CHAPTER 10

Grouping and Model Fitting

In the previous chapter, we collected together pixels that "looked like" one another, using various clustering methods and various ways of measuring similarity. This view could be applied to tokens (such as points, edge points, superpixels). It is an intrinsically local view.

An alternative, which emphasizes a more global view, is to collect together pixels, tokens, or whatever because they conform to some model. This approach appears rather similar to the clustering approach in intent, but the mechanisms and outcomes tend to be quite different. In the clustering approach, the results we produce can have local structure, but will not necessarily have a global structure. For example, if we try to assemble tokens into a line agglomeratively by testing whether the token to be added is close to the line formed by the previous two tokens, we might get a shape that is quite curved. We really need to check whether all tokens "agree" on the parameters of the line; local consistency is not enough.

These problems are rather difficult, and strategies to attack them for one kind of model tend to extend rather well to strategies for other kinds of model. In this chapter, we mainly concentrate on one core problem, which is simple enough to do in detail. We seek to find all the lines represented by a set of tokens. This problem is usually referred to as *fitting*, or sometimes as *grouping*. There are three important sub-problems here: If all the tokens agree on a model, what is the model? Which tokens contribute to a particular model, and which do not? And how many instances of the model are there?

10.1 THE HOUGH TRANSFORM

Assume we wish to fit a structure to a set of tokens (say, a line to a set of points). One way to cluster tokens that could lie on the same structure is to record all the structures on which each token lies and then look for structures that get many votes. This (quite general) technique is known as the *Hough transform.* To fit a structure with a 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.

10.1.1 Fitting Lines with the Hough Transform

A line is easily parametrized as a collection of points (x, y) such that

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

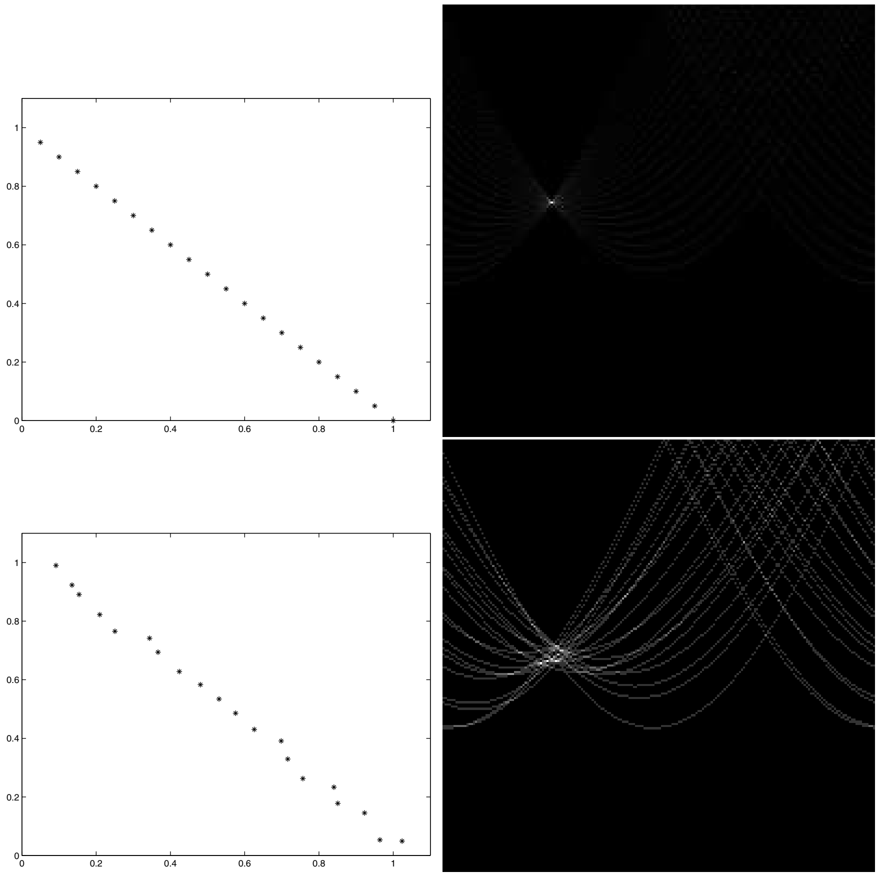

FIGURE 10.1: The Hough transform maps each point like token to a curve of possible lines (or other parametric curves) through that point. These figures illustrate the Hough transform for lines. The **left-hand column** shows points, and the **right-hand column** shows the corresponding accumulator arrays (the number of votes is indicated by the gray level, with a large number of votes being indicated by bright points). The **top** row shows what happens using a set of 20 points drawn from a line. On the **top right**, the accumulator array for the Hough transform of these points. 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 $[0, 1.55]$. On the **bottom**, these points have been offset by a random vector, each element of which is uniform in the range $[0, 0.05]$. Note that this offsets the curves in the accumulator array shown next to the points and 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).

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).

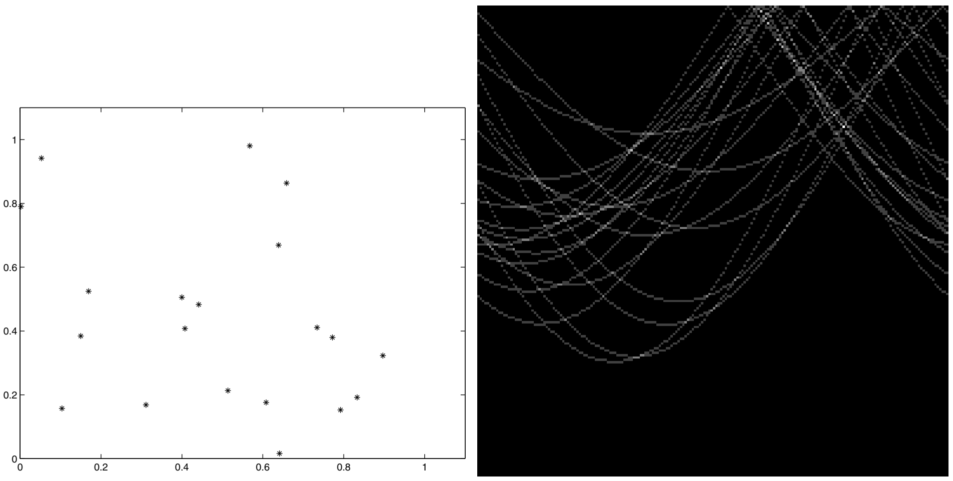

FIGURE 10.2: The Hough transform for a set of random points can lead to quite large sets of votes in the accumulator array. As in Figure 10.1, the **left-hand column** shows points, and the **right-hand column** shows the corresponding accumulator arrays (the number of votes is indicated by the gray 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 10.1).

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. 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.

10.1.2 Using the Hough Transform

The Hough transform is an extremely general procedure. One could use the procedure described to fit, say, circles to points in the plane, or spheres or even ellipsoids to three-dimensional data. This works in principle, but in practice, the Hough transform as described is difficult to use, even for finding lines. There are several sources of difficulty.

- **Grid dimension:** The accumulator array for lines has dimension two, but for circles in the plane, it has dimension three (center location and radius); for axis-aligned ellipses in the plane it has dimension four; for general ellipses in the plane, five; for spheres in 3D, four; for axis-aligned ellipsoids in 3D, seven; and for general ellipsoids in 3D, 10. Even quite simple structures could result in high-dimensional accumulator arrays, which take unmanageable amounts of storage.

- **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 structures correspond to a single bucket. Too fine a grid can lead to structures not being found because votes resulting from tokens that are not exactly aligned end up in different buckets, and no bucket has a large vote (Figure 10.1).
- **Noise:** The attraction of the Hough transform is that it connects widely separated tokens that lie close to some structure. This is also a weakness because it is usually possible to find many quite good phantom structures in a large set of reasonably uniformly distributed tokens (Figure 10.2). This means that regions of texture can generate peaks in the voting array that are larger than those associated with the lines sought.

These difficulties can be avoided, to some extent, by recognizing the Hough transform as an attempt to find a mode in a distribution. The distribution is represented by the voting array, and some of the problems are created by the cells in that array. But to find a mode, we do not necessarily need to use an accumulator array; instead, we could apply mean shift. The algorithm of Section 9.3.4 can be applied directly. It can also be useful to ensure the minimum of irrelevant tokens by, for example, tuning the edge detector to smooth out texture, using lighting that ensures high-contrast edges, or using tokens with a more complex structure with edge points.

One natural application of the Hough transform is in object recognition. We defer the details to Section 18.4.2, but the general picture is as follows. Imagine an object is made out of known parts. These parts might move around a bit with respect to one another, but are substantial enough to be detected on their own. We can then expect each detected part to have an opinion about the location (and, perhaps, the state) of the object. This means we could detect objects by first detecting parts, then allowing each detected part to vote for location (and maybe state) of object instances, and finally using the Hough transform, most likely in mean shift form, to find instances on which many part detectors agree. This approach has been successfully applied in numerous forms (Maji and Malik 2009).

10.2 FITTING LINES AND PLANES

In many applications, objects are characterized by the presence of straight lines. For example, we might wish to build models of buildings using pictures of the buildings (as in the application in Chapter 19). This application uses polyhedral models of buildings, meaning that straight lines in the image are important. Similarly, many industrial parts have straight edges of one form or another; if we wish to recognize industrial parts in an image, straight lines could be helpful. In either case, a report of all straight lines in the image is an extremely useful segmentation. All this means that fitting a line to a set of plane tokens is extremely useful. Fitting planes to tokens in 3D is also useful, and our methods for fitting lines in the plane apply with little change.

10.2.1 Fitting a Single Line

We first 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.

There is a simple strategy for fitting lines, known as *least squares*. This procedure has a long tradition (which is the only reason we describe it!), but has a substantial bias. Most readers will have seen this idea, but many will not be familiar with the problems that it leads to. 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 minimizes

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

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 not much help in vision applications, because the model is an extremely poor one. 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 10.3). In fact, the process is so dependent on coordinate frame that it doesn't represent vertical lines at all.

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

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

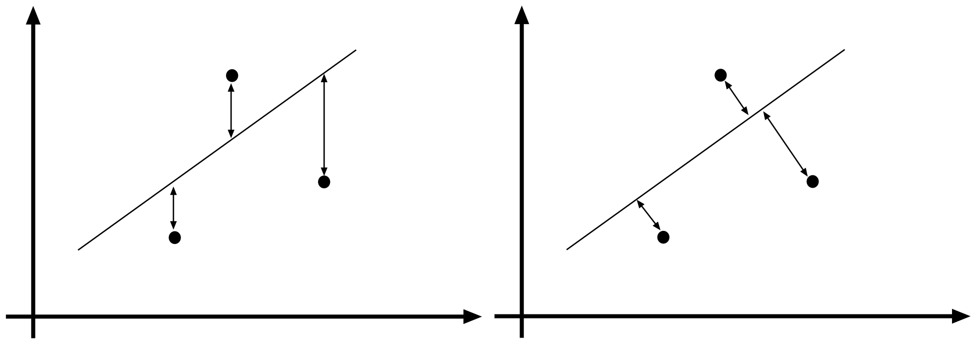

FIGURE 10.3: **Left:** Least squares finds the line that minimizes the sum of squared vertical distances between the line and the tokens (because it assumes that the error appears only in the y coordinate). This yields a (very slightly) simpler mathematical problem at the cost of a poor fit. **Right:** Total least-squares finds the line that minimizes the sum of squared perpendicular distances between tokens and the line; this means that, for example, we can fit near-vertical lines without difficulty.

subject to $a^2 + b^2 = 1$. 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; one maximizes the sum of squared distances and the other minimizes it.

10.2.2 Fitting Planes

Fitting planes is very similar to fitting lines. We could represent a plane as $z = ux + vy + w$, then apply least squares. This will be biased, just like least squares line fitting, because it will not represent vertical planes well. Total least squares is a better strategy, just as in line fitting. We represent the plane as $ax + by + cz + d = 0$; then the distance from a point $\boldsymbol{x}_i = (x_i, y_i, z_i)$ to the plane will be $(ax_i + by_i + cz_i + d)^2$ if $a^2 + b^2 + c^2 = 1$, and we can now use the analysis above with small changes.

10.2.3 Fitting Multiple Lines

Now assume we have a set of tokens (say, points), and we want to fit several lines to this set of tokens. This problem can be difficult because it can involve searching over a large combinatorial space. One approach is to notice that we seldom encounter isolated points; instead, in many problems, we are fitting lines to edge points. We can use the orientation of an edge point as a hint to the position of the next point on the line. If we are stuck with isolated points, then k-means can be applied.

Incremental line fitting algorithms take connected curves of edge points and fit lines to runs of points along the curve. Connected curves of edge points are fairly easily obtained from an edge detector whose output gives orientation (see exercises). An incremental fitter then starts at one end of a curve of edge points and walks along the curve, cutting off runs of pixels that fit a line well (the structure of the algorithm is shown in Algorithm 10.1). Incremental line fitting can work well, despite the lack of an underlying statistical model. One feature is that it reports groups of lines that form closed curves. This is attractive when the lines of interest can reasonably be expected to form a closed curve (e.g., in some object recognition applications) because it means that the algorithm reports natural groups without further fuss. When one uses this strategy, occlusion of an edge can lead to more than one line segment being fitted to the boundary. This difficulty can be addressed by postprocessing the lines to find pairs that (roughly) coincide, but the process is somewhat unattractive because it is hard to give a sensible criterion by which to decide when two lines do coincide.

```
Put all points on curve list, in order along the curve
Empty the line point list
Empty the line list
Until there are too few points on the curve
  Transfer first few points on the curve to the line point list
  Fit line to line point list
  While fitted line is good enough
    Transfer the next point on the curve
      to the line point list and refit the line
  end
  Transfer last point(s) back to curve
  Refit line
  Attach line to line list
end
```

Algorithm 10.1: Incremental Line Fitting.

Now assume that points carry no hints about which line they lie on (i.e., there is no color information or anything like it to help, and, crucially, the points are not linked). Furthermore, assume that we know how many lines there are. We can attempt to determine which point lies on which line using a modified version of k-means. In this case, the model is that there are k lines, each of which generates some subset of the data points. The best solution for lines and data points is obtained by minimizing

$$\sum_{l_i \in \text{lines}} \sum_{x_j \in \text{data due to } i\text{th line}} \text{dist}(l_i, x_j)^2$$

over both correspondences and lines. Again, there are too many correspondences to search this space.

It is easy to modify k-means to deal with this problem. The two phases are as follows:

- Allocate each point to the closest line.
- Fit the best line to the points allocated to each line.

This results in Algorithm 10.2. Convergence can be tested by looking at the size of the change in the lines, at whether labels have been flipped (probably the best test), or at the sum of perpendicular distances of points from their lines.

```
Hypothesize $k$ lines (perhaps uniformly at random)
or
Hypothesize an assignment of lines to points
   and then fit lines using this assignment

Until convergence
   Allocate each point to the closest line
   Refit lines
end
```

Algorithm 10.2: K-means Line Fitting.

10.3 FITTING CURVED STRUCTURES

Curves in 2D are different from lines in 2D. For every token on the plane, there is a unique, single point on a line that is closest to it. This is not true for a curve. Because curves curve, there might be more than one point on the curve that looks locally as though it is closest to the token (Figure 10.4). This means it can be very hard to find the smallest distance between a point and a curve. Similar effects occur for surfaces in 3D. If one ignores this difficulty, fitting curves is similar to fitting lines. We minimize the sum of squared distances between the points and the curve as a function of the choice of curve.

Assume that the curve is implicit, and so 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, which means that $\phi(u, v) = 0$.
2. $\boldsymbol{s} = (d_x, d_y) - (u, v)$ is normal to the curve.

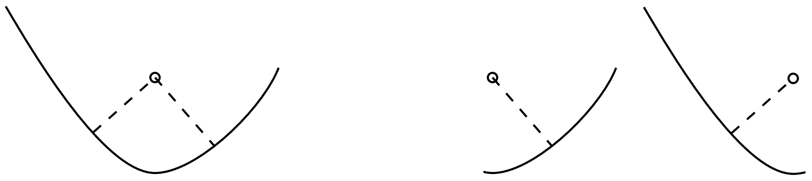

FIGURE 10.4: There can be more than one point on a curve that looks locally as if it is closest to a token. This makes fitting curves to points very difficult. On the **left**, a curve and a token; dashed lines connect the token to the two points on the curve that, by a local test, could be closest. The local test checks that the dashed line and the tangent to the curve are at right angles. **Center** and **right**, we show copies of part of the curve; for each, the closest point on the segment to the token is different, because part of the curve is missing. As a result, we cannot perform a local test that guarantees that a point is closest. We must check all candidates.

Given all such $\boldsymbol{s}$, 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 normal at a point (u, v) is

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

evaluated at (u, v). If the tangent to the curve is $\boldsymbol{T}$, then we must have $\boldsymbol{T}.\boldsymbol{s} = 0$. 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, because there might be many solutions. We expect d^2 in the case that ϕ is a polynomial of degree d, though some of them might be complex.

The situation is not improved by using a parametric curve. The coordinates of a point on a parametric curve are functions of a parameter, so if t is the parameter, the curve could be written as $(x(t), y(t))$. 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 the other end of the curve. Otherwise, the vector from our data point to the closest point is normal to the curve. This means that $\boldsymbol{s}(\tau) = (d_x, d_y) - (x(\tau), y(\tau))$ is normal to the tangent vector, so that $\boldsymbol{s}(\tau).\boldsymbol{T} = 0$. The tangent vector is

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

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. The underlying problem is geometric: there may be many points on a curve that, locally, appear to be the closest point to a given data point. This is because the curve is not flat (Figure 10.4). There is no way to tell which is closest without checking each in turn. In some cases (for example, circles), one can duck around this problem. This difficulty applies to fitting curved surfaces to points in 3D as well.

There are two strategies for dealing with this quite general problem. One is to substitute some approximation for the distance between a point and a curve (or, in 3D, a point and a surface), which is sometimes effective. The other is to modify the representation of the curve or of the surface. For example, one might represent a curve with a set of samples, or with a set of line segments. Similarly, a surface might be represented with a set of samples, or a mesh. We could then use the methods of Chapter 12 to register these representations to the data, or even to deform them to fit the data.

10.4 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 might give errors that dominate those due to many good data points; these errors could result in a substantial bias in the fitting process (Figure 10.5). This effect results from the squaring of the error. 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.

This problem can be resolved either by reducing the influence of distant points on the estimate (Section 10.4.1), or by identifying outlying points and ignoring them. There are two methods to identify outlying points. We could search for good points. A small set of good points will identify the thing we wish to fit; other good points will agree, and the points that disagree are bad points. This is the basis of an extremely important approach, described in Section 10.4.2. Alternatively, we could regard this as a problem with missing data, and use the EM algorithm described in Section 10.5.

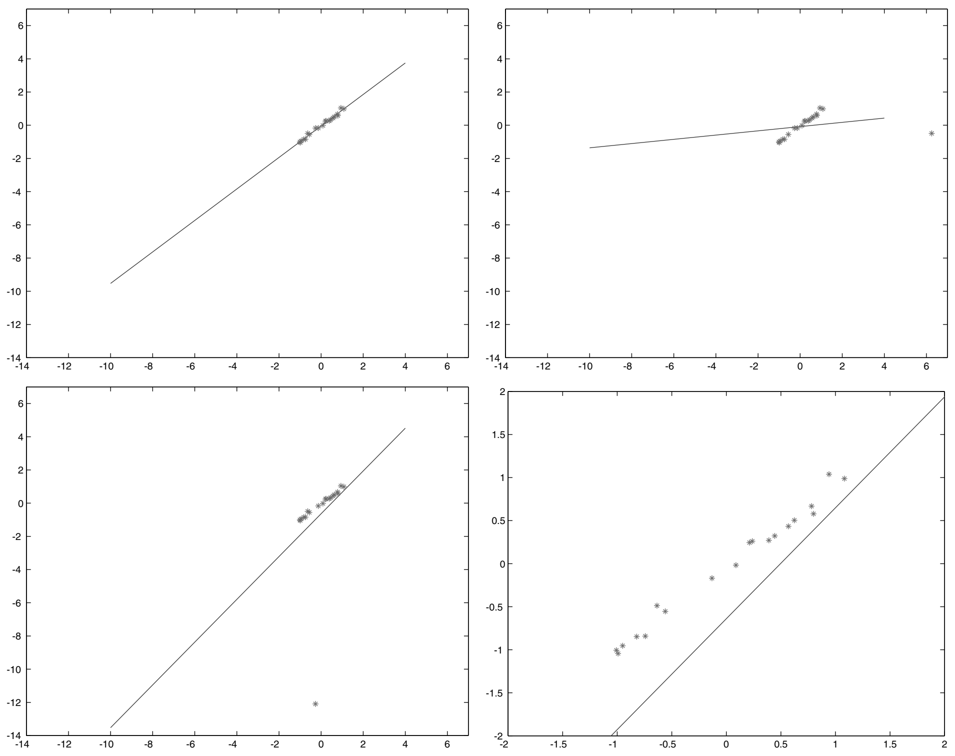

FIGURE 10.5: Line fitting with a squared error is extremely sensitive to outliers, both in x and y coordinates. We show an example using least squares. At the **top left**, a good least-squares fit of a line to a set of points. **Top right** shows the same set of points, but with the x coordinate of one point corrupted; this means that the point has been translated horizontally from where it should be. As a result, it contributes an enormous error term to the true line, and a better least-squares fit is obtained by making a significant change in the line's orientation. Although this makes the errors at most points larger, it reduces the very large error at the outlier. **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, in which the line is clearly a bad fit.

10.4.1 M-Estimators

An *M-estimator* estimates parameters by replacing the squared error term with a term that is better behaved. This means we minimize an expression of the form

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

where θ are the parameters of the model being fitted (for example, in the case of the line, we might have the orientation and the y intercept), and $r_i(\boldsymbol{x}_i, \theta)$ is the residual error of the model on the ith data point. Using this notation, our least

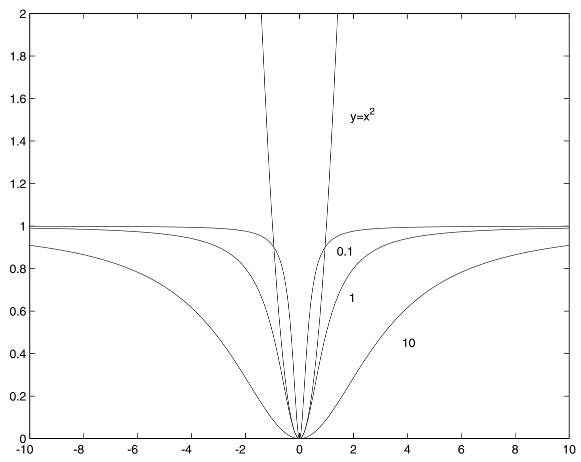

FIGURE 10.6: The function $\rho(x;\sigma) = x^2/(\sigma^2 + x^2)$, plotted for $\sigma^2 = 0.1$, 1, and 10, with a plot of $y = x^2$ for comparison. Replacing quadratic terms with ρ reduces the influence of outliers on a fit. A point that is several multiples of σ away from the fitted curve is going 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.

squares and total least squares line-fitting errors—which differ only in the form of the residual error—both have $\rho(u;\sigma) = u^2$. The trick to M-estimators is to make $\rho(u;\sigma)$ look like u^2 for part of its range and then flattens out; we expect that $\rho(u;\sigma)$ increases monotonically, and is close to a constant value for large u. 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, and we have plotted a variety of examples in Figure 10.6. There are many other M-estimators 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 minimization criterion yields

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

at the solution. 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 minimization problem is non-linear and must be solved iteratively. The standard difficulties apply: there might be more than one local minimum, the method might diverge, and the behavior of the method is likely to be quite dependent on the start point.

```
For $s = 1$ to $s = k$
  Draw a subset of $r$ distinct points, chosen uniformly at random
  Fit to this set of points using least squares to obtain an initial
    set of parameters $\theta_s^0$
  Estimate $\sigma_s^0$ using $\theta_s^0$
  Until convergence (usually $|\theta_s^n - \theta_s^{n-1}|$ is small):
    Take a minimizing step using $\theta_s^{n-1}$, $\sigma_s^{n-1}$
      to get $\theta_s^n$
    Now compute $\sigma_s^n$
  end
end
Report the best fit of this set of $k$ trials, using the median of the residuals
  as a criterion
```

Algorithm 10.3: Using an M-Estimator to Fit a Least Squares Model.

A common strategy for dealing with this problem is to draw a subsample of the dataset, 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 there is at least one subsample that consists entirely of good data points (Algorithm 10.3).

Second, as Figures 10.7 and 10.8 indicate, the estimators require a sensible estimate of σ, which is often referred to as *scale*. Typically, the scale estimate is supplied at each iteration of the solution method; a popular estimate of scale is

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

We summarize a general M-estimator in Algorithm 10.3.

10.4.2 RANSAC: Searching for Good Points

An alternative to modifying the cost function 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, and 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.

For example, assume that we are fitting a line to a dataset that consists of about 50% outliers. We can fit a line to only two points. 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.

Fischler and Bolles (1981) formalized this approach into 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

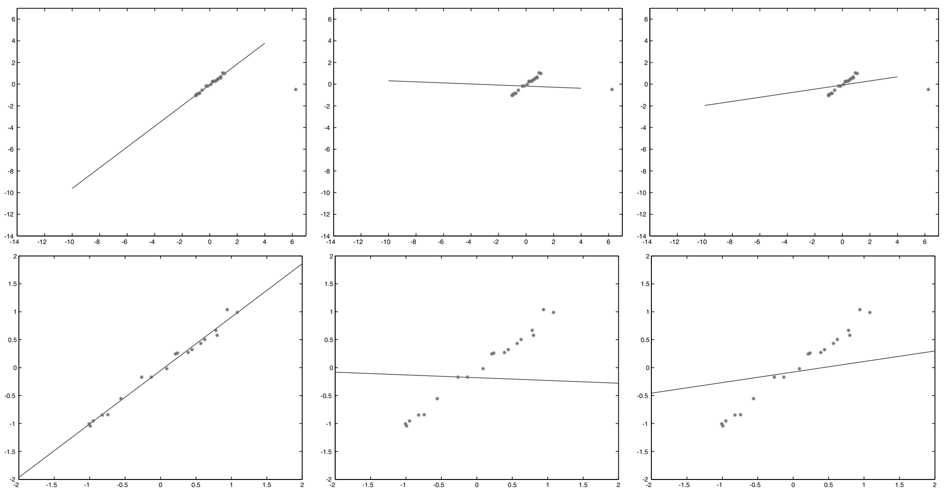

FIGURE 10.7: The **top row** shows lines fitted to the second dataset of Figure 10.5 using a weighting function that deemphasizes the contribution of distant points (the function ϕ of Figure 10.6). On the **left**, μ has about the right value; the contribution of the outlier has been down-weighted, and 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 as it does in least squares. The **bottom row** shows closeups of the fitted line and the non-outlying data points for the same cases.

displayed in Algorithm 10.4. 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 dataset. Each sample contains the minimum number of points required to fit the abstraction of interest. 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

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

(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

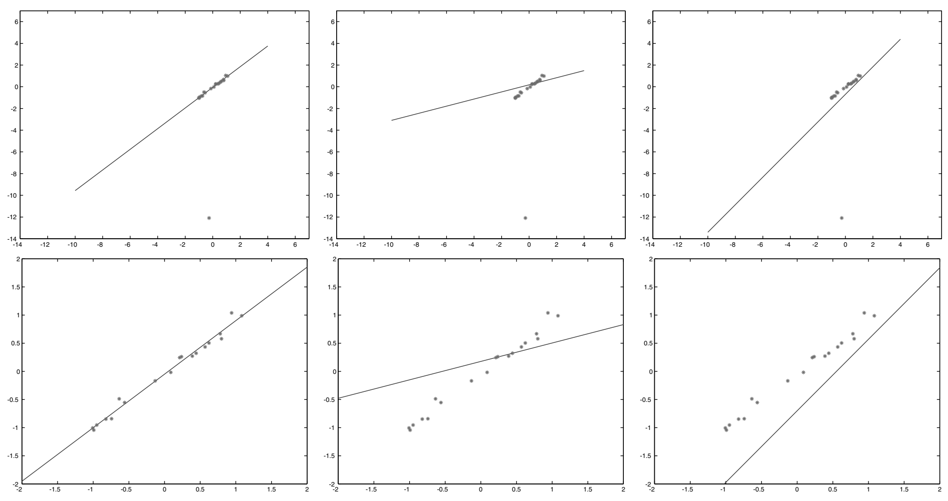

FIGURE 10.8: The **top row** shows lines fitted to the third dataset of Figure 10.5 using a weighting function that deemphasizes the contribution of distant points (the function ϕ of Figure 10.6). On the **left**, μ has about the right value; the contribution of the outlier has been down-weighted, and 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 as it does in least squares. The **bottom row** shows close ups of the fitted line and the non-outlying data points, for the same cases.

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 the new estimate of w predicts, 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.

```
Determine:
    $n$—the smallest number of points required (e.g., for lines, $n = 2$,
      for circles, $n = 3$)
    $k$—the number of iterations required
    $t$—the threshold used to identify a point that fits well
    $d$—the number of nearby points required
      to assert a model fits well
Until $k$ iterations have occurred
    Draw a sample of $n$ points from the data
      uniformly and at random
    Fit to that set of $n$ points
    For each data point outside the sample
      Test the distance from the point to the structure
        against $t$; if the distance from the point to the structure
        is less than $t$, the point is close
    end
    If there are $d$ or more points close to the structure
      then there is a good fit. Refit the structure using all
      these points. Add the result to a collection of good fits.
end
Use the best fit from this collection, using the
  fitting error as a criterion
```

Algorithm 10.4: RANSAC: Fitting Structures Using Random Sample Consensus.

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. 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 datasets, 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, and 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 would like to choose some number of points t such that the probability that all points near the line are outliers, y^t, is small (say, less than 0.05). Notice that $y \leq (1 - w)$ (because some outliers should be far from the line), so we could choose t such that $(1 - w)^t$

is small.

10.5 FITTING USING PROBABILISTIC MODELS

It is straightforward to build probabilistic models from the fitting procedures we have described. Doing so yields a new kind of model, and a new algorithm; both are extremely useful in practice. The key is to view our observed data as having been produced by a *generative model.* The generative model specifies how each data point was produced.

In the simplest case, line fitting with least squares, we can recover the same equations we worked with in Section 10.2.1 by using a natural generative model. Our model for producing data is that the x coordinate is uniformly distributed and the y coordinate is generated by (a) finding the point $ax_i + b$ on the line corresponding to the x coordinate and then (b) adding a zero mean normally distributed random variable. Now write $x \sim p$ to mean that x is a sample from the probability distribution p; write $U(R)$ for the uniform distribution over some range of values R; and write $N(\mu, \sigma^2)$ for the normal distribution with mean mu and variance σ^2. With our notation, we can write:

$$\begin{aligned} x_i &\sim U(R) \\ y_i &\sim N(ax_i + b, \sigma^2). \end{aligned}$$

We can estimate the unknown parameters of this model in a straightforward way. The important parameters are a and b (though knowing σ might be useful). The usual way to estimate parameters in a probabilistic model is to maximize the likelihood of the data, typically by working with the negative log-likelihood and minimizing that. In this case, the log-likelihood of the data is

$$\begin{aligned} \mathcal{L}(a, b, \sigma) &= \sum_{i \in \text{data}} \log P(x_i, y_i | a, b, \sigma) \\ &= \sum_{i \in \text{data}} \log P(y_i | x_i, a, b, \sigma) + \log P(x_i) \\ &= \sum_{i \in \text{data}} -\frac{(y_i - (ax_i + b))^2}{2\sigma^2} - \frac{1}{2}\log 2\pi\sigma^2 + K_b \end{aligned}$$

where K_b is a constant representing $\log P(x_i)$. Now, to minimize the negative log-likelihood as a function of a and b we could minimize $\sum_{i \in \text{data}} (y_i - (ax_i + b))^2$ as a function of a and b (which is what we did for least-squares line fitting in Section 10.2.1).

Now consider total least-squares line fitting. Again, we can recover the equations we worked with in Section 10.2.1 from a natural generative model. In this case, to generate a data point (x_i, y_i), we generate a point (u_i, v_i) uniformly at random along the line (or rather, along a finite length segment of the line likely to be of interest to us), then sample a distance ξ_i (where $\xi_i \sim N(0, \sigma^2)$, and move the point (u_i, v_i) perpendicular to the line by that distance. If the line is $ax + by + c = 0$ and if $a^2 + b^2 = 1$, we have that $(x_i, y_i) = (u_i, v_i) + \xi_i(a, b)$. We can write the log-likelihood of the data under this model as

$$\begin{aligned}\mathcal{L}(a,b,c,\sigma) &= \sum_{i\in\text{data}} \log P(x_i, y_i|a,b,c,\sigma) \\ &= \sum_{i\in\text{data}} \log P(\xi_i|\sigma) + \log P(u_i, v_i|a,b,c).\end{aligned}$$

But $P(u_i, v_i|a,b,c)$ is some constant, because this point is distributed uniformly along the line. Since ξ_i is the perpendicular distance from (x_i, y_i) to the line (which is $\|(ax_i + by_i + c)\|$ as long as $a^2 + b^2 = 1$), we must maximize

$$\begin{aligned}\sum_{i\in\text{data}} \log P(\xi_i|\sigma) &= \sum_{i\in\text{data}} -\frac{\xi_i^2}{2\sigma^2} - \frac{1}{2}\log 2\pi\sigma^2 \\ &= \sum_{i\in\text{data}} -\frac{(ax_i + by_i + c)^2}{2\sigma^2} - \frac{1}{2}\log 2\pi\sigma^2\end{aligned}$$

(again, subject to $a^2 + b^2 = 1$). For fixed (but perhaps unknown) σ this yields the problem we were working with in Section 10.2.1. So far, generative models have just reproduced what we know already, but a powerful trick makes them much more interesting.

10.5.1 Missing Data Problems

A number of important vision problems can be phrased as problems that happen to be missing useful elements of the data. For example, we can think of segmentation as the problem of determining from which of a number of sources a measurement came. This is a general view. More specifically, fitting a line to a set of tokens involves segmenting the tokens into outliers and inliers, then fitting the line to the inliers; segmenting an image into regions involves determining which source of color and texture pixels generated the image pixels; fitting a set of lines to a set of tokens involves determining which tokens lie on which line; and segmenting a motion sequence into moving regions involves allocating moving pixels to motion models. Each of these problems would be easy if we happened to possess some data that is currently missing (respectively, whether a point is an inlier or an outlier, which region a pixel comes from, which line a token comes from, and which motion model a pixel comes from).

A *missing data problem* is a statistical problem where some data is missing. There are two natural contexts in which missing data are important: In the first, some terms in a data vector are missing for some instances and present for others (perhaps someone responding to a survey was embarrassed by a question). In the second, which is far more common in our applications, an inference problem can be made much simpler by rewriting it using some variables whose values are unknown. Fortunately, there is an effective algorithm for dealing with missing data problems; in essence, we take an expectation over the missing data. We demonstrate this method and appropriate algorithms with two examples.

Example: Outliers and Line Fitting

We wish to fit a line to a set of tokens that are at $\boldsymbol{x}_i = (x_i, y_i)$. Some tokens might be outliers, but we do not know which ones are. This means we can model the process of generating a token as first, choosing whether it will come from the line or be an outlier, and then, choosing the token conditioned on the original choice. The first choice will be random, and we can write $P(\text{token comes from line}) = \pi$. We have already given two models of how a point could be generated from a line model. We model outliers as occuring uniformly and at random on the plane. This means that we can write the probability of generating a token as

$$\begin{aligned} P(\boldsymbol{x}_i|a,b,c,\pi) &= P(\boldsymbol{x}_i, \text{line}|a,b,c,\pi) + P(\boldsymbol{x}_i, \text{outlier}|a,b,c,\pi) \\ &= P(\boldsymbol{x}_i|\text{line},a,b,c)P(\text{line}) + P(\boldsymbol{x}_i|\text{outlier},a,b,c)P(\text{outlier}) \\ &= P(\boldsymbol{x}_i|\text{line},a,b,c)\pi + P(\boldsymbol{x}_i|\text{outlier},a,b,c)(1-\pi). \end{aligned}$$

If we knew for every data item whether it came from the line or was an outlier, then fitting the line would be simple; we would ignore all the outliers, and apply the methods of Section 10.2.1 to the other points. Similarly, if we knew the line, then estimating which point is an outlier and which is not would be straightforward (the outliers are far from the line). The difficulty is that we do not; the key to resolving this difficulty is repeated re-estimation (Section 10.5.3), which provides a standard algorithm for this class of problem. Figure 10.9 shows typical results using the standard algorithm.

By a very small manipulation of the equations above (replace "line" with "background" and "outlier" with "foreground"), we can represent a background subtraction problem, too. We model the image in each frame of video as the same, multiplied by some constant to take account of automatic gain control, but with noise added. We model the noise as coming from some uniform source. Figures 10.10 and 10.11 show results, obtained with the standard algorithm for these problems (Section 10.5.3).

Example: Image Segmentation

At each pixel in an image, we compute a d-dimensional feature vector $\boldsymbol{x}$, which might contain position, color, and texture information. We believe the image contains g segments, and each pixel is produced by one of these segments. Thus, to produce a pixel, we choose an image segment and then generate the pixel from the model of that segment. We assume that the lth segment is chosen with probability π_l, and we model the density associated with the lth segment as a Gaussian, with known covariance Σ and unknown mean $\theta_l = (\boldsymbol{\mu}_l)$ that depends on the particular segment. We encapsulate these parameters into a parameter vector to get $\Theta = (\pi_1, \ldots, \pi_g, \theta_1, \ldots, \theta_g)$. This means that we can write the probability of generating a pixel vector $\boldsymbol{x}$ as

$$p(\boldsymbol{x}|\Theta) = \sum_i p(\boldsymbol{x}|\theta_l)\pi_l.$$

Fitting this model would be simple if we knew which segment produced which pixel, because then we could estimate the mean of each segment separately. Similarly, if we knew the means, we could estimate which segment produced the pixel. This is quite a general situation.

10.5.2 Mixture Models and Hidden Variables

Each of the previous examples are instances of a general form of model, known as a *mixture model*, where a data item is generated by first choosing a mixture component (the line or the outlier; which segment the pixel comes from), then generating the data item from that component. Call the parameters for the lth component θ_l, the probability of choosing the lth component π_l, and write $\Theta = (\pi_1, \ldots, \pi_l, \theta_1, \ldots, \theta_l)$. Then, we can write the probability of generating $\boldsymbol{x}$

$$p(\boldsymbol{x}|\Theta) = \sum_j p(\boldsymbol{x}|\theta_j)\pi_j.$$

This is a weighted sum, or *mixture*, of probability models; the π_l are usually called *mixing weights*. One can visualize this model as a density in the space of $\boldsymbol{x}$ that consists of a set of g "blobs" of probability, each of which is associated with a component of the model. We want to determine: (a) the parameters of each of these blobs, (b) the mixing weights, and usually (c) from which component each token came. The log-likelihood of the data for a general mixture model is

$$\mathcal{L}(\Theta) = \sum_{i \in \text{observations}} \log \left(\sum_{j=1}^{g} \pi_j p_j(\boldsymbol{x}_i|\theta_j) \right).$$

This function is hard to maximize, because of the sum inside the logarithm. Just like the last two examples, the problem would be simplified if we knew the mixture component from which each token came, because then we would estimate the components independently.

We now introduce a new set of variables. For each data item, we have a vector of indicator variables (one per component) that tells us from which component each data item came. We write δ_i for the vector associated with the ith data item, and δ_{ij} for the j'th component of δ_i. Then, we have

$$\delta_{ij} = \begin{cases} 1 & \text{if item } i \text{ came from component } j \\ 0 & \text{otherwise} \end{cases}.$$

and these variables are unknown. If we did know these variables, we could maximize the *complete data log-likelihood*,

$$\mathcal{L}_c(\Theta) = \sum_{i \in \text{observations}} \log P(\boldsymbol{x}_i, \delta_i|\Theta),$$

which would be quite easy to do (because it would boil down to estimating the components independently). We regard δ as part of our data that happens to be missing (which is why we call this the complete data log-likelihood). The form of $\mathcal{L}_c(\Theta)$ for mixture models is worth remembering because it involves a neat trick:

using the δ_{ij} to switch on and off terms. We have

$$\begin{aligned}
\mathcal{L}_c(\Theta) &= \sum_{i \in \text{observations}} \log P(\boldsymbol{x}_i, \delta_i|\Theta) \\
&= \sum_{i \in \text{observations}} \log \prod_{j \in \text{components}} \left[p_j(\boldsymbol{x}_i|\theta_j)\pi_j\right]^{\delta_{ij}} \\
&= \sum_{i \in \text{observations}} \left(\sum_{j \in \text{components}} \left[(\log p_j(\boldsymbol{x}_i|\theta_j) \log \pi_j)\, \delta_{ij}\right] \right)
\end{aligned}$$

(keeping in mind that the δ_{ij} are either one or zero, and that $\sum_j \delta_{ij} = 1$, equivalent to requiring that each data point comes from exactly one model).

10.5.3 The EM Algorithm for Mixture Models

For each of our examples, if we knew the missing data, we could estimate the parameters effectively. Similarly, if we knew the parameters, the missing data would follow. This suggests an iterative algorithm:

1. Obtain some estimate of the missing data using a guess at the parameters.
2. Form a maximum likelihood estimate of the free parameters using the estimate of the missing data.

We would iterate this procedure until (hopefully!) it converged. In the case of line fitting, the algorithm would look like this:

1. Obtain some estimate of which points lie on the line and which are off lines, using an estimate of the line.
2. Form a revised estimate of the line, using this information.

For image segmentation, this would look like the following:

1. Obtain some estimate of the component from which each pixel's feature vector came, using an estimate of the θ_l.
2. Update the θ_l and the mixing weights, using this estimate.

Although it would be nice if the procedures given for missing data converged, there is no particular reason to believe that they do. In fact, given appropriate choices in each stage, they do. This is most easily seen by showing that they are examples of a general algorithm—the *expectation-maximization* (EM) algorithm.

The key idea in EM is to obtain a set of working values for the missing data (and so for Θ) by substituting an expectation for each missing value. In particular, we fix the parameters at some value, and then compute the expected value of each δ_{ij}, given the value of $\boldsymbol{x}_i$ and the parameter values. We then plug the expected value of δ_{ij} into the complete data log-likelihood, which is much easier to work with, and obtain a value of the parameters by maximizing that. At this point, the expected values of δ_{ij} may have changed. We obtain an algorithm by alternating the expectation step with the maximization step and iterate until convergence. More formally, given $\Theta^{(s)}$, we form $\Theta^{(s+1)}$ by:

1. Computing an expected value for the *complete* data log-likelihood using the incomplete data and the current value of the parameters. That is, we compute
$$Q(\Theta;\Theta^{(s)}) = E_{\delta|\boldsymbol{x},\Theta^{(s)}}\mathcal{L}_c(\Theta).$$
Notice that this object is a *function* of Θ, obtained by taking an expectation of a function of Θ and δ; the expectation is with respect to $P(\delta|\boldsymbol{x},\Theta^{(s)})$. This is referred to as the *E-step*.
2. Maximizing this object as a function of Θ. That is, we compute
$$\Theta^{(s+1)} = \arg\max_{\Theta} Q(\Theta;\Theta^{(s)}).$$
This is known as the *M-step*.

It can be shown that the incomplete data log-likelihood is increased at each step, meaning that the sequence $\boldsymbol{u}^s$ converges to a (local) maximum of the incomplete data log-likelihood (e.g., Dempster *et al.* (1977) or McLachlan and Krishnan (1996)). Of course, there is no guarantee that this algorithm converges to the *correct* local maximum, and finding the correct local maximum can be a mild nuisance.

EM is considerably easier than it looks for a mixture model. First, recall from Section 10.5.2 that the complete data log-likelihood for a mixture model is
$$\mathcal{L}_c(\Theta) = \sum_{i\in\text{observations}}\ \sum_{j\in\text{components}} \left[(\log p_j(\boldsymbol{x}_i|\theta_j)\log\pi_j)\,\delta_{ij}\right].$$
which is *linear* in δ. Because taking expectations is linear, $Q(\Theta;\Theta^{(s)})$ can be obtained from $\mathcal{L}_c(\Theta)$ by substituting the expected values of δ_{ij}. Now write
$$\alpha_{ij} = E_{\delta|\boldsymbol{x},\Theta^{(s)}}[\delta_{ij}]$$
(which is the expected value of δ_{ij}, taking the expectation using the posterior on δ given data and the current estimate of parameters $\Theta^{(s)}$); these are commonly called *soft weights*. We can now write
$$Q(\Theta;\Theta^{(s)}) = \sum_{i\in\text{observations}}\ \sum_{j\in\text{components}} \left[(\log p_j(\boldsymbol{x}_i|\theta_j)\log\pi_j)\,\alpha_{ij}\right].$$
Second, notice that the ith missing variable is conditionally independent of all others given the ith data point and the parameters of the model. If you find this confusing, think about the examples. In the case of line fitting, the only information you need to tell whether a particular point is an outlier is that point together with your estimate of the line; no other points have anything to say about it. Finally, notice that
$$\begin{aligned}\alpha_{ij} &= E_{\delta|\boldsymbol{x},\Theta^{(s)}}[\delta_{ij}]\\ &= E_{\delta_i|\boldsymbol{x}_i,\Theta^{(s)}}[\delta_{ij}]\\ &= 1\cdot P(\delta_{ij}=1|\boldsymbol{x}_i,\Theta^{(s)}) + 0\cdot P(\delta_{ij}=0|\boldsymbol{x}_i,\Theta^{(s)})\\ &= P(\delta_{ij}=1|\boldsymbol{x}_i,\Theta^{(s)}).\end{aligned}$$

Now we must compute

$$\begin{aligned} P(\delta_{ij}=1|\boldsymbol{x}_i,\Theta^{(s)}) &= \frac{P(\boldsymbol{x}_i,\delta_{ij}=1|\Theta^{(s)})}{P(\boldsymbol{x}_i|\Theta^{(s)})} \\ &= \frac{P(\boldsymbol{x}_i|\delta_{ij}=1,\Theta^{(s)})P(\delta_{ij}=1|\Theta^{(s)})}{P(\boldsymbol{x}_i|\Theta^{(s)})} \\ &= \frac{p_j(\boldsymbol{x}_i|\Theta^{(s)})\pi_j}{\sum_l P(\boldsymbol{x}_i,\delta_{il}=1|\Theta^{(s)})} \\ &= \frac{p_j(\boldsymbol{x}_i|\Theta^{(s)})\pi_j}{\sum_l p_l(\boldsymbol{x}_i|\Theta^{(s)})\pi_l} \end{aligned}$$

because the numerator is the probability of getting a data point out of model j, and the denominator is the probability of getting that point at all. Our steps are then as follows.

The E-Step For each i, j, compute the soft weights

$$\alpha_{ij} = P(\delta_{ij}=1|\boldsymbol{x}_i,\Theta^{(s)}) = \frac{p_j(\boldsymbol{x}_i|\Theta^{(s)})\pi_j}{\sum_l p_l(\boldsymbol{x}_i|\Theta^{(s)})\pi_l}.$$

Then, we have

$$Q(\Theta;\Theta^{(s)}) = \sum_{i\in\text{observations}} \sum_{j\in\text{components}} \left[(\log p_l(\boldsymbol{x}_i|\theta_l)\log\pi_l)\,\alpha_{ij}\right].$$

The M-Step We must maximize

$$Q(\Theta;\Theta^{(s)}) = \sum_{i\in\text{observations}} \sum_{j\in\text{components}} \left[(\log p_l(\boldsymbol{x}_i|\theta_l)\log\pi_l)\,\alpha_{ij}\right].$$

as a function of Θ. Notice that this is equivalent to allocating each data point to the j'th model with weight α_{ij}, then maximizing the likelihood of each model separately. The process behaves as if each model accounts for some fraction of each data point, which is why these terms are called soft weights. This will become more apparent when you study the equations for the examples (see the exercises).

10.5.4 Difficulties with the EM Algorithm

EM is inclined to get stuck in local minima. These local minima typically are associated with combinatorial aspects of the problem being studied. In the example of fitting lines subject to outliers, the algorithm is, in essence, trying to decide whether a point is an outlier. Some incorrect labelings might be stable. For example, if there is only one outlier, the algorithm could find a line through that point and one other, and label all remaining points outliers (Figure 10.9).

One useful strategy is to notice that the final configuration of the algorithm is a deterministic function of its start point and use carefully chosen start points. One might start in many different (randomly chosen) configurations and sift through

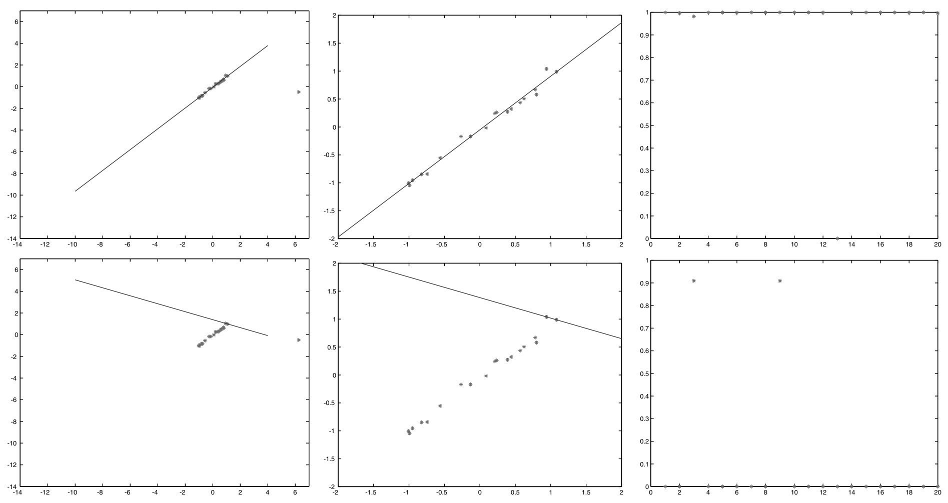

FIGURE 10.9: EM can be used to reject outliers. Here we demonstrate a line fit to the second dataset of Figure 10.5. The **top row** shows the correct local minimum, and the **bottom row** shows another local minimum. The **first column** shows the line superimposed on the data points using the same axes as Figure 10.5; the **second column** shows a detailed view of the line, indicating the region around the data points; and the **third column** shows a plot of the probability that a point comes from the line, rather than from the noise model, plotted against the index of the point. Notice that at the correct local minimum, all but one point is associated with the line, whereas at the incorrect local minimum, there are two points associated with the line and the others are allocated to noise.

the results looking for the best fit, rather like RANSAC. One might preprocess the data using something like a Hough transform to look for the best fit. Neither is guaranteed.

A second difficulty is that some points will have extremely small expected weights. This presents us with a numerical problem: it isn't clear what happens if we regard small weights as being equivalent to zero (this usually isn't a wise thing to do). In turn, we might need to adopt a numerical representation that allows us to add many very small numbers and come up with a nonzero result. This issue is rather outside the scope of this book, but you should not underestimate its nuisance value because we don't treat it in detail.

10.6 MOTION SEGMENTATION BY PARAMETER ESTIMATION

Consider two frames of a motion sequence produced by a moving camera. For a small movement, we will see relatively few new points, and lose relatively few points, so we can join each point in the first frame to its corresponding point on the second frame (which is overlaid) with an arrow. The head is at the point in the second frame, and, if the elapsed time is short, the field of arrows can be thought of as the instantaneous movement in the image. The arrows are known as the *optical flow*, a notion originally due to Gibson (1950). The structure of an optical

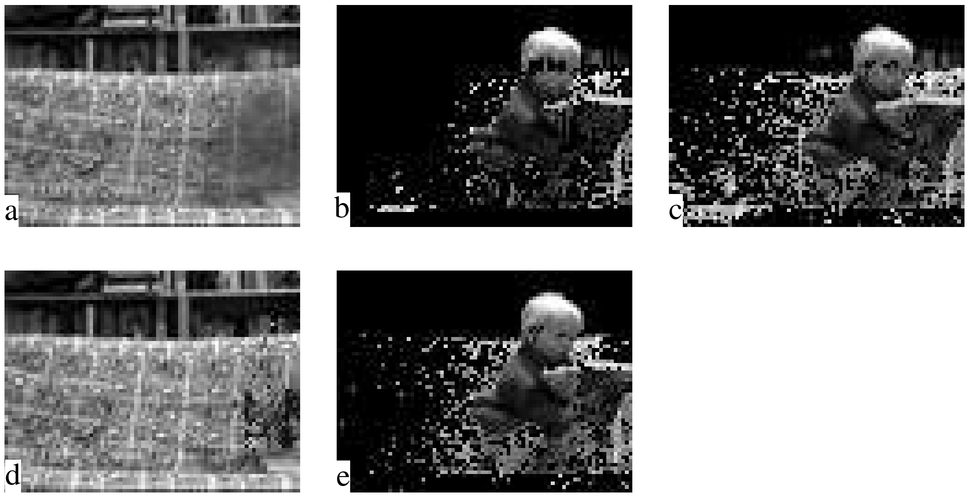

FIGURE 10.10: Background subtraction for the sequence of Figure 9.8, using EM. (**a**), (**b**), and (**c**) are from Figure 9.9, for comparison. (**d**) shows the estimated background and (**e**) shows the estimated foreground. Notice that, in each case, there are some excess pixels and some missing pixels.

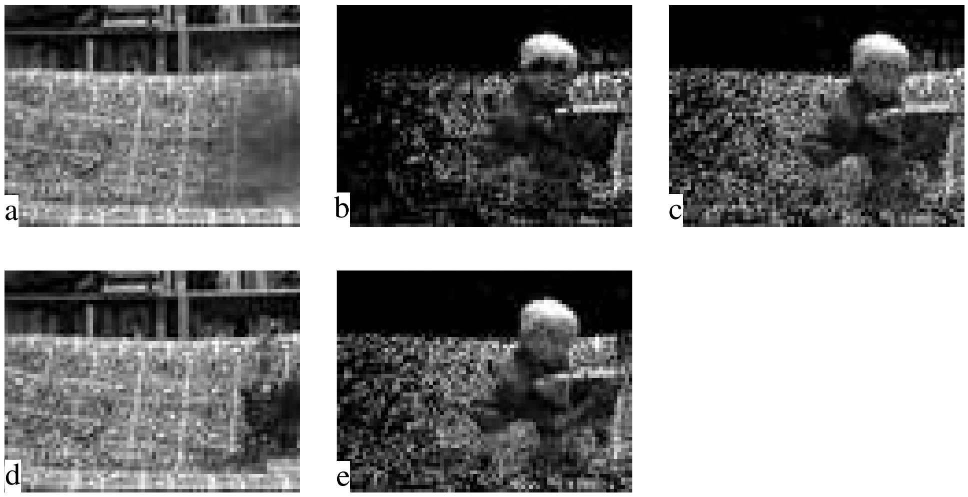

FIGURE 10.11: Background subtraction for the sequence of Figure 9.8, using EM. (**a**), (**b**), and (**c**) are from Figure 9.10, for comparison. (**d**) shows the estimated background, and (**e**) shows the estimated foreground. Notice that the number of problem pixels—where the pattern on the sofa has been mistaken for the child—has markedly increased. This is because small movements can cause the high spatial frequency pattern on the sofa to be misaligned, leading to large differences.

flow field can be quite informative about a scene (Section 10.6.1), and quite simple parametric models of optical flow are often good representations (Section 10.6.2).

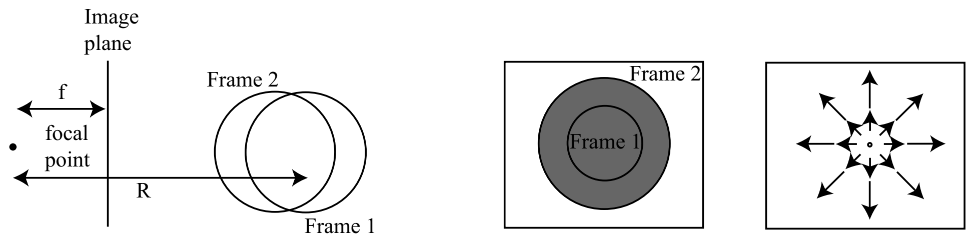

FIGURE 10.12: A sphere of radius R approaches a camera along the Z axis, at velocity V (side view on the **left**). The image is a circle, which grows as the sphere gets closer (**center**). The flow is radial, about a focus of expansion, and provides an estimate of the time to contact (**right**). This estimate works for other objects, too.

As a result, motion sequences often consist of large regions that have similar motion internally. In turn, this gives us a segmentation principle: we want to decompose a motion sequence into a set of moving layers, that compose to make the sequence (Section 10.6.3).

10.6.1 Optical Flow and Motion

Flow is particularly informative about relations between the viewer's motion, usually called *egomotion*, and the 3D scene. For example, when viewed from a moving car, distant objects have much slower *apparent motion* than close objects, so the rate of apparent motion can tell us something about distance. This means that the flow arrows on distant objects will be shorter than those on nearby objects. As another example, assume the egomotion is pure translation in some direction. Then the image point in that direction, which is known as the *focus of expansion*, will not move, and all the optical flow will be away from that point (Figure 10.12). This means that simply observing such a flow field tells us something about how we are moving. Further simple observations tell us how quickly we will hit something. Assume the camera points at the focus of expansion, and make the world move to the camera. A sphere of radius R whose center lies along the direction of motion and is at depth Z will produce a circular image region of radius $r = fR/Z$. If it moves down the Z axis with speed $V = dZ/dt$, the rate of growth of this region in the image will be $dr/dt = -fRV/Z^2$. This means that

$$\text{time to contact} = -\frac{Z}{V} = \frac{r}{\left(\frac{dr}{dt}\right)}.$$

The minus sign is because the sphere is moving down the Z axis, so Z is getting smaller and V is negative. The object doesn't need to be a sphere for this argument to work, and if the camera is spherical, we don't need to be looking in the direction we are traveling either. This means that an animal that is translating quickly can get an estimate of how long until it hits something very quickly and very easily.

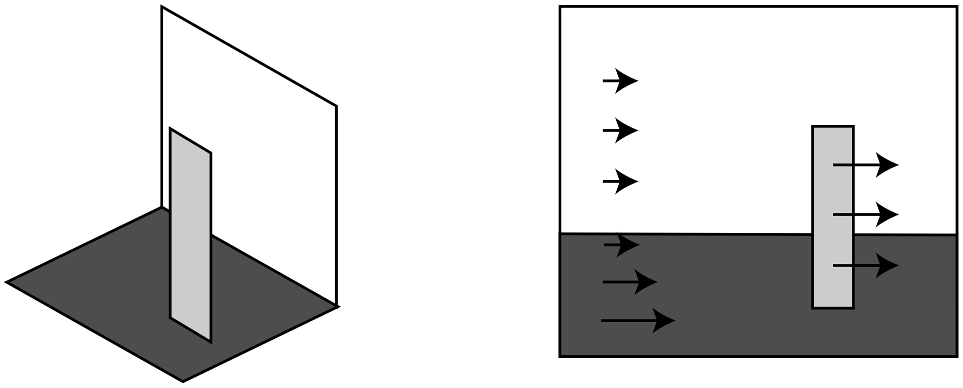

FIGURE 10.13: Optic flow fields can be used to structure or segment a scene. On the **left**, a very simple scene. Now imagine we view this scene with a camera whose image plane is parallel to the white rectangle, and that is moving left; we will see flow fields that look like the image on the **right**. The flow on the white rectangle is constant (because the plane is parallel to the direction of translation and the image plane) and small (it is distant); on the light gray rectangle, it is constant, but larger; and on the inclined plane, it is small at distant points and large at nearby points. With a parametric model of such flow fields, we could segment scenes like this, because different structures would correspond to different flow fields.

10.6.2 Flow Models

Quite simple parametric flow models can group together parts of a scene (Figure 10.13). It is helpful to build models that are linear in their parameters. Writing θ_i for the ith component of the parameter vector, $\boldsymbol{F}_i$ for the ith flow basis vector field, and $\boldsymbol{v}(\boldsymbol{x})$ for the flow vector at pixel $\boldsymbol{x}$, one has

$$\boldsymbol{v}(\boldsymbol{x}) = \sum_i \theta_i \boldsymbol{F}_i$$

In the *affine motion model*, we have

$$\boldsymbol{v}(\boldsymbol{x}) = \begin{pmatrix} 1 & x & y & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & x & y \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \\ \theta_5 \\ \theta_6 \end{pmatrix}.$$

If flows involve what are essentially 2D effects—this is particularly appropriate for lateral views of human limbs—a set of basis flows that encodes translation, rotation and some affine effects is probably sufficient. Write (x, y) for the components of $\boldsymbol{x}$.

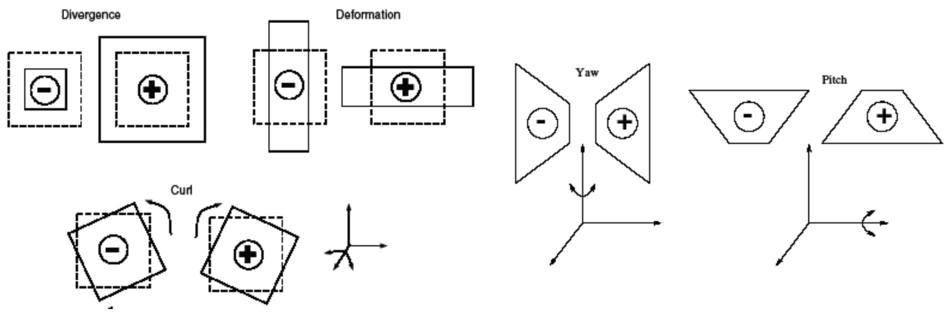

FIGURE 10.14: Typical flows generated by the model $(u(\boldsymbol{x}), v(\boldsymbol{x})^T = (\theta_1 + \theta_2 x + \theta_3 y + \theta_7 x^2 + \theta_8 xy, \theta_4 + \theta_5 x + \theta_6 y + \theta_y xy + \theta_8 y^2)$. Different values of the θ_i give different flows, and the model can generate flows typical of a 2D figure moving in 3D. **Divergence** occurs when the image is scaled; for example, $\theta = (0, 1, 0, 0, 0, 1, 0, 0)$. **Deformation** occurs when one direction shrinks and another grows (for example, rotation about an axis parallel to the view plane in an orthographic camera); for example, $\theta = (0, 1, 0, 0, 0, -1, 0, 0)$. **Curl** can result from in plane rotation; for example, $\theta = (0, 0, -1, 0, 1, 0, 0, 0)$. **Yaw** models rotation about a vertical axis in a perspective camera; for example $\theta = (0, 0, 0, 0, 0, 1, 0)$. Finally, **pitch** models rotation about a horizontal axis in a perspective camera; for example $\theta = (0, 0, 0, 0, 0, 0, 1)$. *This figure was originally published as Figure 2 of "Cardboard People: A Parameterized Model of Articulated Image Motion," S. Ju, M. Black, and Y. Yacoob, IEEE Int. Conf. Face and Gesture, 1996 © IEEE, 1996.*

One can obtain such flows using the simple model

$$\boldsymbol{v}(\boldsymbol{x}) = \begin{pmatrix} 1 & x & y & 0 & 0 & 0 & x^2 & xy \\ 0 & 0 & 0 & 1 & x & y & xy & y^2 \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \\ \theta_5 \\ \theta_6 \\ \theta_7 \\ \theta_8 \end{pmatrix}.$$

This model is linear in θ, and provides a reasonable encoding of flows resulting from 3D motions of a 2D rectangle (see Figure 10.14). Alternatively, we could obtain basis flows by a singular value decomposition of a pool of examples of the types of flow one would like to track, and try to find a set of basis flows that explains most of the variation (for examples, see Ju *et al.* (1996)).

10.6.3 Motion Segmentation with Layers

We now wish to segment a video sequence using a parametric flow model. Assume for the moment that there are just two frames in the sequence and that we know there are k segments (otherwise, we will need to use the methods of Section 10.7 to search over different numbers of segments). We will estimate a flow model for the two frames that is a mixture of k parametric flow models. The motion at each pixel in the first frame will come from this mixture, and will take the pixel to some

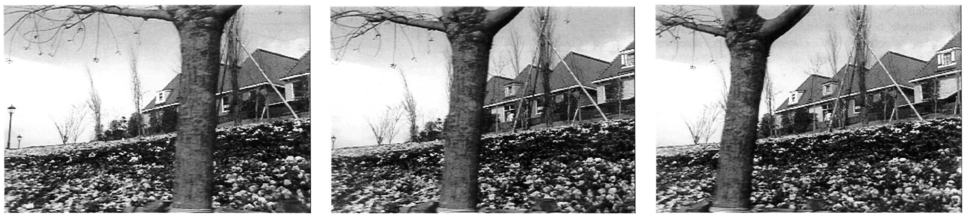

FIGURE 10.15: Frames 1, 15, and 30 of the MPEG flower garden sequence, which is often used to demonstrate motion segmentation algorithms. This sequence appears to be taken from a translating camera, with the tree much closer to the camera than the house and a flower garden on the ground plane. As a result, the tree appears to be translating quickly across the frame, and the house slowly; the plane generates an affine motion field. *This figure was originally published as Figure 6 from "Representing moving images with layers," by J. Wang and E.H. Adelson, IEEE Transactions on Image Processing, 1994, © IEEE, 1994.*

pixel in the second frame, which we expect will have the same brightness value. We could segment the first image (or the second; when we have the flow model, this doesn't really matter) by assigning each pixel to its flow model, so the pixels whose flow came from the first model would be in segment one, and so on. This model encapsulates a set of distinct, internally consistent motion fields, one per flow model. These might come from, say, a set of rigid objects at different depths and a moving camera (Figure 10.15). The separate motion fields are often referred to as *layers* and the model as a *layered motion* model.

Given a pair of images, we wish to determine (a) which motion field a pixel belongs to and (b) the parameter values for each field. All this should look a great deal like the first two examples, in that if we knew the first, the second would be easy, and if we knew the second, the first would be easy. This is again a missing data problem: the missing data is the motion field to which a pixel belongs, and the parameters are the parameters of each field and the mixing weights.

To work out the problem, we also need a probabilistic model of our observations. We assume that the intensity of a pixel in image two is obtained by taking the pixel in image one, moving it along the flow arrow for that pixel, and then adding a zero-mean Gaussian random variable with variance σ^2. Now assume that the pixel at (x, y) in the first image belongs to the lth motion field, with parameters θ_l. This means that this pixel has moved to $(x, y) + \boldsymbol{v}(x, y; \theta_l)$ in the second frame, and so that the intensity at these two pixels is the same, up to measurement noise. We write $I_1(x, y)$ for the image intensity of the first image at the x, yth pixel, and so on. The missing data is the motion field to which the pixel belongs. We can represent this by an indicator variable $V_{xy,j}$, where

$$V_{uv,j} = \left\{ \begin{array}{c} 1, \text{if the } x,\, y\text{th pixel belongs to the } j\text{th motion field} \\ 0, \text{otherwise} \end{array} \right\}.$$

The complete data log-likelihood becomes

$$L(V, \Theta) = -\sum_{xy,j} V_{xy,j} \frac{(I_1(x, y) - I_2(x + v_1(x, y; \theta_j), y + v_2(x, y; \theta_j)))^2}{2\sigma^2} + C,$$

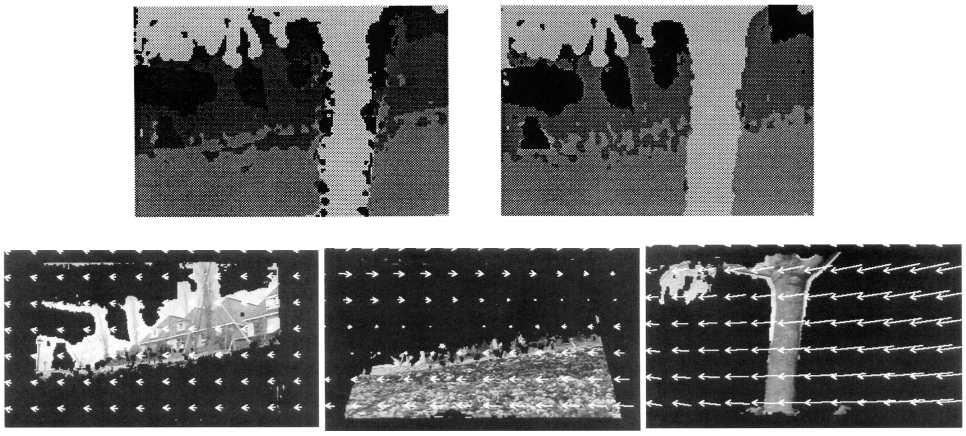

FIGURE 10.16: On the **top left**, a map indicating to which layer pixels in a frame of the flower garden sequence belong, obtained by clustering local estimates of image motion. Each gray level corresponds to a layer, and each layer is moving with a different affine motion model. This map can be refined by checking the extent to which the motion of pixel neighborhoods is consistent with neighborhoods in future and past frames, resulting in the map on the **top right**. Three of the layers and their motion models are shown on the **bottom**. *This figure was originally published as Figures 11 and 12 from "Representing moving images with layers," by J. Wang and E.H. Adelson, IEEE Transactions on Image Processing, 1994, © IEEE, 1994.*

where $\Theta = (\theta_1, \ldots, \theta_k)$. Setting up the EM algorithm from here on is straightforward. As before, the crucial issue is determining

$$P\left\{V_{xy,j} = 1 | I_1, I_2, \Theta\right\}.$$

These probabilities are often represented as *support maps*—maps assigning a gray-level representing the maximum probability layer to each pixel (Figure 10.16).

Layered motion representations are useful for several reasons: First, they cluster together points moving "in the same way." Second, they expose motion boundaries. Finally, new sequences can be reconstructed from the layers in interesting ways (Figure 10.17).

10.7 MODEL SELECTION: WHICH MODEL IS THE BEST FIT?

To date, we have assumed that we knew how many components our model has. For example, we assumed that we were fitting a single line; in the image segmentation example, we assumed we knew the number of segments; for general mixture models, we assumed the number of components was known. Generally, this is not a safe assumption.

We could fit models with different numbers of components (such as lines, segments, and so on), and see which model fits best. This strategy fails, because the model with more components will *always* fit best. In the extreme case, a really good fit of lines to points involves passing one line through each pair of points. This representation will be a perfect fit to the data, but will be useless in almost every

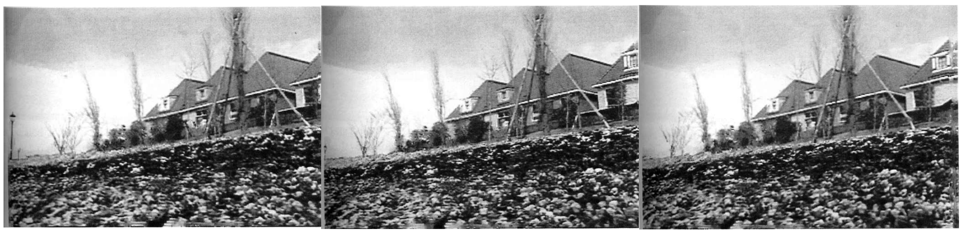

FIGURE 10.17: One feature of representing motion in terms of layers is that one can reconstruct a motion sequence *without* some of the layers. In this example, the MPEG garden sequence has been reconstructed with the tree layer omitted. The figure on the **left** shows frame 1, that in the **center** shows frame 15, and that on the **right** shows frame 30. *This figure was originally published as Figure 13 from "Representing moving images with layers," by J. Wang and E.H. Adelson, IEEE Transactions on Image Processing, 1994, © IEEE, 1994.*

case. It will be useless because it is too complex to manipulate and because it will be very poor at predicting new data.

Another way to look at this point is as a trade off between bias and variance. The data points are a sample that comes from some underlying process, that we are trying to represent. Representing a lot of data points with, say, a single line is a biased representation, because it cannot represent all the complexity of the model that produced the dataset. Some information about the underlying process is inevitably lost. However, we can estimate the properties of the line used to represent the data points very accurately indeed with some care, so there is little variance in our estimate of the model that we do fit. Alternatively, if we were to represent the data points with a zigzag set of lines that joined them up, the representation would have no bias, but would be different for each new sample of data points from the same source. As a result, our estimate of the model we fit changes wildly from sample to sample; it is overwhelmed by variance.

We want a trade off. Fitting error gets smaller with the number of parameters, so we need to add a term to the fitting error that *increases* with the number of components. This penalty compensates for the decrease in fitting error (equivalently, negative log-likelihood) caused by the increasing number of parameters. Instead, we can choose from a variety of techniques, each of which uses a different discount corresponding to a different extremality principle and different approximate estimates of the criterion.

Another way to look at this point is that we wish to predict future samples from the model. Our dataset is a sample from a parametric model that is a member of a family of models. A proper choice of the parameters predicts future samples from the model—a *test set—as well as* the dataset (which is often called the *training set*). Unfortunately, these future samples are not available. Furthermore, the estimate of the model's parameters obtained using the dataset is likely to be biased because the parameters chosen ensure that the model is an optimal fit to the *training set*, rather than to the entire set of possible data. The effect is known as *selection bias*. The training set is a subset of the entire set of data that could have been drawn from the model; it represents the model exactly only if it is infinitely large. This is why the negative log-likelihood is a poor guide to the choice of model: the fit looks better because it is increasingly biased.

Now write the best choice of parameters as Θ^* and the log-likelihood of the fit to the dataset as $L(\boldsymbol{x}; \Theta^*)$, p for the number of free parameters, and N for the number of data items. We will compute a score from the log-likelihood and a penalty that discourages too many parameters. There are several possibilities for the score, but the procedure involves searching a space of models to find the one that optimizes this score (for example, we could increase the number of components).

AIC: An Information Criterion

Akaike proposed a penalty, widely called *AIC* (for "an information criterion," *not* "Akaike information criterion"), that leads to choosing the model with the minimum value of

$$-2L(\boldsymbol{x}; \Theta^*) + 2p.$$

There is a collection of statistical debate about the AIC. The first main point is that it lacks a term in the *number* of data points. This is suspicious because our estimate of the parameters of the real model should get better as the number of data points goes up. Second, there is a body of experience that the AIC tends to *overfit*—that is, to choose a model with too many parameters that fits the training set well but doesn't perform as well on test sets.

Bayesian Methods and Schwartz's BIC

For simplicity, let us write $\mathcal{D}$ for the data, $\mathcal{M}$ for the model, and θ for the parameters. Bayes' rule then yields:

$$\begin{aligned} P(\mathcal{M}|\mathcal{D}) &= \frac{P(\mathcal{D}|\mathcal{M})}{P}(\mathcal{M})P(\mathcal{D}) \\ &= \frac{\int P(\mathcal{D}|\mathcal{M}_i, \theta)P(\theta)d\theta P(\mathcal{M})}{P(\mathcal{D})}. \end{aligned}$$

Now we could choose the model for which the posterior is large. Computing this posterior can be difficult, but, by a series of approximations, we can obtain a criterion

$$-L(\mathcal{D}; \theta^*) + \frac{p}{2}\log N$$

(where N is the number of data items). Again, we choose the model that minimizes this score. This is called the *Bayes information criterion*, or BIC. Notice that this does have a term in the number of data items.

Description Length

Models can be selected by criteria not intrinsically statistical. After all, we are selecting the model, and we can say why we want to select it. A criterion that is somewhat natural is to choose the model that encodes the dataset most crisply. This *minimum description length* criterion chooses the model that allows the most efficient transmission of the dataset. To transmit the dataset, one codes and transmits the model parameters, and then codes and transmits the data given the model parameters. If the data fits the model poorly, then this latter term is large because one has to code a noise-like signal.

A derivation of the criterion used in practice is rather beyond our needs. The details appear in Rissanen (1983), (1987), and in Wallace and Freeman (1987); there are similar ideas rooted in information theory, due to Kolmogorov, and expounded in Cover and Thomas (1991). Surprisingly, the BIC emerges from this analysis, yielding

$$-L(\mathcal{D};\theta^*) + \frac{p}{2}\log N.$$

Again, we choose the model that minimizes this score.

10.7.1 Model Selection Using Cross-Validation

The key difficulty in model selection is that we should be using a quantity we can't measure: the model's ability to predict data not in the training set. Given a sufficiently large training set, we could split the training set into two components, and use one to fit the model and the other the test the fit. This approach is known as *cross-validation.*

We can use cross-validation to determine the number of components in a model by splitting the dataset into training and test data, fitting a variety of different models to training data, and then choosing the model that performs best on the test data. To evaluate performance, we could look at log-likelihood on the test data. We expect this process to estimate the number of components because a model that has too many parameters will fit the training dataset well, but predict the test set badly.

Using a single choice of a split into two components introduces a different form of selection bias, and the safest thing to do is average the estimate over all such splits. This becomes unwieldy if the test set is large, because the number of splits is huge. The most usual version is *leave-one-out cross-validation.* In this approach, we fit a model to each set of $N-1$ of the training set, compute the error on the remaining data point, and sum these errors to obtain an estimate of the model error. The model that minimizes this estimate is then chosen.

10.8 NOTES

The origins of least squares fitting are opaque to us, though we believe that Gauss himself invented the method. Total least squares appears to be due to Deming (1943). There is a large literature on fitting curves or curved surfaces using least squares methods, or approximations (one could start with work on conics (Bookstein 1979, Fitzgibbon *et al.* 1999, Kanatani 2006, Kanatani 1994, Porrill 1990, Sampson 1982); more complicated problems in (Taubin 1991)).

The Hough Transform

The Hough transform is due to Hough (1962) (a note in Keith Price's wonderful bibliography remarks: "The most cited and least read reference"). There was a large literature on the Hough transform, which was seen as having theoretical significance; the interested might start with Ballard (1981), then Ballard (1984). The topic then began to seem dated, but was revived by mean shift methods and by the observation, due to Maji and Malik (2009), that not every token needed to have the same vote, and the weights could be learned. The idea that multiple pieces of an object could reinforce one another by voting on the location of the object is very old (see Ballard (1981); Ballard (1984)); the most important recent version is Bourdev *et al.* (2010).

RANSAC

RANSAC is a hugely important algorithm, very easy to implement and use, and very effective. The original paper (Fischler and Bolles 1981) is still worth reading. There are numerous variants, depending on what one knows about the data and the problem; see Torr and Davidson (2003) and Torr and Zisserman (2000) for a start.

EM and Missing Variable Models

EM was first formally described in the statistical literature by Dempster *et al.* (1977). A very good summary reference is McLachlan and Krishnan (1996), which describes numerous variants. For example, it isn't necessary to find the maximum of $Q(\boldsymbol{u}; \boldsymbol{u}^{(s)})$; all that is required is to obtain a better value. As another example, the expectation can be estimated using stochastic integration methods.

Missing variable models seem to crop up in all sorts of places. All the models we are aware of in computer vision arise from mixture models (and so have complete data log-likelihood that is linear in the missing variables), and so we have concentrated on this case. It is natural to use a missing variable model for segmentation (see, for example Belongie *et al.* (1998*a*); Feng and Perona (1998); Vasconcelos and Lippman (1997); Adelson and Weiss (1996); or Wells *et al.* (1996)). Various forms of layered motion now exist (see Dellaert *et al.* (2000); Wang and Adelson (1994); Adelson and Weiss (1996); Tao *et al.* (2000); and Weiss (1997)); one can also construct layers that lie at the same depth (see Brostow and Essa (1999); Torr *et al.* (1999*b*); or Baker *et al.* (1998)), or have some other common property. Other interesting cases include motions resulting from transparency, specularities, etc. (see Darrell and Simoncelli (1993); Black and Anandan (1996); Jepson and Black (1993); Hsu *et al.* (1994); or Szeliski *et al.* (2000)). The resulting representation can be used for quite efficient image based rendering (see Shade *et al.* (1998)).

EM is an extremely successful inference algorithm, but it isn't magical. The primary source of difficulty for the kinds of problem that we have described is local maxima. It is common for problems that have very large numbers of missing variables to have large numbers of local maxima. This could be dealt with by starting the optimization close to the right answer, which rather misses the point.

Model Selection

Model selection is a topic that hasn't received as much attention as it deserves. There is significant work in motion, the question being which camera model (orthographic, perspective, etc.) to apply (see Torr (1999); Torr (1997); Kinoshita and Lindenbaum (2000); or Maybank and Sturm (1999)). Similarly, there is work in segmentation of range data, where the question is to what set of parametric surfaces the data should be fitted (i.e., are there two planes or three, etc.) (Bubna and Stewart 2000). In reconstruction problems, one must sometimes decide whether a degenerate camera motion sequence is present (Torr *et al.* 1999*a*). The standard problem in segmentation is how many segments are present (see Raja *et al.* (1998); Belongie *et al.* (1998*a*); and Adelson and Weiss (1996)). If one is using models predictively, it is sometimes better to compute a weighted average over model predictions (real Bayesians don't do model selection) (Torr and Zisserman 1998, Ripley 1996). We have described only some of the available methods; one important omission is Kanatani's geometric information criterion (Kanatani 1998).

PROBLEMS

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

10.2. Derive the eigenvalue problem

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

from the generative model for total least squares. This is a simple exercise—maximum likelihood and a little manipulation will do it—but worth doing right and remembering. The technique is extremely useful.

10.3. How do we get a curve of edge points from an edge detector that returns orientation? Give a recursive algorithm.

10.4. An implicit curve is given by $\phi(x, y) = 0$. For this curve, we have the property that for every point (u, v), there is exactly one point (x_0, y_0) on the curve that satisfies the local equations for the close. Show that this curve is a line. (Hint: the normals at each point on the curve must be parallel).

10.5. A slightly more stable variation of incremental fitting cuts the first few pixels and the last few pixels from the line point list when fitting the line because these pixels might have come from a corner.

(a) Why would this lead to an improvement?

(b) How should one decide how many pixels to omit?

10.6. Assume we have a fixed camera with focal length f. Write the coordinates of world points in capital letters, and of image points in lowercase letters. Place the focal point of the camera at $(0, 0, 0)$, and the image plane at $Z = -f$. We have a plane object lying on the plane $Z = aX + b$, with $|a| > 0$, $|b| > 0$.

(a) What translations of this object give image motion fields that are exactly represented by an affine motion model?

(b) Under what circumstances does an affine motion model give a reasonable approximation to the image flow fields produced by translating this object?

10.7. Refer to Section 10.5.1 for notation for the line and outliers example. Write δ_i for an indicator variable for the ith example, where $\delta_i = 1$ if the example comes from the line and $\delta_i = 0$ otherwise. Assume that we wish to fit using total least squares. Assume we know σ, the standard deviation of the errors. Assume that the probability of an outlier is independent of its position. For this example, show that the complete data log-likelihood is

$$\mathcal{L}_c(a, b, c, \pi) = \sum_i \left[-\frac{(ax_i + by_i + c)^2}{2\sigma^2} + \log \pi \right] \delta_i + \left[K + \log(1 - \pi) \right] (1 - \delta_i) + L$$

where K is a constant expressing the probability of obtaining an outlier and L does not depend on a, b, c, or π.

10.8. Refer to Section 10.5.1 and the previous example for notation for the line and outliers example. For this example, produce an expression for $\alpha_i = E_{\delta|\boldsymbol{x},\Theta^{(s)}}[\delta_i]$, where $\Theta = (a, b, c, \pi)$.

10.9. Refer to Section 10.5.1 for notation for the image segmentation example. Write δ_{ij} for an indicator variable for the ith example, where $\delta_{ij} = 1$ if the example comes from the j'th segment, and $\delta_{ij} = 0$ otherwise. Assume we know Σ, the covariance of the image segment probability distributions, and that this is the same per segment. For this example, show that the complete data log-likelihood is

$$\mathcal{L}_c(\pi_1, \ldots, \pi_g, \mu_1, \ldots, \mu_g) = \sum_{ij} \left[-\frac{(\boldsymbol{x}_i - \mu_j)^T \Sigma^{-1} (\boldsymbol{x}_i - \mu_j)}{2} + \log \pi_j \right] \delta_{ij} + L$$

where L does not depend on μ_j or π_j.

10.10. Refer to Section 10.5.1 for notation for the image segmentation example. For this example, produce an expression for $\alpha_{ij} = E_{\delta|\boldsymbol{x},\Theta^{(s)}}[\delta_{ij}]$, where $\Theta = (\pi_1, \ldots, \pi_g, \mu_1, \ldots, \mu_g)$.

10.11. Refer to Section 10.5.1 for notation for the line and outliers example. For this example, show that the updates produced in the M-step will be

$$\pi(s+1) = \frac{\sum_i \alpha_{ij}}{\sum_{i,j} \alpha_{ij}}$$

and

$$\mu_j^{(s+1)} = \frac{\sum_i \alpha_{ij} \boldsymbol{x}_i}{\sum_i \alpha_{ij}}$$

10.12. Refer to Section 10.5.1 for notation for the image segmentation example. For this example, show that the updates produced in the M-step will be

$$\pi_j^{(s+1)} = \frac{\sum_i \alpha_{ij}}{\sum_{i,j} \alpha_{ij}}$$

and

$$\mu_j^{(s+1)} = \frac{\sum_i \alpha_{ij} \boldsymbol{x}_i}{\sum_i \alpha_{ij}}$$

PROGRAMMING EXERCISES

10.13. Implement an incremental line fitter. Determine how significant a difference results from leaving out the first few pixels and the last few pixels from the line point list (put some care into building this; in our experience, it's a useful piece of software to have lying around).

10.14. Implement a Hough transform based line finder.

10.15. Count lines with an HT line finder. How well does it work?

10.16. Implement the algorithm for incremental line fitting.

10.17. Refer to Section 10.5.1 for notation for the image segmentation example. Use your expression for $\alpha_{ij} = E_{\delta|\boldsymbol{x},\Theta^{(s)}}[\delta_{ij}]$, where $\Theta = (\pi_1, \ldots, \pi_g, \mu_1, \ldots, \mu_g)$, to implement an EM algorithm to segment images. It is sufficient to use RGB color and location as a feature vector.