CHAPTER 10

Fitting

Sometimes data points (pixels; edge points; and so on) belong together because together they mostly conform to some explicit model. So, for example, many of the points in Figure 35.2 lie on a line (at least by eye). We must then find the model most points conform to. This activity is usually called *fitting*.

Typically, there are three problems in fitting a model to data points. First, given the points that belong to the model, what is the model? Second, which points belong to which model? Finally, how many models are there?

10.1 LEAST SQUARES LINE FITTING

Line fitting is extremely useful. In many applications, objects are characterized by the presence of straight lines. Assume that all the points that belong to a particular line are known, and the parameters of the line must be found. We adopt the notation

$$\overline{u} = \frac{\sum u_i}{k}$$

to simplify the presentation.

10.1.1 Least Squares

Least squares is a fitting procedure with a long tradition (which is the only reason we describe it!). It yields a simple analysis but has a substantial bias. For this approach, we represent a line as $y = ax + b$. At each data point, we have (x_i, y_i); we decide to choose the line that best predicts the measured y coordinate for each measured x coordinate.

This means we want to choose the line that minimises

$$\sum_i (y_i - ax_i - b)^2.$$

By differentiation, the line is given by the solution to the problem

$$\begin{pmatrix} \overline{y^2} \\ \overline{y} \end{pmatrix} = \begin{pmatrix} \overline{x^2} & \overline{x} \\ \overline{x} & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}.$$

Although this is a standard linear solution to a classical problem, it's actually not much help in vision applications because the model is an extremely poor model. The difficulty is that the measurement error is dependent on coordinate frame — we are counting vertical offsets from the line as errors, which means that near vertical lines lead to quite large values of the error and quite funny fits (Figure 10.1). In fact, the process is so dependent on coordinate frame that it doesn't represent vertical lines at all.

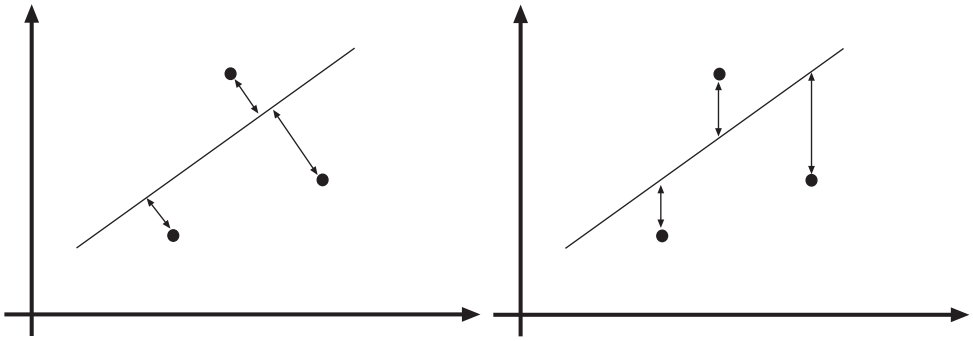

FIGURE 10.1: **Left:** *Total least-squares models data points as being generated by an abstract point along the line to which is added a vector perpendicular to the line. We wish to choose a line that minimizes the sum of distances to tokens measured (as distance usually is!) perpendicular to the line.* **Right:** *Least squares follows the same general outline, but assumes that the error appears only in the y coordinate. This yields a (very slightly) simpler mathematical problem at the cost of a poor fit.*

10.1.2 Total Least Squares

We could work with the actual distance between the point and the line (rather than the vertical distance). This leads to a problem known as *total least squares*. We can represent a line as the collection of points where $ax + by + c = 0$. Every line can be represented in this way, and we can think of a line as a triple of values (a, b, c). Notice that for $\lambda \neq 0$, the line given by $\lambda(a, b, c)$ is the same as the line represented by (a, b, c). In the exercises, you are asked to prove the simple, but extremely useful, result that the perpendicular distance from a point (u, v) to a line (a, b, c) is given by

$$\text{abs}(au + bv + c) \text{ if } a^2 + b^2 = 1.$$

In our experience, this fact is useful enough to be worth memorizing. To minimize the sum of perpendicular distances between points and lines, we need to minimize

$$\sum_i (ax_i + by_i + c)^2,$$

where $a^2 + b^2 = 1$ and C is some normalizing constant of no interest. Thus, a maximum-likelihood solution is obtained by maximizing this expression. Now using a Lagrange multiplier λ, we have a solution if

$$\begin{pmatrix} \overline{x^2} & \overline{xy} & \overline{x} \\ \overline{xy} & \overline{y^2} & \overline{y} \\ \overline{x} & \overline{y} & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \lambda \begin{pmatrix} 2a \\ 2b \\ 0 \end{pmatrix}$$

This means that

$$c = -a\overline{x} - b\overline{y}$$

and we can substitute this back to get the eigenvalue problem

$$\begin{pmatrix} \overline{x^2} - \overline{x}\,\overline{x} & \overline{xy} - \overline{x}\,\overline{y} \\ \overline{xy} - \overline{x}\,\overline{y} & \overline{y^2} - \overline{y}\,\overline{y} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \mu \begin{pmatrix} a \\ b \end{pmatrix}$$

Because this is a 2D eigenvalue problem, two solutions up to scale can be obtained in closed form (for those who care - it's usually done numerically!). The scale is obtained from the constraint that $a^2 + b^2 = 1$. The two solutions to this problem are lines at right angles, and one maximises the sum of squared distances and the other minimises it.

10.2 ROBUSTNESS

All of the line fitting methods described involve squared error terms. This can lead to poor fits in practice because a single wildly inappropriate data point can give errors that dominate those due to many good data points; these errors can result in a substantial bias in the fitting process (Figure 10.2). It is difficult to avoid such data points — usually called *outliers* — in practice. Errors in collecting or transcribing data points is one important source of outliers. Another common source is a problem with the model — perhaps some rare but important effect has been ignored or the magnitude of an effect has been badly underestimated. Finally, errors in correspondence are particularly prone to generating outliers. Practical vision problems usually involve outliers.

One approach to this problem puts the model at fault: The model predicts these outliers occurring perhaps once in the lifetime in the universe, and they clearly occur much more often than that. The natural response is to improve the model either by giving the noise "heavier tails" (Section 10.2.1) or by allowing an explicit outlier model. The second strategy requires a study of missing data problems — we don't know which point is an outlier and which isn't — and we defer discussion until Section **??** in the following chapter. An alternative approach is to search for points that appear to be good (Section 25.2.2).

10.2.1 M-estimators

The difficulty with modeling the source of outliers is that the model might be wrong. Generally, the best we can hope for from a probabilistic model of a process is that it is quite close to the right model. Assume that we are guaranteed that our model of a process is close to the right model — say, the distance between the density functions in some appropriate sense is less than ϵ. We can use this guarantee to reason about the design of estimation procedures for the parameters of the model. In particular, we can choose an estimation procedure by assuming that nature is malicious and well informed about statistics. These are generally sound assumptions for any enterprise; the world is full of opportunities for painful and expensive lessons in practical statistics. In this line of reasoning, we assess the goodness of an estimator by assuming that somewhere in the collection of processes close to our model is the real process, and it just happens to be the one that makes the estimator produce the worst possible estimates. The best estimator is the one that behaves best on the worst distribution close to the parametric model. This is a criterion that can be used to produce a wide variety of estimators.

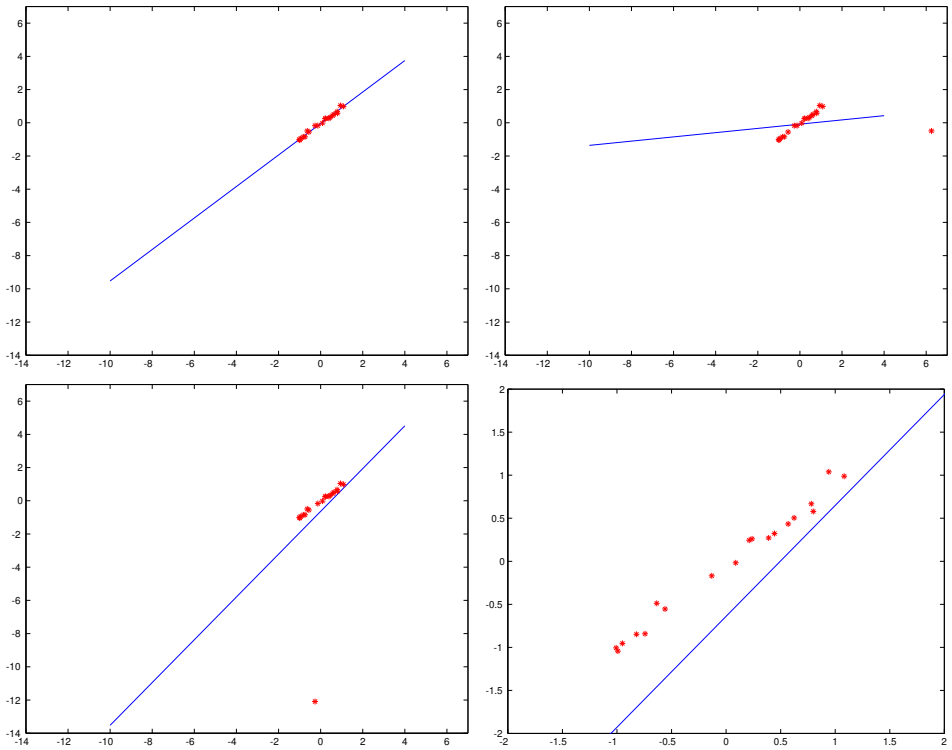

FIGURE 10.2: *Least-squares line fitting is extremely sensitive to outliers, both in x and y coordinates. At the* **top left**, *a good least-squares fit of a line to a set of points.* **Top right** *shows the same set of points, but with the x coordinate of one point corrupted. In this case, the slope of the fitted line has swung wildly.* **Bottom left** *shows the same set of points, but with the y-coordinate of one point corrupted. In this particular case, the x intercept has changed. These three figures are on the same set of axes for comparison, but this choice of axes does not clearly show how bad the fit is for the third case;* **bottom right** *shows a detail of this case — the line is clearly a bad fit.*

An *M-estimator* estimates parameters by minimizing an expression of the form

$$\sum_i \rho(r_i(\mathbf{x}_i, \theta); \sigma),$$

where θ are the parameters of the model being fitted and $r_i(\mathbf{x}_i, \theta)$ is the residual error of the model on the ith data point. Generally, $\rho(u; \sigma)$ looks like u^2 for part of its range and then flattens out. A common choice is

$$\rho(u; \sigma) = \frac{u^2}{\sigma^2 + u^2}.$$

The parameter σ controls the point at which the function flattens out; we have plotted a variety of examples in Figure 10.3. There are many other M-estimators

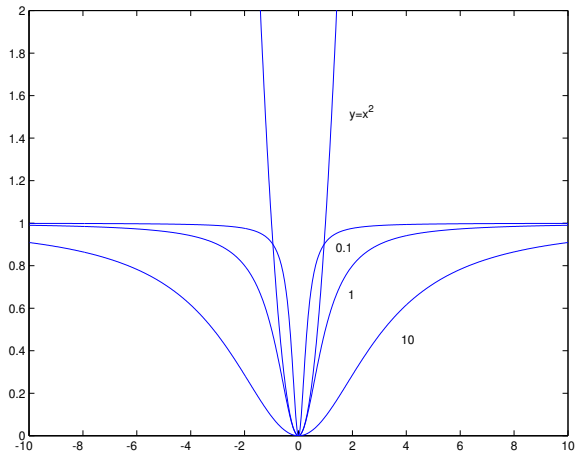

FIGURE 10.3: *The function* $\rho(x;\sigma) = x^2/(\sigma^2 + x^2)$*, plotted for* $\sigma^2 = 0.1$*,* 1*, and* 10*, with a plot of* $y = x^2$ *for comparison. Replacing quadratic terms with* ρ *reduces the influence of outliers on a fit — a point that is several multiples of* σ *away from the fitted curve is going to have almost no effect on the coefficients of the fitted curve because the value of* ρ *will be close to* 1 *and will change extremely slowly with the distance from the fitted curve.*

available. Typically, they are discussed in terms of their *influence function*, which is defined as

$$\frac{\partial \rho}{\partial \theta}.$$

This is natural because our criterion is

$$\sum_i \rho(r_i(\mathbf{x}_i, \theta); \sigma) \frac{\partial \rho}{\partial \theta} = 0.$$

For the kind of problems we consider, we would expect a good influence function to be antisymmetric — there is no difference between a slight overprediction and a slight underprediction — and to tail off with large values — because we want to limit the influence of the outliers.

There are two tricky issues with using M-estimators. First, the extremization problem is non-linear and must be solved iteratively. The standard difficulties apply: There may be more than one local minimum, the method may diverge, and the behavior of the method is likely to be quite dependent on the start point. A common strategy for dealing with this problem is to draw a subsample of the data set, fit to that subsample using least squares, and use this as a start point for the fitting process. We do this for a large number of different subsamples — enough to ensure that there is a high probability that in that set there is at least one that consists entirely of good data points.

Second, as Figures 10.4 and 10.5 indicate, the estimators require a sensible estimate of σ, which is often referred to as *scale*. Typically, the scale estimate is

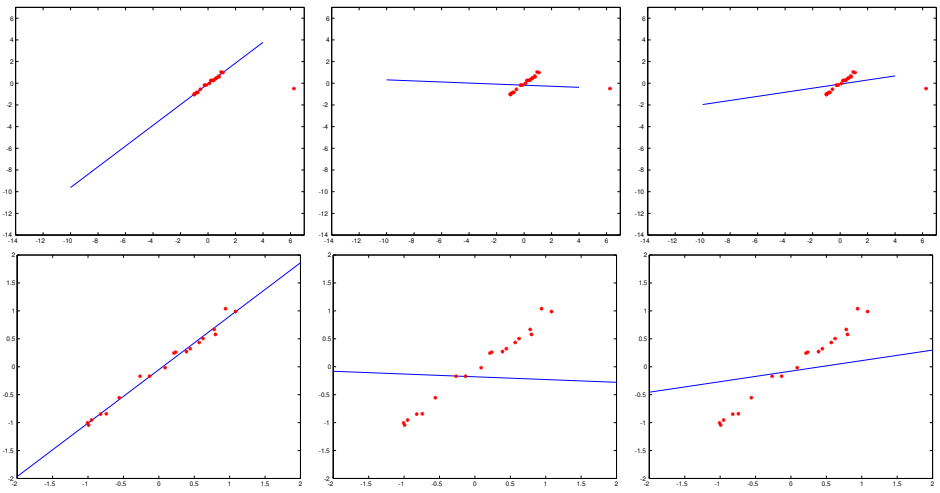

FIGURE 10.4: *The* **top row** *shows lines fitted to the second dataset of Figure 10.2 using a weighting function that deemphasizes the contribution of distant points (the function ϕ of Figure 10.3). On the* **left**, *μ has about the right value; the contribution of the outlier has been down-weighted, and the fit is good. In the* **center**, *the value of μ is too small so that the fit is insensitive to the position of all the data points, meaning that its relationship to the data is obscure. On the* **right**, *the value of μ is too large, meaning that the outlier makes about the same contribution that it does in least-squares. The* **bottom row** *shows closeups of the fitted line and the non-outlying data points for the same cases.*

supplied at each iteration of the solution method; a popular estimate of scale is

$$\sigma^{(n)} = 1.4826 \text{ median}_i \, |r_i^{(n)}(x_i; \theta^{(n-1)})| \, .$$

An M-estimator can be thought of as a trick for ensuring that there is more probability in the tails than would otherwise occur with a quadratic error. The function that is minimized looks like distance for small values of $\mathbf{x}$ — thus, for valid data points, the behavior of the M-estimator should be rather like maximum likelihood — and like a constant for large values of $\mathbf{x}$ — meaning that a component of probability is given to the tails of the distribution. The strategy of the previous section can be seen as an M-estimator, but with the difficulty that the influence function is discontinuous, meaning that obtaining a minimum is tricky.

10.2.2 RANSAC

An alternative to modifying the generative model to have heavier tails is to search the collection of data points for good points. This is quite easily done by an iterative process: First, we choose a small subset of points and fit to that subset, then we see how many other points fit to the resulting object. We continue this process until we have a high probability of finding the structure we are looking for.

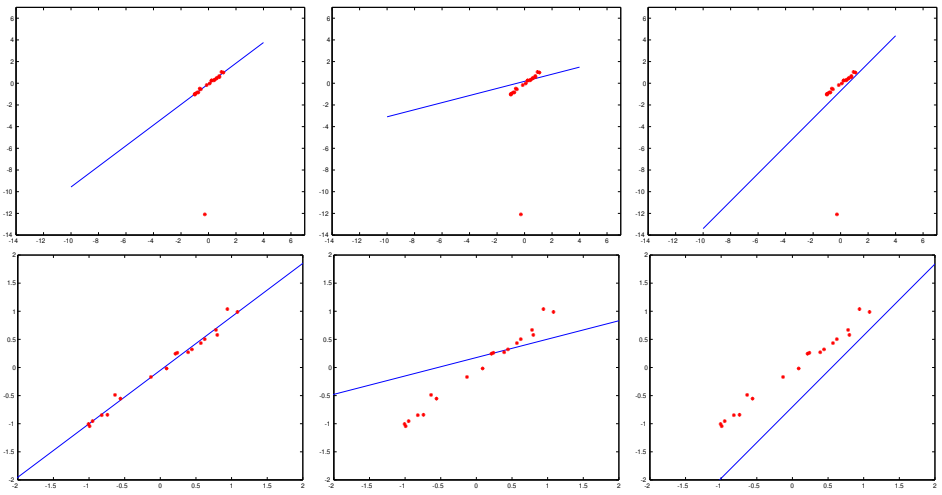

FIGURE 10.5: *The* **top row** *shows lines fitted to the third dataset of Figure 10.2 using a weighting function that deemphasizes the contribution of distant points (the function ϕ of Figure 10.3). On the* **left**, *μ has about the right value; the contribution of the outlier has been down-weighted, and the fit is good. In the* **center**, *the value of μ is too small, so that the fit is insensitive to the position of all the data points, meaning that its relationship to the data is obscure. On the* **right**, *the value of μ is too large, meaning that the outlier makes about the same contribution that it does in least-squares. The* **bottom row** *shows closeups of the fitted line and the non-outlying data points, for the same cases.*

For example, assume that we are fitting a line to a data set that consists of about 50% outliers. If we draw pairs of points uniformly and at random, then about a quarter of these pairs will consist entirely of good data points. We can identify these good pairs by noticing that a large collection of other points lie close to the line fitted to such a pair. Of course, a better estimate of the line could then be obtained by fitting a line to the points that lie close to our current line.

This approach leads to an algorithm — search for a random sample that leads to a fit on which many of the data points agree. The algorithm is usually called `RANSAC`, for RANdom SAmple Consensus, and is displayed in Algorithm 25.1. To make this algorithm practical, we need to choose three parameters.

The Number of Samples Required Our samples consist of sets of points drawn uniformly and at random from the data set. Each sample contains the minimum number of points required to fit the abstraction we wish to fit. For example, if we wish to fit lines, we draw pairs of points; if we wish to fit circles, we draw triples of points, and so on. We assume that we need to draw n data points, and that w is the fraction of these points that are good (we need only a reasonable estimate of this number). Now the expected value of the number of

draws k required to get one point is given by

$$\begin{aligned}\mathrm{E}[k] &= 1P(\text{one good sample in one draw}) + 2P(\text{one good sample in two draws}) + \dots \\ &= w^n + 2(1-w^n)w^n + 3(1-w^n)^2 w^n + \dots \\ &= w^{-n}\end{aligned}$$

(where the last step takes a little manipulation of algebraic series). We would like to be fairly confident that we have seen a good sample, so we wish to draw more than w^{-n} samples; a natural thing to do is to add a few standard deviations to this number. The standard deviation of k can be obtained as

$$SD(k) = \frac{\sqrt{1-w^n}}{w^n}.$$

An alternative approach to this problem is to look at a number of samples that guarantees a low probability z of seeing only bad samples. In this case, we have

$$(1-w^n)^k = z,$$

which means that

$$k = \frac{\log(z)}{\log(1-w^n)}.$$

It is common to have to deal with data where w is unknown. However, each fitting attempt contains information about w. In particular, if n data points are required, then we can assume that the probability of a successful fit is w^n. If we observe a long sequence of fitting attempts, we can estimate w from this sequence. This suggests that we start with a relatively low estimate of w, generate a sequence of attempted fits, and then improve our estimate of w. If we have more fitting attempts than we need for the new, the process can stop. The problem of updating the estimate of w reduces to estimating the probability that a coin comes up heads or tails given a sequence of fits.

Telling Whether a Point Is Close We need to determine whether a point lies close to a line fitted to a sample. We do this by determining the distance between the point and the fitted line, and testing that distance against a threshold d; if the distance is below the threshold, the point lies close. In general, specifying this parameter is part of the modeling process. For example, when we fitted lines using maximum likelihood, there was a term σ in the model (which disappeared in the manipulations to find an maximum). This term gives the average size of deviations from the model being fitted.

In general, obtaining a value for this parameter is relatively simple. We generally need only an order of magnitude estimate, and the same value applies to many different experiments. The parameter is often determined by trying a few values and seeing what happens; another approach is to look at a few characteristic data sets, fitting a line by eye, and estimating the average size of the deviations.

The Number of Points That Must Agree Assume that we have fitted a line to some random sample of two data points. We need to know whether that line

is good. We do this by counting the number of points that lie within some distance of the line (the distance was determined in the previous section). In particular, assume that we know the probability that an outlier lies in this collection of points; write this probability as y. We should like to choose some number of points t such that y^t is small (say less than 0.05).

There are two ways to proceed. One is to notice that $y \leq (1 - w)$ and to choose t such that $(1 - w)^t$ is small. Another is to get an estimate of y from some model of outliers — for example, if the points lie in a unit square, the outliers are uniform, and the distance threshold is d, then $y \leq 2\sqrt{2}d$.

10.3 THE HOUGH TRANSFORM

The Hough transform is a method that promises a solution to all three problems (although in practice rarely delivers it). It is something worth understanding because the underlying method is quite general and appears in a number of applications.

One way to cluster points that could lie on the same structure is to record all the structures on which each point lies and then look for structures that get many votes. This (quite general) technique is known as the *Hough transform*. We take each image token and determine all structures *that could pass through that token*. We make a record of this set — you should think of this as voting — and repeat the process for each token. We decide on what is present by looking at the votes. For example, if we are grouping points that lie on lines, we take each point and vote for all lines that could go through it; we now do this for each point. The line (or lines) that are present should make themselves obvious because they pass through many points and so have many votes.

10.3.1 Fitting Lines with the Hough Transform

Hough transforms tend to be most successfully applied to line finding. We do this example to illustrate the method and its drawbacks. A line is easily parametrized as a collection of points (x, y) such that

$$x \cos\theta + y \sin\theta + r = 0.$$

Now any pair of (θ, r) represents a unique line, where $r \geq 0$ is the perpendicular distance from the line to the origin and $0 \leq \theta < 2\pi$. We call the set of pairs (θ, r) *line space*; the space can be visualized as a half-infinite cylinder. There is a family of lines that passes through any point token. In particular, the lines that lie on the curve *in line space* given by $r = -x_0 \cos\theta + y_0 \sin\theta$ all pass through the point token at (x_0, y_0).

Because the image has a known size, there is some R such that we are not interested in lines for $r > R$ — these lines are too far away from the origin for us to see them. This means that the lines we are interested in form a bounded subset of the plane, and we discretize this with some convenient grid (which we'll discuss later). The grid elements can be thought of as buckets into which we place votes. This grid of buckets is referred to as the *accumulator array*. For each point token, we add a vote to the total formed for every grid element on the curve corresponding to the point token. If there are many point tokens that are collinear, we expect there to be many votes in the grid element corresponding to that line.

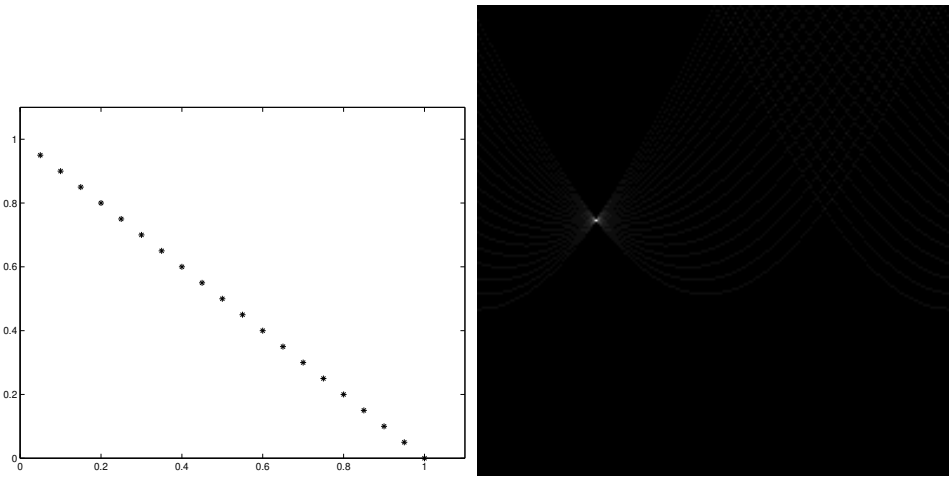

FIGURE 10.6: *The Hough transform maps each point like token to a curve of possible lines (or other parametric curves – but here we show lines) through that point.* **Left** *shows points, and* **right** *shows the corresponding accumulator array (the number of votes is indicated by the grey level, with a large number of votes being indicated by bright points). Here we see what happens using a set of 20 points drawn from a line. Corresponding to each point is a curve of votes in the accumulator array; the largest set of votes is 20 (which corresponds to the brightest point). The horizontal variable in the accumulator array is θ and the vertical variable is r; there are 200 steps in each direction, and r lies in the range $[0, 1.55]$. We have a clean peak corresponding to the line.*

10.3.2 Practical Problems with the Hough Transform

Unfortunately, the Hough transform comes with a number of important practical problems:

- **Quantization errors:** An appropriate grid size is difficult to pick. Too coarse a grid can lead to large values of the vote being obtained falsely because many quite different lines correspond to a bucket. Too fine a grid can lead to lines not being found because votes resulting from tokens that are not exactly collinear end up in different buckets, and no bucket has a large vote (Figure **??**).

- **Difficulties with noise:** The attraction of the Hough transform is that it connects widely separated tokens that lie close to some form of parametric curve. This is also a weakness; it is usually possible to find many quite good phantom lines in a large set of reasonably uniformly distributed tokens (Figure 10.8). This means that regions of texture can generate peaks in the voting array that are larger than those associated with the lines sought (Figures 10.9 and 10.10).

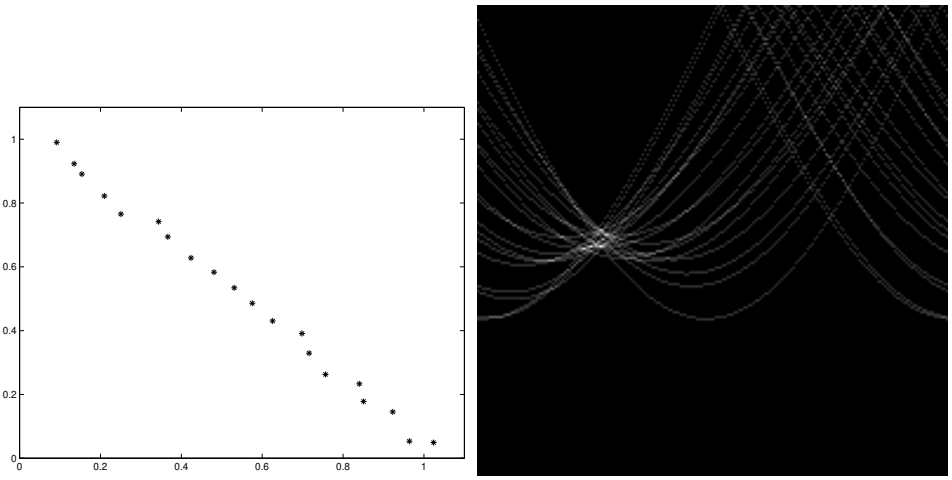

FIGURE 10.7: *When the points of Figure 10.6 are perturbed by a random vector each element of which is uniform in the range* $[0, 0.05]$*, the curves in the accumulator array are each slightly offset from one another. The maximum vote is now 6 (which corresponds to the brightest value in this image — this value would be difficult to see on the same scale as the top image).*

10.4 CLASSIFYING IMAGES BY VOTING

The Hough transform is worth talking about because, despite the problems, it can be used. You should think of the Hough transform as a voting procedure. For line fitting, each token has an opinion about where a line could be, and registers that opinion in the accumulator array. Then if any lines have enough support, we decide they are present. Problems occur because there are too many tokens; because each token can vote for many different lines; and because it is hard to choose a grid size.

Now imagine each token is more interesting than a point. For example, tokens might be a family of patterns around interest points, represented by (say) a SIFT feature. Then there might be relatively few tokens, and each might cast relatively few votes. In this case, we would expect it to be relatively straightforward to identify peaks in the accumulator array.

It is possible to build an image classifier like this. Assume we wish to know whether an image contains (say) a car or not. We will also assume that, if it does contain a car, the car is pretty prominent and is seen from a reasonable viewpoint. We start by finding a collection of images of cars, and marking the location of the car in each image. We then find interest points for a large number of car and non-car images, and compute SIFT (or other) feature descriptors for these interest points. We now look for interest points with feature descriptors that (a) appear very often in the car images and (b) appear very seldom in the non car images. One example might be an interest point that sits at the center of a wheel. We will try to predict the location of the car using these interest points. To do so, we need to deal with two issues: first, corresponding interest points might look somewhat different (wheels change in appearance); second, different versions of the same interest point

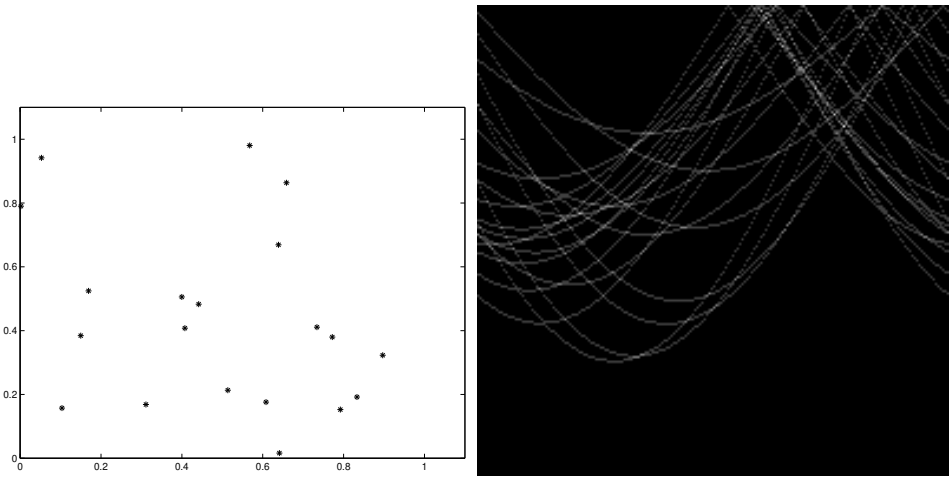

FIGURE 10.8: *The Hough transform for a set of random points can lead to quite large sets of votes in the accumulator array. As in Figure* **??**, *the* **left-hand column** *shows points, and the* **right-hand column** *shows the corresponding accumulator arrays (the number of votes is indicated by the grey level, with a large number of votes being indicated by bright points). In this case, the data points are noise points (both coordinates are uniform random numbers in the range* $[0, 1]$*); the accumulator array in this case contains many points of overlap, and the maximum vote is now 4 (compared with 6 in Figure* **??**). *Figures 10.9 and 10.10 explore noise issues somewhat further.*

might predict the center of the car in different places (so the front wheel thinks the center is behind it, and the back wheel thinks the center is in front of it).

10.4.1 Vector Quantizing Descriptors with K-Means

Wheels (and interest points) can look different from car to car. Vector quantization is a strategy to suppress small differences between interest points, so as to come up with a representation that is good for most cars.

We observe N interest points in many different images. Write $\mathbf{v}_i$ for the feature vector describing the i'th interest point. We expect that there are interest points that mostly look similar from image to image (though they might not be exactly the same) *and* that there are different kinds of such interest point. So, for example, wheels look mostly like one another and doorhandles look mostly like one another but doorhandles do not look like wheels. We assume that there are k different kinds of interest point. Every example of a given kind of interest point has feature vectors that are similar to one another. Each kind of interest point has feature vectors that are quite different. We want to (a) estimate what each kind of interest point looks like and (b) decide what kind of interest point each example belongs to. This is a clustering problem.

Assume that there are k different kinds of interest point (I will describe one strategy to choose k in the next section – for the moment, assume it is known). We

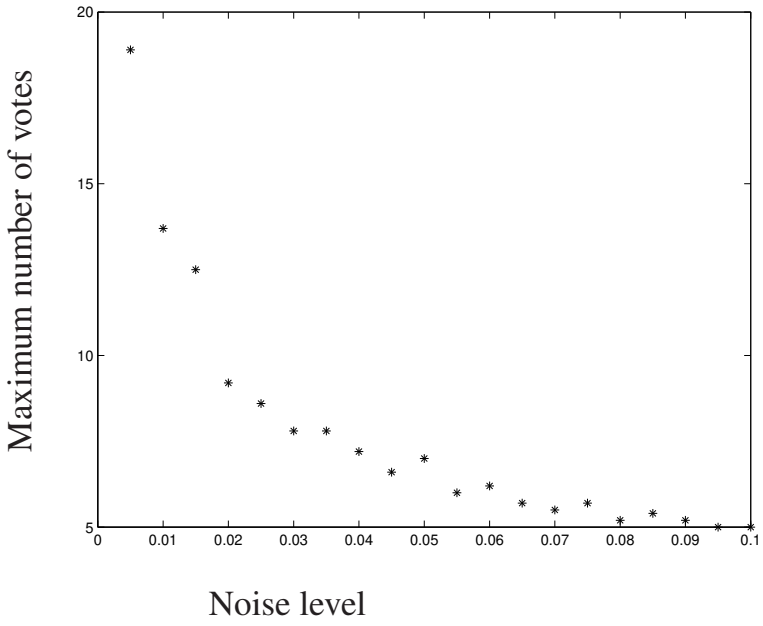

FIGURE 10.9: *The effects of noise make it difficult to use a Hough transform robustly. The plot shows the maximum number of votes in the accumulator array for a Hough transform of 20 points on a line perturbed by uniform noise plotted against the magnitude of the noise. The noise displaces the curves from each other and quickly leads to a collapse in the number of votes. The plot has been averaged over 10 trials. The accumulator array had the same quantization for each case shown here.*

want to construct a set of k centers $\mathbf{c}_j$ that represent the typical feature vector of each kind of interest point. Now define a $N \times k$ table of discrete variables δ_{ij} which maps each interest point to its kind. We have $\delta_{ij} = 1$ if the i'th interest point is of kind k, and 0 otherwise. Because each interest point can have only one kind, we have that $\sum_j \delta_{ij} = 1$. We do not know δ_{ij} and we do not know $\mathbf{c}_j$. But we believe that every example has a feature vector that is close to the center for its kind. This means that δ_{ij} and $\mathbf{c}_j$ should minimize

$$\sum_{ij} \delta_{ij} \left(\mathbf{v}_i - \mathbf{c}_j\right)^T \left(\mathbf{v}_i - \mathbf{c}_j\right).$$

Getting an exact solution to this minimization problem is intractable, but a very good approximate solution is quite easy to get. Notice that if we knew the centers, it would be easy to compute the best δ_{ij}. For the i'th data point, set the δ_{ij} corresponding to the closest center to 1, and that corresponding to all others to zero. Similarly, if we know the δ_{ij}, the centers are easy to compute. The j'th center is the average of all the data points whose δ_{ij} is one.

This suggests an algorithm. Come up with an initial estimate of the centers. Now repeat: compute the δ_{ij}, and re-estimate the centers. There are some impor-

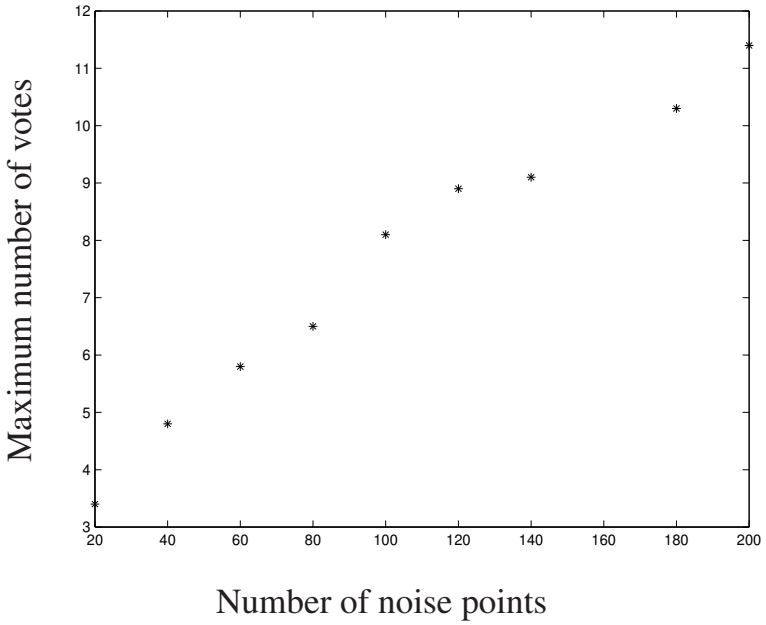

FIGURE 10.10: *A plot of the maximum number of votes in the accumulator array for a Hough transform of a set of points whose coordinates are uniform random numbers in the range* $[0, 1]$ *plotted against the number of points. As the level of noise goes up, the number of votes in the right bucket goes down, and the prospect of obtaining a large spurious vote in the accumulator array goes up. The plots have again been averaged over 10 trials. Compare this figure with Figure 10.9, but notice the slightly different scales; the comparison suggests that it can be quite difficult to pull a line out of noise with a Hough transform (because the number of votes for the line might be comparable with the number of votes for a line due to noise). These figures illustrate the importance of ruling out as many noise tokens as possible before performing a Hough transform.*

tant details to get right. We need to know when to stop, and averaging data points to get the j'th center won't work if there are no data points with $\delta_{ij} = 1$. Stopping turns out to be straightforward. If the δ_{ij} don't change from the r'th iteration to the $r + 1$'th, the cluster centers won't change from then on. It is usually enough to stop when the cluster centers don't change much.

Procedure: 10.1 *K-Means Clustering*

Choose k. Now choose k data points $\mathbf{c}_j$ to act as cluster centers. Until the cluster centers change very little

- Allocate each data point to cluster whose center is nearest.
- Now ensure that every cluster has at least one data point; one way to do this is by supplying empty clusters with a point chosen at random from points far from their cluster center.
- Replace the cluster centers with the mean of the elements in their clusters.

It turns out the choice of initial estimates of the centers can matter a lot, too. One natural strategy for initializing k-means is to choose k data items at random, then use each as an initial cluster center. This approach is widely used, but has some difficulties. The quality of the clustering can depend quite a lot on initialization, and an unlucky choice of initial points might result in a poor clustering. One (again quite widely adopted) strategy for managing this is to initialize several times, and choose the clustering that performs best in your application. Another strategy, which has quite good theoretical properties and a good reputation, is known as *k-means++*. You choose a point $\mathbf{x}$ uniformly and at random from the dataset to be the first cluster center. Then you compute the squared distance between that point and each other point; write $d_i^2(\mathbf{x})$ for the distance from the i'th point to the first center. You now choose the other $k-1$ cluster centers as IID draws from the probability distribution

$$\frac{d_i^2(\mathbf{x})}{\sum_u d_u^2(\mathbf{x})}.$$

K-Means allows us to estimate the center $\mathbf{c}_j$ for each of k kinds of feature vector. *Vector quantization* consists of taking a feature vector and deciding which kind it belongs to. We do this by finding the *index* of the closest center. We can do so for feature vectors describing interest points in the training set, or for feature vectors describing new interest points. Furthermore, we can tell which kinds of interest point are most strongly associated with cars. For each kind of interest point, we record the fraction of car images that have at least one such interest point. This is an estimate of $P(\text{see } j\text{'th kind of interest point}|\text{car in image}) = P(j|\text{car})$. Similarly, we record the fraction of non-car images that have at least one such interest point (which would be an estimate of $P(\text{see } j\text{'th kind of interest point}|\text{not car in image}) = P(j|\text{ car})$).

At this point, we could classify car vs non-car images with naive bayes. This will work better than you'd expect (because naive bayes always does) but not very well (because we are ignoring important information). An easy voting trick improves classification significantly.

10.4.2 Voting with a Generalized Hough Transform

Naive Bayes doesn't use the fact that interest points are likely to be able to predict where the car is. If they do so even approximately, the fact that several interest points *agree* on the location of the car would strongly enhance our belief it was there. Here is one procedure to use that information.

Choose kinds of interest point that are strongly associated with cars ($P(j|\text{car}$ is big and $P(j|$ car) is small). For each example of this kind of interest point in each example car image, record where the center of the car is *relative to the interest point's coordinate system*. Remember, you know the location, orientation and scale of the window around the interest point. If the scale is big, you expect the center of the car to be far away in pixels; and if it is small, you expect the center of the car to be nearby in pixels. Now cluster these predictions to a small number using k-means (quite possibly using an entirely different value of k from the one you used to find kinds of interest points).

Now given an image that might contain a car, identify every example of each kind of interest point that is strongly associated with a car. Record a vote for the location of the center of the car for

10.5 FITTING CURVES

In principle, fitting curves is similar to fitting lines. We minimize the sum of squared distances between the points and the curve. However, it is usually very hard to tell the distance between a point and a curve. We can either solve this problem or apply various approximations (which are usually chosen because they are computationally simple, not because they result from clean models). We sketch some solutions for the distance problem for the two main representations of curves.

10.5.1 Implicit Curves

The coordinates of *implicit curves* satisfy some parametric equation; if this equation is a polynomial, then the curve is said to be *algebraic*, and this case is by far the most common. Some common cases are given in Table 10.1.

The Distance from a Point to an Implicit Curve Now we would like to know the distance from a data point to the closest point on the implicit curve. Assume that the curve has the form $\phi(x, y) = 0$. The vector from the closest point on the implicit curve to the data point is normal to the curve, so the closest point is given by finding all the (u, v) with the following properties:

1. (u, v) is a point on the curve — this means that $\phi(u, v) = 0$;
2. $\mathbf{s} = (d_x, d_y) - (u, v)$ is normal to the curve.

Given all such $\mathbf{s}$, the length of the shortest is the distance from the data point to the curve. The second criterion requires a little work to determine the normal. The normal to an implicit curve is the direction in which we leave the curve fastest; along this direction, the value of ϕ must change fastest, too. This means that the

TABLE 10.1: *Some implicit curves used in vision applications. Note that not all of these curves are guaranteed to have any real points on them — e.g.,* $x^2+y^2+1=0$ *doesn't. Higher degree curves are seldom used because it can be difficult to get stable fits to these curves.*

Curve	Equation
Line	$ax+by+c=0$
Circle, center (a, b), and radius r	$x^2+y^2-2ax-2by+a^2+b^2-r^2=0$
Ellipses (including circles)	$ax^2+bxy+cy^2+dx+ey+f=0$ where $b^2-4ac<0$
Hyperbolae	$ax^2+bxy+cy^2+dx+ey+f=0$ where $b^2-4ac>0$
Parabolae	$ax^2+bxy+cy^2+dx+ey+f=0$ where $b^2-4ac=0$
General conic sections	$ax^2+bxy+cy^2+dx+ey+f=0$

normal at a point (u,v) is

$$(\frac{\partial\phi}{\partial x},\frac{\partial\phi}{\partial y}),$$

evaluated at (u,v). If the tangent to the curve is $\mathbf{T}$, then we must have $\mathbf{T}.\mathbf{s}=0$. Because we are working in 2D, we can determine the tangent from the normal, so that we must have

$$\psi(u,v;d_x,d_y)=\frac{\partial\phi}{\partial y}(u,v)\{d_x-u\}-\frac{\partial\phi}{\partial x}(u,v)\{d_y-v\}=0$$

at the point (u,v). We now have two equations in two unknowns and, *in principle* can solve them. However, this is very seldom as easy as it looks, as Example **??** indicates.

The distance between a point and a conic

A conic section is given by $ax^2+bxy+cy^2+dx+ey+f=0$. Given a data point (d_x,d_y), the nearest point on the conic satisfies two equations:

$$au^2+buv+cv^2+du+ev+f=0$$

and

$$2(a-c)uv-(2ad_y+e)u+(2cd_x+d)v+(ed_x-dd_y)=0.$$

There can be up to four real solutions of this pair of equations (in the exercises, you are asked to demonstrate this, given an algorithm for obtaining the solutions, and asked to sketch various cases). As an example, choose the ellipse $2x^2+y^2-1=0$, which yields the equations

$$2u^2+v^2-1=0 \text{ and } 2uv-4d_yu+2d_xv=0.$$

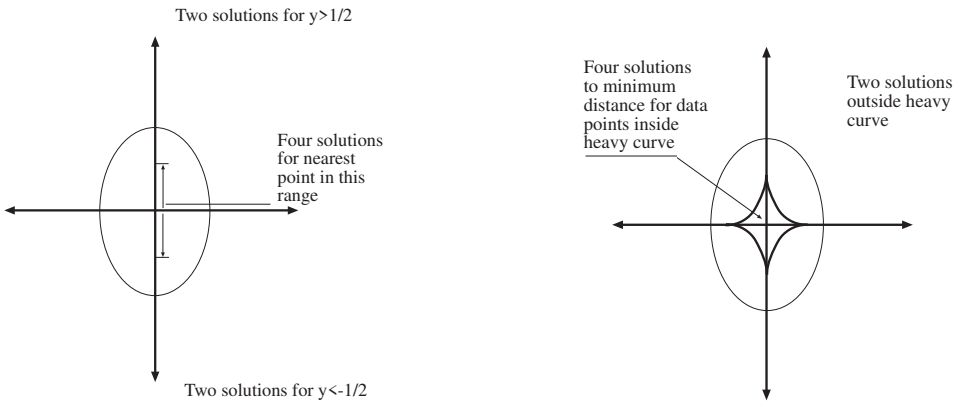

FIGURE 10.11: *On the* **left**, *the example worked in the text, where we study the number of possible solutions for the distance between a point and an ellipse for data points lying on the vertical axis. The figure on the* **right** *indicates the general case for this ellipse.*

Let us consider a family of data points $(d_x, d_y) = (0, \lambda)$; then we can rearrange these equations to get

$$2u^2 + v^2 - 1 = 0 \text{ and } 2uv - 4\lambda u = 2u(v - 2\lambda) = 0.$$

The second equation helps: Either $u = 0$ or $v = 2\lambda$. Two of our solutions will be $(0, 1)$, $(0, -1)$. The other two are obtained by solving $2u^2 + 4\lambda^2 - 1 = 0$, which has solutions only if $-1/2 \leq \lambda \leq 1/2$. The situation is illustrated in Figure 10.11.

Approximations to the Distance Notice that for a relatively simple curve, we already have an unpleasant problem to solve. A curve with a slightly more complicated geometry — obtained by choosing ϕ to be a polynomial of higher degree, say d — leads to openly nasty problems. This is because the closest point on the curve would be obtained by solving two simultaneous polynomial equations, *both* of degree d. It can be shown that this can lead to as many as d^2 solutions, which are usually hard to obtain in practice. Various approximations to the distance between a point and an implicit algebraic curve have come into practice.

The best known is *algebraic distance*: In this case, we measure the distance between a curve and a point by evaluating the polynomial equation at that point, that is, we make the approximation

$$\text{distance between } (d_x, d_y) \text{ and } \phi(x, y) = 0 = \phi(d_x, d_y).$$

This approximation can be (rather roughly!) justified when the data points are quite close to the curve. For a point sufficiently close to the curve *and to first order*, $\phi(d_x, d_y)$ increases as (d_x, d_y) moves normal to the curve — because the normal to the curve is given by the gradient of ϕ — and does not increase as (d_x, d_y) moves tangent to the curve. One significant difficulty is that, as it stands,

algebraic distance is ill defined because many polynomials correspond to the same curve. In particular, the curve given by $\mu\phi(x, y) = 0$ is the same as the curve given by $\phi(x, y) = 0$. This problem can be solved by normalizing the coefficients of the polynomial in some way.

We have already seen one example of this process in Section 10.1, where we fitted a line ($\phi(x, y) = ax + by + c = 0$) to a set of points by minimizing the algebraic distance subject to the constraint that $a^2 + b^2 = 1$. In this case, the algebraic distance is the same as the actual distance. The choice of normalization is important. For example, if we try to fit conics ($ax^2 + bxy + cy^2 + dx + ey + f = 0$) using the constraint $b = 1$, we cannot fit circles. An alternative approximation is to use

$$\frac{\phi(d_x, d_y)}{|\nabla\phi(d_x, d_y)|},$$

which has the advantage of not requiring a normalizing constant; in the case of a line, this approximation is exact. Notice that this approximation has the same properties as algebraic distance — it goes up as one moves along the normal, and so on. The advantage of the approximation is that it is somewhat more accurate than algebraic distance because it is normalised by the length of the normal. This means that it can be read — roughly! — as giving the percentage distance along the normal from the curve to the point. In practice, this approximation is seldom used mainly because the use of algebraic distance yields simpler numerical problems.

Both of these approximations are dangerous because their behavior for data points that are far from the curve is strange and not well understood. As a result, the relationship between a fitted curve and a set of data points becomes a bit mysterious if the data points don't lie close to a curve of that class. Algebraic distance is used quite widely in practice because it yields easy numerical problems and can be used for higher dimensional problems like approximating the distance between points and implicit surfaces. The exact distance is often difficult to compute for such problems.

10.5.2 Parametric Curves

The coordinates of a *parametric curve* are given as parametric functions of a parameter that varies along the curve. Parametric curves have the form

$$(x(t), y(t)) = (x(t; \theta), y(t; \theta)) \quad t \in [t_{\min}, t_{\max}].$$

Table 10.2 shows the form of a variety of useful parametric curves.

The Distance from a Point to a Parametric Curve Assume we have a data point (d_x, d_y). The closest point on a parametric curve can be identified by its parameter value, which we shall write as τ. This point could lie at one or other end of the curve. Otherwise, the vector from our data point to the closest point is normal to the curve. This means that $\mathbf{s}(\tau) = (d_x, d_y) - (x(\tau), y(\tau))$ is normal to the tangent vector, so that $\mathbf{s}(\tau).\mathbf{T} = 0$. The tangent vector is

$$(\frac{dx}{dt}(\tau), \frac{dy}{dt}(\tau)),$$

TABLE 10.2: *A selection of parametric curves often used in vision applications. It is quite common to put together a set of cubic curves, with constraints on their coefficients such that they form a single continuous differentiable curve; the result is known as a* cubic spline.

Curves	Parametric Form	Parameters
Circles centered at the origin	$(r\sin(t), r\cos(t))$	$\theta = r$ $t \in [0, 2\pi)$
Circles	$(r\sin(t) + a, r\cos(t) + b)$	$\theta = (r, a, b)$ $t \in [0, 2\pi)$
Axis aligned ellipses	$(r_1\sin(t) + a, r_2\cos(t) + b)$	$\theta = (r_1, r_2, a, b)$ $t \in [0, 2\pi)$
Ellipses	$(\cos\phi\,(r_1\sin(t) + a) - \sin\phi\,(r_2\cos(t) + b)\,,$ $\sin\phi\,(r_1\sin(t) + a) + \cos\phi\,(r_2\cos(t) + b))$	$\theta = (r_1, r_2, a, b, \phi)$ $t \in [0, 2\pi)$
cubic segments	$(at^3 + bt^2 + ct + d, et^3 + ft^2 + gt + h)$	$\theta = (a, b, c, d, e, f, g, h)$ $t \in [0, 1]$

which means that τ must satisfy the equation

$$\frac{dx}{dt}(\tau)\left\{d_x - x(\tau)\right\} + \frac{dy}{dt}(\tau)\left\{d_y - y(\tau)\right\} = 0.$$

Now this is only one equation, rather than two, but the situation is not much better than that for parametric curves. It is almost always the case that $x(t)$ and $y(t)$ are polynomials because it is usually easier to do root finding for polynomials. At worst, $x(t)$ and $y(t)$ are ratios of polynomials because we can rearrange the left-hand side of our equation to come up with a polynomial in this case, too. However, we are still faced with a possibly large number of roots.

There is a second difficulty that makes fitting to parametric curves unpopular. Parametric curves with different coefficients may represent the same curve — for example, the curve $(x(t), y(t))$ for $t \in [0, 1]$ is the same as the curve $(x(2t), y(2t))$ for $t \in [0, 1/2]$. This situation can be very bad depending on the class of parametric curves that we use.