# 5

# More Image Features

There was an old man of West Dumpet
Who possessed a large nose like a trumpet

Edward Lear, *The Book of Nonsense*

This chapter develops further the discussion on image features, introducing more features and related detection algorithms.

## Chapter Overview

**Section 5.1** introduces *grouping* and *model fitting*, and their relation.

**Section 5.2** presents the *Hough transform*, a class of algorithms for locating lines and curves in images.

**Section 5.3** describes three algorithms for *fitting ellipses* to noisy image data.

**Section 5.4** introduces *deformable contours*, a special class of algorithms for curve detection and description.

**Section 5.5** tackles the problem of forming groups of line segments likely to belong to the same object.

## What You Need to Know to Understand this Chapter

- Working knowledge of the previous chapters.
- Least-squares parameter fitting (Appendix, sections A.6 and A.7).

## 5.1 Introduction: Line and Curve Detection

Lines and curves are important features in computer vision because they define the contours of objects in the image. This chapter presents methods to detect lines, ellipses, and general closed contours. Here is a statement of our task.

---

**Problem Statement: Line and Curve Detection**

Given the output of an edge detector run on an image, $I$, find all the instances of a given curve (e.g., a line or an ellipse) or parts thereof (e.g., line segments or arcs of ellipse) in $I$.

---

Strictly speaking, the detection of image lines can be performed directly; that is, by *template matching*: One can look for the peaks of the convolution between the image and a set of masks matched to long, linear edges in all possible orientations. There are at least two disadvantages: First, accurate line location requires a large set of masks; second, we again run into all the problems connected to filter design that we discussed in Chapter 4. Trying to apply template matching to curves worsen the problem. Therefore, we follow an alternative plan: We start with CANNY_EDGE_DETECTOR, and we feed the resulting edge image to the line or ellipse detectors. We shall depart from this plan only when looking for general-shape, closed image contours (section 5.4), which will be fit to intensity images directly.

Under this assumption, line and curve detection splits logically into two subproblems:

**Grouping:** Which image points compose each instance of the target curve in the image?

**Model fitting:** Given a set of image points probably belonging to a single instance of the target curve, find the best curve interpolating the points.

In the previous chapter, we have seen that edge detection procedures like HYSTERESIS_THRESH, for example, solve the grouping problem by providing an ordered list of edge points (or *chain*). Let us now understand the problem of *model fitting*. Assume we know that certain image points lie, say, on a line. Because of pixelization and errors introduced by image acquisition and edge detection, there is no line going *exactly* through all the points; we must look for the best compromise line we can find. So we write the equation of a generic line (the *mathematical model*), $\mathbf{a}^\top \mathbf{x} = ax + by + c = 0$, and look for the parameter vector, $\mathbf{a}_o$, which results in a line going as near as possible to each image point. The vector $\mathbf{a}_o$ is computed by defining a distance function, $D$, between the line and the set of image points, and looking for the minimum of $D$ over all possible $\mathbf{a}$. Very often, $D$ is a squared distance, and finding $\mathbf{a}_o$ implies solving a least squares problem.

This chapter introduces algorithms for curve fitting and grouping, as well as algorithms that perform both grouping and model fitting at the same time, but are suitable only for lines and simple curves.

## 5.2 The Hough Transform

The Hough transform (henceforth called HT) was introduced to detect complex patterns of points in binary images and became quickly a very popular algorithm to detect lines and simple curves. The key idea is to *map a difficult pattern detection problem* (finding instances of a given curve) *into a simple peak detection problem* in the space of the parameters of the curve. We start with an image of contour points, such as the output of an edge detector. The contour points needs not to be grouped in edge chains.

### 5.2.1 The Hough Transform for Lines

We begin by introducing the two basic steps of HT work using the case of lines:

1. *Transform line detection into a line intersection problem.* Any line, $y = mx + n$, is identified by a unique parameter pair, $(m, n)$. Therefore, the line is represented by a point in the $m, n$ plane (the *parameter space*). Conversely, any point $\mathbf{p} = [x, y]^\top$ in the image corresponds to a line $n = x(-m) + y$ in parameter space, which, as $m$ and $n$ vary, represents all possible image lines through $\mathbf{p}$. Therefore, a line defined by $N$ collinear image points, $\mathbf{p}_1 \ldots \mathbf{p}_N$, is identified in parameter space by the intersection of the lines associated with $\mathbf{p}_1 \ldots \mathbf{p}_N$, as illustrated in Figure 5.1 for $N = 2$.
2. *Transform line intersection in a simple peak detection problem*, or search for a maximum. Imagine to divide the $m, n$ plane into a finite grid of cells, the resolution of which depends on the accuracy we need, and to associate a counter, $c(m, n)$, initially set to zero, to each cell. Assume for simplicity that the image contains only one line, $(m', n')$, formed by points $\mathbf{p}_1 \ldots \mathbf{p}_N$. For each image point, $\mathbf{p}_i$, increment

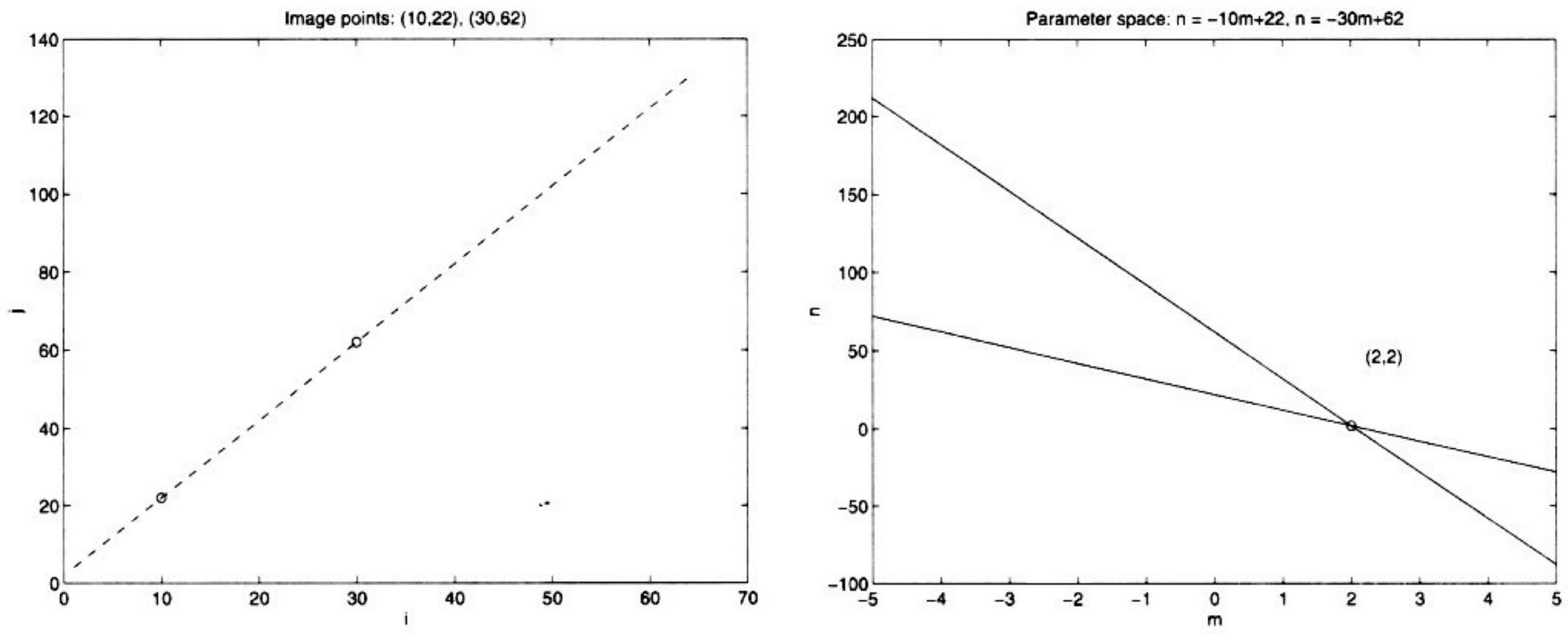



**Figure 5.1** Illustration of the basic idea of the Hough transform for lines. The two image points (a) are mapped onto two lines in parameter space (b). The coordinates of the intersection of these lines are the parameters $(m, n)$ of the image line through the two points.

all counters on the corresponding line in parameter space. All the parameter-space lines, $l_1 \ldots l_N$, associated to $\mathbf{p}_1 \ldots \mathbf{p}_N$, go through $(m', n')$, so that $c(m', n') = N$. Any other counter on $l_1 \ldots l_N$ is 1. Therefore, the image line is identified simply by the peak of $c(m, n)$ in parameter space.

To make things work, several (if straightforward) extensions are necessary.

***Keeping the Parameter Space Finite.*** Both $m$ and $n$ can take on values in $[-\infty, \infty]$, which implies that we cannot sample the whole parameter space. To minimize the risk of missing interesting parameter ranges, we can sample wide intervals for both $m$ and $n$, but at the price of reducing resolution, as there is a limit to the size of the discrete parameter space that we can search in acceptable time. Moreover, the $(m, n)$ parametrization does not capture the bundle $x = k$, with $k$ a constant. The *polar representation* $\rho = x \cos\theta + y \sin\theta$ where $\rho$ represents the distance between the image origin and the line, and $\theta$ the line orientation, solves both problems: The intervals of variation of $\rho$ and $\theta$ are finite, and any line can be represented.

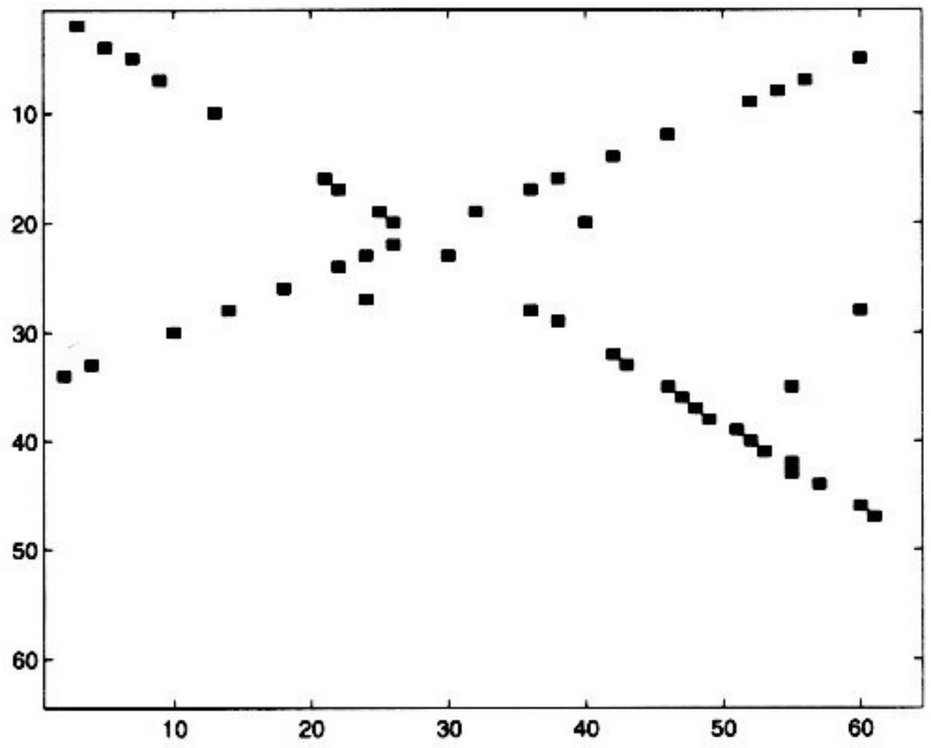
☞ Notice that an image point is now represented by a *sinusoid*, not a line, in parameter space.

***Simultaneous Detection of Multiple Lines.*** Edge images may contain many lines. To find them all, you simply look for all local maxima of $c(m, n)$.

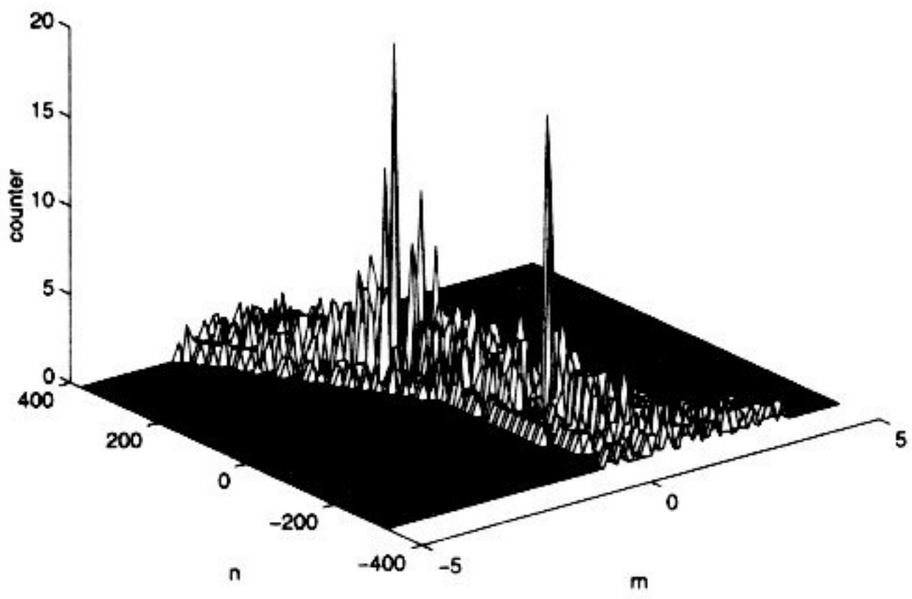
***Effect of Nonlinear Contours.*** Edge images usually contain points not belonging to any line, for instance curved contours, or just noise introduced by the edge detector. These points result in a spread of low, random counter values throughout the parameter space. Consequently, a multitude of local, noisy peaks appears, and we must divide the peaks created by noise from those identifying lines. The simplest way to achieve this is to threshold $c(m, n)$.

***Noise.*** Because of pixelization and the limited accuracy of edge detection, not all the parameter-space lines corresponding to the points of an image line intersect at the same point. Consequently, the image points do not contribute to one counter only, but to *several* counters within a small neighborhood of the correct one, so that the peak identifying the line is spread over that neighborhood. This phenomenon is reinforced by the presence of noisy points. Depending on the resolution of the parameter space and the accuracy required, one can estimate the true parameters just as the local maximum $(m', n')$, or as a weighted average of the values $(m, n)$ in a neighborhood of $(m', n')$ such that $c(m, n) > \tau c(m', n')$, where $\tau$ is a fixed fraction (e.g., 0.9) and the weights are proportional to the counters' values.

As an example, Figure 5.2 (a) shows a synthetic $64 \times 64$ image of two lines. Only a subset of the lines' points are present, and spurious points appear at random locations. Figure 5.2 (b) shows the counters in the associated $(m, n)$ parameter space.




(a)




(b)

**Figure 5.2** (a) An image containing two lines, sampled irregularly, and several random points. (b) Plot of the counters in the corresponding parameter space (how many points contribute to each cell $(m, n)$). Notice that the main peaks are obvious, but there are many secondary peaks.

We are now ready for the following algorithm:

---

**Algorithm HOUGH_LINES**

The input is $E$, a $M \times N$ binary image in which each pixel $E(i, j)$ is 1 if an edge pixel, 0 otherwise. Let $\boldsymbol{\rho}_d, \boldsymbol{\theta}_d$ be the arrays containing the discretized intervals of the $\rho, \theta$ parameter spaces ($\rho \in [0, \sqrt{M^2 + N^2}]$, $\theta \in [0, \pi]$), and $R, T$, respectively, their number of elements.

1. Discretize the parameter spaces of $\rho$ and $\theta$ using sampling steps $\delta\rho$, $\delta\theta$, which must yield acceptable resolution and manageable size for $\boldsymbol{\rho}_d$ and $\boldsymbol{\theta}_d$.
2. Let $A(R, T)$ be an array of integer counters (accumulators); initialize all elements of $A$ to zero.
3. For each pixel, $E(i, j)$, such that $E(i, j) = 1$ and for $h = 1 \ldots T$:
   (a) let $\rho = i \cos \boldsymbol{\theta}_d(h) + j \sin \boldsymbol{\theta}_d(h)$;
   (b) find the index, $k$, of the element of $\boldsymbol{\rho}_d$ closest to $\rho$;
   (c) increment $A(k, h)$ by one.
4. Find all local maxima $(k_p, h_p)$ such that $A(k_p, h_p) > \tau$, where $\tau$ is a user-defined threshold.

The output is a set of pairs $(\boldsymbol{\rho}_d(k_p), \boldsymbol{\theta}_d(h_p))$, describing the lines detected in $E$ in polar form.

---

☞ If an estimate $m_g(\mathbf{p})$ of the edge direction at image point $\mathbf{p}$ is available, and we assume that $m_g(\mathbf{p})$ is also the direction of the line through $\mathbf{p}$, a *unique* cell $(m_g, n_g = y - m_g x)$ can be identified. In this case, instead of the whole line, we increment only the counter at $(m_g, n_g)$; to allow for the uncertainty associated with edge direction estimates, we increment all the cells on a small segment centered in $(m_g, n_g)$, the length of which depends inversely on the

reliability of the direction estimates. This can speed up considerably the construction of the parameter space.

### 5.2.2 The Hough Transform for Curves

The HT is easily generalized to detect curves $y = f(x, \mathbf{a})$, where $\mathbf{a} = [a_1, \ldots, a_P]^\top$ is a vector of $P$ parameters. The basic algorithm is very similar to HOUGH_LINES.

---

**Algorithm HOUGH_CURVES**

The input is as in HOUGH_LINES. Let $y = f(x, \mathbf{a})$ be the chosen parametrization of a target curve.

1. Discretize the intervals of variation of $a_1, \ldots, a_P$ with sampling steps yielding acceptable resolution and manageable size for the parameter space. Let $s_1, \ldots, s_p$ be the sizes of the discretized intervals.
2. Let $A(s_1, s_2 \ldots s_P)$ be an array of integer counters (accumulators), and initialize all its elements to zero.
3. For each pixel $E(i, j)$ such that $E(i, j) = 1$, increment all counters on the curve defined by $y = f(x, \mathbf{a})$ in $A$.
4. Find all local maxima $\mathbf{a}_m$ such that $A(\mathbf{a}_m) > \tau$, where $\tau$ is a user-defined threshold.

The output is a set of vectors, $\mathbf{a}_1 \ldots \mathbf{a}_P$, describing the curve instances detected in $E$.

---

☞ The size of the parameter space increases exponentially with the number of model parameters, and the time needed to find all maxima becomes rapidly unacceptable. This is a serious limitation. In particular, assuming for simplicity that the discretized intervals of all parameters have the same size $N$, the cost of an exhaustive search for a curve with $p$ parameters is proportional to $N^p$. This problem can be tackled by variable-resolution parameter spaces (see Question 5.6).

### 5.2.3 Concluding Remarks on Hough Transforms

The HT algorithm is a *voting algorithm*: Each point "votes" for all combinations of parameters which may have produced it if it were part of the target curve. From this point of view, the array of counters in parameter space can be regarded as a *histogram*. The final total of votes, $c(\mathbf{m})$, in a counter of coordinates $\mathbf{m}$ indicates the relative likelihood of the hypothesis "a curve with parameter set $\mathbf{m}$ exists in the image."

The HT can also be regarded as *pattern matching*: the class of curves identified by the parameter space is the class of patterns. Notice that the HT is more efficient than direct template matching (comparing all possible appearances of the pattern with the image).

The HT has several attractive features. First, as all points are processed independently, it copes well with occlusion (if the noise does not result in peaks as high as those created by the shortest true lines). Second, it is relatively robust to noise, as spurious

points are unlikely to contribute consistently to any single bin, and just generate background noise. Third, it detects multiple instances of a model in a single pass.

The major limitation of the HT is probably the rapid increase of the search time with the number of parameters in the curve's representation. Another limitation is that non-target shapes can produce spurious peaks in parameter space: For instance, line detection can be disturbed by low-curvature circles.

## 5.3 Fitting Ellipses to Image Data

Many objects contain circular shapes, which almost always appear as *ellipses* in intensity images (but see Exercise 5.5); for this reason, ellipse detectors are useful tools for computer vision. The ellipse detectors we consider take an image of edge points in input, and find the best ellipse fitting the points. Therefore this section concentrates on *ellipse fitting*, and *assumes that we have identified a set of image points plausibly belonging to a single arc of ellipse.*

---

**Problem Statement: Ellipse Fitting**

Let $\mathbf{p}_1 \ldots \mathbf{p}_N$ be a set of $N$ image points, $\mathbf{p}_i = [x_i, y_i]^\top$. Let $\mathbf{x} = \left[x^2, xy, y^2, x, y, 1\right]^\top$, $\mathbf{p} = [x, y]^\top$, and

$$f(\mathbf{p}, \mathbf{a}) = \mathbf{x}^\top \mathbf{a} = ax^2 + bxy + cy^2 + dx + ey + f = 0$$

the implicit equation of the generic ellipse, characterized by the parameter vector $\mathbf{a} = [a, b, c, d, e, f]^\top$.

Find the parameter vector, $\mathbf{a}_o$, associated to the ellipse which fits $\mathbf{p}_1 \ldots \mathbf{p}_N$ best in the least squares sense, as the solution of

$$\min_{\mathbf{a}} \sum_{i=1}^{N} [D(\mathbf{p}_i, \mathbf{a})]^2 \tag{5.1}$$

where $D(\mathbf{p}_i, \mathbf{a})$ is a suitable distance.

---

☞ Notice that the equation we wrote for $f(\mathbf{p}, \mathbf{a})$ is really a *generic conic*. We shall have more to say about this point later.

What is a suitable distance? There are two main answers for ellipse fitting, the *Euclidean distance* and the *algebraic distance*.

### 5.3.1 Euclidean Distance Fit

The first idea is to try and minimize the Euclidean distance between the ellipse and the measured points. In this case, problem (5.1) becomes

$$\min_{\mathbf{a}} \sum_{i=1}^{N} \|\mathbf{p} - \mathbf{p}_i\|^2 \tag{5.2}$$

under the constraint that $\mathbf{p}$ belongs to the ellipse:

$$f(\mathbf{p}, \mathbf{a}) = 0.$$

Geometrically, the Euclidean distance seems the most appropriate. Unfortunately, it leads only to an approximate, numerical algorithm. How does this happen? Let us try Lagrange multipliers to solve problem (5.2). We define an objective function

$$L = \sum_{i=1}^{N} \|\mathbf{p} - \mathbf{p}_i\|^2 - 2\lambda f(\mathbf{p}, \mathbf{a}),$$

and set $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} = 0$ which yield

$$\mathbf{p} - \mathbf{p}_i = \lambda \nabla f(\mathbf{p}, \mathbf{a}). \tag{5.3}$$

Since we do not know $\mathbf{p}$, we try to express it as a function of computable quantities. To do this, we introduce two approximations:

1. We consider a first-order approximation of the curve

$$0 = f(\mathbf{p}, \mathbf{a}) \approx f(\mathbf{p}_i, \mathbf{a}) + [\mathbf{p} - \mathbf{p}_i]^\top \nabla f(\mathbf{p}_i, \mathbf{a}). \tag{5.4}$$

2. We assume that the $\mathbf{p}_i$ are close enough to the curve, so that $\nabla f(\mathbf{p}) \approx \nabla f(\mathbf{p}_i)$.

Approximation 2 allows us to rewrite (5.3) as

$$\mathbf{p} - \mathbf{p}_i = \lambda \nabla f(\mathbf{p}_i, \mathbf{a}),$$

which, plugged into (5.4), gives

$$\lambda = \frac{-f(\mathbf{p}_i, \mathbf{a})}{\|\nabla f(\mathbf{p}_i, \mathbf{a})\|^2}.$$

Substituting in (5.3), we finally find

$$\|\mathbf{p} - \mathbf{p}_i\| = \frac{|f(\mathbf{p}_i, \mathbf{a})|}{\|\nabla f(\mathbf{p}_i, \mathbf{a})\|}.$$

This is the equation we were after: It allows us to replace, in problem (5.2), the unknown quantity $\|\mathbf{p} - \mathbf{p}_i\|$ with a function we can compute. The resulting algorithm is as follows:

---

**Algorithm EUCL_ELLIPSE_FIT**

The input is a set of $N$ image points, $\mathbf{p}_1, \ldots, \mathbf{p}_N$. We assume the notation introduced in the problem statement box for ellipse fitting.

1. Set $\mathbf{a}$ to an initial value $\mathbf{a}_0$.

2. Using $\mathbf{a}_0$ as initial point, run a numerical minimization to find the solution of

$$\min_{\mathbf{a}} \sum_{i=1}^{N} \frac{f(\mathbf{p}_i, \mathbf{a})^2}{\|\nabla f(\mathbf{p}_i, \mathbf{a})\|^2}.$$

The output is the solution vector $\mathbf{a}_m$, defining the best-fit ellipse.

---

☞ A reasonable initial value is the solution of the closed-form algorithm discussed next (section 5.3.2).

How satisfactory is EUCL_ELLIPSE_FIT? Only partially. We started with the *true* (Euclidean) distance, the best possible, but were forced to introduce approximations, and arrived at a nonlinear minimization that can be solved only numerically. We are not even guaranteed that the best-fit solution is an ellipse: It could be *any* conic, as we imposed no constraints on $\mathbf{a}$. Moreover, we have all the usual problems of numerical optimization, including how to find a good initial estimate for $\mathbf{a}$ and how to avoid getting stuck in local minima.

☞ The good news, however, is that EUCL_ELLIPSE_FIT can be used for *general conic fitting*. Of course, there is a risk that the result is not the conic we expect (see Further Readings).

A logical question at this point is: If using the *true* distance implies anyway approximations and a numerical solution, can we perhaps find an *approximate* distance leading to a closed-form solution without further approximations? The answer is yes, and the next section explains how to do it.

### 5.3.2 Algebraic Distance Fit

---

**Definition: Algebraic Distance**

The *algebraic distance* of a point $\mathbf{p}$ from a curve $f(\mathbf{p}, \mathbf{a}) = 0$ is simply $|f(\mathbf{p}, \mathbf{a})|$.

---

The algebraic distance is different from the true geometric distance between a curve and a point; in this sense, we start off with an approximation. However, this is the *only* approximation we introduce, since the algebraic distance turns problem (5.1) into a linear problem that we can solve in closed form and with no further approximations.

Problem (5.1) becomes

$$\min_{\mathbf{a}} \sum_{i=1}^{N} |\, \mathbf{x}_i^\top \mathbf{a} \,|^2 \tag{5.5}$$

To avoid the trivial solution $\mathbf{a} = 0$, we must enforce a constraint on $\mathbf{a}$. Of the several constraints possible (see Further Readings), we choose one which forces the solution to be an ellipse:

$$b^2 - 4ac = \mathbf{a}^\top \begin{bmatrix} 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{a} = \mathbf{a}^\top C \mathbf{a} = -1. \tag{5.6}$$

☞ Notice that this can be regarded as a "normalized" version of the elliptical constraint $b^2 - 4ac < 0$, as $\mathbf{a}$ is only defined up to a scale factor.

We can find a solution to this problem *with no approximations*. First, we rewrite problem (5.5) as

$$\min_{\mathbf{a}} \|\mathbf{a}^\top X^\top X \mathbf{a}\| = \min_{\mathbf{a}} \|\mathbf{a}^\top S \mathbf{a}\|, \tag{5.7}$$

where

$$X = \begin{bmatrix} x_1^2 & x_1 y_1 & y_1^2 & x_1 & y_1 & 1 \\ x_2^2 & x_2 y_2 & y_2^2 & x_2 & y_2 & 1 \\ & & \cdots & & & \\ x_N^2 & x_N y_N & y_N^2 & x_N & y_N & 1 \end{bmatrix} \tag{5.8}$$

In the terminology of constrained least squares, $X$ is called the *design matrix*, $S = X^\top X$ the *scatter matrix*, and $C$ the *constraint matrix*. Again using Lagrange multipliers, we obtain that problem (5.7) is solved by

$$S\mathbf{a} = \lambda C \mathbf{a}. \tag{5.9}$$

This is a so-called *generalized eigenvalue problem*, which can be solved in closed form. It can be proven that the solution, $\mathbf{a}_o$, is the eigenvector corresponding to the *only* negative eigenvalue. Most numerical packages will find the solution of problem (5.9) for you, taking care of the fact that $C$ is rank-deficient. The resulting algorithm is very simple.

---

**Algorithm ALG_ELLIPSE_FIT**

The input is a set of $N$ image points, $\mathbf{p}_1, \ldots, \mathbf{p}_N$. We assume the notation introduced in the problem statement box for ellipse fitting.

1. Build the design matrix, $X$, as per (5.8).
2. Build the scatter matrix, $S = X^\top X$.
3. Build the constraint matrix, $C$, as per (5.6).
4. Use a numerical package to compute the eigenvalues of the generalized eigenvalue problem, and call $\lambda_n$ the only negative eigenvalue.

The output is the best-fit parameter vector, $\mathbf{a}_o$, given by the eigenvector associated to $\lambda_n$.

---

Figure 5.3 shows the result of ALG_ELLIPSE_FIT run on an elliptical arc corrupted by increasing quantities of Gaussian noise.

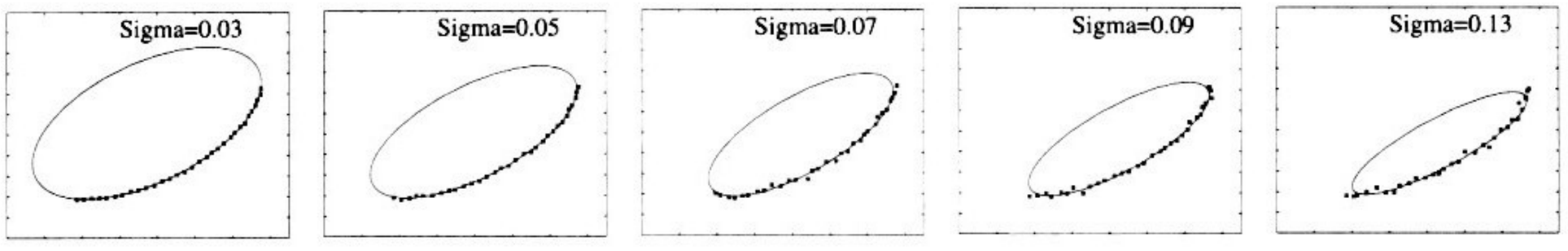



**Figure 5.3** Example of best-fit ellipses found by ALG_ELLIPSE_FIT for the same arc of ellipse, corrupted by increasingly strong Gaussian noise. From left to right, the noise varies from 3% to 20% of the data spread (figure courtesy of Maurizio Pilu, University of Edinburgh).

ALG_ELLIPSE_FIT tends to be *biased towards low-eccentricity solutions*, indeed a characteristic of *all* methods based on the algebraic distance. Informally, this means that the algorithm prefers fat ellipses to thin ellipses, as shown in Figure 5.4. The reason is best understood through the geometric interpretation of the algebraic distance.

---

**Geometric Interpretation of the Algebraic Distance (Ellipses)**

Consider a point $\mathbf{p}_i$ *not* lying on the ellipse $f(\mathbf{p}, \mathbf{a}) = 0$. The algebraic distance, $f(\mathbf{p}_i, \mathbf{a})$, is proportional to

$$Q = 1 - \left[ \frac{(r+d)^2}{r^2} \right],$$

where $r$ is the distance of the ellipse from its center along a line which goes through $\mathbf{p}_i$, and $d$ is the distance of $\mathbf{p}_i$ from the ellipse along the same line (Figure 5.5).

---

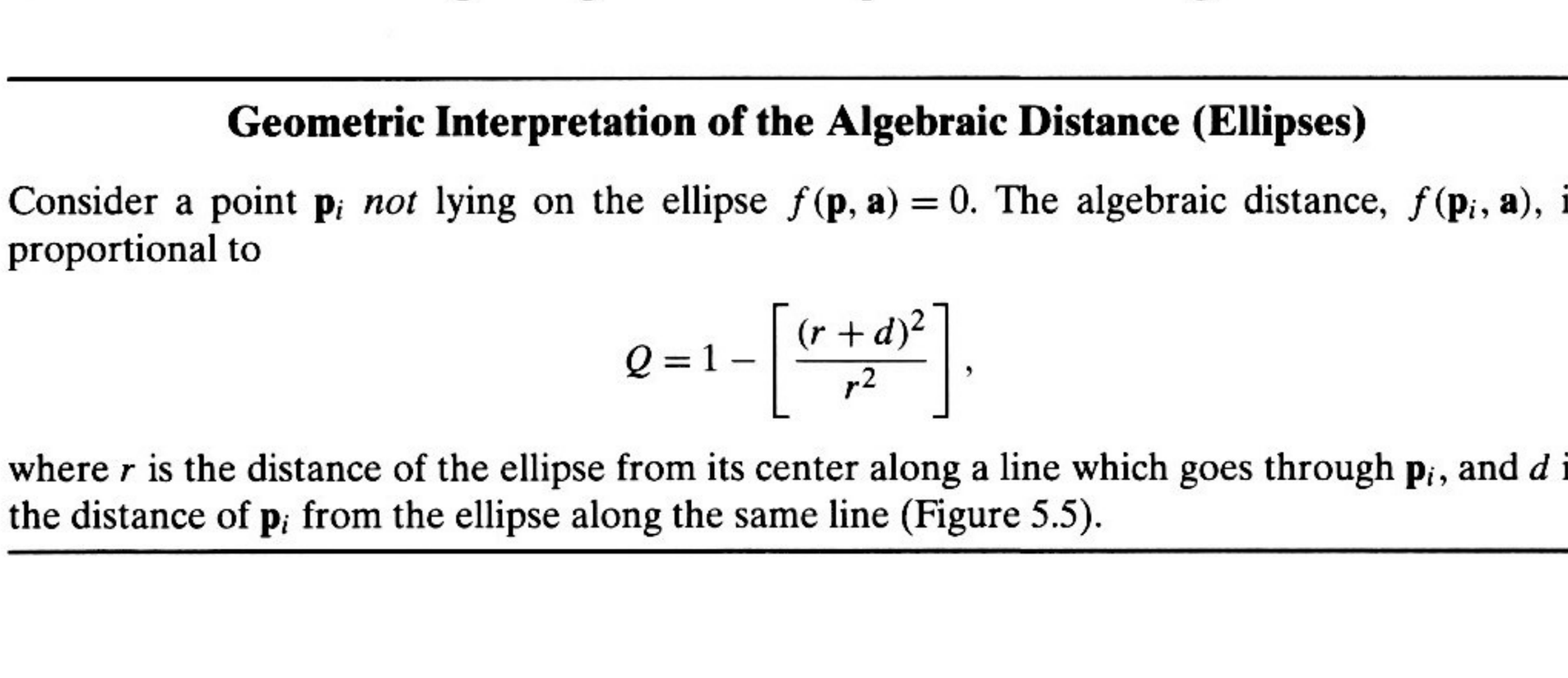


**Figure 5.4** Illustration of the low-eccentricity bias introduced by the algebraic distance. ALG_ELLIPSE_FIT was run on 20 samples covering half an ellipse, spaced uniformly along $x$, and corrupted by different realizations of rather strong, Gaussian noise with constant standard deviation ($\sigma = 0.08$, about 10% of the smaller semiaxis). The best-fit ellipse (solid) is systematically biased to be "fatter" than the true one (dashed).

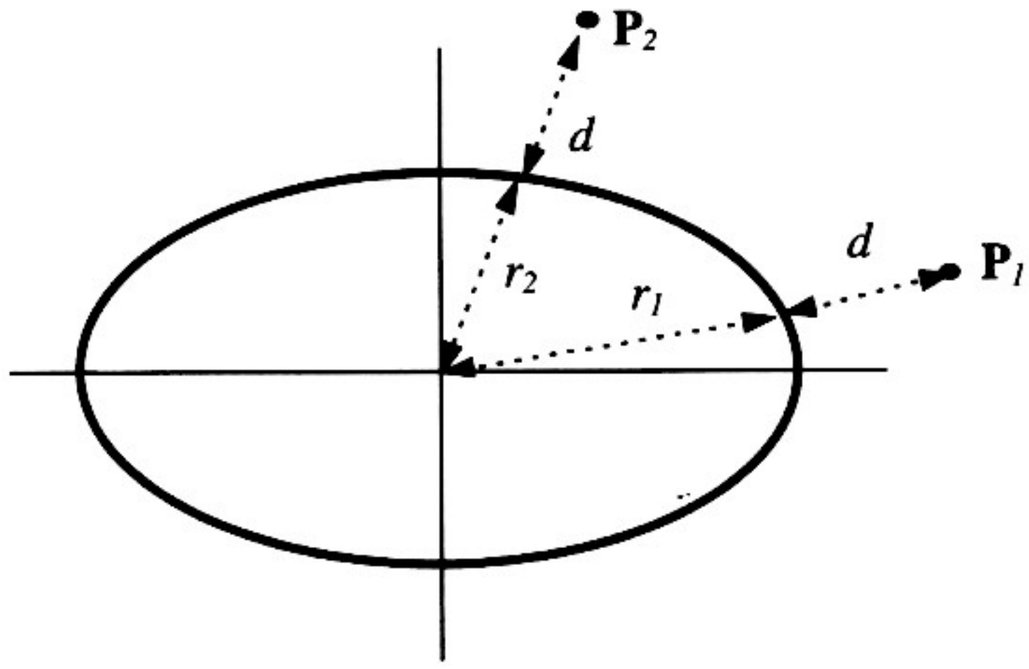



Figure 5.5 Illustration of the distances $d$ and $r$ in the geometric interpretation of the algebraic distance, $Q$. At a parity of $d$, $Q$ is larger at $\mathbf{P}_2$ than at $\mathbf{P}_1$.

☞ Notice that this interpretation is valid for *any* conic. For hyperbolae, the center is the intersection of the asymptotes; for parabolae, the center is at infinity.

For any fixed $d$, $Q$ is maximum at the intersection of the ellipse with its smaller axis (e.g., $\mathbf{P}_2$ in Figure 5.5) and minimum at the intersection of the ellipse with its larger axis (e.g., $\mathbf{P}_1$ in Figure 5.5). Therefore, the algebraic distance is maximum (high weight) for observed points around the flat parts of the ellipse, and minimum (low weight) for observed points around the pointed parts. As a consequence, a fitting algorithm based on $Q$ tends to believe that most data points are concentrated in the flatter part of the ellipse, which results in "fatter" best-fit ellipses.

### 5.3.3 Robust Fitting

One question which might have occurred to you is: Where do the data points for ellipse fitting come from? In real applications, and without *a priori* information on the scene, finding the points most likely to belong to a specific ellipse is a difficult problem. In some cases, it is reasonable to expect that the data points can be selected by hand. Failing that, we rely on edge chaining as described in HYSTERESIS_THRESH. In any case, it is very likely that the data points contain *outliers*.

Outliers are data points which violate the statistical assumptions of the estimator. In our case, an outlier is an edge point *erroneously* assumed to belong to an ellipse arc. Both EUCL_ELLIPSE_FIT and ALG_ELLIPSE_FIT, as least-squares estimators, assume that all data points can be regarded as true points corrupted by additive, Gaussian noise; hence, even a small number of outliers can degrade their results badly. *Robust estimators* are a class of methods designed to tolerate outliers.[1] A robust distance that

[1] Section A.7 in the Appendix gives a succinct introduction to robust estimators.

often works well is the absolute value, which is adopted by the following algorithm for robust ellipse fitting.

---

**Algorithm ROB_ELLIPSE_FIT**

The input is a set of $N$ image points, $\mathbf{p}_1, \ldots, \mathbf{p}_N$. We assume the notation introduced in the problem statement box for ellipse fitting.

1. run ALG_ELLIPSE_FIT, and call its solution $\mathbf{a}_0$.
2. Using $\mathbf{a}_0$ as initial value, run a numerical minimization to find a solution of

$$\min_{\mathbf{a}} \sum_{i=1}^{N} | \mathbf{x}_i^\top \mathbf{a} |$$

The output is the solution, $\mathbf{a}_m$, which identifies the best-fit ellipse.

---

Figure 5.6 illustrates the problems caused by outliers to ALG_ELLIPSE_FIT, and allows you to compare the result of ALG_ELLIPSE_FIT with those of ROB_ELLIPSE_FIT, started from the solution of ALG_ELLIPSE_FIT, in conditions of severe noise (that is, lots of outliers *and* Gaussian noise). Both algorithms were run on 40 points from half an ellipse, spaced uniformly along $x$, and corrupted by different realizations of Gaussian noise with constant standard deviation ($\sigma = 0.05$, about 7% of the smaller semiaxis, $a$). About 20% of the points were turned into outliers by adding a uniform deviate in $[-a, a]$ to their coordinates. Notice the serious errors caused to ALG_ELLIPSE_FIT by the outliers, which are well tolerated by ROB_ELLIPSE_FIT.

### 5.3.4 Concluding Remarks on Ellipse Fitting

With moderately noisy data, ALG_ELLIPSE_FIT should be your first choice. With seriously noisy data, the eccentricity of the best-fit ellipse can be severely underestimated (the more so, the smaller the arc of ellipse covered by the data). If this is a problem for your application, you can try EUCL_ELLIPSE_FIT, starting it from the solution of ALG_ELLIPSE_FIT. With data containing many outliers, the results of both EUCL_ELLIPSE_FIT and ALG_ELLIPSE_FIT will be skewed; in this case, ROB_ELLIPSE_FIT, started from the solution of ALG_ELLIPSE_FIT should do the trick (but you are advised to take a look at the references in section A.7 in the Appendix if robustness is a serious issue for your application). If speed matters, your best bet is ALG_ELLIPSE_FIT alone, assuming you use a reasonably efficient package to solve the eigenvalue problem, and the assumptions of the algorithms are plausibly satisfied.

What "moderately noisy" and "seriously noisy" mean quantitatively depends on your data (number and density of data points along the ellipse, statistical distribution, and standard deviation of noise). In our experience, ALG_ELLIPSE_FIT gives good fits with more than 10 points from half an ellipse, spaced uniformly along $x$, and

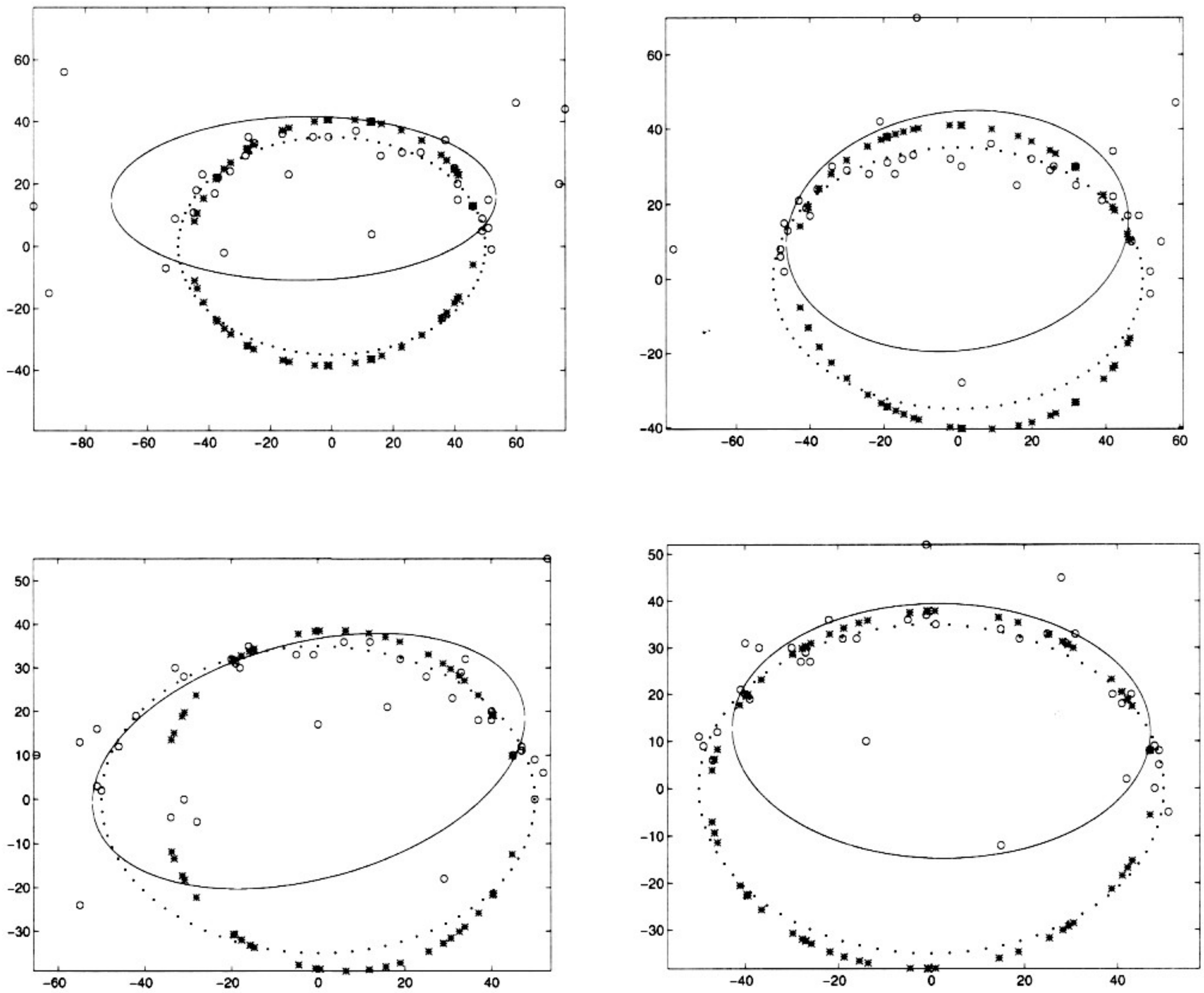




Figure 5.6 Comparison of ALG_ELLIPSE_FIT and ROB_ELLIPSE_FIT when fitting to data severely corrupted by outliers. The circles show the data points, the asterisks suggest the robust fit, the solid line show the algebraic fit, and the dots the true (uncorrupted) ellipse.

corrupted by Gaussian noise of standard deviation up to about 5% of the smaller semiaxis. Section 5.7 suggests Further Readings on the evaluation and comparison of ellipse-fitting algorithms.

## 5.4 Deformable Contours

Having discussed how to fit simple curves, we now move on to the general problem of fitting a curve of arbitrary shape to a set of image edge points. We shall deal with *closed contours* only.

A widely used computer vision model to represent and fit general, closed curves is the *snake*, or *active contour*, or again *deformable contour*. You can think of a snake as an elastic band of arbitrary shape, sensitive to the intensity gradient. The snake is located initially near the image contour of interest, and is attracted towards the target contour by forces depending on the intensity gradient.