Chapter 17

FITTING

There are a variety of segmentation criteria that involve models at a larger scale. Typically, one wants to decompose an image or a set of tokens — which could be pixels, isolated points, sets of edge points, etc. — into components that belong to one or another simple family. For example, we might want to cluster tokens together because they form a circle (which seems to be what's going on in figure 17.1). Finding such groups is often called **fitting**. We see fitting as part of the segmentation process, because it uses a model to produce compact representations that emphasize the relevant image structures.

Fitting lines as a model problem: there is a very wide range of possible fitting strategies; to keep track of what's important, and to see ideas in a reasonable context, we need a model problem. Fitting lines is a good model problem, because it is clear what the main issues are, and the technical aspects are relatively simple.

Generally, fitting involves determining what possible structures could have given rise to a set of tokens observed in an image. For example, we might have a set of edge points (the tokens) and wish to determine which lines fit them best. There are three quite general problems that occur in fitting:

- **Parameter estimation:** In this case, we assume we know *which tokens came from a particular structure*, and we want to know what the parameters of the structure are. For example, we might have a set of edge points, *all of which are known to have come from a line*, and we wish to know what line they came from.
- **Which token came from which structure:** In this case, we assume we know *how many structures are present*, and we wish to determine which tokens came from which structure. For example, we might have a set of edge points, and we need to know the best set of lines fitting these points; this involves (1) determining which points belong together on a line and (2) figuring out what each line is. Generally, these problems are not independent (because one good way of knowing whether points belong together on a line is checking how well the best fitting line approximates them).
- **Counting:** In this case, we would like to know (1) how many structures are present (2) which points are associated with which structure and (3) what

the structures are. For example, given a set of edge points, we might want to return a set of lines that fits them well. This is, in general, a substantially difficult problem the answer to which depends strongly on the type of model adopted (for example, we could simply pass a line through every pair of edge points — this gives a set of lines that fit extremely well, but are a poor representation).

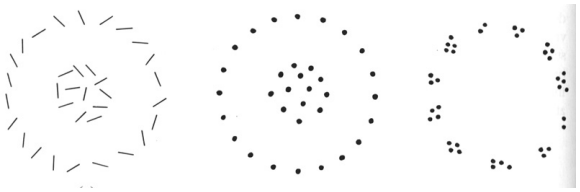

Figure 17.1. On occasion, tokens appear to be grouped together because they form useful primitives. For example, the main reason these tokens belong together appears to be that they form circles. *figure from Marr, Vision, page101, in the fervent hope that permission will be granted*

We first discuss a simple method for clustering tokens that lie on structures (section 17.1)— in practice, it is almost always used to find points that lie on lines — and then look at methods for fitting lines to point sets (section 17.2). We investigate fitting curves other than lines in section 17.3, and then discuss clustering edge points that could lie on the outlines of interesting objects (section 17.4).

17.1 The Hough Transform

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

17.1.1 Fitting Lines with the Hough Transform

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

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

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

Because the image has a known size, there is some R such that we are not interested in lines for $r > R$ — these lines will be too far away from the origin for us to see them. This means that the lines we are interested in form a bounded subset of the plane, and we discretize this with some convenient grid (which we'll discuss later). The grid elements can be thought of as buckets, into which we will sort votes. This grid of buckets is referred to as the **accumulator array**. Now 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 that there will be many votes in the grid element corresponding to that line.

17.1.2 Practical Problems with the Hough Transform

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

- **Quantization errors:** an appropriate grid size is difficult to pick. Too coarse a grid can lead to large values of the vote being obtained falsely, because many quite different lines correspond to a bucket. Too fine a value of the grid can lead to lines not being found, because votes resulting from tokens that are not exactly collinear end up in different buckets, and no bucket has a large vote (figure 17.2).

- **Difficulties with noise:** the attraction of the Hough transform is that it connects widely separated tokens that lie "close" to some form of parametric curve. This is also a weakness; it is usually possible to find many quite good phantom lines in a large set of reasonably uniformly distributed tokens. This means that, for example, regions of texture can generate peaks in the voting array that are larger than those associated with the lines sought (figures 17.4 and 17.5).

The Hough transform is worth talking about, because, despite these difficulties, it can often be implemented in a way that is quite useful for well-adapted problems. In practice, it is almost always used to find lines in sets of edge points. Useful implementation guidelines are:

- **Ensure the minimum of irrelevant tokens** this can often be done by tuning the edge detector to smooth out texture, setting the illumination to produce high contrast edges, etc.

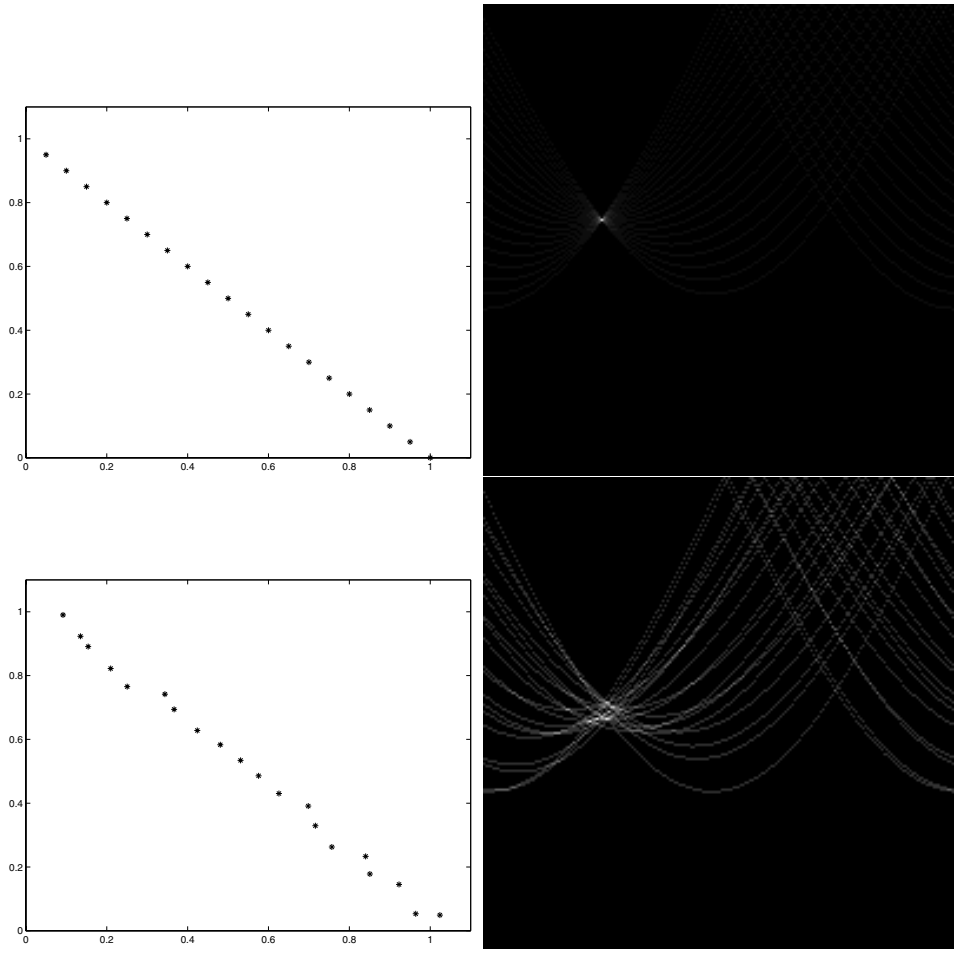

Figure 17.2. The Hough transform maps each point like token to a curve of possible lines (or other parametric curves) through that point. These figures illustrate the Hough transform for lines. The left hand column shows points, and the right hand column shows the corresponding accumulator arrays (the number of votes is indicated by the grey level, with a large number of votes being indicated by bright points). The top shows a set of 20 points drawn from a line next to 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. 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 $[0, 1.55]$. In the center, 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; the maximum vote is now 6.

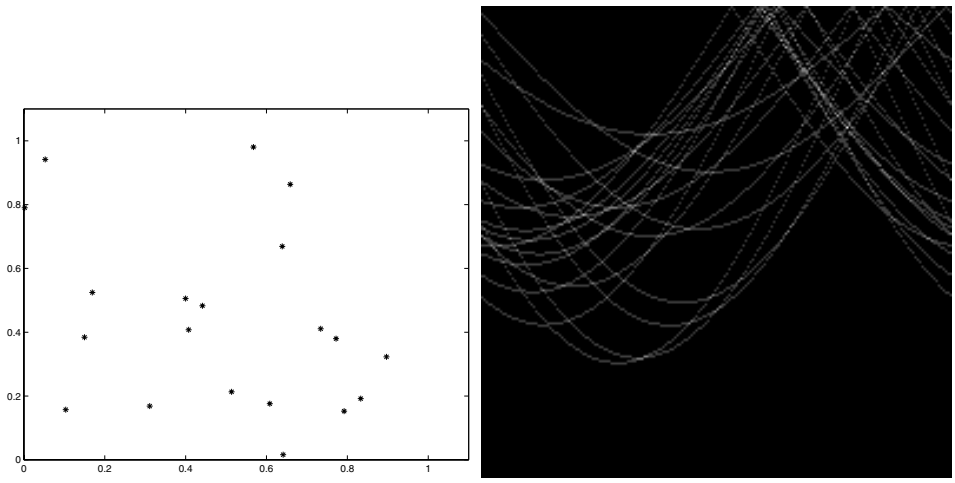

Figure 17.3. The Hough transform for a set of random points can lead to quite large sets of votes in the accumulator array. As in figure 17.2, the left hand column shows points, and the right hand column shows the corresponding accumulator arrays (the number of votes is indicated by the grey level, with a large number of votes being indicated by bright points). In this case, the data points are noise points (both coordinates are uniform random numbers in the range $[0, 1]$); the accumulator array in this case contains many points of overlap, and the maximum vote is now 4. Figures 17.4 and explore noise issues somewhat further.

- **Choose the grid carefully** this is usually done by trial and error. It can be helpful to vote for all neighbours of a grid element at the same time one votes for the element.

17.2 Fitting Lines

Line fitting is extremely useful. In many applications, objects are characterised 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 **??**). 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, and if we wish to recognise industrial parts in an image, straight lines could be helpful. This suggests a segmentation that reports all straight lines in the image.

The first step in line fitting is to establish a probabilistic model that indicates how our data relates to any underlying line; with that model in place, we will proceed to determine how points can be allocated to particular lines, and how to count lines.

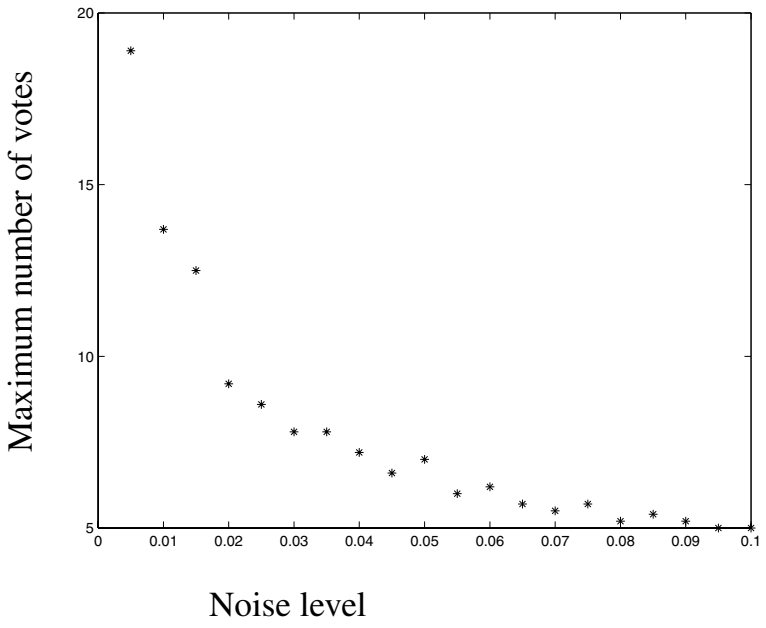

Figure 17.4. The effects of noise make it difficult to use a Hough transform robustly. The plot shows the maximum number of votes in the accumulator array for a Hough transform of 20 points on a line perturbed by uniform noise, plotted against the magnitude of the noise. The noise displaces the curves from each other, and quite quickly leads to a collapse in the number of votes. The plot has been averaged over 10 trials.

17.2.1 Least Squares, Maximum Likelihood and Parameter Estimation

Assume that all the points that belong to a particular line are known, and the parameters of the line must be found. We now need a **generative model** that indicates how our measurements were generated, given that the line was present. This generative model will give us an expression for the likelihood. In the vast majority of practical cases there is no reason to believe any one line is more likely to occur than any other, so that the distinction between maximum likelihood and MAP inference is moot. This means we wish to perform maximum likelihood inference. We adopt the notation that

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

to simplify the presentation.

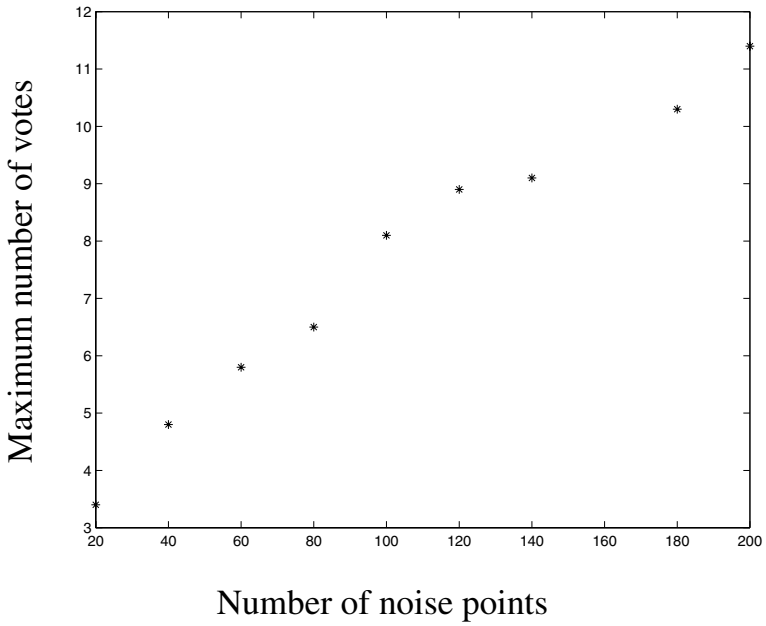

Figure 17.5. A plot of the maximum number of votes in the accumulator array for a Hough transform of a set of points whose coordinates are uniform random numbers in the range $[0, 1]$, plotted against the number of points. As the level of noise goes up, the number of votes in the right bucket goes down and the prospect of obtaining a large spurious vote in the accumulator array goes up. The plots have again been averaged over 10 trials. Compare this figure with figure 17.4, but notice the slightly different scales; the comparison suggests that it can be quite difficult to pull a line out of noise with a Hough transform (because the number of votes for the line might be comparable with the number of votes for a line due to noise). These figures illustrate the importance of ruling out as many noise tokens as possible before performing a Hough transform.

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). Question **??** asks you 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 $\text{abs}(au + bv + c)$ *if* $a^2 + b^2 = 1$. In our experience, this fact is useful enough to be worth memorizing.

Generative model: We assume that our measurements are generated by choosing a point along the line, and then perturbing it perpendicular to the line using Gaussian noise. We assume that the process that chooses points along the line is uniform — in principle, it can't be, because the line is infinitely long, but in practice

we can assume that any difference from uniformity is too small to bother with. This means we have a sequence of k measurements, (x_i, y_i), which are obtained from the model

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix} + n \begin{pmatrix} a \\ b \end{pmatrix}$$

where $n \sim N(0, \sigma)$, $au + bv + c = 0$ and $a^2 + b^2 = 1$.

Inference algorithm: The log-likelihood function is

$$-\frac{\sum_i (ax_i + by_i + c)^2}{2\sigma^2} + C$$

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

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

This means that

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

and we can substitute this back to get the eigenvalue problem

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

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

Least Squares

Least squares is a fitting procedure with a long tradition (which is the only reason we describe it!). It has the virtue of yielding a simple analysis and the very significant disadvantage of a generative model that very seldom makes sense in vision applications. For this approach, we represent a line as $y = ax + b$.

Generative model: The k measurements (x_i, y_i) are obtained from the model $y = ax + b + n$, where $n \sim N(0, \sigma)$. This means that only the y-coordinate of each measurement is affected by noise, which is why it is a rather dubious model.

Inference algorithm: The maximum likelihood estimate of a and b is easily obtained from the solution to

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

While this is a standard linear solution to a classical problem, it's actually not much help in vision applications because the model is an extremely poor model. The difficulty is that the measurement error is dependent on coordinate frame — we are counting vertical offsets from the line as errors, which means that near vertical lines lead to quite large values of the error and quite funny fits (figure 17.6). In fact, the process is so dependent on coordinate frame that it doesn't represent vertical lines at all.

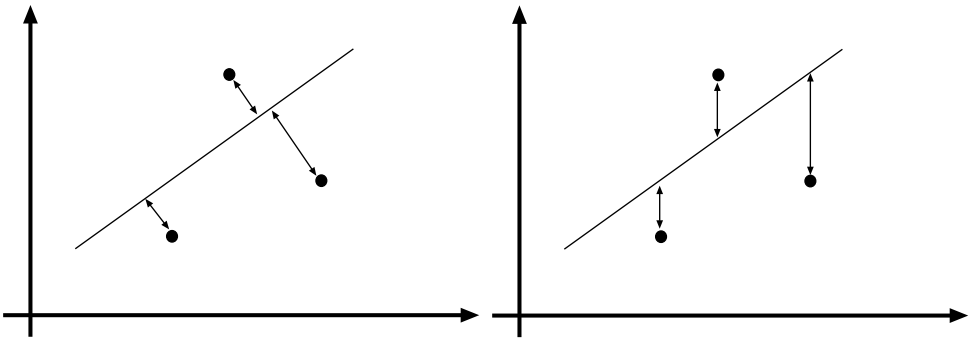

Figure 17.6. Left: Perpendicular least squares models data points as being generated by an abstract point along the line to which is added a vector perpendicular to the line, with a length given by a zero mean, Gaussian random variable. This means that the distance from data points to the line has a normal distribution. By setting this up as a maximum likelihood problem, we obtain a fitting criterion that chooses a line that minimizes the sum of distances between data points and the line. **Right:** Least squares follows the same general outline, but assumes that the error appears only in the y-coordinate. This yields a (very slightly) simpler mathematical problem, at the cost of a poor fit.

17.2.2 Which Point is on Which Line?

This problem can be very difficult, because it can involve search over a very large combinatorial space. One approach is to notice that we very seldom encounter isolated points; instead, 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 both k-means and EM algorithms can be applied.

Incremental Fitting

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 1). Incremental line fitting can work very well indeed, 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 one is interested in can reasonably be expected to form a closed curve (for example, in some object recognition applications)because it means that the algorithm reports natural groups without further fuss. This strategy often leads to occluded edges resulting in more than one fitted line. 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 two 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 back to curve
  attach line to line list
end
```

Algorithm 17.1: *Incremental line fitting by walking along a curve, fitting a line to runs of pixels along the curve, and breaking the curve when the residual is too large*

17.3 Fitting Curves

In principle, fitting curves is not very different from fitting lines. The usual generative model is that data points are generated uniformly at random on the curve, and then perturbed by Gaussian noise *normal* to the curve. This means that the distance between points and the underlying curve is normally distributed. In turn, a maximum likelihood approach minimizes the sum of distances between the points and the curve.

This generates quite difficult practical problems: it is usually very hard to tell the distance between a point and a curve. We can either solve this problem, or apply

various approximations (which are usually chosen because they are computationally simple, not because they result from clean generative models). In this section, we will suppress the probabilistic issues — which were comprehensively discussed for the case of fitting lines — and concentrate on how to solve the distance problem for the two main representations of curves.

17.3.1 Implicit Curves

The coordinates of **implicit curves** satisfy some parametric equation; if this equation is a polynomial, then the curve is said to be **algebraic**, and this case is by far the most common. Some common cases are given in table 17.1.

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

Table 17.1. Some implicit curves used in vision applications. Note that not all of these curves are guaranteed to have any real points on them — for example, $x^2 + y^2 + 1 = 0$ doesn't. Higher degree curves are seldom used, because it can be difficult to get stable fits to these curves.

The Distance from a Point to an Implicit Curve

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

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

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

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

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, as example 1 indicates.

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

$$\begin{aligned} au^2 + buv + cv^2 + du + ev + f &= 0 \\ \text{and} & \\ 2(a-c)uv - (2ad_y + e)u + (2cd_x + d)v + (ed_x - dd_y) &= 0 \end{aligned}$$

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

$$2u^2 + v^2 - 1 = 0 \text{ and } 2uv - 4d_y u + 2d_x v = 0$$

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

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

The second equation helps: either $u = 0$, or $v = 2\lambda$. Two of our solutions will be $(0, 1)$, $(0, -1)$. The other two are obtained by solving $2u^2 + 4\lambda^2 - 1 = 0$, which has solutions only if $-1/2 \leq \lambda \leq 1/2$. The situation is illustrated in figure 17.7.

Example 17.1: *The distance between a point and a conic*

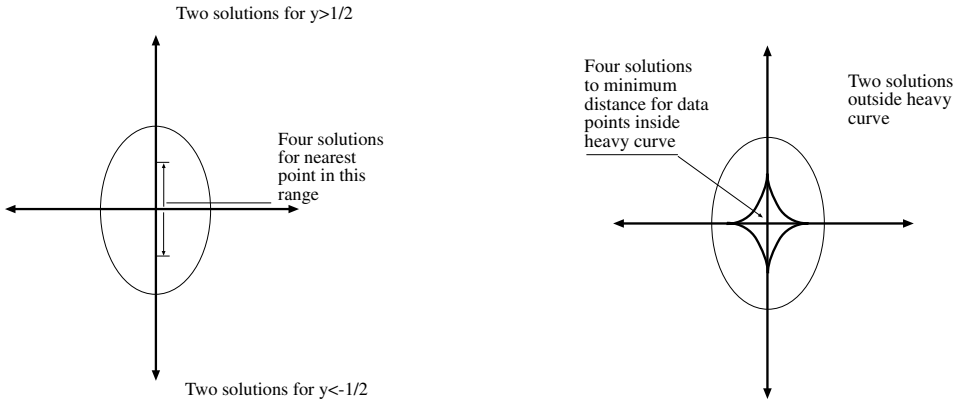

Figure 17.7. *On the left, the example worked in the text, where we study the number of possible solutions for the distance between a point and an ellipse for data points lying on the vertical axis. The figure on the right, indicates the general case for this ellipse.*

Approximations to the Distance

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

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

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

This approximation can be (rather roughly!) justified when the data points are quite close to the curve. For a point sufficiently close to the curve *and to first order*, $\phi(d_x, d_y)$ increases as (d_x, d_y) moves normal to the curve — because the normal to the curve is given by the gradient of ϕ — and does not increase as (d_x, d_y) moves tangent to the curve. One significant difficulty is that, as it stands, algebraic distance is ill-defined, because many polynomials correspond to the same curve. In particular, the curve given by $\mu\phi(x, y) = 0$ is the same as the curve given by $\phi(x, y) = 0$. This problem can be solved normalising the coefficients of the polynomial in some way.

We have already seen one example of this process in section 17.2, where we

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

An alternative approximation is to use

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

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

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

17.3.2 Parametric Curves

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

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

Table 17.2 shows the form of a variety of useful parametric curves.

The Distance from a Point to a Parametric Curve

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

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

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

Table 17.2. A selection of parametric curves often used in vision applications. It is quite common to put together a set of cubic curves, with constraints on their coefficients such that they form a single continuous differentiable curve; the result is known as a **cubic spline**.

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 ratio's of polynomials, because we can rearrange the left hand side of our equation to come up with a polynomial in this case, too. However, we are still faced with a possibly large number of roots.

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

17.4 Fitting to the Outlines of Surfaces

Generally, if we view a surface drawn from a constrained class — say, a surface of revolution — then its outline will satisfy some set of constraints, too. This observation is the key to a powerful idea: cluster edge points to form collections that "look like" the outline of a surface (i.e. satisfy the relevant constraints). The approach appears to work in practice only for quite simple cases, but some of these cases are important.

17.4.1 Some Relations Between Surfaces and Outlines

Recall that the outline of a surface is formed by slicing a cone of rays with the image plane. The cone of rays consists of rays tangent to the surface that pass through the focal point of the camera — for a perspective camera — or are parallel — for an affine camera. Call this cone the **viewing cone**. If the affine camera is orthographic, which is by far the most common case, then the slice is taken perpendicular to the rays. The viewing cone is usually easier to analyze than the outline.

Cones

A **cone** is a surface obtained by sweeping a family of rays through a point — the **vertex** of the cone — along a plane curve, called the **generator**. Notice that this definition is more general than that of a **right circular cone**, which many people incorrectly call a cone; right circular cones have a rotational symmetry (figure 17.8). A cone consists of scaled and translated copies of its generator. Choose a coordinate frame where the generator can be written as $(x(t), y(t), 1)$. Then the cone can be written as

$$(x(t)s, y(t)s, s)$$

and the vertex occurs at $(0, 0, 0)$ (figure 17.8).

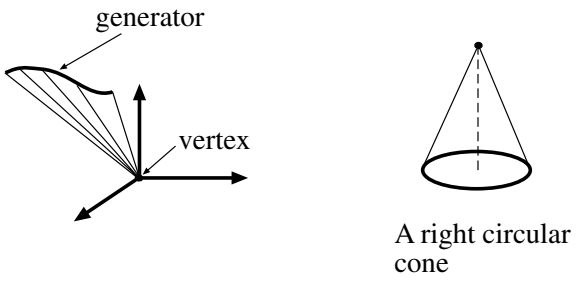

Figure 17.8. Cones are surfaces obtained by sweeping rays through a vertex along a generator. A right circular cone is a special cone, where the generator is a circle and the line joining the vertex with the center of the circle is normal to the circle's plane.

The viewing cone for a cone is a family of planes, all of which pass through the focal point and the vertex of the cone. This means that the outline of a cone consists of a set of lines passing through a vertex (figure 17.9). All this should be obvious (you are asked for a proof in the exercises), but is surprisingly useful.

Straight Homogenous Generalised Cylinders

A **straight homogenous generalised cylinder** (or **SHGC** for short) is a surface obtained by sweeping a plane generating curve along a line at right angles to the

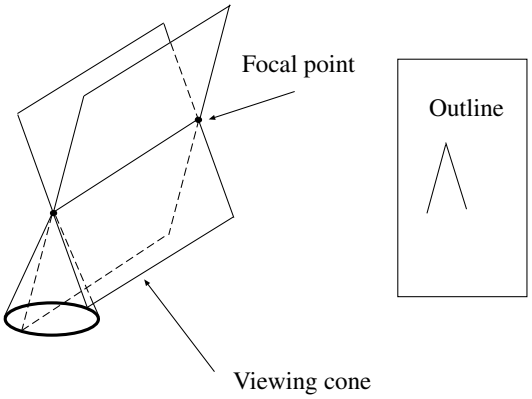

Figure 17.9. The viewing cone is a set of planes tangent to the cone, and passing through both the vertex of the cone and the focal point. The outline of a cone is obtained by slicing the viewing cone with a plane, and is a set of lines through a single point.

curve, and scaling it as one sweeps. With an appropriate choice of coordinate system, the generating curve is $(x(t), y(t), 0)$, and the surface can be written as

$$(x(t)f(s), y(t)f(s), g(s))$$

An SHGC is *locally* a cone. Fix a value of $s = s_0$ and consider the strip of surface from s_0 to $s_0 + \epsilon$. If the strip is small enough, then $f(s)$ can be approximated by $us + v$, and $g(s)$ can be approximated by $cs + d$, for u, v, c and d some constants (this is what derivatives are all about!). The d can be disposed of by translation and the c by reparametrisation, so this strip is a cone. This means that all the tangent vectors in the s direction at $s = s_0$ pass through some vertex.

It is a remarkable fact that the collection of vertices obtained for different values of s is collinear. The easiest way to see this is to work in coordinates. Take the coordinate form given above, and slice it with an arbitrary vertical plane, yielding two plane curves $(\sqrt{x(t_0)^2 + y(t_0)^2}f(s), g(s))$ and $(-\sqrt{x(t_1)^2 + y(t_1)^2}f(s), g(s))$. Take the tangents to these curves at $s = s_0$, and produce them to form tangent rays; a quick calculation shows that the rays meet at $(0, g(s_0) - (f(s_0)/f'(s_0))g'(s_0))$. This means that for any cross-section and any value of s, the tangents meet on the line $x = 0$, meaning that the vertices are collinear. Although we obtained this result in coordinates, this was merely for convenience. Because the result is about incidence properties of tangents — which aren't affected by change of coordinates — it applies to any SHGC in any coordinate frame. So for any SHGC, in any coordinate frame, there is a well defined axis, which is the line along which the vertices of the tangent cones fall.

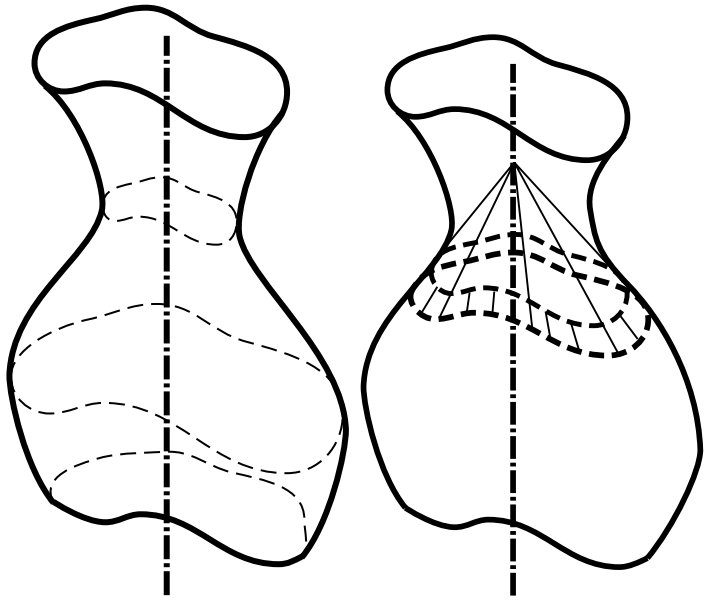

Figure 17.10. An SHGC is obtained by sweeping a plane generator along a line perpendicular to its plane, and growing or shrinking it as it sweeps (left). This gives a surface that is *locally* a cone, in the sense that there is a cone tangent to the surface along any generator. In the right hand figure, we have cut a narrow strip of the surface between two generators, and produced its tangents. These tangents meet at a vertex on the axis, meaning that each such strip of an SHGC is a strip of cone.

Surfaces of Revolution

A **surface of revolution** — or **SOR** for short — is a special SHGC, whose generator is a circle. The viewing cone for an SOR has a symmetry. Imagine the plane through the axis of the SOR and the focal point; the cone must have a flip symmetry about this plane. This *does not* mean that the image curve has this symmetry, because we are slicing the viewing cone with the image plane to get the image curve, and the slicing process can disrupt the symmetry (figure 17.12). The effect is governed by the field of view of the camera, and for the vast majority of practical cameras, it is tiny.

17.4.2 Clustering to Form Symmetries

Clustering the outline of a cone is relatively simple — a collection of image line segments that would pass through a single point, if extended, could be the outline of a cone.

Similarly, we can reason about the outline of an SHGC using our knowledge of cones. In particular, the tangents at components of the outline corresponding to the

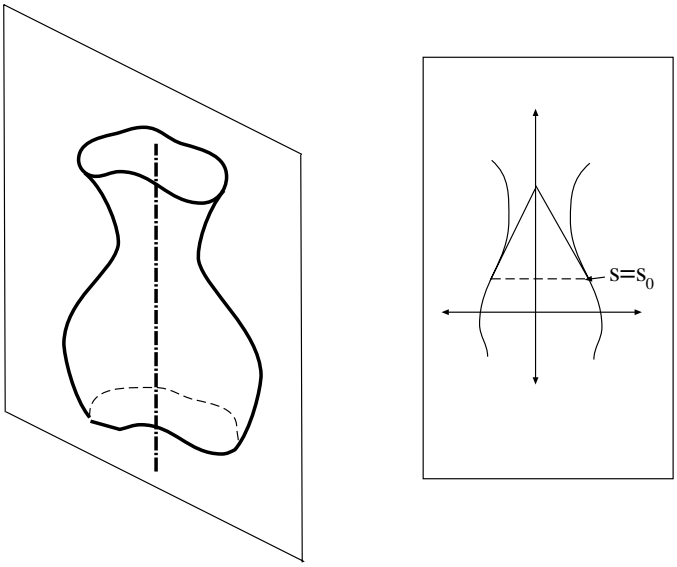

Figure 17.11. The vertices of the tangent cones to an SHGC all fall along a single line; this defines the axis for the SHGC. The easiest way to prove this is to work with the SHGC in the coordinatised form $(x(t)f(s), y(t)f(s), g(s))$, and slice it with an arbitrary plane through the z-axis (left). For any such slice, lines tangent to the cross-section at a particular s-value meet at a point on the z-axis, and the result follows.

same value of s, when produced, will meet at the projected vertex of the tangent cone for that value of s. In turn, there is some correspondence between components of the outline such that tangents at corresponding sides, when produced, will meet along a straight line. This is a segmentation criterion, because not all sets of curves will satisfy it. It is somewhat tricky to use in this form, however (but see [**?**; **?**]).

Surfaces of Revolution

Corresponding points on either side of the outline of an SOR can be identified by a local test on pairs of points on image curves. In particular, two points on image curves where the tangent is at about the same angle to the line joining the points (figure 17.13) could be on opposite sides of a symmetry — we will call this configuration a **local symmetry**, and the line segment joining the points the **symmetry line**.

We could find the outline of a surface of revolution by looking for local symmetries whose midpoints lie on a straight line roughly perpendicular to their symmetry lines. The main difficulty with this strategy is that most images contain an awful lot of symmetries, and there may be many groups of symmetries that satisfy this test.

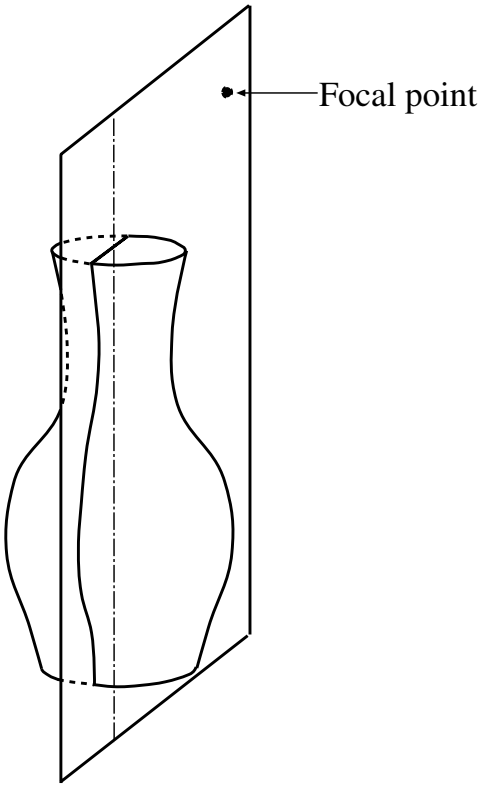

Figure 17.12. A surface of revolution and a focal point together give a plane of symmetry, that passes through the axis of the SOR and the focal point. The contour generator must have a mirror symmetry in this plane. This doesn't mean that the outline has an exact mirror symmetry, because the outline is obtained by slicing the viewing cone — which isn't shown, for simplicity — by a plane *that may not be at right angles to the plane of symmetry.* However, the effect is small, and to all intents and purposes the outline of an SOR can be regarded as having a mirror symmetry.

Cylinders and Body Segments

A particularly interesting SOR is a cylinder. This is because, to a crude first approximation human body segments look like this, as do body segments from some animals. In this case, the local symmetries will have midpoints that (roughly) lie on a straight line, *and* this line will be (roughly) perpendicular to the symmetry lines, *and* the lengths of all the symmetry lines will be (roughly) the same. While typical images contain an awful lot of local symmetries with these properties, it is just about practical to winnow through them looking for groups that satisfy these

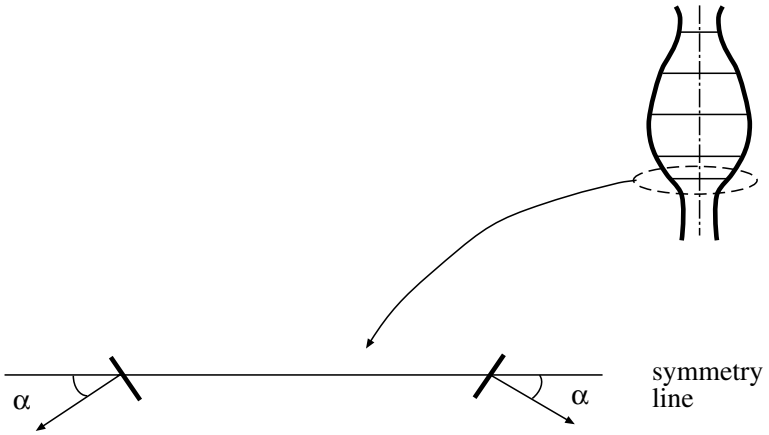

Figure 17.13. A local symmetry is a pair of contour points where the tangents to the contours are at roughly the same angle to the line joining the contours (the symmetry line). Such symmetries are often seen in the outlines of surfaces of revolution.

constraints.

Figure 17.14. An example of the kind of representation that can be obtained from straight segments with near parallel sides. On the left, a colour image of two horses; on the center left, pixels with the right colour and texture to be hide have been retained and the others masked off; on the center right, the edges of this set of pixels; and on the far right, all straight segments with near parallel sides obtained using the mechanisms described above. These segments have been displayed using the abstraction maintained by the program (i.e. the sides of the abstract segment are shown superimposed on the edges that yield the segments). Notice that the components do not exactly correspond to body segments. Although it looks unpromising, this representation can be used to find the horses — which are represented as assemblies of segments — despite the fact it captures image information very poorly.

Cylinders can be found with a relatively crude algorithm (too crude to display!). We use a form of agglomerative clustering. Make each symmetry a cluster. We will build bigger clusters by looking forward and backward along the axis predicted by the symmetries in the cluster. Given a cluster, we can predict the orientation of the next symmetries in the cluster (roughly parallel to the symmetries in the cluster), and the position of their midpoints (along a line roughly perpendicular to

the symmetry lines in the cluster), and their width (roughly the same as the width of the symmetries in the cluster). If the next symmetries are sufficiently nearby and sufficiently similar, we add them to the cluster, and proceed until the cluster cannot be grown further. We do this for each cluster. It is usually a good idea to have a second pass that engages in greedy merges between clusters, using the same criteria.

17.5 Discussion

Fitting should be seen as a form of segmentation exercise — we are concentrating attention on a set of image tokens that have some global structural properties. Section **??** gives the flavour of this line of reasoning, which is currently very poorly developed. We were deliberately vague in describing algorithms in that section; typically, words like "roughly" and "approximately" in our description are interpreted as tests against some threshold, which is set by hand. This means it is rather hard to be precise about what the representation means, or how well it performs. At the same time, symmetries do seem to yield representations that can be useful in practice — for example, it is possible to decide, with quite limited accuracy, whether a picture has unclad people in it or not using a representation based on symmetries **??**. The idea is useful because many interesting objects really do look like cylinders in images. Furthermore, cylinders are associated with a line of reasoning that is very attractive: we first check relatively small sets of image components to tell whether they are part of a cylinder, then assemble the survivors into a larger region. This means that we are using a model of what (some!) objects look like to help assemble together the evidence we need to tell whether an object is present. You should notice that, despite the fact that we have no real probabilistic model here, we seem to be doing something rather like inference: in the following chapter, we will describe the use of inference algorithms in segmentation.

Assignments

Exercises

- 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* $a^2 + b^2 = 1$.

- 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.

- How do we get a curve of edge points from an edge detector that returns orientation? - give a recursive algorithm.
- 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 may have come from a corner
  1. Why would this lead to an improvement?
  2. How should one decide how many pixels to omit?
- A conic section is given by $ax^2 + bxy + cy^2 + dx + ey + f = 0$.
  1. Given a data point (d_x, d_y), show that the nearest point on the conic (u, v) satisfies two equations:

$$\begin{aligned} au^2 + buv + cv^2 + du + ev + f &= 0 \\ &\text{and} \\ 2(a-c)uv - (2ad_y + e)u + (2cd_x + d)v + (ed_x - dd_y) &= 0 \end{aligned}$$

  2. These are two quadratic equations. Write $\boldsymbol{u}$ for the vector $(u, v, 1)$. Now show that we can write these equations as $\boldsymbol{u}^T\mathcal{M}_1\boldsymbol{u} = 0$ and $\boldsymbol{u}^T\mathcal{M}_2\boldsymbol{u} = 0$, for $\mathcal{M}_1$ and $\mathcal{M}_2$ symmetric matrices.
  3. Show that there is a transformation $\mathcal{T}$, such that $\mathcal{T}^T\mathcal{M}_1\mathcal{T} = Id$ *and* $\mathcal{T}^T\mathcal{M}_2\mathcal{T}$ is diagonal.
  4. Now show how to use this transformation to obtain a set of solutions to the equations; in particular, show that there can be up to four real solutions.
  5. Show that there are either four, two or zero real solutions to these equations.
  6. Sketch an ellipse, and indicate the points for which there are four or two solutions.
- Show that the curve

$$(\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2})$$

  is a circular arc (the length of the arc depending on the interval for which the parameter is defined).
  1. Write out the equation in t for the closest point on this arc to some data point (d_x, d_y); what is the degree of this equation? How many solutions in t could there be?
  2. Now substitute $s^3 = t$ in the parametric equation, and write out the equation for the closest point on this arc to the same data point. What is the degree of the equation? why is it so high? What conclusions can you draw?

- Show that the outline of a cone consists of a set of lines passing through the projection of the vertex (you can do this in a short sentence if you think about it; it's a bad idea to try and do it with equations).

Programming Assignments

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