **do**:
    target $\leftarrow$ *midpoints*$[j]$
    for $k = 0$ **to** $j - 1$ **do**:
        mdp $\leftarrow$ *midpoints*$[k]$
        if the set F contains a curve f for the set of points
                from *mdp* to *target* **then**
            Obtain $c = (g, p, mdp)$ from *back-pointers*$[k]$
            (g is a conic, whose end-points are p and *mdp*)
            score $\leftarrow$ *score*[k] + *error*(f) + *join-quality*(f, g, mdp)
                + fixed cost + extra costs for bad fits, if any
            (Unless *mdp* is the same as *source*, in which case
                no join needs to be tested)
            if this score is the new best, update *back-pointers*$[j]$
        end if (existence of f)
    end (k loop)
end (j loop)
Finally, construct the best path by following the back-pointers from *sink*.

2.6 Curves Without Corners

So far we have described the segmentation algorithm as it works between two corners. The implication is that we have a starting point and an end point, but no end conditions. However, some contours will not have corners, such as those encountered in the letters o or s (when done in a smooth, non-serif font).

Solving the segmentation problem in this case is similar to the problem of finding

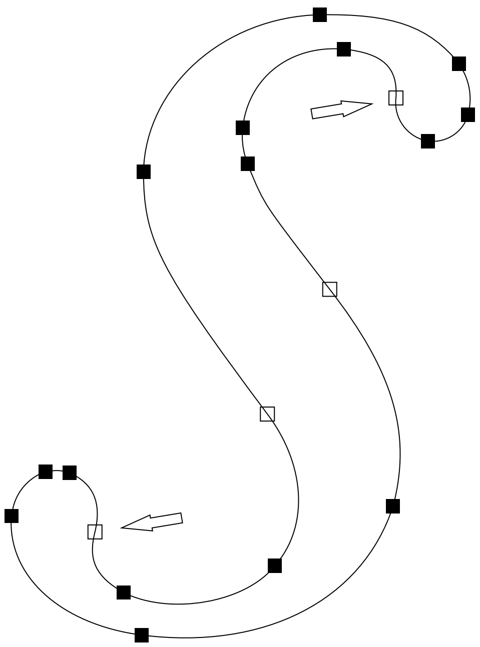

Figure 2.7: Segmentation with inflection points.

the best cycle in a directed graph, as opposed to merely finding the best path from A to B. Essentially, the way to solve this problem is to try all possible starting places, and find the optimal path that starts and ends in that place. That amounts to running the algorithm of Section 2.5 for every possible segmentation, each time using a different point as both the source and the sink; so the total work is multiplied by the number of segmentation candidates.

The algorithm must be modified, however, to take into account the quality of the fit at the starting point itself. And it is not enough to just calculate the score for a cyclic path, starting at point A, and then add the quality of the join between the first and last curves. The join at A must be incorporated in the optimization process.

Indeed, the correct way to do so is to treat the *curves,* rather than the locations on the contour, as starting points in the abstract graph. To do so, we select one curve segment c, going from P to Q, and assume that it is the first element in a cyclical path. The algorithm from Section 2.5 proceeds as before, except that even when the working mid-point mdp is equal to P or Q, the appropriate join must be evaluated.

(See Figure 2.8.)

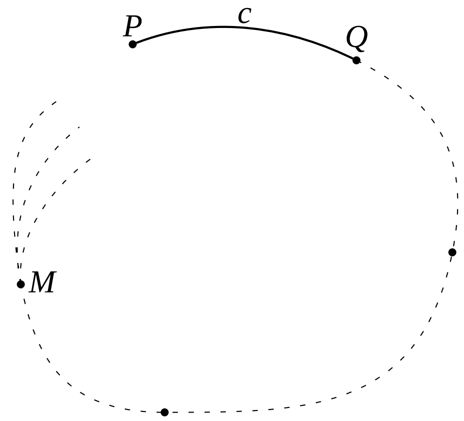

Figure 2.8: A partial path starting at a given curve.

Clearly, not all the possible partial fits need be tested. For every location along the contour, there a small set of curves that cover it; since any cycle must cover this location, one member of the small set will be part of that cycle.

Thus, we save a lot of redundancy by using as candidates only a set of curves that cover a given point in the contour. Obviously, it is best to choose the point that is covered by the fewest curves.

For example, many contours that don't have corners do have inflection points. These points are natural places for segmentation (Figure 2.7). As a matter of fact, one of the salient features of conics is that they do not have inflections! Therefore, a conic is not likely to be a good fit to a region that includes an inflection point.

Because of this, the neighborhood of an inflection point is going to be one place where only a few conics can be fitted. For example, in the case of the letter s of Figure 2.7, only 38 out of 1903 curves cover the lower left inflection point. Clearly, restricting the search to paths that begin with these curves is crucial for the efficiency of the segmentation algorithm.

These curves may further be sorted according the error of the fit; we could then limit ourselves to only the best of these curves. We would not be assured of finding the optimal path, of course, but this appears to be a reasonable time-saving heuristic.

In the cases that were tested, the optimal path used a curve in the top half of the candidate set, and even a restriction to only the top 10% still yielded a nearly-optimal result.

This is especially important in the case of almost-circular contours, like the letter O. Even the smallest set of curves that covers a given point has a lot of candidates. But most of the solutions are very similar to each other, and using only the top portion of the list of candidates does not cause a big drop in the quality of the fit.

2.7 Final Modifications

In the last step, the curves are modified to ensure the C^1 continuity. The two tangents at an end point, which are already known to be close to each other, are replaced by an average of both, and the curves are re-computed to agree with the new end conditions. Similarly, the corners are re-defined by extending and intersecting the tangents of the curves that lead into them.

Figure 2.9 shows the final segmentation and fit to a character that contains two long, but smooth, arcs. This was the letter C from a font called *Diploma*, which was scanned from a calligraphy book. Its contour has two long parts, and each is described by 10 conic sections. In the figure, the thin straight lines describe the tangents to the conic curves at the segmentation points.

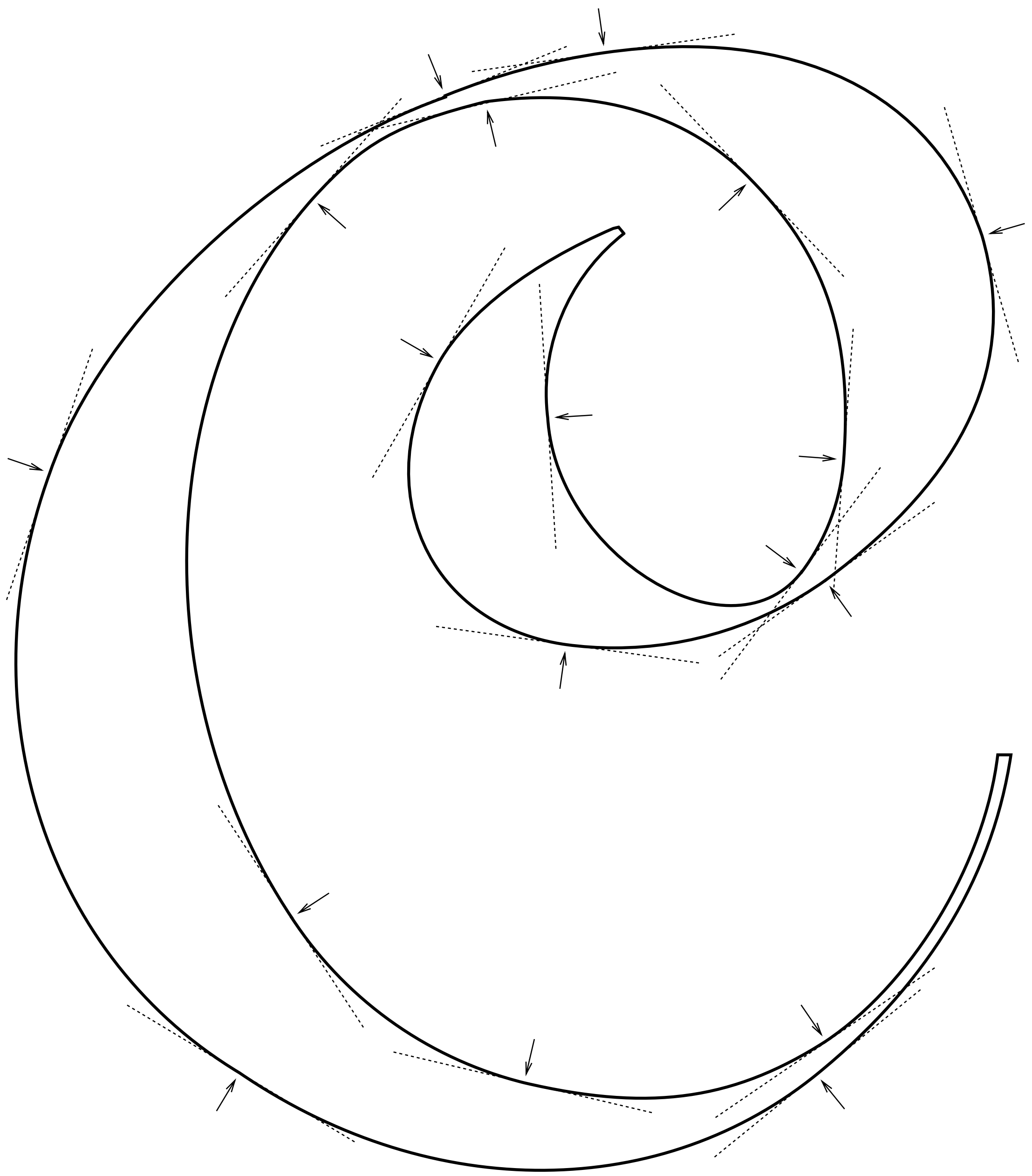

Figure 2.9: Final segmentation of a *Diploma* letter.

Chapter 3

Programs and Tools

In this chapter we discuss some of the practical aspects of implementing the algorithms that were described in the previous chapters. We trace here all the steps that lead from a scanned image to a font file, in a format that can be used by any PostScript[1] printer or interpreter.

3.1 The Edge Detection Program

The program `edges` takes an image file (in Sun's rasterfile format) and creates a corresponding file with lists of edge points, one list for every contour in the image. In order to do so, the program needs to find a starting point in each contour, track that contour, and then make sure not to track it again.

To do so, the program keeps a bitmap with the same dimensions as the image (but only one bit, rather than eight, per pixel) and uses it to mark all the locations that were visited by the edge detection algorithm. This bitmap serves as a "flag" matrix, since each bit indicates whether the corresponding pixel in the image has been seen by the tracking algorithm.

To locate the starting points of the contours, we simply scan the image, line by line, looking for neighboring pixels — either horizontal or vertical — such that the difference between their gray level is large enough. We then check the flags bitmap

[1]PostScript is a trademark of Adobe Systems Inc.

mentioned above, and look at the bits that correspond to this pair. If either one of these bits is on, we know that a previous contour has passed through this place and move on. This makes sure that we will not process the same contour more than once. (Note that this new pair might actually belong to a different contour, which just happens to pass very close to another contour. But if that is the case, there will surely be other starting places in the new contour that are distinct enough to be used as starting places.)

If the bitmap shows that the dark-light pair of pixels has not been used before, we use it to begin the edge tracking algorithm. The tracking procedure reports every edge point along the way, and at the same time marks the pixel that it used in the flags matrix. The tracking continues, moving clockwise around the dark image, until we return to the starting point. Then we return to the main loop, and continue scanning the image for more contours.

It has already been mentioned that images from different sources may have different dynamic ranges, meaning that the highest and lowest gray levels in the image, corresponding, respectively, to white and black, are not the same for all images (and all scanners). To handles this variety, the `edges` program always looks for a "sibling" of the image file, a histogram file that contains, in its first line, a suggestion for the gray levels that are assumed to represent the black and white levels.

In order to create a histogram file for an image, `edges` invokes the utility program `clip`. This program calculates the histogram of the image, and then selects the threshold levels of white and black. levels. The selection is based on the heuristic that was suggested in Chapter 1, Figure 1.6.

The separation between edge detection and histogram analysis is designed to allow for different situations: For example, the operator may modify the black and white levels based on some other criterions, or let various image files use the same values without needing to actually go through the histogram stage.

At the heart of the edge tracking and detection program we find the calculation of sub-pixel edge points, as described in a previous chapter. We have not tried to optimize this code, but the rate of production is about 15,000 edge points per second. This means that even large sheets with many symbols and markings (as in Figure 3.2) can be fully processed in less than a minute. This time is usually negligible even when compared to the scan time, not to mention the curve fitting program.

3.1.1 Data Packing

The points are saved in a tightly packed format, taking advantage of the sequential nature of the edge. Once the first point of the contour is fully described, the following points can be fully described only by the small increments relative to the previous ones.

Because of the way the edge-tracking algorithm operates, consecutive edge points are at most one pixel away from each other. Moreover, each point lies on the line that connects the centers of two pixels vertical or horizontal neighbors. Thus, one of the coordinates of every edge point must be an integer: the x coordinate if the neighbors are vertical, and the y coordinate if the neighbors are horizontal. (Occasionally, both coordinates can be integers, corresponding to an edge point that is identical with one of the pixel's center.)

This enables the packing scheme to be very efficient, even if we choose to encode the coordinates in units units of 1/64 of a pixel (which is more than enough, given the accuracy of the edge detection algorithm.) The runs of consecutive edge points are encoded by one byte per point, with a small overhead to start and stop each contour.

The packed format is common to the whole suit of programs: the edge detector, the curve fitting and segmentation program, and the intercative editor (described below). The computation involved in packing and unpacking the data is minimal, easily going through the large files of 100,000 points in a few seconds.

3.2 Interactive Additions

Figure 3.2 shows a snapshot of the interactive program that is used to select the characters in an image and assign them font-related information. In this example, we have just scanned a part of a professional typographic sheet that contains the Baskerville-italic font. After edge detection, the file contains more than 300,000 edge points in almost 2000 contours.

Of course, most of these contours correspond to the various marks on the page, so the objective here is to isolate the important contours for the next step in the process. Still, having these markings on the screen is useful, because they provide valuable information about the correct spacing and location of the characters.

In the upper left corner of the window we see a scaled-down version of the file. The main part of the window shows the part that contains the letter Q, almost in full scale. The bottom line in the panel tells us that two contours, numbers 8 and 9 were selected by the user. This is done simply by clicking the mouse near the contour: The numbers on the right give the details of the last click of the mouse. Contour 8 describes the outside of the letter, and the inner contour was selected automatically because it was completely included in the other's bounding box.

The rectangle around the Q was then adjusted to fit the markings in the image. The lower edge of the rectangle is aligned with the horizontal mark to the right of the box, which represents the level of the printing base-line. This means that the "leg" of the Q is located below the line, and is therefore a "descender." As for the left and right sides of the rectangle, they were placed in accordance with the small arrow (on the left, near the numbers) and vertical mark (top right). They represent the character's origin and width, defining its relationship to the characters that come before and after it on the printed line. (This is similar to the font metrics required by PostScript [Ad86].)

Finally, the two contours and metrics information (i.e., the dimensions described by the rectangle) are assigned the name 'Q' and saved to a new file called `bask-it`. The next step in the digitization process, which is the curve fitting and segmentation program, will be applied only to the contours that were specified as part of the new font.

3.3 Conversion to DPS

The most common Digital Printing Systems involve the PostScript language and printing engines. Thus, the final stage in our process translates the curves and their associated metrics into a Type 3 PostScript font. Once this is done, the font can be used like any other PostScript font — scaled, rotated and printed (Figure 3.1).

Figure 3.1: Scaled and rotated text in the *Diploma* font.

The conversion of contour descriptions to PostScript format is not completely straightforward: The PostScript graphic engine has a built-in function for drawing Bezier cubics, but none for conics. Therefore, we had to come up with a scheme for approximating conics by Bezier cubics. This scheme is described in the Appendix, and its implementation is simple and direct. As a matter of fact, all the drawings of conics throughout this document were done with this scheme.

3.4 Some Numbers

The current implementation of the fitting program takes from 20 to 100 seconds to work on one character, depending on its complexity. The complexity is not so much the number of edge points involved, but mostly the number of polygonal sides between successive corners. In that sense, a simple shape like the letter C in the *Diploma* font is actually quite complex, since it requires much more segmentation than, say, the letter k with its various corners.

The complete *Diploma* font has 75 characters, and their contours are made of more than 300,000 edge points. The process of curve fitting for the whole font took about 35 minutes. This should be compared to the standard, manual process of fitting and segmentation, which requires hours and days of interactive human work.

3.5 Summary

The font production process was implemented by a suite of programs, including one interactive tool. They were all written in the C language, and run on a Sun SPARC station.

- `edges` : Detection and tracking of edges in scanned images, producing points-files separated to distinct contours. About 1500 lines of code.
- `fittool` : The interactive tool for displaying points and conics, and for adding font-related data. Runs under Sun's SunView window system. 4500 lines.
- `fit` : The main curve fitting and segmentation program, with several testing and debugging variations. 5200 lines.
- `con2ps` : Converting a conics-file to PostScript code, including the approximation of conics. 1100 lines of codes.
- `clip` : A utility program for clipping and editing 8-bit images, which is also invoked from `edges` to calculate the histogram of an image. 700 lines.
- `pack` and `unpack` : The data-packing scheme utilities, used by all programs that access the points-files. 800 lines.

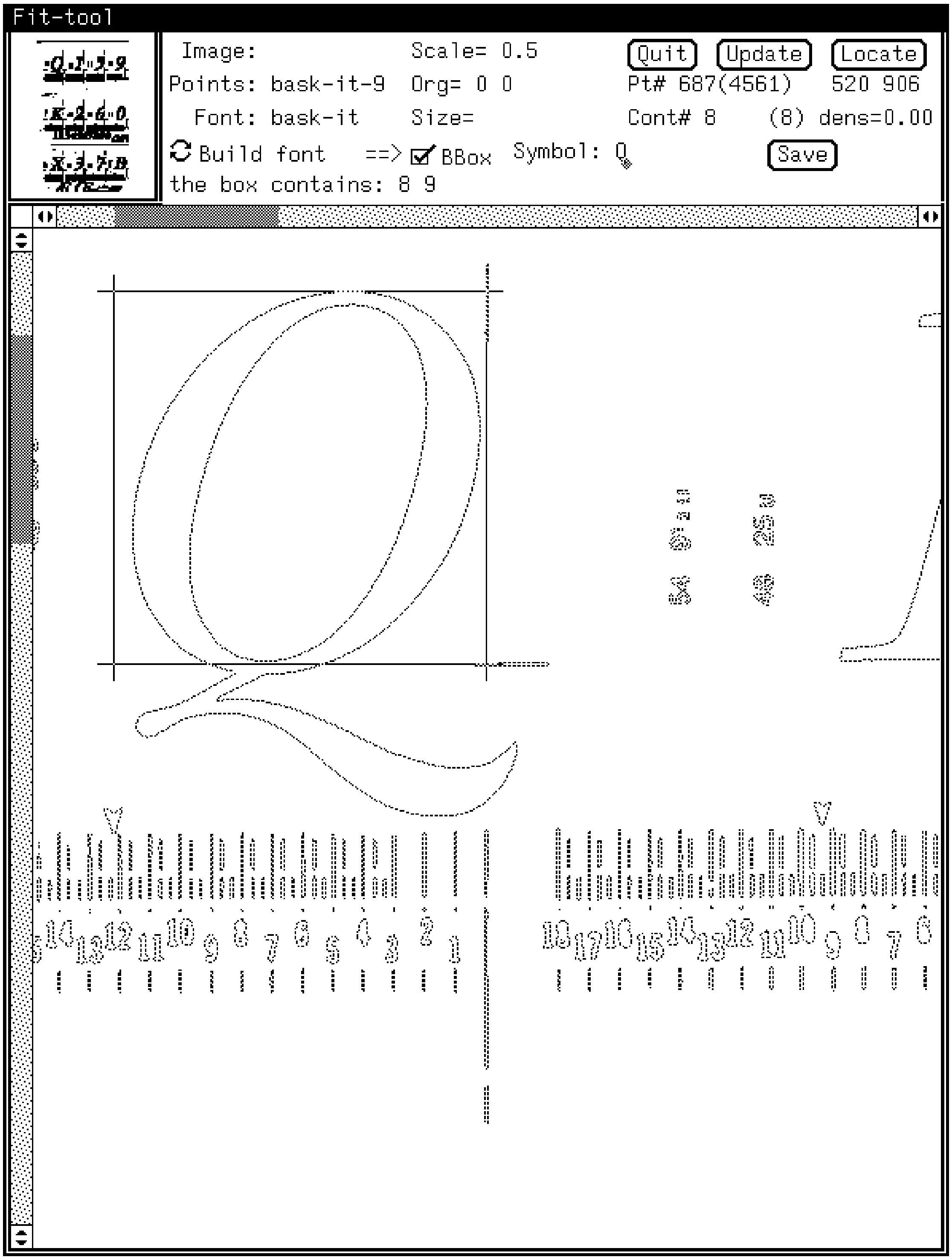

Figure 3.2: A snapshot of the interactive program `fittool`.

Chapter 4

Examples

In this chapter we describe in some detail the performance of the system on large sets of images, and show the resulting fonts. In all the examples, the input images were scanned with a Microtek MSF-400GS scanner, at 400 dots per inch, using software developed by Perfect Byte, Inc. The work was performed on a Sparc workstation, by software written in C with no special optimization.

4.1 Professional Type Face

The complete Baskerville-Italic font is described in two sheets of photographic templates. The sheets are 23 inches wide and 19 inches tall, and contain about 100 symbols: in addition to the 52 letters (capital and small) and 10 digits, there are special variations (such as *ç* and *æ*) ligatures (*fi*) and symbols (§, \$, etc.) The characters are each about one inch tall

The sheets were scanned in parts, with each scan covering roughly a 5" by 8" portion of the sheet, with 4 to 10 characters in the image. In addition, each partial image contains hundreds of marks, numbers and symbols, which are not part of the font but provide information about the positioning of the letters.

At 400 dpi, 5" by 8" becomes an image of 2000 by 3200, or 6.4 million pixels. To conserve disk space, we save these images only long enough to apply the edge detector to them. On such an image, the edge detector produces between 60,000 to 100,000 edge points, grouped in a few hundred contours. Most of these contours, of course, describe the marks and need not be analyzed to build the font.

We then use the interactive program to separate the letters from the marks and add the metrics information. The result is a file that contains the outlines of 92 characters, in 134 contours. These contours contain about 245,000 edge points, and so the compressed representation can be as small as one quarter of a megabyte.

The next step is to run the fitting program on the contours. As noted in the discussion of the fitting algorithm, the running time depends quadratically on the number of candidate points for segmentation, and this number depends, in turn, on the error limits set for the polygonal approximation. The running times for the whole font file are listed in the following table:

Max. polygon error	Running time (seconds)
0.4	6150
0.5	4980
0.6	4000
1.0	2765
1.5	2240

In using the smaller error limit, we get better recognition of corners, and we also have an improvement in the quality of the fit and the segmentation. For example, Figure 4.1 shows two results of fitting the letter g: on the left, the polygon error parameter was set to 1.0 pixel, and on the right it was 0.4. In both cases, we see the final curves and the tangents in each segmentation point.

In particular, we note that when the error is 1.0, the inner oval shape in the upper part of the g gets a segmented into five parts, instead of four. Clearly, we did not have enough candidates for segmentation that would allow the program to recognize the simpler solution of only four parts.

After applying the curve fitting algorithm with the small error parameter to the full font, we ended up with a description made of 1970 curves. 1398 of these are given as conics, and 572 as straight lines. (Some of these straight lines are very short, as they correspond to a clipped description of a corner.)

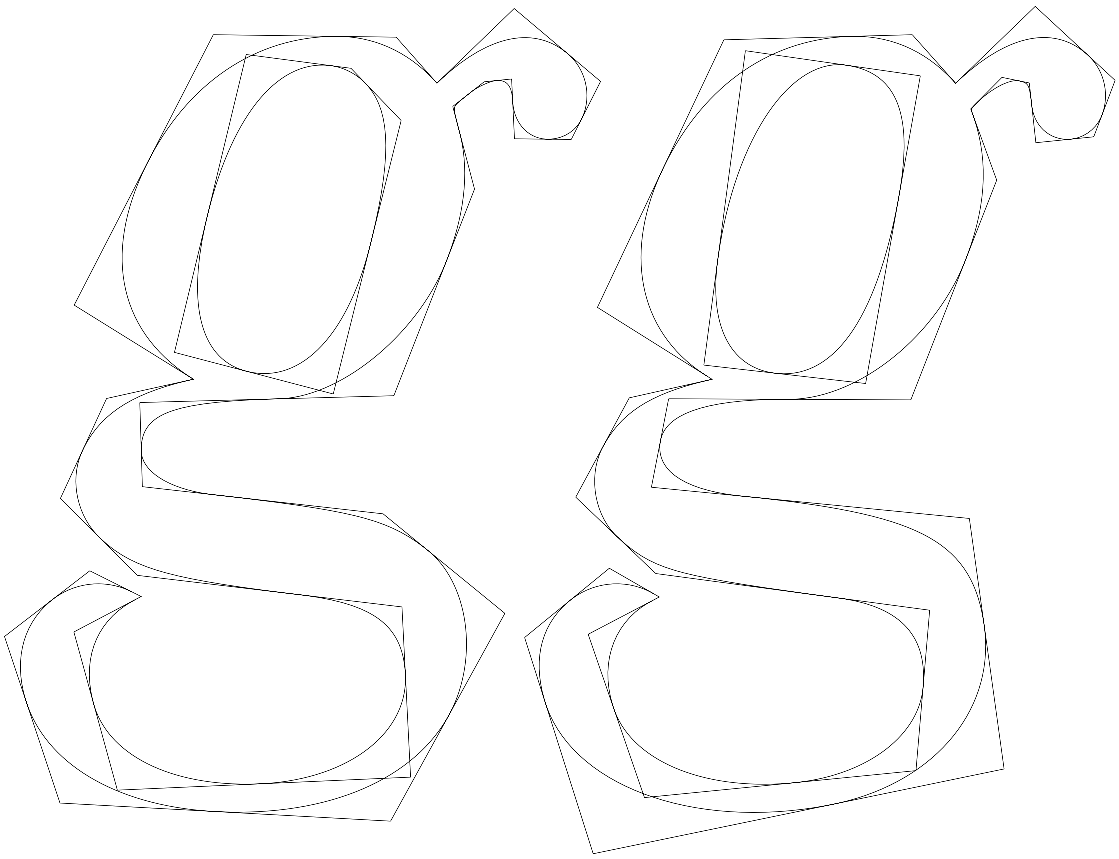

Figure 4.1: Two fits for the letter g, and the tangents at the segmentation points. (Left: P=1.0. Right: P=0.4.)

Since the outlines are continuous, each straight line can be specified by two coordinates, to denote the next point) while a conic requires six numbers and two coordinates. Adding a starting point for each of the 134 contours, we see that the resulting description requires $134 \cdot 2 + 1970 \cdot 2$ coordinates, plus $1398 \cdot 6$ coefficients for the conic sections. In all, there are 12,596 numbers (coordinates and coefficients), for a 40-fold reduction in the data size.

Unlike the coordinates of consecutive edge points, these numbers cannot be easily packed in a one-per-byte representation, especially the conic coefficients. But if we represent each conic as a triangle and a weight factor, we need only two extra coordinates and the weight, rather than six full coefficients. As described in Appendix A, we actually convert the conics to approximating Bezier cubics, so each section requires two additional points, and occasionally one conic needs to be split into two or more parts. Still, the effective compression rate, after all these transformations, is significant.

Figure 4.11 shows the final version of the Baskerville Italic, with small and capital letters, symbols and punctuation marks.

4.2 Ink On Paper

As a second example, let us take a close look at a symbol drawn in ink on white paper, with the type of contours that may be used by a calligraphic designer.

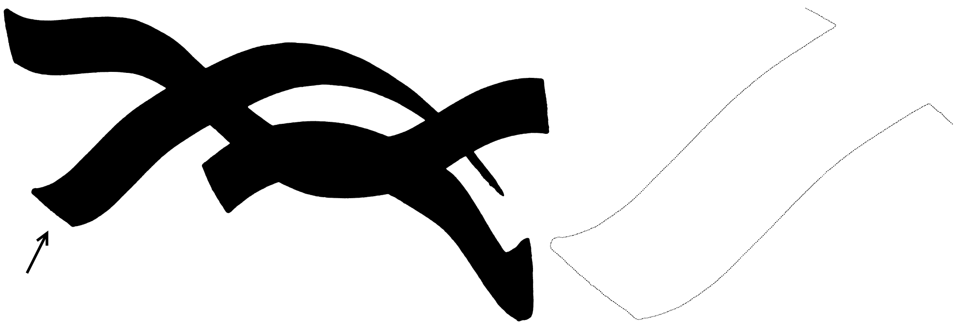

Figure 4.2: A calligraphic input, and an enlarged detail.

This image was also scanned from an original roughly one inch in size. As can be seen in Figure 4.2, the quality of the image is good enough so that the sub-pixel edges provide a smooth edge. Figure 4.3 shows the polygonal fit to the part that was detailed in Figure 4.2, at two settings of the maximal polygon error parameter: 1.5 pixel units and 0.5 units.

Finally, Figure 4.4 shows the resulting segmentation and conic fits to the same detail area. We can see that the finer polygonal approximation removed a corner,

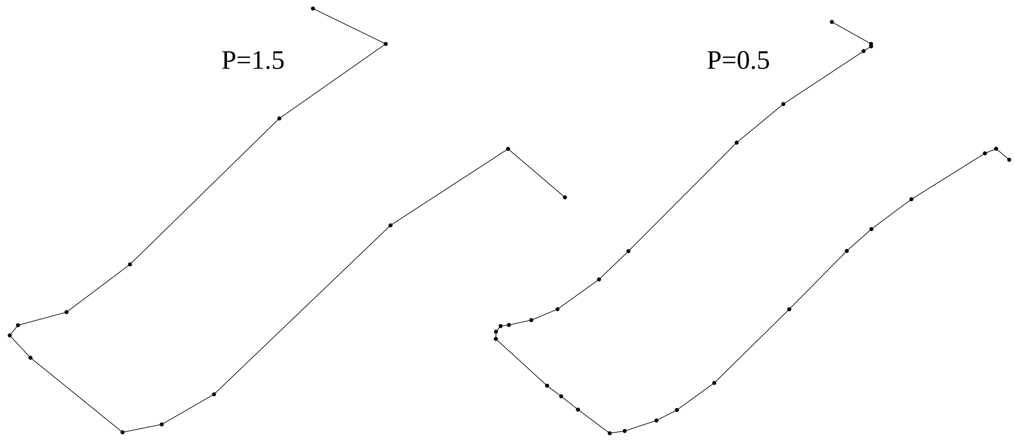

Figure 4.3: The polygonal fits with different polygon errors.

and replaced it with a short hyperbola. In the lower part of this contour we can see how the 0.5 result was indeed able to use one less segment.

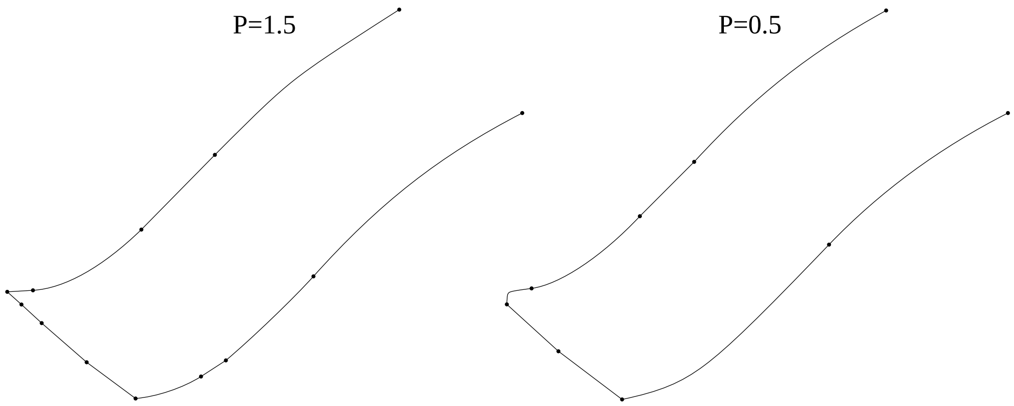

Figure 4.4: The final segmentation and conic fits.

The following table sums up the different amount of work performed by the fitting program, depending on the value of the polygon error parameter:

Polygon error parameter:		0.5	0.75	1.0	1.5
Outside contour	Polygon	156	114	95	73
	Corners	16	16	17	18
	Midpoints	307	230	193	152
	Partial curves	1360	749	539	313
	Final segments	62	62	58	61
Inside contour	Polygon	24	19	15	12
	Corners	2	2	2	2
	Midpoints	50	38	32	26
	Partial curves	313	164	116	77
	Final segments	6	7	9	7
Total running time (sec):		43.7	29.1	24.3	18.2

4.3 The Diploma Font

Some examples from the Diploma font were already shown in previous chapters. The images were scanned from a book on calligraphy, so the quality of the image was not as good as in the professional sheets. Figure 4.5 shows two characters from this font, and Figure 4.6 shows an enlarged view of the middle section of the 3.

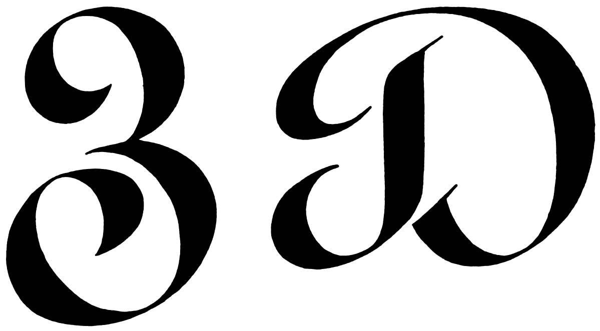

Figure 4.5: Two characters from the "Diploma" font.

Once again, the performance of the segmentation program on these contours increases when the polygonal fit is tighter. However, in this case the tightest fit, with

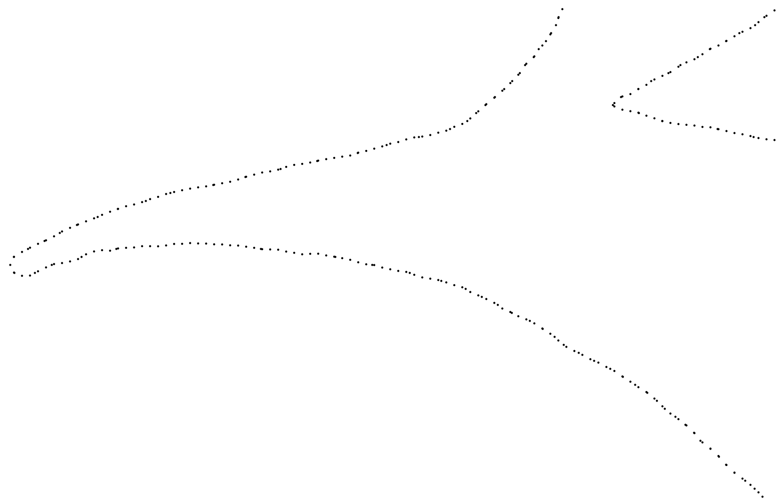

Figure 4.6: A detailed view of the edge points in the "Diploma" 3.

an error of at most 0.5 pixel, seems to be too sensitive to the small bumps and jaggies in the contour. The best fit is probably achieved with an error of 0.75, which still follows the contour faithfully, but also provides a good segmentation.

Figure 4.7 shows four results of the segmentation program, with different polygon error parameters. Table 4.1 summarizes the work performed by the program with these parameters, on the characters 3 and D. (Note that the D in this font is made of two parts.)

4.4 Kanji, Low Quality

The four Kanji characters shown in Figure 4.8 were scanned from the logo of a martial arts class. The edges in the image were not sharp enough for meaningful edge detection at a sub-pixel level. However, we could still run the fitting program with loose parameters, and obtain a reasonable font version of the characters.

This input tested the ability of the fitting program to handle contours that twist and turn with very small radii of curvature. One way to smooth the data is to use a large error parameter in the polygon. For example, on the left part of Figure 4.9 we

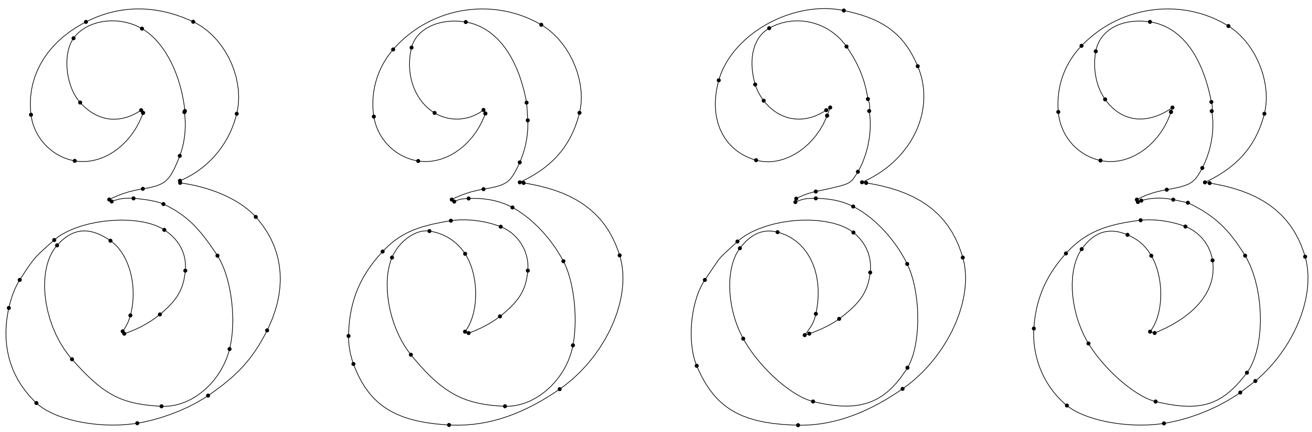

Figure 4.7: Four segmentations of the "Diploma" 3.

can see the polygon that describes one part of one character from the Kanji set. The error parameter was 1.5, meaning that each polygonal side is allowed to be 1.5 pixel sizes away from the data points it describes.

Similarly, the curve fitting itself was allowed an error of 1.1 pixel sizes, resulting in the outline on the right of Figure 4.9. The main problem with this description is that the near-vertical straight line in the middle (see arrow) is a poor fit for the data. Examination of the algorithm details shows that the corner heuristics caused that region to be detected as a corner, based on the high estimated curvature and short polygon side length. Thus, the short line denoted by the arrow is construed as a corner, and is sharpened as such.

Figure 4.10 shows the final font version of this small Kanji example. Even though the quality is lacking, it still demonstrates the effectiveness of digitizing a font by curve fitting: the edge detection algorithm had produced 10,129 edge points (as seen in Figure 4.8) but the final description of the 17 contours requires only 347 conics, including 211 straight lines. This means that the full description can be given by $17 \cdot 2 + 211 \cdot 2 + 136 \cdot 8 = 1510$ numbers.

This example may not be impressive in terms of the final product, but it does

Polygon error parameter:		0.5	0.75	1.0	1.5
3 (3644 points)	Polygon	174	134	117	94
	Corners	5	4	4	4
	Partial curves	3872	2322	1748	1129
	Final segments	34	34	32	34
Running time (sec):		92.3	70.6	62.7	49.8
D (2823 points)	Polygon	120	88	76	59
	Corners	3	3	3	3
	Partial curves	2574	1370	988	618
	Final segments	24	27	27	29
D (1518 points)	Polygon	63	49	41	31
	Corners	2	2	2	2
	Partial curves	1014	649	449	266
	Final segments	14	14	14	15
Total running time (sec):		98.4	73.9	61.3	48.7

Table 4.1: Running time and work performed on contours in the "Diploma" font.

demonstrate the robustness of our algorithm. Given a cleaner input, we would have been able to apply the algorithm with smaller error parameters, to obtain an output closer in quality to those shown above.

4.5 Summary

The automatic font digitization procedure was applied to a variety of inputs. Once the images were scanned and the edge detection produced a reasonably smooth outline, the only parameter that had to be adjusted was the polygon error tolerance, which reflected an estimate for the smoothness of the contours.

The algorithm proved to be robust and handle all inputs with success, producing good quality fonts, as shown in the illustrations in this chapter.

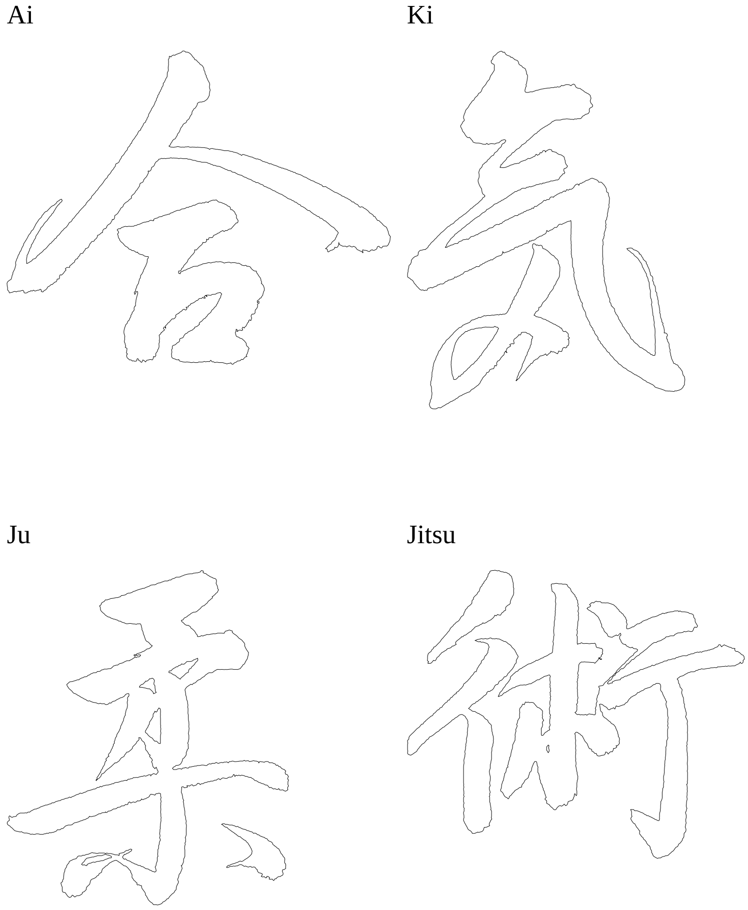

Figure 4.8: The Kanji symbols for "Ai Ki Ju Jitsu."

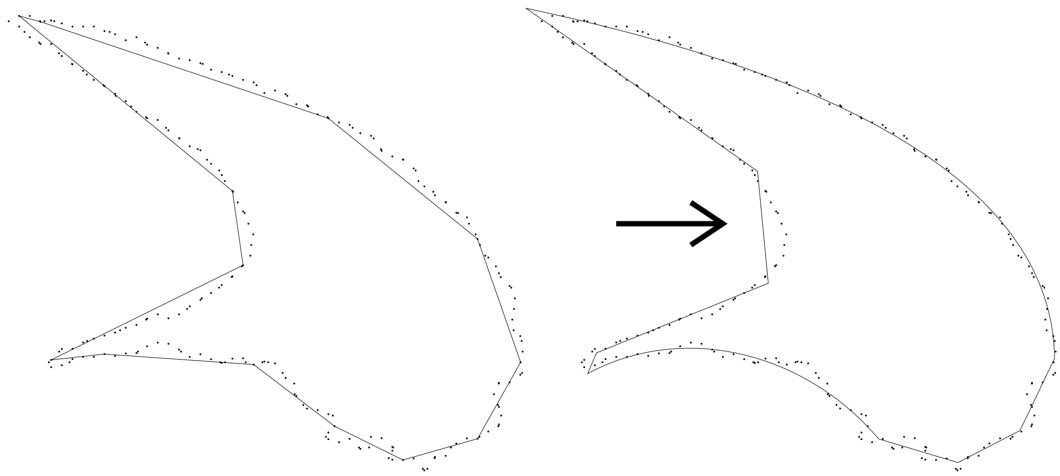

Figure 4.9: A detail from the Kanji character "Ju."

Figure 4.10: Digitized font version of the Kanji characters.

Baskerville Italic
The dagger is †
More: ‡ and §
Chapter & verse
Quote 4%
No?

Figure 4.11: The final Baskerville Italic font.

Chapter 5

Conclusion

The system described in this work can be used to perform the full task of capturing and digitizing fonts, from scanner to printer. The procedure is essentially automatic, and only the tasks that require interactive input are those of specifying names and data. It provides a full solution to the problem of capturing fonts and bringing them to use in a Digital Printing System.

The fonts produced by this system preserve the quality of the input, and can even improve it — by smoothing some noise and sharpening corners. The fonts are represented by a small number of curves, each described by an average of about six numbers. Thus, the final representation maintain a very manageable size.

There currently exist commercial systems that perform similar tasks, most notably Fontographer [M92], from Altsys Corporation. The state of the art in these systems emphasizes interactive design and offers many options for manipulating font files. However, the "auto-trace' capabilities of these systems — i.e., the task of digitizing scanned input, which is our focus here — are less advanced than ours. The common criticism from users of these systems is that they produce too many segmentation points, with short straight line segments. Very often, manual adjustments are necessary for a high quality result.

By contrast, our system is robust enough to handle a wide range of inputs, and it produces good results even for relatively low-quality images. At the same time, it needs only a few parametric adjustments and is fully automatic, requiring no manual post-editing.

5.1 Future Directions

The idea of optimizing the description of curves in digital images has applications outside of the font domain. For example, in Computer Aided Design and Manufacturing, there is a need to test a product (by looking at a picture) and compare it to the design (given as a mathematical curve). It would be interesting to examine the potential of our techniques in that, and similar settings.

In the field of digital font production, it may be important to optimize the description of contours not only with respect to inherent mathematical qualities, but also by considering the reproduction technology. For example, PostScript printers use a "hinting" scheme to ensure the quality of reproduction in small and medium scales. This scheme may have its own preferences about the placement of knots, and it might be interesting to examine their effect on the optimization process.

Appendix A

Approximating Conics by Bezier Cubics

The standard references to Bezier cubics and conics in general are Farin [F90] and Bartels et. al. [BBB87]. We also follow here the work of Pavlidis [P83] and Pratt [P85]. Like they do, we describe a conic by a triangle ABC, such that the conic passes through the points A and C and the tangents at those points coincide with the directions AB and CB. An additional "weight" parameter selects one of the family of conics that corresponds to a given triangle (Figure A.1).

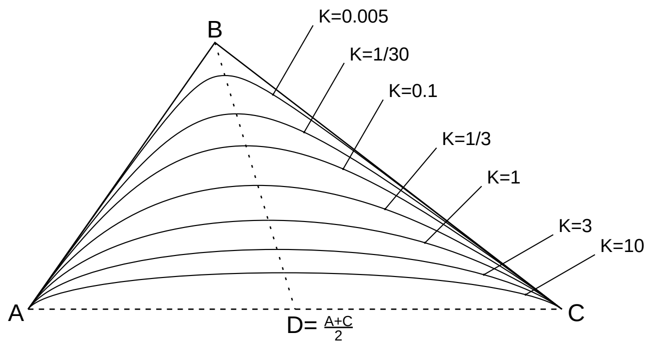

Figure A.1: Conics of various weights in a triangle.

Using aerial coordinates (i.e., normalized barycentric), the conic is defined as the set of points that satisfy $ac = Kb^2$, for some constant K. This constant is the "weight," or "sharpness" parameter, and selects among a family of curves that correspond to the given triangle. Figure A.1 shows such a family: Small values of K define hyperbolas that almost reach the point B, while large values define ellipses that stay close to the line AC. (The weight K here is the same as used by Pavlidis [P83].

Bezier cubics are usually specified by four points, $APQC$, with the curve passing through A and C and the tangents coinciding with AP and CQ. Parametrized by t in $[0, 1]$, the Bezier cubic is given by:

$$f(t) = (1-t)^3 A + 3(1-t)^2 tP + 3(1-t)t^2 Q + t^3 C. \tag{A.1}$$

Even though cubics have more degrees of freedom then conics, they do not contain them as a subset. The result described here shows how well some Bezier cubics can approximate conics, and how a subdivision procedure can improve this approximation to any desired accuracy:

Theorem 1 (Construction) *Let a conic be defined by a triangle ABC and weight K. Let D be the halfway point on the side AC, and define*

$$j = \frac{1}{1 + 2\sqrt{K}} \quad \text{and} \quad z = \frac{4}{3}j. \tag{A.2}$$

Mark the points P and Q on the sides AB and CB, so that $P = (1-z)A + zB$ and $Q = (1-z)C + zB$, and let M be the point $(1-j)D + jB$. Then the Bezier curve based on the points $APQC$ has the following properties:

1. *At the points A, C and M, the Bezier and the curve coincide, and share the same tangent direction.*

2. *The point M corresponds to the Bezier parameter $t = \frac{1}{2}$.*

3. *In the special case where $K = \frac{1}{4}$, the conic and the Bezier are identical to each other, and the curve is a parabola.*

See Figure A.2 for an illustration of the points described in the theorem. The second result addresses the quality of the approximation:

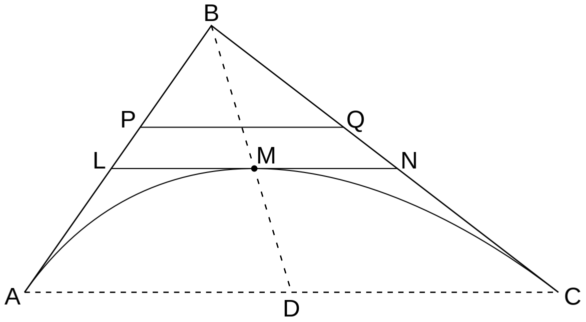

Figure A.2: The triangle ABC and the additional points P, Q, M, L and N.

Theorem 2 (Approximation) *For all values of K in the range $[\frac{1}{9}, 1]$, the Bezier conic approximates the conic with an error of less than one percent of the distance between D and M.*

The error is the maximal distance from any point on one of the curves to the other curve; in other words, for every point on either the Bezier or the conic, there is a point on the other curve at a distance of less than $0.01\|D - M\|$. The length of the segment DM, against which we measure the relative error, can be loosely described as the "thickness" of the conic. (See Figure A.2.)

Ways to estimate the errors on other ranges of values of K are discussed in Section A.2. But as the following theorem shows, we need not worry about values that are outside the "good" range $[\frac{1}{9}, 1]$:

Theorem 3 (Subdivision) *Let M be as above, and let L and N be the points on the sides AB and CB such that the line LN is tangent to the conic at M. Then*

1. *The triangles ALM and MNC can be used to describe the two halves of the conic, from A to M and from M to C, with a weight parameter:*

$$K' = \frac{\sqrt{K}}{1 + 2\sqrt{K}}$$

2. *If $K > 1$, then $\frac{1}{4} < K' < \frac{1}{2}$; and if $K < \frac{1}{9}$, then successive subdivisions approach $\frac{1}{9}$ from below in a number of steps proportional to $\log(\log(K))$. In particular, if $K > 10^{-6}$ then 3 steps suffice.*

Thus, when the weight is outside the range of the Approximation Theorem, the situation can be quickly remedied: If K is too large, then one subdivision step brings it inside the good range; but if K is too small, then more than one step may be required, and in practice usually only two or three. Once again, Figure A.2 shows the points L and N that are used to construct the new triangles.

In the following sections we explain and prove the results stated above. Section A.1 motivates and proves the details of the construction theorem, Section A.2 generalizes and proves the error bounds of the approximation theorem and subdivision procedure is explained in Section A.3.

A.1 Construction

The construction begins by making sure that the Bezier curve passes through the points A and C with tangents that agree with AB and CB. To do so, choose the two points P and Q to be on the sides AB and BC. Thus, there are numbers w and z such that $P = (1-w)A + wB$, and $Q = (1-z)C + zB$.

Substituting in Equation A.1, we find that the curve $f(t)$ is:

$$\begin{aligned} f(t) &= (1-t)^3 A + 3(1-t)^2 tP + 3(1-t)t^2 Q + t^3 C \\ &= ((1-t)^3 + 3(1-t)^2 t(1-w))\ A \\ &\quad +(3(1-t)^2 tw + 3(1-t)t^2 z)\ B \\ &\quad +(3(1-t)t^2(1-z) + t^3)\ C \end{aligned} \tag{A.3}$$

This formulation defines the point $f(t)$ as a linear combination of A, B and C, with coefficients whose sum is 1. Hence, theses coefficients must be the areal coordinates of the point $f(t)$ with respect to the triangle ABC!

It is convenient to change the parameter from t to $s = t - \frac{1}{2}$. Thus, the areal coordinates are:

$$\begin{aligned} a = a_{f(t)} &= (\frac{1}{2} - s)^2(2 + 2s - 3(\frac{1}{2} + s)w) \\ b = b_{f(t)} &= 3(\frac{1}{4} - s^2)(\frac{w+z}{2} + (z-w)s) \end{aligned} \tag{A.4}$$

$$c = c_{f(t)} \quad = \quad (\frac{1}{2}+s)^2(2-2s-3(\frac{1}{2}-s)z)$$

Now we want the Bezier to agree with the conic and its tangent at M. First, we show that M is indeed on the conic. We note that, at every point X on the median, the areal coordinates a and c are equal. (This is true because the area of the triangle AXD is equal to the area of DXC; and since B is on the median, it follows that the areas ABX and XBC, which define the coordinates a and c, are also equal.)

At the intersection of the median with the conic, where the areal coordinates satisfy:

$$K = \frac{ac}{b^2} = \frac{a^2}{(1-2a)^2}$$

Thus, at the intersection $a = c = \frac{\sqrt{K}}{1+2\sqrt{K}}$, and $b = \frac{1}{1+2\sqrt{K}} = j$. The last equality is from the definition of j in Equation A.2. Thus, the intersection is the point $jB + \frac{1-j}{2}(A+C)$, and according to the definition of Theorem 1, this is M.

Moreover, if we look at $Kb^2 = ac = a(1-b-a)$ as describing an implicit function $b(a)$, then this function has a maximum when $a = c$. So j is the maximum value of the b coordinate on the conic, and that implies that the tangent to the conic at M is parallel to AC.

So now, if we want the Bezier curve to agree with the conic at M, the following equations must be satisfied:

$$a = c \quad ; \quad b = j \quad ; \quad \text{and} \quad \frac{db}{ds} = 0.$$

The first two equations make the areal coordinates of $f(t)$ equal to those of M, and the third means that the value of b at that point is the maximum over the Bezier, ensuring that the tangent is parallel to AC.

The first and third equations can be transformed into:

$$\begin{aligned} \frac{3}{2}(s^2-\frac{1}{4})(w-z) + 3s(\frac{1}{4}-s^2)(w+z) \quad &= \quad (3-4s^2)s \\ (\frac{1}{4}-3s^2)(w-z) + s(w+z) \quad &= \quad 0 \end{aligned}$$

The obvious solution is $s = 0$ and $w = z$. (There is usually one more solution, with $s \neq 0$, but it often involves negative values of w or z, and we really don't need it.)

Substituting in the equation for b, we find that

$$w = z = \frac{4}{3}j$$

which concludes the proof of the construction.

We now turn our attention to the value of $\frac{ac}{b^2}$ on the Bezier curve. It is a function of s, and equal to:

$$\begin{aligned} k(s) &= \frac{(\frac{1}{2}-s)^2(2+2s-3(\frac{1}{2}+s)w)\cdot(\frac{1}{2}+s)^2(2-2s-3(\frac{1}{2}-s)z)}{9(\frac{1}{4}-s^2)(\frac{w+z}{2}+(z-w)s)^2} \\ &= \frac{(2-\frac{3}{2}z+(2-3z)s)\cdot(2-\frac{3}{2}z-(2-3z)s)}{9z^2} \\ &= \frac{(2-\frac{3}{2}z)^2-(2-3z)^2s^2}{9z^2} = (\frac{2}{3z}-\frac{1}{2})^2-(\frac{2}{3z}-1)^2s^2 \end{aligned} \tag{A.5}$$

As expected, when $s = 0$ we have $k(0) = K$. We note also, that if $z = \frac{2}{3}$ (corresponding to $j = \frac{1}{2}$ and $K = \frac{1}{4}$) then $k(s) = K$ is a constant, and every point of the Bezier satisfies the conic equation. Indeed, in this case, both the Bezier and the conic are equal to

$$(1-t)^2A + 2(1-t)tB + t^2C,$$

which is a parabola. That proves part 3 of Theorem 1.

It can also be shown that no other combinations of z and w will make $k(s)$ a constant. This is an indirect proof to the fact that the parabola is the only curve that is both a conic and a Bezier cubic.

A.2 Error Analysis

We saw that the Bezier curve is equal to the conic when the shape is a parabola. So it makes sense that the Bezier curve will also provide a good approximation to conics whose weight, K, is close to $\frac{1}{4}$. In order to analyze the quality of this approximation, we substitute $z = \frac{4}{3}j$ and $j = \frac{1}{1+2\sqrt{K}}$ in Equation A.5, obtaining

$$\begin{aligned} k(s) &= (\frac{2}{3z}-\frac{1}{2})^2-(\frac{2}{3z}-1)^2s^2 \\ &= (\frac{1}{2j}-\frac{1}{2})^2-(\frac{1}{2j}-1)^2s^2 \end{aligned}$$

$$\begin{aligned} &= (\frac{1}{2}(1+2\sqrt{K})-\frac{1}{2})^2-(\frac{1}{2}(1+2\sqrt{K})-1)^2s^2 \\ &= K-(\sqrt{K}-\frac{1}{2})^2s^2 \le K. \end{aligned} \tag{A.6}$$

Now, let the areal coordinates of $f(t)$ be (a,b,c). We look at points whose coordinates are of the form $(a+\epsilon, b-2\epsilon, c+\epsilon)$, for small positive values of ϵ. If we find a point like that such that $\frac{(a+\epsilon)(c+\epsilon)}{(b-2\epsilon)^2}=K$, then that point is on the conic, and the distance between the Bezier curve and the conic is no more than the distance between these two points, which is:

$$\begin{aligned} \|(aA+bB+cC) &- ((a+\epsilon)A+(b-2\epsilon)B+(c+\epsilon)C)\| = \|-\epsilon A+2\epsilon B-\epsilon C\| = \\ &= 2\epsilon\|B-\frac{A+C}{2}\| = 2\epsilon\|B-D\|. \end{aligned}$$

This equation gives the distance as a fraction of the length of the median of the triangle, the line BD. But this length could be much larger than the size of the conic arc itself (see, for example, Figures A.2 and Figure A.4). Instead, we compare the error to the length of the line from D to M. By the construction of M (Theorem 1), we have $\|M-D\|=\frac{1}{j}\|B-D\|$. Thus, we can simply estimate the relative error ρ, defined by:

$$\rho=\frac{2\epsilon\|B-D\|}{\|M-D\|}=\frac{2\epsilon}{j} \tag{A.7}$$

By the same analysis, it is clear that if (a,b,c) is a point on the conic, then the opposite translation, to the point $(a-\epsilon, b+2\epsilon, c-\epsilon)$, will yield a point on the Bezier curve. Thus, ρ is an upper bound on the distance from any point, on both curves, to the other curve — the kind of distance we talk about in Theorem 2.

A.2.1 The Relative Error Bounds

We now try to see which ϵ will make the following expression close to K:

$$\begin{aligned} \frac{(a+\epsilon)(c+\epsilon)}{(b-2\epsilon)^2} &= \frac{ac}{b^2}\cdot\frac{(1+\frac{\epsilon}{a})(1+\frac{\epsilon}{c})}{(1-2\frac{\epsilon}{b})^2} = k(s)\cdot\frac{(1+\frac{\epsilon}{a})(1+\frac{\epsilon}{c})}{(1-2\frac{\epsilon}{b})^2} \\ &\ge k(s)\cdot(1+\frac{\epsilon}{a}+\frac{\epsilon}{c}+4\frac{\epsilon}{b}). \end{aligned}$$

The last inequality is valid whenever $\frac{2\epsilon}{b} \leq \frac{3}{2}$, and since we are looking for small ϵ, we can assume that this is the case. Substituting the value of $k(s)$ from Equation A.5, we see that ϵ must satisfy:

$$\begin{aligned} k(s) \cdot (1+\epsilon(\frac{1}{a}+\frac{1}{c}+\frac{4}{b})) &= K-(\sqrt{K}-\frac{1}{2})^2 s^2 \cdot (1+\epsilon(\frac{1}{a}+\frac{1}{c}+\frac{4}{b})) = \\ &= K(1-(1-\frac{1}{2\sqrt{K}})^2 s^2)(1+\epsilon(\frac{1}{a}+\frac{1}{c}+\frac{4}{b})) = K. \end{aligned}$$

But since ϵ is only meant to be an upper bound on the distance, we will require that ϵ satisfy the inequality:

$$(1-(1-\frac{1}{2\sqrt{K}})^2 s^2)(1+\epsilon(\frac{1}{a}+\frac{1}{c}+\frac{4}{b})) \geq 1. \tag{A.8}$$

Now we can estimate the size of ϵ given some range of values for K. We look for an interval around $K=\frac{1}{4}$ that makes $(1-\frac{1}{2\sqrt{K}})^2$ small; call that interval $[k_0,k_1]$. Let p be an upper bound on this expression in the selected range:

$$p = \max_{k_0 \leq K \leq k_1} (1-\frac{1}{2\sqrt{K}})^2 = \max\left\{(1-\frac{1}{2\sqrt{k_0}})^2, (1-\frac{1}{2\sqrt{k_1}})^2\right\} \tag{A.9}$$

Note that if we want to have a specific value of p, then the largest interval with that bound will have $k_0 = \frac{1}{4(1+\sqrt{p})^2}$, and $k_1 = \frac{1}{4(1-\sqrt{p})^2}$. Such an interval may be thought of as "symmetric" around $\frac{1}{4}$ with respect to p. In particular, $p=\frac{1}{4}$ yields the values $k_0=\frac{1}{9}$ and $k_1=1$ that were cited in Theorem 2.

We now look for a number q such that, if we set

$$\epsilon = \frac{qs^2}{(\frac{1}{a}+\frac{1}{c}+\frac{4}{b})} \tag{A.10}$$

then Equation A.8 becomes:

$$\begin{aligned} (1-(1-\frac{1}{2\sqrt{K}})^2 s^2)(1+\epsilon(\frac{1}{a}+\frac{1}{c}+\frac{4}{b})) &\geq (1-ps^2)(1+qs^2) = 1-ps^2+qs^2-pqs^4 = \\ &= 1+pq(p^{-1}-q^{-1}-s^2)s^2. \end{aligned}$$

To make this ≥ 1, we note that $s^2 \leq \frac{1}{4}$; so it is enough to take q such that $p^{-1}-q^{-1}=\frac{1}{4}$, or:

$$q = \frac{4p}{4-p} \tag{A.11}$$

The next step is to obtain an upper bound on $(\frac{1}{a}+\frac{1}{c}+\frac{4}{b})^{-1}$. Since this expression is evaluated when $a+b+c=1$ and $ac=Kb^2$, it turns out that:

$$\frac{1}{\frac{1}{a}+\frac{1}{c}+\frac{4}{b}} = \frac{Kb^2}{1+(4K-1)b} \leq \begin{cases} Kb^2 & (\text{if} K \geq \frac{1}{4}) \\ \frac{\sqrt{K}}{2}b^2 & (\text{if} K \leq \frac{1}{4}). \end{cases} \tag{A.12}$$

To simplify matters, we want to use a single bound. Define:

$$u = \frac{1}{2\sqrt{k_0}}. \tag{A.13}$$

Since we always take $k_0 \leq \frac{1}{4}$, it is easy to see that for all K in the interval $[k_0, k_1]$, both $K \leq uK$ and $\frac{\sqrt{K}}{2} \leq uK$. So, by Equation A.12,

$$\frac{1}{\frac{1}{a}+\frac{1}{c}+\frac{4}{b}} = \frac{Kb^2}{1+(4K-1)b} \leq uKb^2 \quad (k_0 \leq K \leq k_1).$$

Finally, we go back to Equation A.3 and see that $b = 3z(1-t)t = 3z(\frac{1}{4}-s^2) = 4j(\frac{1}{4}-s^2)$. Substituting in Equations A.7 and A.10 we get:

$$\begin{aligned} \rho = \frac{2\epsilon}{j} &= \frac{2q}{j(\frac{1}{a}+\frac{1}{c}+\frac{4}{b})}s^2 \leq \\ &\leq \frac{2q}{j}u \cdot Kb^2 \cdot s^2 = \frac{2q}{j}u \cdot K \cdot 16j^2(\frac{1}{4}-s^2)^2 s^2 = \\ &= 32qu \cdot Kj \cdot ((\frac{1}{4}-s^2)s)^2. \end{aligned}$$

We need two more upper bounds: the maximum of the function $((\frac{1}{4}-s^2)s)^2$ when $|s| < \frac{1}{2}$ turns out to be $\frac{1}{432}$; and the function $Kj = \frac{K}{1+2\sqrt{K}}$, which is monotonically increasing, has its maximum when $K = k_1$, so we can define:

$$v = \max_{k_0 \leq K \leq k_1} Kj = \frac{k_1}{1+2\sqrt{k_1}}. \tag{A.14}$$

We now combine Equations A.7, A.14 and A.14, and obtain the upper bound on the relative error:

$$\begin{aligned} \rho &\leq \frac{32}{432}quv = \frac{2}{27}\frac{qk_1}{2\sqrt{k_0}(1+2\sqrt{k_1})} \\ &= \frac{1}{27} \cdot \frac{qk_1}{\sqrt{k_0}(1+2\sqrt{k_1})} \end{aligned} \tag{A.15}$$

where q is defined by Equation A.11.

A.2.2 Error Bounds on Specific Intervals

As mentioned already, the interval $[\frac{1}{9}, 1]$ is the largest that makes $p = \frac{1}{4}$. Continuing with Equations A.11, A.13, and A.14, we get:

$$q = \frac{1}{p} - \frac{1}{4} = \frac{4}{15} \quad ; \quad \sqrt{k_0} = \frac{1}{3} \quad ; \quad \sqrt{k_1} = 1.$$

And so, Equation A.15 becomes:

$$\rho \leq \frac{1}{27} \cdot \frac{4}{15} \cdot 3 \cdot \frac{1}{1 + 2 \cdot 1} = \frac{4}{405} < \frac{1}{100}$$

which proves the claim of Theorem 2.

Let us now consider two other ranges of K: if we decrease k_1 to $\frac{1}{2}$ without changing $k_0 = \frac{1}{9}$, then only the value of $v = \max\{Kj\}$ decreases, and the upper bound on ρ falls only to about $\frac{1}{160}$. To make the interval symmetric again, we must have $k_0 = (4 - \frac{1}{\sqrt{k_1}})^{-2} \approx 0.15$. In this case we get $p = (1 - \sqrt{\frac{1}{2}})^2 \approx 0.086$, $q = \frac{4p}{4-p} \approx 0.088$, $u = \frac{1}{2\sqrt{0.15}} \approx 0.129$, and $v = \frac{\frac{1}{2}}{1+2\sqrt{\frac{1}{2}}} \approx 0.207$. So the bound on the relative error in this smaller range of K falls to $\rho \approx 0.00174 < \frac{1}{575}$.

A.3 Subdivision

We see that the error bounds cannot be kept small if K strays too far away from $\frac{1}{4}$. An even worse problem manifests itself when $K \leq \frac{1}{36}$, because then $j = \frac{1}{1+2\sqrt{K}} \geq \frac{3}{4}$ and $z = \frac{4}{3}j \geq 1$. This means that the points P and Q are extrapolated beyond B. The resulting Bezier curve may fall outside the triangle, will have inflection points, and could even be self-intersecting — all of which make for a very bad representation of the conic (Figure A.3).

The solution is to divide the conic into two parts, and represent each part by a separate Bezier curve. All we need is that after this division, the weight of each part will be closer to $\frac{1}{4}$. Of course, we also need that a small number of steps will produce a weight close to $\frac{1}{4}$, for example, to within the range $[\frac{1}{9}, 1]$.

The subdivision replaces the triangle ABC by the two triangles ALM and MNC (Figure A.2.) The new triangles are valid for representing the conic, because the

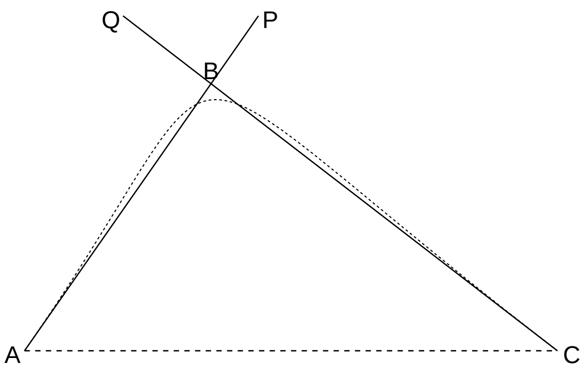

Figure A.3: The Bezier curve resulting from $K = 0.001, z = 1.254$.

straight line LN is indeed tangent to the conic at M. The weight factor in both the new triangles turns out to be the same:

$$K' = \frac{hm}{l^2} = \frac{\sqrt{K}}{1 + 2\sqrt{K}} = \frac{1-j}{2}.$$

It is easy to see that for all $K > \frac{1}{4}$, $K > K' > \frac{1}{4}$. In fact, $K' < \frac{1}{2}$ for all K. Thus, large values of K are immediately reduced to very small ones, that are still above $\frac{1}{4}$, just as claimed in part 2 of Theorem 3.

On the other side of the weight spectrum, small values of K will also increase toward $\frac{1}{4}$, as $K < \frac{1}{4}$ implies $K < K' < \frac{1}{4}$. In fact, it is easy to see that if $K < \frac{1}{9}$ then

$$K' = \frac{\sqrt{K}}{1 + 2\sqrt{K}} < \frac{\sqrt{K}}{1 + 2 \cdot \frac{1}{3}} = \frac{3}{5}\sqrt{K}.$$

For very small values of K, K' is very close to $\sqrt{K}$ (from below), and so it is clear that the convergence towards $\frac{1}{9}$ will take a number of steps proportional to $\log(\log(K))$.

Note that the inverse relationship is

$$K = (\frac{K'}{1 - 2\sqrt{K'}})^2.$$

Let $K(0)$ be $\frac{1}{9}$ and define $K(m+1)$, recursively, as $(\frac{K(m)}{1-2\sqrt{K(m)}})^2$. Clearly, $K(m)$ is the largest value of K which requires at least m steps to reach $\frac{1}{9}$. Looking at the first few values of $K(m)$, we see that this function decreases very rapidly with m: $K(1) = \frac{1}{49}, K(2) = \frac{1}{2209} = 4.5 \cdot 10^{-4}$, and $K(3) = 2.05 \cdot 10^{-7}$. As a matter of fact, for $m > 2$, the value of $K(m)$ is almost equal to $10^{0.836 \cdot 2^m}$. In particular, since $K(3) < 10^{-6}$, we have proved part 2 of Theorem 3.

In any case, it should be noted that when the value of K is very small the conic is a hyperbola that is almost indistinguishable from the straight lines AB and BC. So, in practice, any K that is smaller than 10^{-6} or 10^{-6} is essentially a singular case. For the reasonable values of K, two subdivision steps would always be enough.

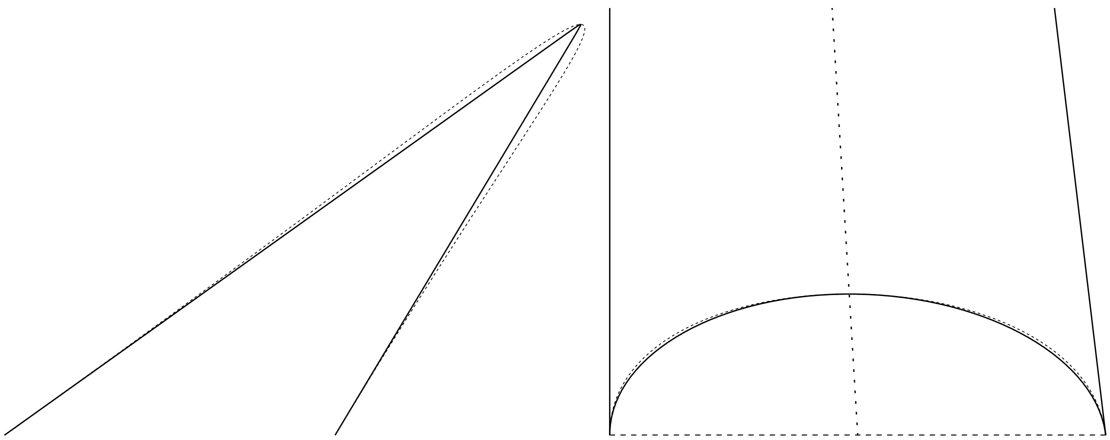

Figure A.4: A hyperbola with $K = 0.00001$ and an ellipse with $K = 200$.

Figure A.4 shows two examples of the hardest cases possible: On the left, a hyperbola with $K = 10^{-5}$, practically indistinguishable from the triangle in which it is inscribed; and on the right, an ellipse with $K = 200$, inside a triangle whose height-to-width ratio is 25 to 3. The hyperbola required two subdivisions (into four parts), and the ellipse only one. In both cases, direct measurement of the errors shows that after two subdivisions, the ratio of the maximal error to the length $\|DM\|$ becomes less than 10^{-6}. This means that, on a device that prints 400 dots per inch, the error of a curve whose size is 25 inches—twice the size of a regular page—is less than one hundredth of a dot, and therefore quite insignificant.

Bibliography

[AP91] **Gideon Avrahami and Vaughan Pratt,** Sub-Pixel Edge Detection in Character Digitization. In *Raster Imaging And Digital Typography II,* edited by Robert A. Morris and Jacques André, pp. 54-64. Cambridge University Press, 1991.

[BBB87] **Richard H. Bartels, John C. Beatty and Brian A. Barsky,** *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling.* Morgan Kaufmann, 1987.

[B79] **Fred L. Bookstein,** Fitting Conic Sections to Scattered Data. *Computer Graphics and Image Processing,* No. 9, pp. 56-71, 1979.

[E79] **Shimon Even** *Graph Algorithms,* p.18. Computer Science Press, 1979.

[F90] **Gerald Farin** *Curves and Surfaces for Computer Aided Geometric Design—A Practical Guide,* Chapter 14. Academic Press, 1990.

[HS92] **Robert M. Haralick and Linda G. Shapiro** *Computer And Robot Vision,* Chapter 11. Addison-Wesley, 1992.

[MI92] **M. Douglas McIlroy,** Getting Raster Ellipses Right. *ACM Transactions on Graphics*, Vol. 11, No. 3, July 1992, pp. 259-275.

[C83] **John F. Canny,** Finding Edges and Lines in Images. *Tech Report 720*, MIT AI Lab, June 1983.

[D88] **R. Deriche,** Using Canny's Criteria to Derive a Recursively Implemented Optimal Edge Detector. *International Journal of Computer Vision*, Vol. 1, No. 2, 1988.

[HM85] **Andres Huerta and Gerard Medioni,** Edge Detection With Subpixel Precision. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, San Francisco, CA. 1985.

[M92] **Bruce Fraser,** Fontographer. *MacUser*, October 1992, p. 56.

[N87] **Vishvit S. Nalwa,** Edge-Detector Resolution Improvement by Image Interpolation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-9, No. 3, May 1987.

[OS91] **John P. Oakley and Richard T. Shann,** Efficient method for finding the position of object boundaries to sub-pixel precision. *Image and Vision Computing*, Vol. 9, No. 4, August 1991.

[P83] **Theo Pavlidis,** Curve Fitting with Conic Splines. *ACM Transactions on Graphics*, Vol. 2, No. 1, Jan. 1983, pp. 1-31.

[PS83] **Michael Plass and Maureen Stone** Curve-Fitting with Piecewise Parametric Cubics. *Computer Graphics* Vol. 17, No. 3 (Siggraph '83), pp. 229-239, July 1983.

[Ad86] *PostScript Language Reference Manual.* Adobe Systems Inc. Addison-Wesley, 1986.

[R88] **Richard Rubinstein.** *Digital Typography*, Chapter 4. Addison-Wesley, 1988.

[P85] **Vaughan Pratt,** Techniques for Conic Splines. *Computer Graphics* Vol. 19, No. 3 (Siggraph '85), pp. 151-159, July 1985.

[P87] **Vaughan Pratt,** Direct least-Squares Fitting of Algebraic Surfaces. *Computer Graphics* Vol. 21, No. 4 (Siggraph '87), pp. 145-152, July 1987.

[W72] **G. Westheimer,** Visual Acuity and Spatial Modulation Thresholds. In *Visual Psychophysics*, pp. 171-187. (*Handbook of sensory physiology,* Vol 7, Part 4.) Edited by Dorothea Jameson and Leo M. Hurvich, Springer-Verlag 1972.