Chapter 16

SEGMENTATION BY CLUSTERING

An attractive broad view of vision is that it is an inference problem: we have some measurements and a model, and we wish to determine what caused the measurement. There are crucial features that distinguish vision from many other inference problems: firstly, there is an awful lot of data, and secondly, we don't know which of these data items may help with solving the inference problem and which may not. For example, one huge difficulty in building good object recognition programs is knowing *which* pixels to recognise and which to ignore. It is very difficult to tell whether a pixel lies on the dalmatian in figure 16.1 simply by looking at the pixel. This problem can be addressed by working with a compact representation of the "interesting" image data that emphasizes the properties that make it "interesting". Obtaining such representation is known variously as **segmentation**, **grouping**, **perceptual organisation** or **fitting**.

We use the term segmentation for a wide range of activities, because, while techniques may differ, the motivation for all these activities is the same: obtain a compact representation of what is helpful in the image. It's hard to see that there could be a comprehensive theory of segmentation, not least because what is interesting and what is not depends on the application. There is certainly no comprehensive theory of segmentation at time of writing, and the term is used in different ways in different quarters. In this chapter we describe segmentation processes that currently have no probabilistic interpretation. In the following chapter, we deal with more complex probabilistic algorithms.

16.1 What is Segmentation?

Assume that we would like to recognise objects in an image. There are too many pixels to handle each individually; instead, we should like some form of compact, summary representation. The details of what that representation should be depend on the task, but there are a number of quite general desirable features. Firstly, there should be relatively few (= not more than later algorithms can cope with) components in the representation computed for "typical" pictures. Secondly, these components should be suggestive. It should be pretty obvious from these components whether the objects we are looking for are present or not, again for "typical" pictures.

Methods deal with different kinds of data set: some are intended for images, some are intended for video sequences and some are intended to be applied to **tokens** — placeholders that indicate the presence of an interesting pattern in an image, say a spot or a dot or an edge point. Tokens can, in fact, occur in video, too; an example might be a spot moving according to some parametric rule.

While superficially these methods may seem quite different, there is a strong similarity amongst them[1]. Each method attempts to obtain a compact representation of its data set using some form of model of similarity (in some cases, one has to look quite hard to spot the model). These general features manifest themselves in very different problems. We review a few examples.

- **Summarizing video:** Users may wish to browse large collections of video sequences. We need to supply a representation that encapsulates "what's in a sequence". One way to do this is to break each sequence into shots — subsequences that "look similar" — and then represent it with a montage of frames, one for each shot. This suggests segmenting the sequences into shots.

- **Finding machined parts:** Assume we wish to find a machined part in an image (a circumstance that arises far less often than one might think). Machined objects tend to contain lines — where plane faces meet — and circles — where holes have been drilled. This suggests segmenting the image into sets of lines and circles; typically, one would find edges first and then fit lines and circles to them.

- **Finding people:** Assume we wish to find people in images. This problem remains open as of writing, but the general outlines of the solution are clear. One should look for body segments first, then assemble them. These segments appear in the image as extended regions; if the people are wearing clothing that isn't textured, then they are extended regions of a single colour — we must look for bars of a constant colour.

- **Finding buildings in satellite images:** The vast majority of buildings are polyhedral, particularly at the scale at which they appear in satellite images. This suggests representing the image in terms of polygonal regions on some background. Typically, this is done by looking for collections of edge points that can be assembled into line segments, and then assembling line segments into polygons.

- **Searching a collection of images:** For users to be able to search a collection of images, the images must be represented in a way that both "makes sense" to

[1]Which is why they appear together!

the user and is related to the content of the picture. Since the content typically involves the objects that are present, and objects tend to have coherent colour and texture, it is natural to try and break the images into regions of coherent colour and texture, and use these regions as a representation.

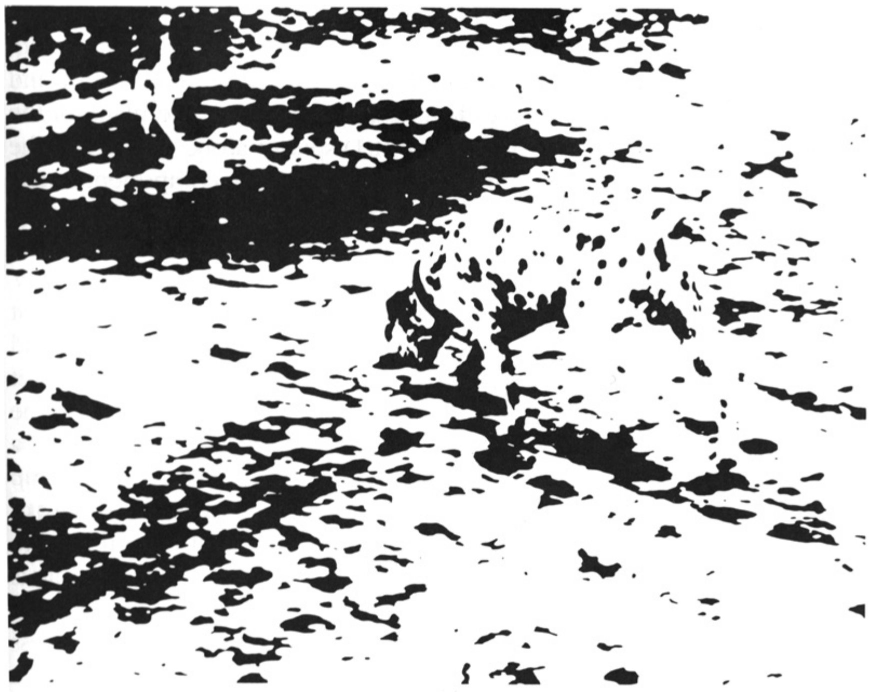


Figure 16.1. As the image of a dalmatian on a shadowed background indicates, an important component of vision involves organising image information into meaningful assemblies. The human vision system seems to be able to do so surprisingly well. The blobs that form the dalmatian appear to be assembled "because they form a dalmatian," hardly a satisfactory explanation, and one that begs difficult computational questions. This process of organisation can be applied to many different kinds of input.

16.1.1 Four Model Problems

Segmentation is a big topic. To help the reader keep track of the diverse methods and problems involved, this account is structured around four model problems. These problems are "natural", in the sense that it is valuable to know more than one method for solving them, and they appear commonly in applications. Our problems are:

- **Forming image segments:** we should like to decompose an image into "super pixels", image regions which have roughly coherent colour and texture.

Typically, the shape of these regions isn't particularly important, but the coherence is very important. This process is quite widely studied — it is often referred to as the exclusive meaning of the term "segmentation" — and usually thought of as a first step in recognition. As we indicated above, one use of these regions is in organising images in a digital library.

- **Finding body segments:** assume we wish to find people in images. One way to do this is to find putative body segments (torso, upper arm, lower arm, etc.) in the image, and then reason about the configuration of these segments. We expect that the body segments will have coherent colour and texture (it helps to be a bit vague about what the word "coherent" means) and also will be extended. The segments should look like the images of rather rough cylinders. This problem is different from the previous problem, because the shape of the regions is now important.

- **Fitting lines to edge points:** as we saw above, there are a number of reasons why it might be useful to fit a set of lines to a set of points. This problem goes from being quite simple (in the case where we know how many lines there are, and which point belongs to which line) to being remarkably subtle (in most other cases). If we try and fit a line to a set of points some of which do not lie close to any line, the resulting line can be meaningless unless we are very careful indeed. This illustrates an important, quite general, principle: ignorance of correspondence can behave like noise. Typically, we need to estimate both the parameters of the lines and the correspondence between points and lines simultaneously.

- **Fitting a fundamental matrix to a set of feature points:** assume we have two views of a set of feature points. It is typically difficult to be sure which points correspond to which, though we may have some cues. One important cue is that, if the correspondence is right, there is a fundamental matrix connecting the points. We should like to determine this fundamental matrix, without knowing the correspondence in advance. There are several reasons this problem is worth solving: firstly, it is impossible to construct sensible shape representations from multiple views without a solution to this problem; secondly, solutions to this problem can be used as a cue to whether a set of points is moving rigidly or not — if a sequence shows two moving objects, they'll have different fundamental matrices. Again, correspondence errors will look like noise in this problem.

We will use these problems to illustrate various segmentation algorithms, but you should keep in mind that not every technique offers a plausible solution to each of these model problems.

16.1.2 Segmentation as Clustering

One natural view of segmentation is that we are attempting to determine which components of a data set naturally “belong together”. This is a problem known as **clustering**; there is a wide literature. Generally, we can cluster in two ways:

- **Partitioning:** here we have a large data set, and carve it up according to some notion of the association between items inside the set. We would like to decompose it into pieces that are “good” according to our model. For example, we might:
  - decompose an image into regions which have coherent colour and texture inside them;
  - decompose an image into extended blobs, consisting of regions that have coherent colour, texture and motion and look like limb segments;
  - take a video sequence and decompose it into **shots** — segments of video showing about the same stuff from about the same viewpoint.
- **Grouping:** here we have a set of distinct data items, and wish to collect sets of data items that “make sense” together according to our model. Effects like occlusion mean that image components that belong to the same object are often separated. Examples of grouping include:
  - collecting together tokens that, taken together, form a line;
  - collecting together tokens that seem to share a fundamental matrix.

Of course, the key issue here is to determine what representation is suitable for the problem in hand. Occasionally this is pretty obvious; more often, the question is subtle. Typically, one has to know what various methods do, and make an informed choice. Even so, we need to know by what criteria a segmentation method should decide which pixels (or tokens) belong together and which do not. A fruitful source of insight is the human vision system, which has to solve a completely general form of this problem and, remarkably, displays strong and easily evoked preferences for how tokens are grouped.

16.2 Human vision: Grouping and Gestalt

A key feature of the human vision system is that “context” affects how things are perceived (for example, see the illusion of figure 16.2). This observation — which is easily demonstrated experimentally — led the Gestalt school of psychologists to reject the study of responses to stimuli, and to emphasize grouping as the key to understanding visual perception. To them, grouping meant the tendency of the visual system to assemble some components of a picture together and to perceive them together (this supplies a rather rough meaning to the word “context” used above). Grouping, for example, is what causes the Müller-Lyer illusion of figure 16.2

— the vision system assembles the components of the two arrows, and the horizontal lines look different from one another because they are peceived as components of a whole, rather than as lines. Furthermore, many grouping effects can't be disrupted by cognitive input; for example, you can't make the lines in figure 16.2 look equal in length by deciding not to group the arrows.

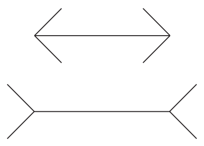


Figure 16.2. The famous Muller-Lyer illusion; the horizontal lines are in fact the same length, though that belonging to the upper figure looks longer. Clearly, this effect arises from some property of the relationships that form the whole (the *gestaltqualität*), rather than from properties of each separate segment.

A common experience of segmentation is the way that an image can resolve itself into a **figure** — typically, the significant, important object — and a **ground** — the background on which the figure lies. However, as figure 16.3 illustrates, what is figure and what is ground can be profoundly ambiguous, meaning that a richer theory is required.

The Gestalt school used the notion of a **gestalt** — a whole or a group — and of its **gestaltqualität** — the set of internal relationships that makes it a whole (e.g. figure 16.2) as central components in their ideas. Their work was characterised by attempts to write down a series of rules by which image elements would be associated together and interpreted as a group. There were also attempts to construct algorithms, which are of purely historical interest (see [Gordon, 1997] for an introductory account that places their work in a broad context).

The Gestalt psychologists identified a series of factors, which they felt predisposed a set of elements to be grouped. These factors are important, because it is quite clear that the human vision system uses them in some way. Furthermore, it is reasonable to expect that they represent a set of preferences about when tokens belong together that lead to a useful intermediate representation.

There are a variety of factors, some of which postdate the main Gestalt movement:

- **Proximity:** tokens that are nearby tend to be grouped.
- **Similarity:** similar tokens tend to be grouped together.
- **Common fate:** tokens that have coherent motion tend to be grouped together.
- **Common region:** tokens that lie inside the same closed region tend to be grouped together.

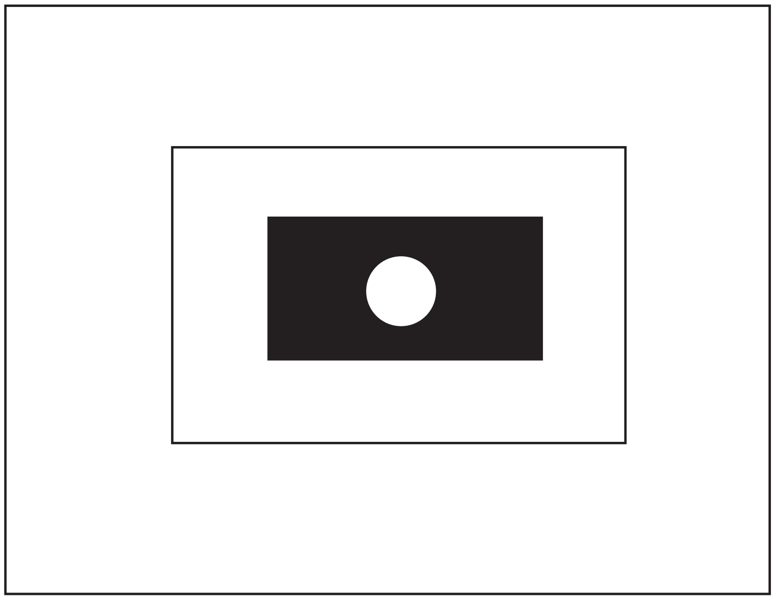


Figure 16.3. One view of segmentation is that it determines which component of the image forms the figure, and which the ground. The figure illustrates one form of ambiguity that results from this view; the white circle can be seen as figure on the black rectangular ground, or as ground where the figure is a black rectangle with a circular hole in it — the ground is then a white square.

- **Parallelism:** parallel curves or tokens tend to be grouped together.
- **Closure:** tokens or curves that tend to lead to closed curves tend to be grouped together.
- **Symmetry:** curves that lead to symmetric groups are grouped together.
- **Continuity:** tokens that lead to "continuous" — as in "joining up nicely", rather than in the formal sense — curves tend to be grouped.
- **Familiar Configuration:** tokens that, when grouped, lead to a familiar object, tend to be grouped together.

These laws are illustrated in figures 16.4, 16.5 and 16.1.

These rules can function fairly well as explanations, but they are insufficiently crisp to be regarded as forming an algorithm. The Gestalt psychologists had serious difficulty with the details, such as when one rule applied and when another. It is very difficult to supply a satisfactory algorithm for using these rules — the Gestalt movement attempted to use an extremality principle.

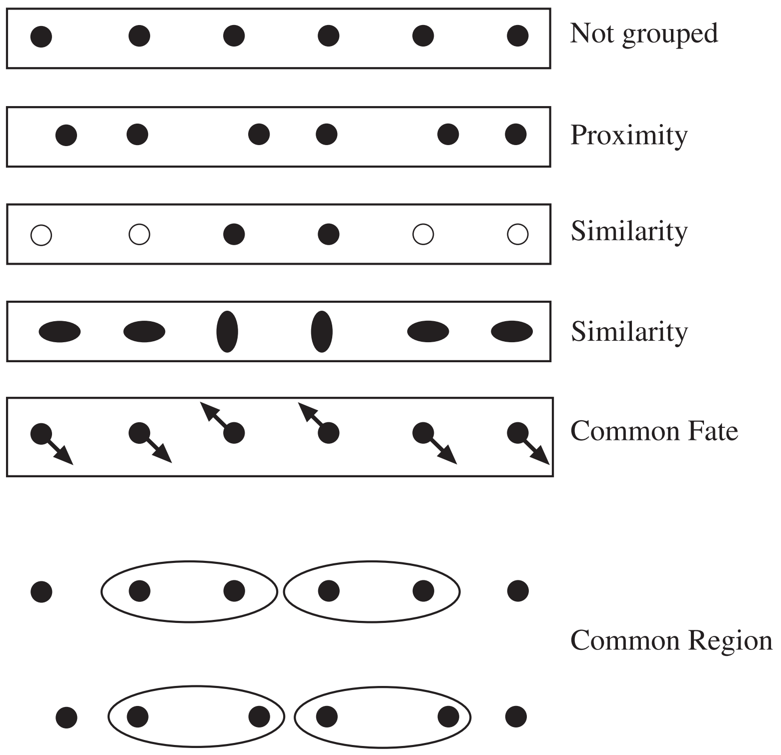


Figure 16.4. Examples of Gestalt factors that lead to grouping (which are described in greater detail in the text).

Familiar configuration is a particular problem. The key issue is to understand just *what* familiar configuration applies in a problem, and how it is selected. For example, look at figure 16.1; one might argue that the blobs are grouped because they yield a dog. The difficulty with this view is explaining how this occurred — where did the hypothesis that a dog is present come from? a search through all views of all objects is one explanation, but one must then explain how this search is organised — do we check *every view* of *every* dog with *every* pattern of spots? how can this be done efficiently?

The Gestalt rules do offer some insight, because they offer some explanation for what happens in various examples. These explanations seem to be sensible, because they suggest that the rules help solve problems posed by visual effects that arise commonly in the real world — that is, they are **ecologically valid**. For example, continuity may represent a solution to problems posed by occlusion — sections of the contour of an occluded object could be joined up by continuity (see figure 16.6).

This tendency to prefer interpretations that are explained by occlusion leads to interesting effects. One is the **illusory contour**, illustrated in figure 16.8. Here

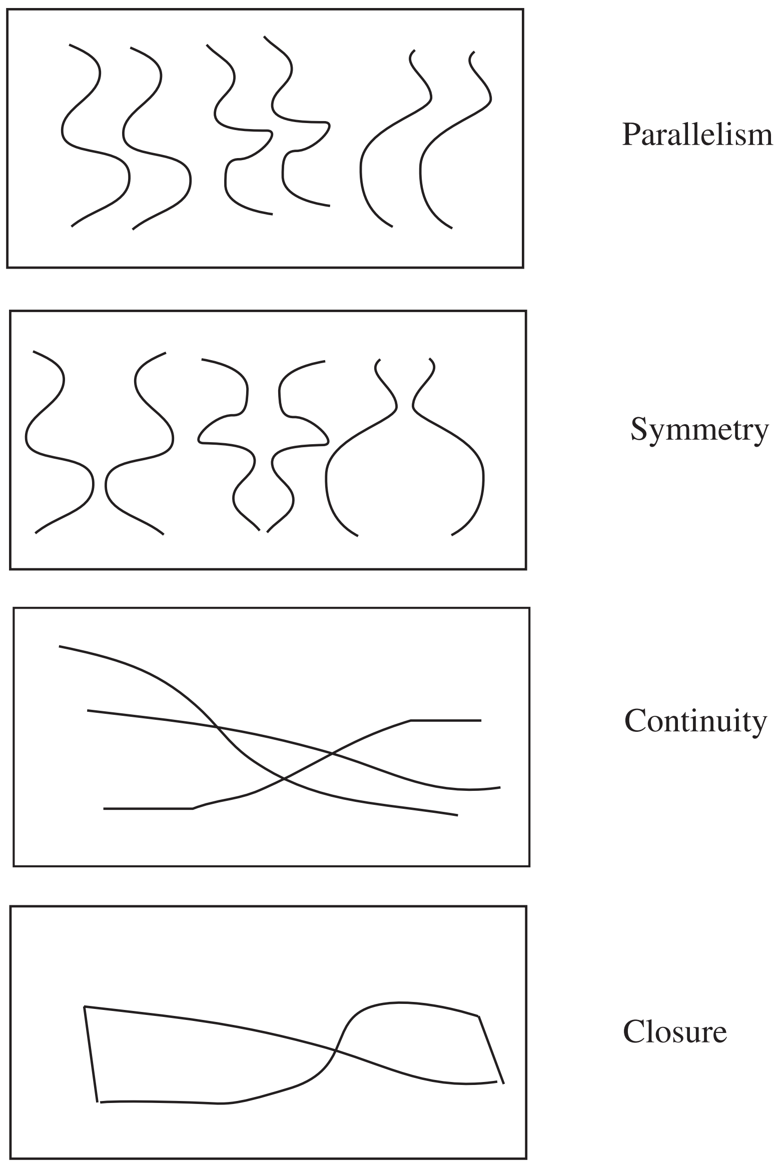


Figure 16.5. Examples of Gestalt factors that lead to grouping (which are described in greater detail in the text).

a set of tokens suggests the presence of an object most of whose contour has no contrast. The tokens appear to be grouped together because they provide a cue to the presence of an occluding object, which is so strongly suggested by these tokens

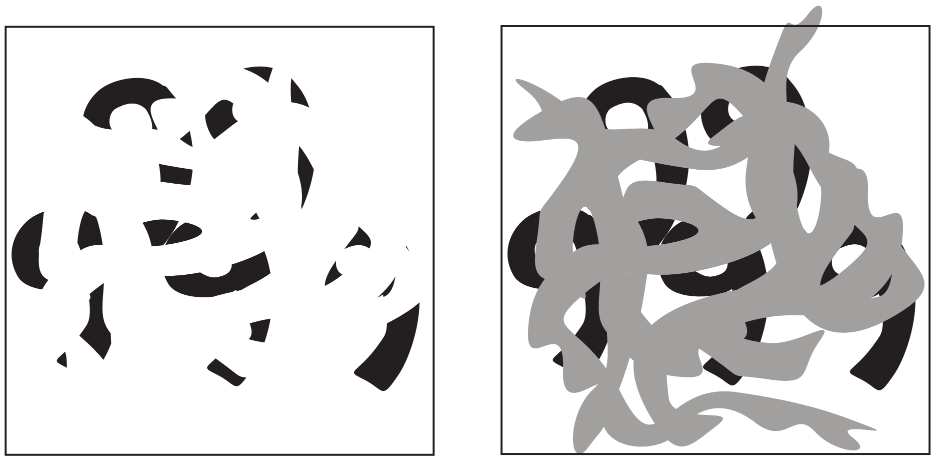


Figure 16.6. Occlusion appears to be an important cue in grouping. It may be possible to see the pattern on the left as a collection of digits; that on the right is quite clearly some occluded digits. The black regions on the left and on the right are the same. The visual system appears to be helped by evidence that separated tokens are separated for a reason, rather than just scattered.

that one could fill in the no-contrast regions of contour.

This ecological argument has some force, because it is possible to interpret most grouping factors using it. Common fate can be seen as a consequence of the fact that components of objects tend to move together. Equally, symmetry is a useful grouping cue because there are a lot of real objects that have symmetric or close to symmetric contours. Essentially, the ecological argument says that tokens are grouped because doing so produces representations that are helpful for the visual world that people encounter. The ecological argument has an appealing, though vague, statistical flavour. From our perspective, Gestalt factors provide interesting hints, but should be seen as the *consequences* of a larger grouping process, rather than the process itself.

16.3 Applications: Shot Boundary Detection and Background Subtraction

Simple segmentation algorithms are often very useful in significant applications. Generally, simple algorithms work best when it is very easy to tell what a “useful” decomposition is. Three important cases are **background subtraction** — where anything that doesn’t look like a known background is interesting — **shot boundary detection** — where substantial changes in a video are interesting — and **skin finding** — where pixels that look like human skin are interesting.

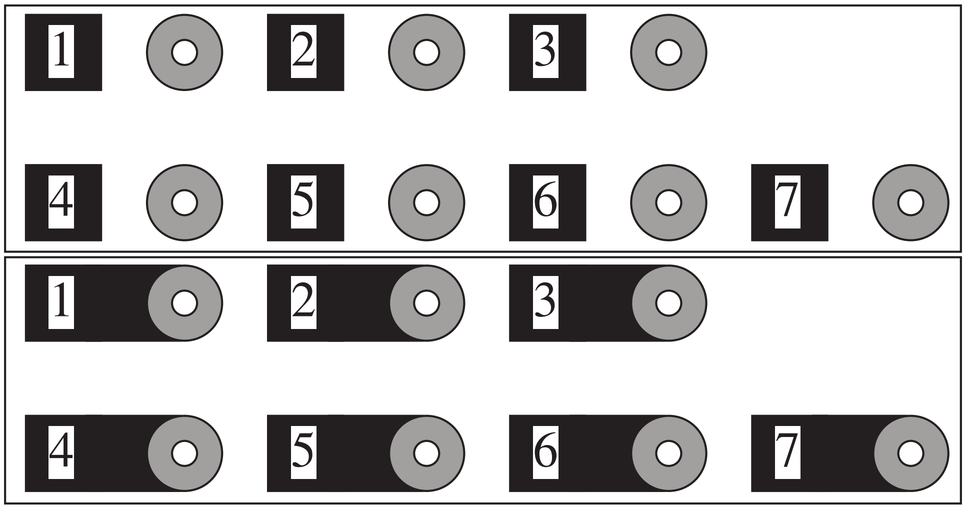


Figure 16.7. An example of grouping phenomena in real life. The buttons on an elevator in the computer science building at U.C. Berkeley used to be laid out as in the **top** figure. It was common to arrive at the wrong floor and discover that this was because you'd pressed the wrong button — the buttons are difficult to group unambiguously with the correct label, and it is easy to get the wrong grouping at a quick glance. A public-spirited individual filled in the gap between the numbers and the buttons, as in the **bottom** figure, and the confusion stopped because the proximity cue had been disambiguated.

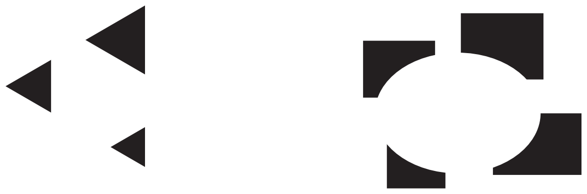


Figure 16.8. The tokens in these images suggest the presence of occluding objects, whose boundaries don't contrast with much of the image. Notice that one has a clear impression of the position of the entire contour of the occluding figures. These contours are known as *illusory contours*.

16.3.1 Background Subtraction

In many applications, objects appear on a background which is very largely stable. The standard example is detecting parts on a conveyor belt. Another example is counting motor cars in an overhead view of a road — the road itself is pretty stable in appearance. Another, less obvious, example is in human computer interaction. Quite commonly, a camera is fixed (say, on top of a monitor) and views a room. Pretty much anything in the view that doesn't look like the room is interesting.

In these kinds of applications, a useful segmentation can often be obtained by subtracting an estimate of the appearance of the background from the image, and looking for large absolute values in the result. The main issue is obtaining a good estimate of the background. One method is simply to take a picture. This approach works rather poorly, because the background typically changes slowly over time. For example, the road may get more shiny as it rains and less when the weather dries up; people may move books and furniture around in the room, etc.

```
Form a background estimate $\mathcal{B}^{(0)}$.
At each frame $\mathcal{F}$

    Update the background estimate, typically by
    forming $\mathcal{B}^{(n+1)} = \frac{w_a\mathcal{F}+\sum_i w_i\mathcal{B}^{(n-i)}}{w_c}$
    for a choice of weights $w_a$, $w_i$ and $w_c$.

    Subtract the background estimate from the
    frame, and report the value of each pixel where
    the magnitude of the difference is greater than some
    threshold.

end
```

Algorithm 16.1: *Background Subtraction*

An alternative which usually works quite well is to estimate the value of background pixels using a **moving average**. In this approach, we estimate the value of a particular background pixel as a weighted average of the previous values. Typically, pixels in the very distant past should be weighted at zero, and the weights increase smoothly. Ideally, the moving average should track the changes in the background, meaning that if the weather changes very quickly (or the book mover is frenetic) relatively few pixels should have non-zero weights, and if changes are slow, the number of past pixels with non-zero weights should increase. This yields algorithm 1 For those who have read the filters chapter, this is a filter that smooths a function of time, and we would like it to suppress frequencies that are larger than the typical frequency of change in the background and pass those that are at or below that frequency. The approach can be quite successful, but needs to be used on quite coarse scale images, as figures 16.10 and 16.11 illustrate.

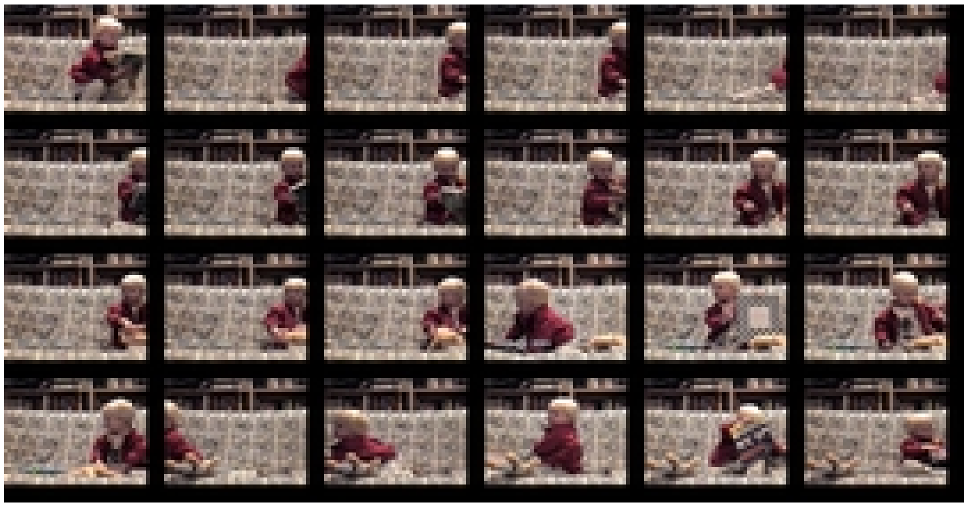


Figure 16.9. The figure shows every fifth frame from a sequence of 120 frames of a child playing on a patterned sofa. The frames are used at an 80x60 resolution, for reasons we discuss in figure 16.11. Notice that the child moves from one side of the frame to the other during the sequence.

16.3.2 Shot Boundary Detection

Long sequences of video are composed of **shots** — much shorter subsequences that show largely the same objects. These shots are typically the product of the editing process. There is seldom any record of where the boundaries between shots fall. It is helpful to represent a video as a collection of shots; each shot can then be represented with a **key frame**. This representation can be used to search for videos or to encapsulate their content for a user to browse a video or a set of videos.

Finding the boundaries of these shots automatically — **shot boundary detection** — is an important practical application of simple segmentation algorithms. A shot boundary detection algorithm must find frames in the video that are "significantly" different from the previous frame. Our test of significance must take account of the fact that within a given shot both objects and the background can move around in the field of view. Typically, this test takes the form of a distance; if the distance is larger than a threshold, a shot boundary is declared (algorithm 2).

There are a variety of standard techniques for computing a distance:

- **Frame differencing** algorithms take pixel-by-pixel differences between each two frames in a sequence, and sum the squares of the differences. These algorithms are unpopular, because they are slow — there are many differences — and because they tend to find many shots when the camera is shaking.

- **Histogram based** algorithms compute colour histograms for each frame, and

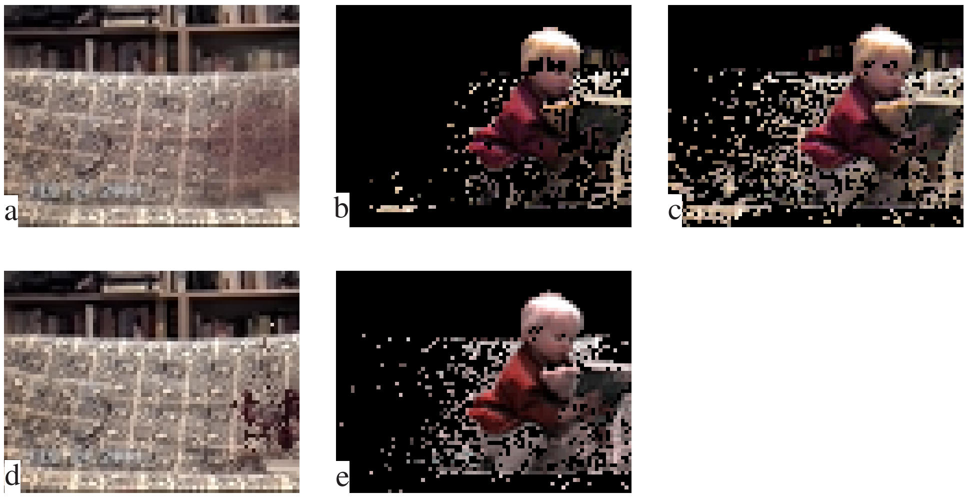


Figure 16.10. Background subtraction results for the sequence of figure 16.9, using 80x60 frames. We compare two methods of computing the background; in **a**, we show the average of all 120 frames — notice that the child spent more time on one side of the sofa than the other, leading the a faint blur in the average there. Pixels whose difference from the average exceeds a small threshold are given in **b**, and **c** shows those whose difference from the average exceeds a somewhat larger threshold. Notice that, in each case, there are some excess pixels and some missing pixels. In **d**, we show a background computed using a somewhat more sophisticated method (described briefly in section 18.2.5), and in **e** we show pixels that this method believes are different from the background. Again, notice the missing pixels.

compute a distance between the histograms. A difference in colour histograms is a sensible measure to use, because it is insensitive to the spatial arrangement of colours in the frame — for example, small camera jitters will not affect the histogram.

- **Block comparison** algorithms compare frames by cutting them into a grid of boxes, and comparing the boxes. This is to avoid the difficulty with colour histograms, where (for example) a red object disappearing off-screen in the bottom left corner is equivalent to a red object appearing on screen from the top edge. Typically, these block comparison algorithms compute an inter-frame distance that is a composite — taking the maximum is one natural strategy — of inter-block distances, computed using the methods above.

- **Edge differencing** algorithms compute edge maps for each frame, and then compare these edge maps. Typically, the comparison is obtained by counting the number of potentially corresponding edges (nearby, similar orientation, etc.) in the next frame. If there are few potentially corresponding edges,

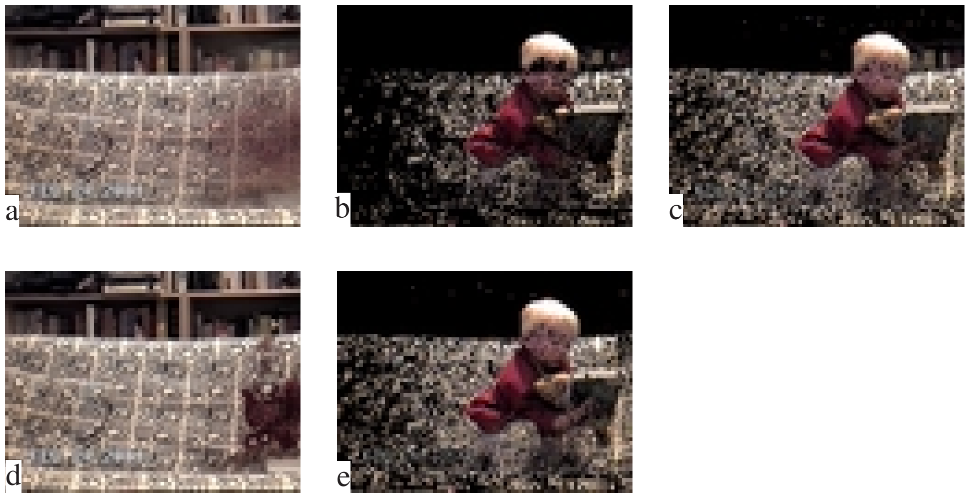


Figure 16.11. Registration can be a significant nuisance in background subtraction, particularly for textures. These figures show results for the sequence of figure 16.9, using 160x120 frames. We compare two methods of computing the background; in **a**, we show the average of all 120 frames — notice that the child spent more time on one side of the sofa than the other, leading the a faint blur in the average there. Pixels whose difference from the average exceeds a small threshold are given in **b**, and **c** shows those whose difference from the average exceeds a somewhat larger threshold. In **d**, we show a background computed using a somewhat more sophisticated method (described briefly in section 18.2.5), and in **e** we show pixels that this method believes are different from the background. 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 very small movements can cause the high spatial frequency pattern on the sofa to be misaligned, leading to large differences.

there is a shot boundary. A distance can be obtained by transforming the number of corresponding edges.

These are relatively *ad hoc* methods, but are often sufficient to solve the problem at hand.

16.4 Image Segmentation by Clustering Pixels

Clustering is a process whereby a data set is replaced by **clusters**, which are collections of data points that "belong together". It is natural to think of image segmentation as clustering; we would like to represent an image in terms of clusters of pixels that "belong together". The specific criterion to be used depends on the application. Pixels may belong together because they have the same colour and/or they have the same texture and/or they are nearby, etc.

```
For each frame in an image sequence

    Compute a distance between this frame and the
    previous frame

    If the distance is larger than some threshold,

      classify the frame as a shot boundary.

end
```

Algorithm 16.2: *Shot boundary detection using interframe differences*

16.4.1 Segmentation Using Simple Clustering Methods

It is relatively easy to take a clustering method and build an image segmenter from it. Much of the literature on image segmentation consists of papers that are, in essence, papers about clustering (though this isn't always acknowledged).

Simple Clustering Methods

There are two natural algorithms for clustering. In **divisive clustering**, the entire data set is regarded as a cluster, and then clusters are recursively split to yield a good clustering (algorithm 4). In **agglomerative clustering**, each data item is regarded as a cluster and clusters are recursively merged to yield a good clustering (algorithm 3).

```
Make each point a separate cluster

Until the clustering is satisfactory

    Merge the two clusters with the
    smallest inter-cluster distance

end
```

Algorithm 16.3: *Agglomerative clustering, or clustering by merging*

There are two major issues in thinking about clustering:

- *what is a good inter-cluster distance?* Agglomerative clustering uses an inter-cluster distance to fuse "nearby" clusters; divisive clustering uses it to split

```
Construct a single cluster containing all points

Until the clustering is satisfactory

    Split the cluster that yields the two
    components with the largest inter-cluster distance

end
```

Algorithm 16.4: *Divisive clustering, or clustering by splitting*

insufficiently "coherent" clusters. Even if a natural distance between data points is available (which may not be the case for vision problems), there is no canonical inter-cluster distance. Generally, one chooses a distance that seems appropriate for the data set. For example, one might choose the distance between the closest elements as the inter-cluster distance — this tends to yield extended clusters (statisticians call this method **single-link clustering**). Another natural choice is the maximum distance between an element of the first cluster and one of the second — this tends to yield "rounded" clusters (statisticians call this method **complete-link clustering**). Finally, one could use an average of distances between elements in the clusters — this will also tend to yield "rounded" clusters (statisticians call this method **group average clustering**).

- *and how many clusters are there?* This is an intrinsically difficult task if there is no model for the process that generated the clusters. The algorithms we have described generate a hierarchy of clusters. Usually, this hierarchy is displayed to a user in the form of a **dendrogram** — a representation of the structure of the hierarchy of clusters that displays inter-cluster distances — and an appropriate choice of clusters is made from the dendrogram (see the example in figure 16.12).

Building Segmenters Using Clustering Methods

The distance used depends entirely on the application, but measures of colour difference and of texture are commonly used as clustering distances. It is often desirable to have clusters that are "blobby"; this can be achieved by using difference in position in the clustering distance.

The main difficulty in using either agglomerative or divisive clustering methods directly is that there are an awful lot of pixels in an image. There is no reasonable prospect of examining a dendrogram, because the quantity of data means that it will be too big. In practice, this means that the segmenters decide when to stop

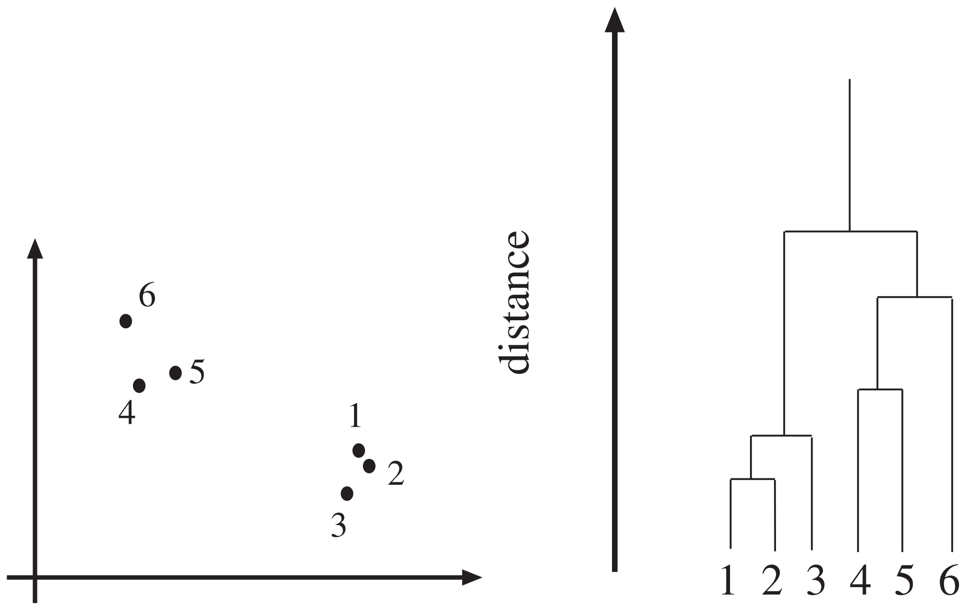


Figure 16.12. Left, a data set; right, a dendrogram obtained by agglomerative clustering using single link clustering. If one selects a particular value of distance, then a horizontal line at that distance will split the dendrogram into clusters. This representation makes it possible to guess how many clusters there are, and to get some insight into how good the clusters are.

splitting or merging by using a set of threshold tests — for example, an agglomerative segmenter may stop merging when the distance between clusters is sufficiently low, or when the number of clusters reaches some value.

Another difficulty created by the number of pixels is that it is impractical to look for the best split of a cluster (for a divisive method) or the best merge (for an agglomerative method). **Divisive methods** are usually modified by using some form of summary of a cluster to suggest a good split. A natural summary to use is a histogram of pixel colours (or grey levels).

Agglomerative methods also need to be modified. Firstly, the number of pixels means that one needs to be careful about the intercluster distance (the distance between cluster centers of gravity is often used). Secondly, it is usual to try and merge only clusters with shared boundaries (we probably don't wish to represent the US flag as three clusters, one red, one white and one blue). Finally, it can be useful to merge regions simply by scanning the image and merging all pairs whose distance falls below a threshold, rather than searching for the closest pair.

16.4.2 Clustering and Segmentation by K-means

Simple clustering methods use greedy interactions with existing clusters to come up with a good overall representation. For example, in agglomerative clustering we repeatedly make the best available merge. However, the methods are not explicit about the objective function that the methods are attempting to optimize. An alternative approach is to write down an objective function that expresses how good a representation is, and then build an algorithm for obtaining the best representation.

A natural objective function can be obtained by assuming that we know there are k clusters, where k is known. Each cluster is assumed to have a center; we write the center of the i'th cluster as $\boldsymbol{c}_i$. The j'th element to be clustered is described by a feature vector $\boldsymbol{x}_j$. For example, if we were segmenting scattered points, then $\boldsymbol{x}$ would be the coordinates of the points; if we were segmenting an intensity image, $\boldsymbol{x}$ might be the intensity at a pixel.

We now assume that elements are close to the center of their cluster, yielding the objective function

$$\Phi(\text{clusters}, \text{data}) = \sum_{i \in \text{clusters}} \left\{ \sum_{j \in i\text{'th cluster}} (\boldsymbol{x}_j - \boldsymbol{c}_i)^T (\boldsymbol{x}_j - \boldsymbol{c}_i) \right\}$$

Notice that if the allocation of points to clusters is known, it is easy to compute the best center for each cluster. However, there are far too many possible allocations of points to clusters to search this space for a minimum. Instead, we define an algorithm which iterates through two activities:

- Assume the cluster centers are known, and allocate each point to the closest cluster center.
- Assume the allocation is known, and choose a new set of cluster centers. Each center is the mean of the points allocated to that cluster.

We then choose a start point by randomly choosing cluster centers, and then iterate these stages alternately. This process will eventually converge to a local minimum of the objective function (the value either goes down, or is fixed, at each step; it is bounded below; and we discount the prospect of symmetries in the objective function). It is not guaranteed to converge to the global minimum of the objective function, however. It is also not guaranteed to produce k clusters, unless we modify the allocation phase to ensure that each cluster has some non-zero number of points. This algorithm is usually referred to as **k-means**. It is possible to search for an appropriate number of clusters by applying k-means for different values of k, and comparing the results; we defer a discussion of this issue until section 18.3.

One difficulty with using this approach for segmenting images is that segments are not connected and can be scattered very widely (figures 16.13 and 16.14). This effect can be reduced by using pixel coordinates as features, an approach that tends to result in large regions being broken up (figure 16.15).

```
Choose $k$ data points to act as cluster centers

Until the cluster centers are unchanged

    Allocate each data point to cluster whose center is nearest

    Now ensure that every cluster has at least
    one data point; possible techniques for doing this include .
    supplying empty clusters with a point chosen at random from
    points far from their cluster center.

    Replace the cluster centers with the mean of the elements
    in their clusters.

end
```

Algorithm 16.5: *Clustering by K-Means*

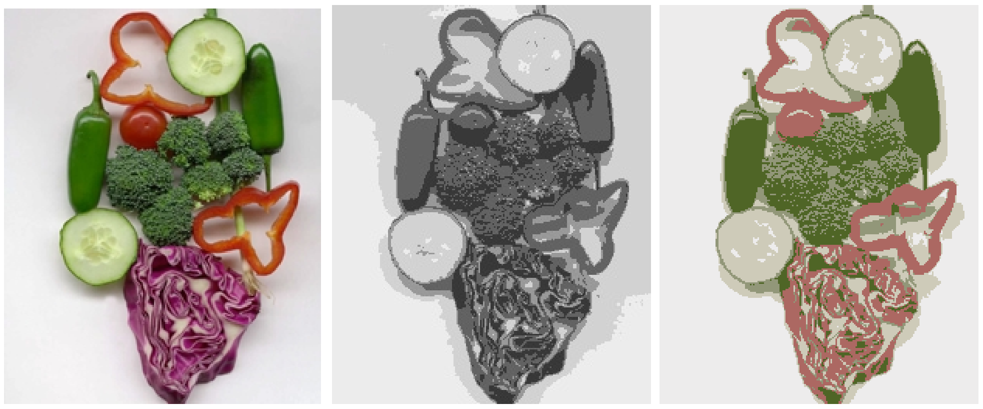


Figure 16.13. On the left, an image of mixed vegetables, which is segmented using k-means to produce the images at center and on the right. We have replaced each pixel with the mean value of its cluster; the result is somewhat like an adaptive requantization, as one would expect. In the center, a segmentation obtained using only the intensity information. At the right, a segmentation obtained using colour information. Each segmentation assumes five clusters.

16.5 Segmentation by Graph-Theoretic Clustering

Clustering can be seen as a problem of cutting graphs into "good" pieces. In effect, we associate each data item with a vertex in a weighted graph, where the weights

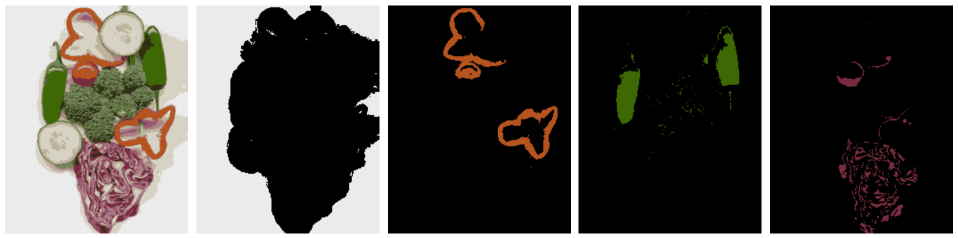


Figure 16.14. Here we show the image of vegetables segmented with k-means, assuming a set of 11 components. The top left figure shows all segments shown together, with the mean value in place of the original image values. The other figures show four of the segments. Note that this approach leads to a set of segments that are not necessarily connected. For this image, some segments are actually quite closely associated with objects but one segment may represent many objects (the peppers); others are largely meaningless. The absence of a texture measure creates serious difficulties, as the many different segments resulting from the slice of red cabbage indicate.

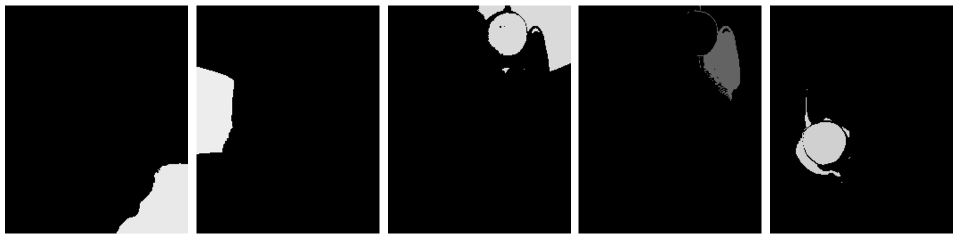


Figure 16.15. Five of the segments obtained by segmenting the image of vegetables with a k-means segmenter that uses position as part of the feature vector describing a pixel, now using 20 segments rather than 11. Note that the large background regions that should be coherent has been broken up because points got too far from the center. The individual peppers are now better separated, but the red cabbage is still broken up because there is no texture measure.

on the edges between elements are large if the elements are "similar" and small if they are not. We then attempt to cut the graph into connected components with relatively large interior weights — which correspond to clusters — by cutting edges with relatively low weights. This view leads to a series of different, quite successful, segmentation algorithms.

16.5.1 Terminology for Graphs

We review terminology here very briefly, as it's quite easy to forget.

- A **graph** is a set of vertices V and edges E which connect various pairs of vertices. A graph can be written $G = \{V, E\}$. Each edge can be represented by a pair of vertices, that is $E \subset V \times V$. Graphs are often drawn as a set of points with curves connecting the points.

- A **directed graph** is one in which edges (a, b) and (b, a) are distinct; such a graph is drawn with arrowheads indicating which direction is intended.
- An **undirected graph** is one in which no distinction is drawn between edges (a, b) and (b, a).
- A **weighted graph** is one in which a weight is associated with each edge.
- A **self-loop** is an edge that has the same vertex at each end; self-loops don't occur in practice in our applications.
- Two vertices are said to be **connected** if there is a sequence of edges starting at the one and ending at the other; if the graph is directed, then the arrows in this sequence must point the right way.
- A **connected graph** is one where every pair of vertices is connected.
- Every graph consists of a disjoint set of **connected components**, that is $G = \{V_1 \cup V_2 \dots V_n, E_1 \cup E_2 \dots E_n\}$, where $\{V_i, E_i\}$ are all connected graphs and there is no edge in E that connects an element of V_i with one of V_j for $i \neq j$.

16.5.2 The Overall Approach

It is useful to understand that a weighted graph can be represented by a square matrix (figure 16.16). There is a row and a column for each vertex. The i, j'th element of the matrix represents the weight on the edge from vertex i to vertex j; for an undirected graph, we use a symmetric matrix and place half the weight in each of the i, j'th and j, i'th element.

The application of graphs to clustering is this: take each element of the collection to be clustered, and associate it with a vertex on a graph. Now construct an edge from every element to every other, and associate with this edge a weight representing the extent to which the elements are similar. Now cut edges in the graph to form a "good" set of connected components — ideally, the within-component edges will be large compared to the across-component edges. Each component will be a cluster. For example, figure 16.17 shows a set of well separated points and the weight matrix (i.e. undirected weighted graph, just drawn differently) that results from a particular similarity measure; a desirable algorithm would notice that this matrix looks a lot like a block diagonal matrix — because intercluster similarities are strong and intracluster similarities are weak — and split it into two matrices, each of which is a block. The issues to study are the criteria that lead to good connected components and the algorithms for forming these connected components.

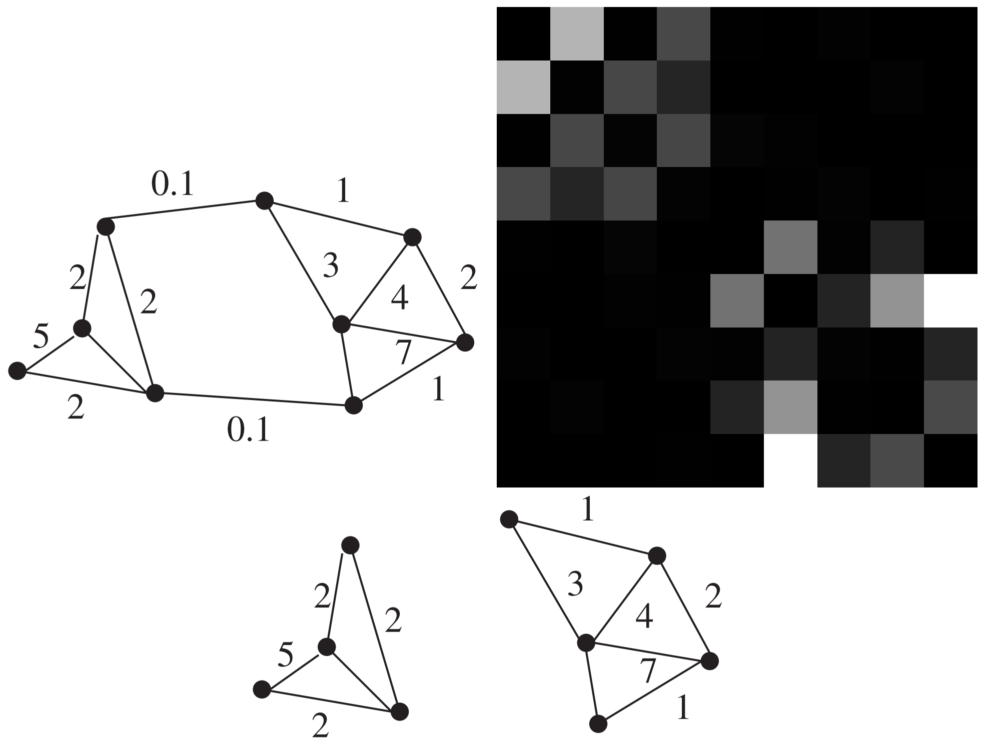


Figure 16.16. On the **top left**, a drawing of an undirected weighted graph; on the **top right**, the weight matrix associated with that graph. Larger values are lighter. By associating the vertices with rows (and columns) in a different order, the matrix can be shuffled. We have chosen the ordering to show the matrix in a form that emphasizes the fact that it is very largely block-diagonal. The figure on the **bottom** shows a cut of that graph that decomposes the graph into two tightly linked components. This cut decomposes the graph's matrix into the two main blocks on the diagonal.

16.5.3 Affinity Measures

When we viewed segmentation as simple clustering, we needed to supply some measure of how similar clusters were. The current model of segmentation simply requires us a weight to place on each edge of the graph; these weights are usually called **affinity measures** in the literature. Clearly, the affinity measure depends on the problem at hand. The weight of an arc connecting similar nodes should be large, and the weight on an arc connecting very different nodes should be small.

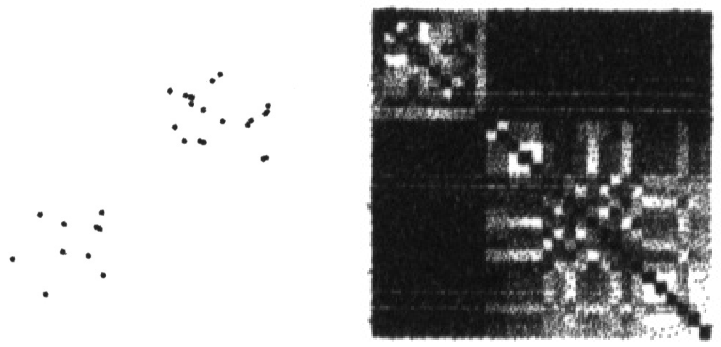


Figure 16.17. On the left, a set of points on the plane. On the right, the affinity matrix for these points computed using a decaying exponential in distance (section 16.5.3), where large values are light and small values are dark. Notice the near block diagonal structure of this matrix; there are two off-diagonal blocks that contain terms that are very close to zero. The blocks correspond to links internal to the two obvious clusters, and the off diagonal blocks correspond to links between these clusters. *figure from Perona and Freeman, A factorization approach to grouping, page 2 figure from Perona and Freeman, A factorization approach to grouping, page 4*

Affinity by Distance

Affinity should go down quite sharply with distance, once the distance is over some threshold. One appropriate expression has the form

$$\mathrm{aff}(\boldsymbol{x},\boldsymbol{y}) = \exp\left\{-\left((\boldsymbol{x}-\boldsymbol{y})^t(\boldsymbol{x}-\boldsymbol{y})/2\sigma_d^2\right)\right\}$$

where σ_d is a parameter which will be large if quite distant points should be grouped and small if only very nearby points should be grouped (this is the expression used for figure 16.17).

Affinity by Intensity

Affinity should be large for similar intensities, and smaller as the difference increases. Again, an exponential form suggests itself, and we can use:

$$\mathrm{aff}(\boldsymbol{x},\boldsymbol{y}) = \exp\left\{-\left((I(\boldsymbol{x})-I(\boldsymbol{y}))^t(I(\boldsymbol{x})-I(\boldsymbol{y}))/2\sigma_I^2\right)\right\}$$

Affinity by Colour

We need a colour metric to construct a meaningful colour affinity function. It's a good idea to use a uniform colour space, and a bad idea to use RGB space, — for reasons that should be obvious, otherwise, reread section 4.3.2 — and an appropriate expression has the form

$$\mathrm{aff}(\boldsymbol{x},\boldsymbol{y}) = \exp\left\{-\left(\mathrm{dist}(\boldsymbol{c}(\boldsymbol{x}),\boldsymbol{c}(\boldsymbol{y}))^2/2\sigma_c^2\right)\right\}$$

where $\boldsymbol{c}_i$ is the colour at pixel i.

Affinity by Texture

The affinity should be large for similar textures and smaller as the difference increases. We adopt a collection of filters $f_1, \ldots, f_n$, and describe textures by the outputs of these filters, which should span a range of scales and orientations. Now for most textures, the filter outputs will not be the same at each point in the texture — think of a chessboard — but a histogram of the filter outputs constructed over a reasonably sized neighbourhood will be well behaved. This suggests a process where we firstly establish a local scale at each point — perhaps by looking at energy in coarse scale filters, or using some other method — and then compute a histogram of filter outputs over a region determined by that scale — perhaps a circular region centered on the point in question. We then write $\boldsymbol{h}$ for this histogram, and use an exponential form:

$$\text{aff}(\boldsymbol{x}, \boldsymbol{y}) = \exp\left\{-\left((\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{y}))^t(\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{y}))/2\sigma_I^2\right)\right\}$$

16.5.4 Eigenvectors and Segmentation

In the first instance, assume that there are k elements and k clusters. We can represent a cluster by a vector with k components. We will allow elements to be associated with clusters using some continuous weight — we need to be a bit vague about the semantics of these weights, but the intention is that if a component in a particular vector has a small value, then it is weakly associated with the cluster, and if it has a large value, then it is strongly associated with a cluster.

Extracting a Single Good Cluster

A good cluster is one where elements that are strongly associated with the cluster also have large values connecting one another in the affinity matrix. Write the matrix representing the element affinities as $\mathcal{A}$, and the vector of weights linking elements to the n'th cluster as $\boldsymbol{w}_n$. In particular, we can construct an objective function

$$\boldsymbol{w}_n^T \mathcal{A} \boldsymbol{w}_n$$

This is a sum of terms of the form

$$\begin{aligned} \{\text{association of element } i \text{ with cluster } n\} \; \times \\ \{\text{affinity between } i \text{ and } j\} \; \times \\ \{\text{association of element } j \text{ with cluster } n\} \end{aligned}$$

We can obtain a cluster by choosing a set of association weights that maximise this objective function. The objective function is useless on its own, because scaling $\boldsymbol{w}_n$ by λ scales the total association by λ^2. However, we can normalise the weights by requiring that $\boldsymbol{w}_n^T \boldsymbol{w}_n = 1$.

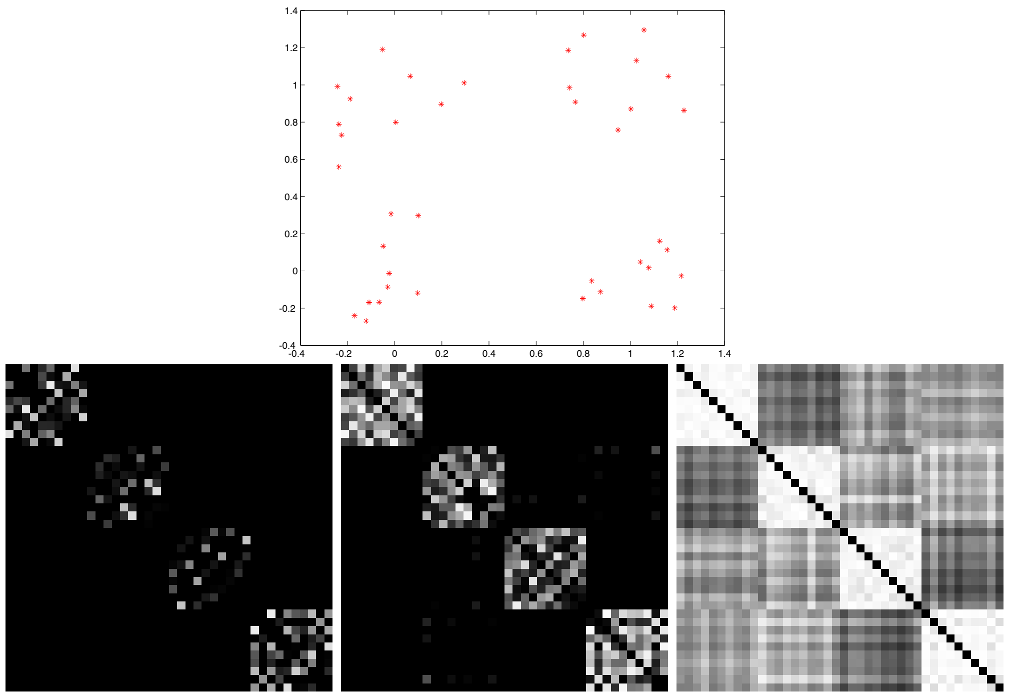


Figure 16.18. The choice of scale for the affinity affects the affinity matrix. The top row shows a dataset, which consists of four groups of 10 points drawn from a rotationally symmetric normal distribution with four different means. The standard deviation in each direction for these points is 0.2. In the second row, affinity matrices computed for this dataset using different values of σ_d. On the **left**, $\sigma_d = 0.1$, in the **center** $\sigma_d = 0.2$ and on the **right**, $\sigma_d = 1$. For the finest scale, the affinity between all points is rather small; for the next scale, there are four clear blocks in the affinity matrix; and for the coarsest scale, the number of blocks is less obvious.

This suggests maximising $\boldsymbol{w}_n^T \mathcal{A} \boldsymbol{w}_n$ subject to $\boldsymbol{w}_n^T \boldsymbol{w}_n = 1$. The Lagrangian is

$$\boldsymbol{w}_n^T \mathcal{A} \boldsymbol{w}_n + \lambda \left(\boldsymbol{w}_n^T \boldsymbol{w}_n - 1 \right)$$

(where λ is a Lagrange multiplier). Differentiation and dropping a factor of two yields

$$\mathcal{A} \boldsymbol{w}_n = \lambda \boldsymbol{w}_n$$

meaning that $\boldsymbol{w}_n$ is an eigenvector of $\mathcal{A}$. This means that we could form a cluster by obtaining the eigenvector with the largest eigenvalue — the cluster weights are the elements of the eigenvector. For problems where reasonable clusters are apparent, we expect that these cluster weights are large for some elements — which belong to the cluster — and nearly zero for others — which do not. In fact, we can get the weights for other clusters from other eigenvectors of $\mathcal{A}$ as well.

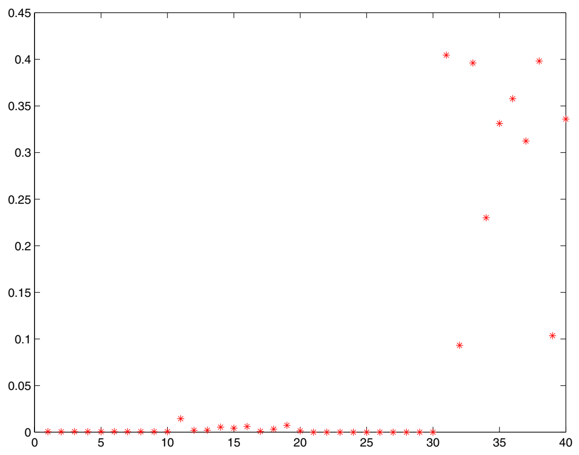


Figure 16.19. The eigenvector corresponding to the largest eigenvalue of the affinity matrix for the dataset of example 16.18, using $\sigma_d = 0.2$. Notice that most values are small, but some — corresponding to the elements of the main cluster — are large. The sign of the association is not significant, because a scaled eigenvector is still an eigenvector.

Extracting Weights for a Set of Clusters

In typical vision problems, there are strong association weights between relatively few pairs of elements. We can reasonably expect to be dealing with clusters that are (a) quite tight and (b) distinct.

These properties lead to a fairly characteristic structure in the affinity matrix. In particular, if we relabel the nodes of the graph, then the rows and columns of the matrix $\mathcal{A}$ are shuffled. We expect to be dealing with relatively few collections of nodes with large association weights; furthermore, that these collections actually form a series of relatively coherent, largely disjoint clusters. This means that we could shuffle the rows and columns of M to form a matrix that is roughly block-diagonal (the blocks being the clusters). Shuffling M simply shuffles the elements of its eigenvectors, so that we can reason about the eigenvectors by thinking about a shuffled version of M (i.e. figure 16.16 is a fair source of insight).

The eigenvectors of block-diagonal matrices consist of eigenvectors of the blocks, padded out with zeros. We expect that each block has an eigenvector corresponding to a rather large eigenvalue — corresponding to the cluster — and then a series of small eigenvalues of no particular significance. From this, we expect that, if there are c significant clusters (where $c < k$), the eigenvectors corresponding to the c largest eigenvalues each represent a cluster.

This means that each of these eigenvectors is an eigenvector of a block, padded with zeros. In particular, a typical eigenvector will have a small set of large values — corresponding to its block — and a set of near-zero values. We expect that only one of these eigenvectors will have a large value for any given component; all the others

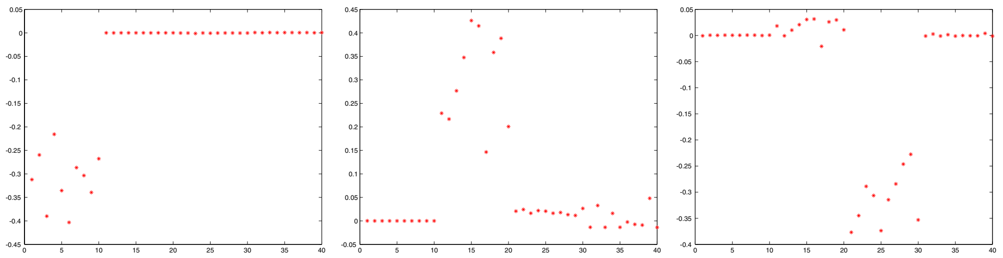


Figure 16.20. The three eigenvectors corresponding to the next three largest eigenvalues of the affinity matrix for the dataset of example 16.18, using $\sigma_d = 0.2$ (the eigenvector corresponding to the largest eigenvalue is given in figure 16.19). Notice that most values are small, but for (disjoint) sets of elements, the corresponding values are large. This follows from the block structure of the affinity matrix. The sign of the association is not significant, because a scaled eigenvector is still an eigenvector.

will be small (figure 16.20). Thus, we can interpret eigenvectors corresponding to the c largest magnitude eigenvalues as cluster weights for the first c clusters. One can usually quantize the cluster weights to zero or one, to obtain discrete clusters; this is what has happened in the figures.

```
Construct an affinity matrix

Compute the eigenvalues and eigenvectors of the affinity matrix

Until there are sufficient clusters

    Take the eigenvector corresponding to the
    largest unprocessed eigenvalue; zero all components corresponding
    to elements that have already been clustered, and threshold the
    remaining components to determine which element
    belongs to this cluster, choosing a threshold by
    clustering the components, or
    using a threshold fixed in advance.

    If all elements have been accounted for, there are
    sufficient clusters

end
```

Algorithm 16.6: *Clustering by Graph Eigenvectors*

This is a qualitative argument, and there are graphs for which the argument is decidedly suspect. Furthermore, we have been decidedly vague about how to determine c, though our argument suggests that poking around in the spectrum of $\mathcal{A}$ might be rewarding — one would hope to find a small set of large eigenvalues, and a large set of small eigenvalues (figure 16.21).

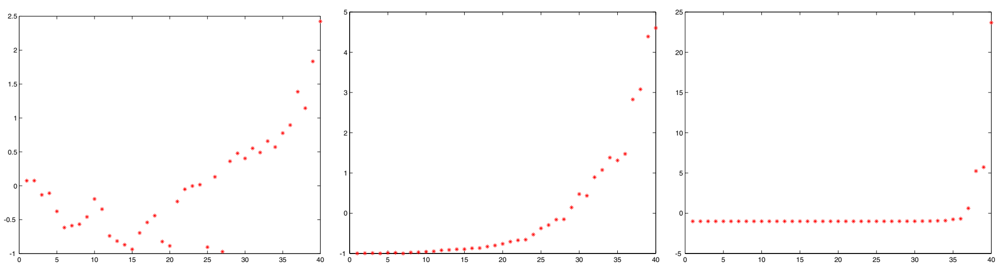


Figure 16.21. The number of clusters is reflected in the eigenvalues of the affinity matrix. The figure shows eigenvalues of the affinity matrices for each of the cases in figure 16.18. On the **left**, $\sigma_d = 0.1$, in the **center** $\sigma_d = 0.2$ and on the **right**, $\sigma_d = 1$. For the finest scale, there are many rather large eigenvalues — this is because the affinity between all points is rather small; for the next scale, there are four eigenvalues rather larger than the rest; and for the coarsest scale, there are only two eigenvalues rather larger than the rest.

16.5.5 Normalised Cuts

The qualitative argument of the previous section is somewhat soft. For example, if the eigenvalues of the blocks are very similar, we could end up with eigenvectors that do not split clusters, because any linear combination of eigenvectors with the same eigenvalue is also an eigenvector (figure 16.22).

An alternative approach is to cut the graph into two connected components such that the cost of the cut is a small fraction of the total affinity within each group. We can formalise this as decomposing a weighted graph V into two components A and B, and scoring the decomposition with

$$\frac{cut(A,B)}{assoc(A,V)} + \frac{cut(A,B)}{assoc(B,V)}$$

(where $cut(A,B)$ is the sum of weights of all edges in V that have one end in A and the other in B, and $assoc(A,V)$ is the sum of weights of all edges that have one end in A). This score will be small if the cut separates two components that have very few edges of low weight between them and many internal edges of high weight. We would like to find the cut with the minimum value of this criterion, called a **normalized cut**.

We write $\boldsymbol{y}$ is a vector of elements, one for each graph node, *whose values are either* 1 *or* $-b$. The values of $\boldsymbol{y}$ are used to distinguish between the components

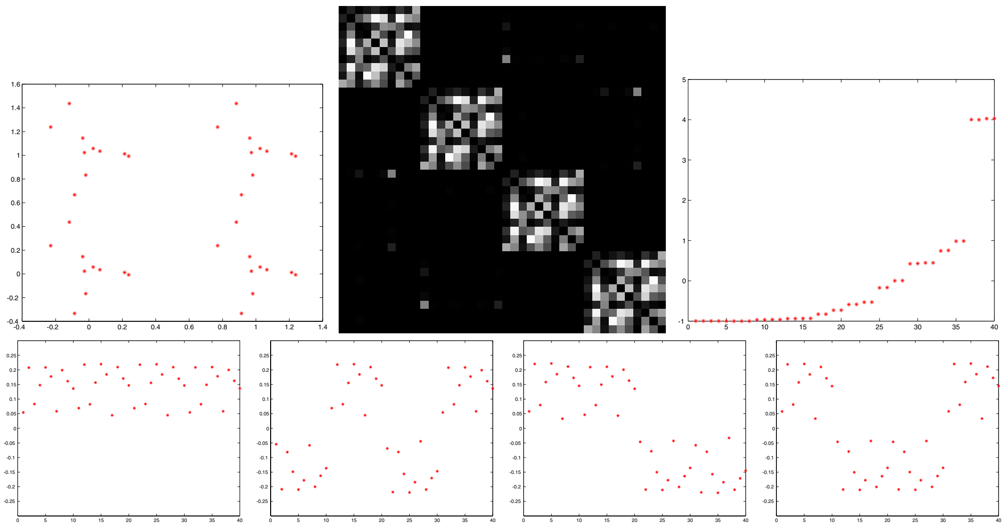


Figure 16.22. Eigenvectors of an affinity matrix can be a misleading guide to clusters. The dataset on the **top left** consists of four copies of the same set of points; this leads to a repeated block structure in the affinity matrix shown in the **top center**. Each block has the same spectrum, and this results in a spectrum for the affinity matrix that has (roughly) four copies of the same eigenvalue (**top right**). The bottom row shows the eigenvectors corresponding to the four largest eigenvalues; notice (a) that the values don't suggest clusters and (b) a linear combination of the eigenvectors might lead to a quite good clustering.

of the graph: if the i'th component of $\boldsymbol{y}$ is 1, then the corresponding node in the graph belongs to one component, and if it is $-b$, the node belongs to the other. We write the affinity matrix as $\mathcal{A}$ is the matrix of weights between nodes in the graph and $\mathcal{D}$ is the **degree matrix**; each diagonal element of this matrix is the sum of weights coming into the corresponding node, that is

$$D_{ii} = \sum_j A_{ij}$$

and the off-diagonal elements of $\mathcal{D}$ are zero. In this notation, and with a little manipulation, our criterion can be rewritten as:

$$\frac{\boldsymbol{y}^T(\mathcal{D} - \mathcal{A})\boldsymbol{y}}{\boldsymbol{y}^T\mathcal{D}\boldsymbol{y}}$$

We now wish to find a vector $\boldsymbol{y}$ that minimizes this criterion. The problem we have set up is an **integer programming** problem, and because it is exactly equivalent to the graph cut problem, it isn't any easier. The difficulty is the discrete values for elements of $\boldsymbol{y}$ — in principle, we could solve the problem by testing every possible

$\boldsymbol{y}$, but this involves searching a space whose size is exponential in the number of pixels which will be slow[2]. A common approximate solution to such problems is to compute a *real* vector $\boldsymbol{y}$ that minimizes the criterion. Elements are then assigned to one side or the other by testing against a threshold. There are then two issues: firstly, we must obtain the real vector, and secondly, we must choose a threshold.

Obtaining a Real Vector

The real vector is easily obtained. It is an exercise to show that a solution to

$$(\mathcal{D} - \mathcal{A})\boldsymbol{y} = \lambda \mathcal{D}\boldsymbol{y}$$

is a solution to our problem *with real values.* The only question is which generalised eigenvector to use? It turns out that the smallest eigenvalue is guaranteed to be zero, so the eigenvector corresponding to the second smallest eigenvalue is appropriate. The easiest way to determine this eigenvector is to perform the transformation $\boldsymbol{z} = \mathcal{D}^{1/2}\boldsymbol{y}$, and so get:

$$\mathcal{D}^{-1/2}(\mathcal{D} - \mathcal{A})\mathcal{D}^{-1/2}\boldsymbol{z} = \lambda \boldsymbol{z}$$

and $\boldsymbol{y}$ follows easily. Note that solutions to this problem are also solutions to

$$\mathcal{N}\boldsymbol{z} = \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}\boldsymbol{z} = \mu \boldsymbol{z}$$

and $\mathcal{N}$ is sometimes called the **normalised affinity matrix**.

Choosing a Threshold

Finding the appropriate threshold value is not particularly difficult; assume there are N nodes in the graph, so that there are N elements in $\boldsymbol{y}$, and at most N different values. Now if we write $ncut(v)$ for the value of the normalised cut criterion at a particular threshold value v, there are at most $N + 1$ values of $ncut(v)$. We can form each of these values, and choose a threshold that leads to the smallest. Notice also that this formalism lends itself to recursion, in that each component of the result is a graph, and these new graphs can be split, too. A simpler criterion, which appears to work in practice, is to walk down the eigenvalues and use eigenvectors corresponding to smaller eigenvalues to obtain new clusters.

16.6 Discussion

Segmentation is a difficult topic, and there are a huge variety of methods. Surveys of mainly historical interest are [Riseman and Arbib, 1977; Fu and Mui, 1981; Haralick and Shapiro, 1985; Nevatia, 1986; Mitiche and Aggarwal, 1985]; more recent survey are rare, but there is [Pal and Pal, 1993]. One reason is that it is typically quite hard to assess the performance of a segmenter at a level more useful than that of showing some examples. Evaluation is easier in the context of a specific

[2] As in, probably won't finish before the universe burns out.

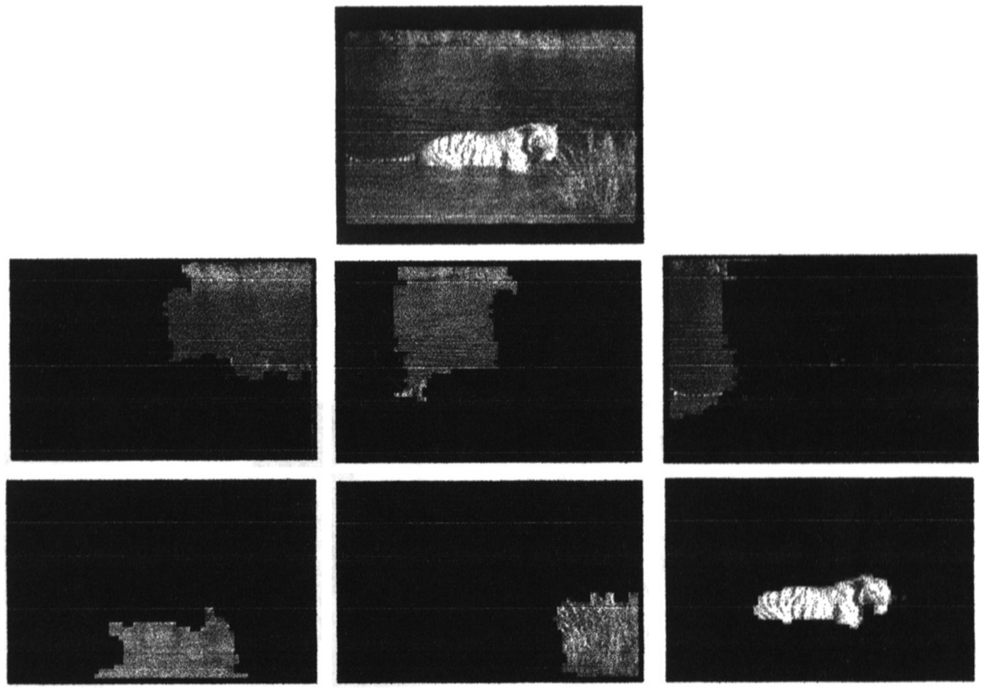


Figure 16.23. The image on top is segmented using the normalised cuts framework, described in the text, into the components shown. The affinity measures used involved intensity and texture, as in section 16.5.3. The image of the swimming tiger yields one segment that is essentially tiger, one that is grass, and four components corresponding to the lake. Note the improvement over k-means segmentation obtained by having a texture measure.

task, and there are several papers dealing with assorted tasks [Hartley *et al.*, 1982; Yasnoff *et al.*, 1977; Zhang, 1996; Ashjaei and Soltanian-Zadeh, 1997; Ranade and Prewitt, 1980].

The original clustering segmenter is [Ohlander, 1975; Ohlander *et al.*, 1978]. Clustering methods tend to be rather arbitrary — remember, this doesn't mean they're not useful — because there really isn't much theory available to predict what should be clustered and how. It is clear that what we should be doing is forming clusters that are helpful to a particular application, but this criterion hasn't been formalised in any useful way. In this chapter, we have attempted to give the big picture while ignoring detail, because a detailed record of what has been done would be unenlightening.

A variety of graph theoretical clustering methods have been used in vision [Wu and Leahy, 1993; Sarkar and Boyer, 1996; Sarkar and Boyer, 1998; Cox *et al.*, 1996] and the summary in [Weiss, 1999]. The normalized cuts formalism is due

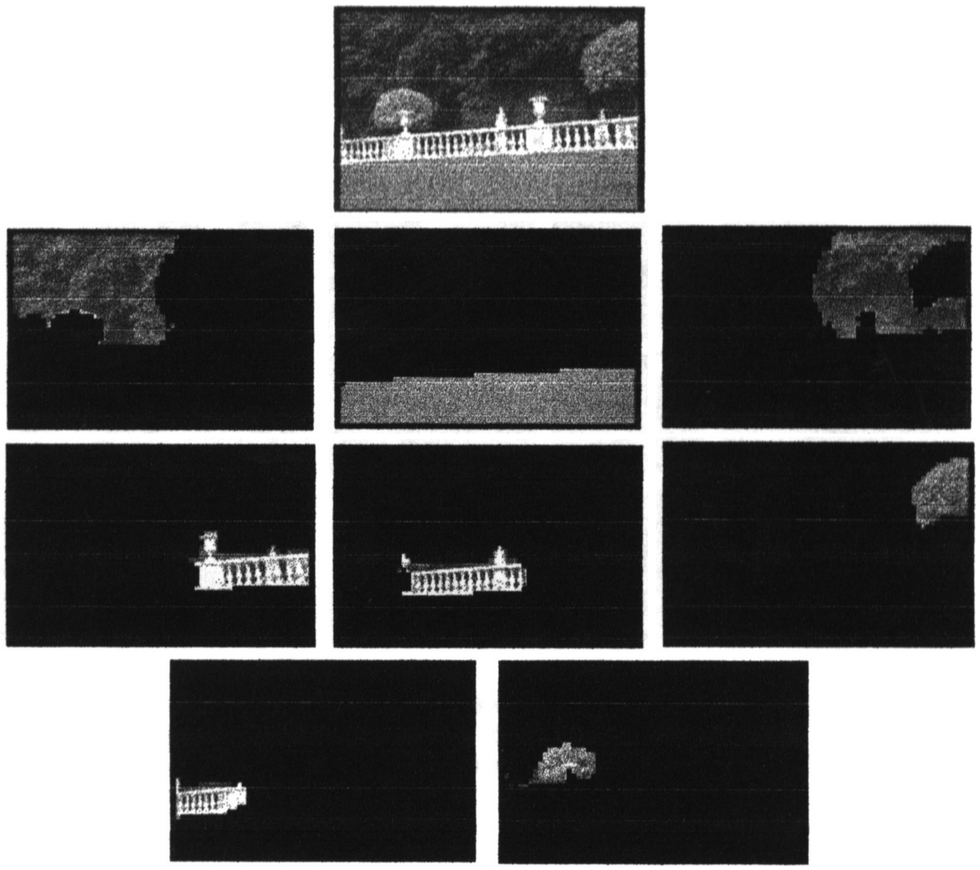


Figure 16.24. The image on top is segmented using the normalised cuts framework, described in the text, into the components shown. The affinity measures used involved intensity and texture, as in section 16.5.3. Again, note the improvement over k-means segmentation obtained by having a texture measure; the railing now shows as three reasonably coherent segments.

to [Shi and Malik, 1997; Shi and Malik, 2000]; variants include applications to motion segmentation [Shi and Malik, 1998a] and methods for deducing similarity metrics from outputs [Shi and Malik, 1998b]. There are numerous alternate criteria (e.g. [Perona and Freeman, 1998; Cox *et al.*, 1996]). We have stressed the graph theoretical clustering methods because their ability to deal with any affinity function one cares to name is an attractive feature.

Segmentation is also a key open problem in vision, which is why a detailed record of what has been done would be huge. Up until quite recently, it was usual to talk about recognition and segmentation as if they were distinct activities. This view is

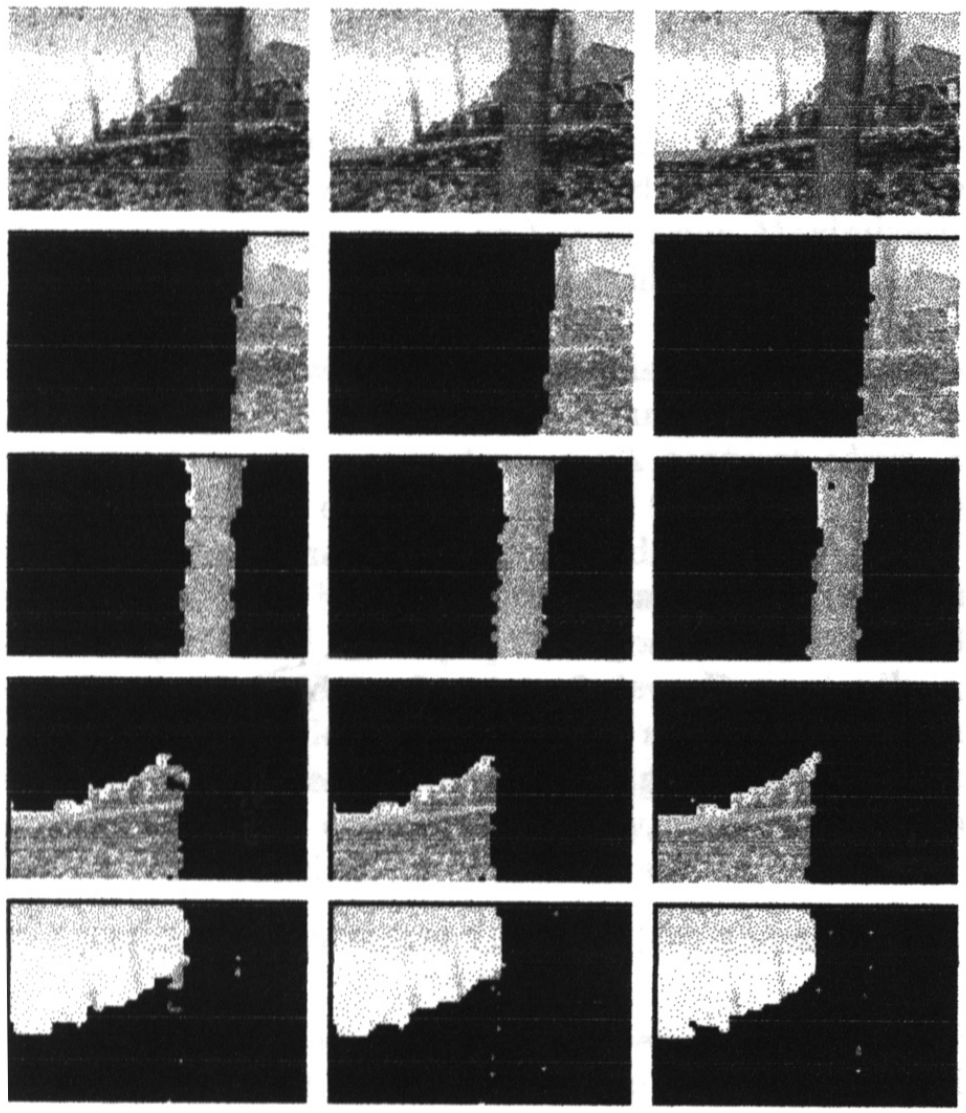


Figure 16.25. Three of the first six frames of a motion sequence, which shows a moving view of a house; the tree sweeps past the front of the house. Below, we see spatio-temporal segments established using normalised cuts and a spatio-temporal affinity function (section 16.5.3).

going out of fashion — as it should — because there isn't much point in creating a segmented representation that doesn't help with some application; furthermore, if we can be crisp about what should be recognised, that should make it possible to be crisp about what a segmented representation should look like.

16.6.1 Segmentation and Grouping in People

There is a large literature on the role of grouping in human visual perception. Standard Gestalt handbooks include [Koffka, 1935; Kanizsa, 1979]. Subjective contours were first described by Kanisza; there is a broad summary discussion in [Kanizsa, 234]. Palmer's authoritative book gives a much broader picture than we can supply here [Palmer, 1999]. There is a great deal of information about the development of different theories of vision and the origins of Gestalt thinking in [Gordon, 1997]. Some groups appear to be formed remarkably early in the visual process, a phenomenon known as "pop out" [Triesman, 1982].

16.6.2 Perceptual Grouping

On occasion, a distinction is drawn between perceptual organisation — which is seen as clustering image tokens into useful groups — and segmentation — which is seen as decomposing images into regions. We don't accept this distinction; the advantage of seeing these problems as manifestations of the same activity is that one can convert algorithmic advances from one problem to another freely. We haven't discussed some aspects of perceptual organisation in great detail, mainly because our emphasis is on exposition rather than historical accuracy, and these methods follow from the unified view. For example, there is a long thread of literature on clustering image edge points or line segments into configurations that are unlikely to have arisen by accident; we cover some of these ideas in the following chapter, but also draw the readers attention to [Lowe, 1985; Mohan and Nevatia, 1992; Sarkar and Boyer, 1994; Sarkar and Boyer, 1992; Sarkar and Boyer, 1993; Amir and Lindenbaum, 1996; Huttenlocher and Wayner, 1992]. In building user interfaces, it can (as we hinted above) be very helpful to know what is perceptually salient (e.g. [Saund and Moran, 1995]).

Assignments

Exercises

- We wish to cluster a set of pixels using colour and texture differences. The objective function

$$\Phi(\text{clusters}, \text{data}) = \sum_{i \in \text{clusters}} \left\{ \sum_{j \in i\text{`th cluster}} (\boldsymbol{x}_j - \boldsymbol{c}_i)^T (\boldsymbol{x}_j - \boldsymbol{c}_i) \right\}$$

  used in section 16.4.2 may be inappropriate — for example, colour differences could be too strongly weighted if colour and texture are measured on different scales.

1. Extend the description of the k-means algorithm to deal with the case of an objective function of the form

$$\Phi(\text{clusters}, \text{data}) = \sum_{i \in \text{clusters}} \left\{ \sum_{j \in i\text{`th cluster}} (\boldsymbol{x}_j - \boldsymbol{c}_i)^T \mathcal{S} (\boldsymbol{x}_j - \boldsymbol{c}_i) \right\}$$

   where $\mathcal{S}$ is an a symmetric, positive definite matrix.
2. For the simpler objective function, we had to ensure that each cluster contained at least one element (otherwise we can't compute the cluster center). How many elements must a cluster contain for the more complicated objective function?
3. As we remarked in section 16.4.2, there is no guarantee that k-means gets to a global minimum of the objective function; show that it must always get to a local minimum.
4. Sketch two possible local minima for a k-means clustering method clustering data points described by a two-dimensional feature vector. Use an example with only two clusters, for simplicity. You shouldn't need many data points. You should do this exercise for both objective functions.

- Read [Shi and Malik, 97] and follow the proof that the normalised cut criterion leads to the integer programming problem given in the text. Why does the normalised affinity matrix have a null space? give a vector in its kernel.

- Show that choosing a *real* vector that maximises the expression

$$\frac{\boldsymbol{y}^T(\mathcal{D} - \mathcal{W})\boldsymbol{y}}{\boldsymbol{y}^T \mathcal{D} \boldsymbol{y}}$$

  is the same as solving the eigenvalue problem

$$\mathcal{D}^{-1/2} \mathcal{W} \mathcal{W} \boldsymbol{z} = \mu \boldsymbol{z}$$

  where $\boldsymbol{z} = \mathcal{D}^{-1/2}\boldsymbol{y}$.

- Grouping based on eigenvectors presents one difficulty: how to obtain eigenvectors for a large matrix quickly. The standard method is **Lanczos' algorithm**; read [], p.xxx-yyy, and implement this algorithm. Determine the time taken to obtain eigenvectors for a series of images of different sizes. Is your data consistent with the (known) order of growth of the algorithm?

- This exercise explores using normalised cuts to obtain more than two clusters. One strategy is to construct a new graph for each component separately, and call the algorithm recursively. You should notice a strong similarity between this approach and classical divisive clustering algorithms. The other strategy is to look at eigenvectors corresponding to smaller eigenvalues.

1. Explain why these strategies are not equivalent.
2. Now assume that we have a graph that has two connected components. Describe the eigenvector corresponding to the largest eigenvalue.
3. Now describe the eigenvector corresponding to the second largest eigenvalue.
4. Turn this information into an argument that the two strategies for generating more clusters should yield quite similar results under appropriate conditions; what are appropriate conditions?

Programming Assignments

- Build a background subtraction algorithm using a moving average and experiment with the filter.
- Build a shot boundary detection system using any two techniques that appeal, and compare performance on different runs of video.
- Implement a segmenter that uses k-means to form segments based on colour and position. Describe the effect of different choices of the number of segments; investigate the effects of different local minima.

Chapter 17

SEGMENTATION BY FITTING A MODEL

An alternative view of segmentation is to assert that pixels (tokens, etc.) belong together, because together they conform to some model. This view is rather similar to the clustering view; the main difference is that the model is now explicit, and may involve relations at a larger scale than from token to token. For example, imagine a program that attempts to assemble tokens into groups that "look like" a line (in some sense we don't need to make precise at this point). It isn't possible to do this by looking only at pairwise relations between tokens — instead, we must look at some properties of the collective of tokens.

Typically, fitting involves choosing a model, and then declaring some criterion by which a good fit can be identified. In the first instance, we will use criteria which are good but should look rather arbitrary. In section 17.1, we discuss fitting lines to tokens, an apparently straightforward exercise that quickly becomes rather subtle; we shall revisit it in the following chapter. In section 17.2, we extend these ideas to fitting curves, and in section 17.3 we show how to use them to find body segments. The criteria that we use to fit lines are, in fact, justifiable using a probabilistic argument, and we introduce the probabilistic approach in section 17.4. In section 17.5, we introduce the vexed issue of outliers, leading to an algorithm that searches a data set for components that are consistent with a model. Finally, in section 17.6, we show the use of this algorithm to fit a fundamental matrix.

17.1 Fitting Lines

Tokens may cluster for other reasons than they are close or have similar colour. One reason to cluster tokens is that, together, they form a familiar geometric configuration — for example, they all lie on a line or on a circle. Clustering tokens into structures is often called **fitting**. Line fitting, in particular, 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 28). 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. In either case, a report of all straight lines in the image is an extremely useful segmentation.

There are three problems in line fitting. Firstly, given the points that belong to a line, what is the line? Secondly, which points belong to which line? Finally, how many lines are there. We describe the Hough transform, is a method that promises a solution to all three, in section 17.1.1. In practice, Hough transforms are not terribly satisfactory. We discuss standard solutions to the first problem in section 17.1.2, and two of many solutions to the second in section 17.1.3. A more sophisticated discussion requires a probabilistic model, and we will revisit the topic in chapter 18.

17.1.1 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.

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

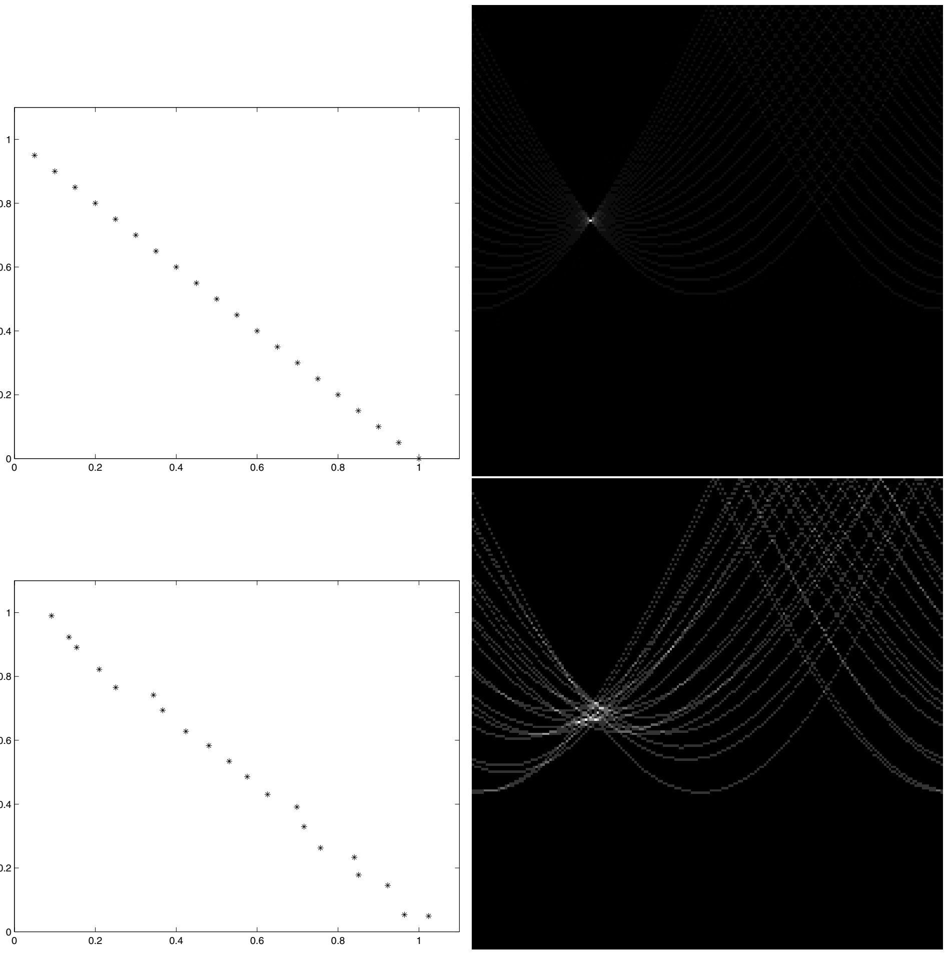


Figure 17.1. The Hough transform maps each point like token to a curve of possible lines (or other parametric curves) through that point. These figures illustrate the Hough transform for lines. The left hand column shows points, and the right hand column shows the corresponding accumulator arrays (the number of votes is indicated by the 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.

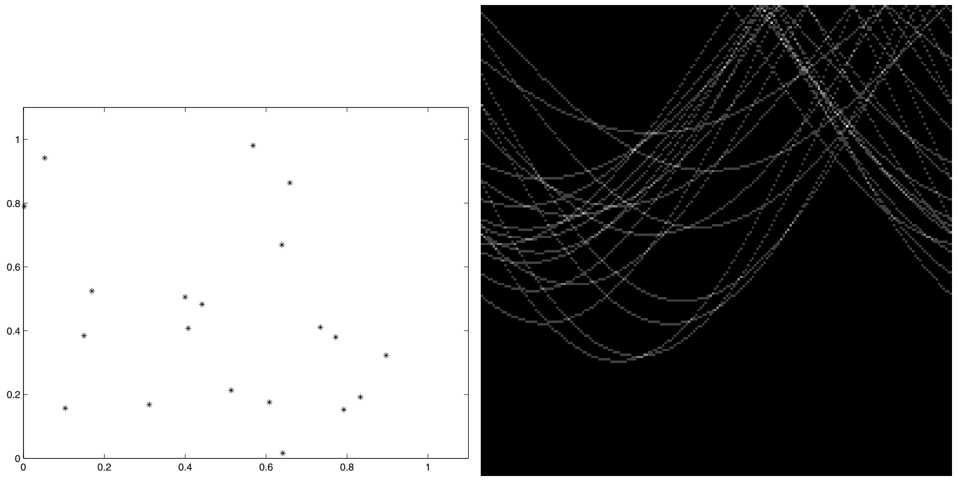


Figure 17.2. The Hough transform for a set of random points can lead to quite large sets of votes in the accumulator array. As in figure 17.1, the left hand column shows points, and the right hand column shows the corresponding accumulator arrays (the number of votes is indicated by the grey level, with 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 17.3 and 17.4 explore noise issues somewhat further.

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.

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 17.1).
- **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 17.3 and 17.4).

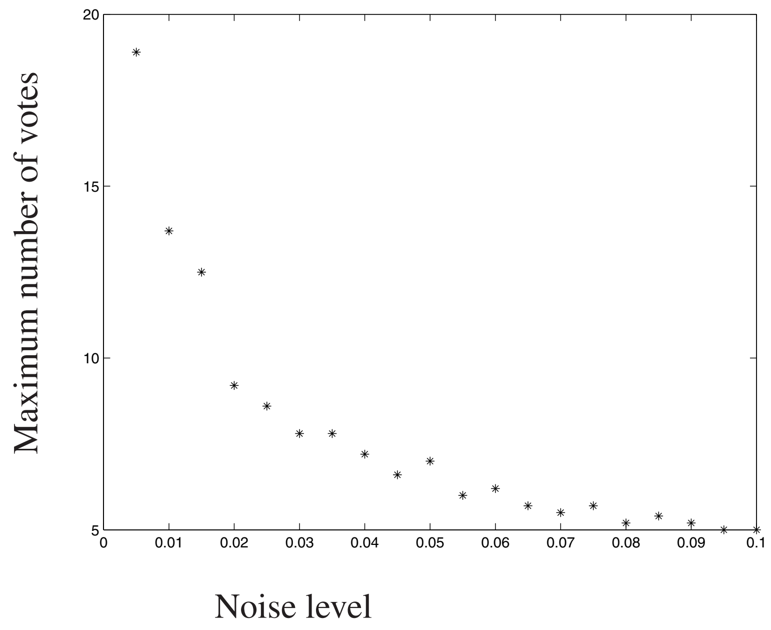


Figure 17.3. The effects of noise make it difficult to use a Hough transform robustly. The plot shows the maximum number of votes in the accumulator array for a Hough transform of 20 points on a line perturbed by uniform noise, plotted against the magnitude of the noise. The noise displaces the curves from each other, and quite quickly leads to a collapse in the number of votes. The plot has been averaged over 10 trials.

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.

- **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.

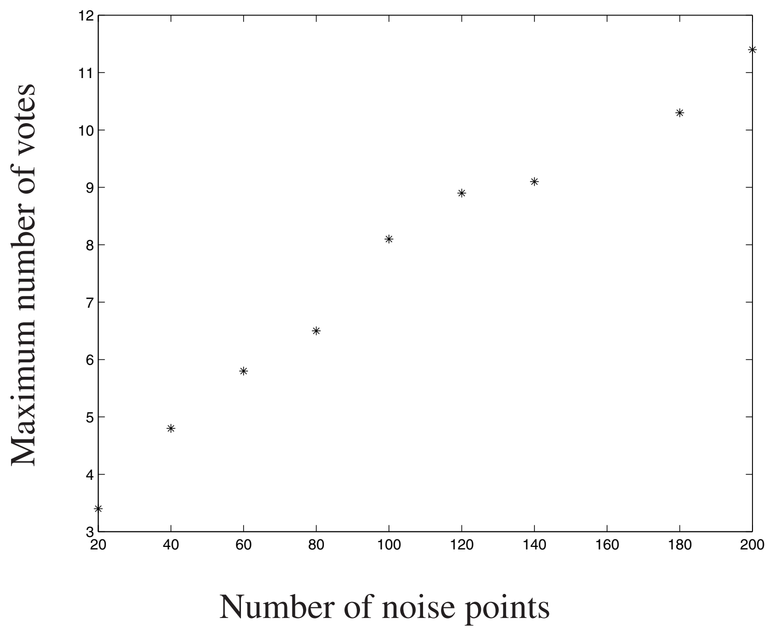


Figure 17.4. A plot of the maximum number of votes in the accumulator array for a Hough transform of a set of points whose coordinates are uniform random numbers in the range $[0, 1]$, plotted against the number of points. As the level of noise goes up, the number of votes in the right bucket goes down 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 17.3, 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.

17.1.2 Line Fitting with Least Squares

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 that

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

to simplify the presentation.

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 substantial bias. For this approach, we represent a line as $y =$

$ax + b$. At each data point, we have (x_i, y_i); we decide to choose the line that best predicts the measured y-coordinate for each measured x coordinate.

This means we want to choose the line that minimises

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

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

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

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 17.5). In fact, the process is so dependent on coordinate frame that it doesn't represent vertical lines at all.

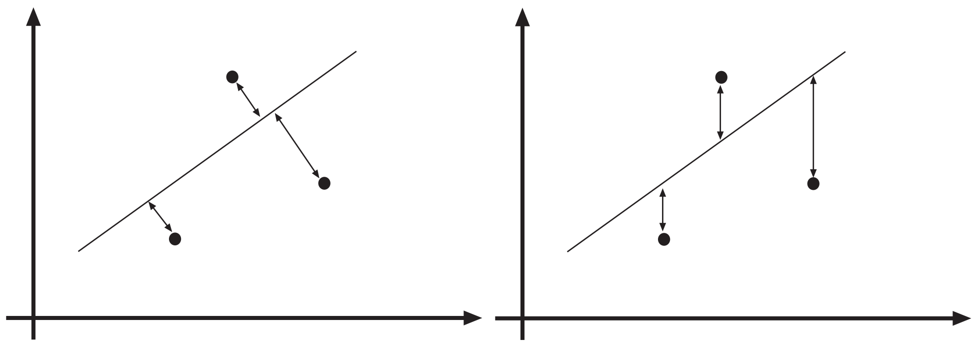


Figure 17.5. Left: Total least squares models data points as being generated by an abstract point along the line to which is added a vector perpendicular to the line, with a length given by a zero mean, Gaussian random variable. This means that the distance from data points to the line has a normal distribution. 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.

Total Least Squares

We could work with the actual distance between the point and the line (rather than the vertical distance). This leads to a problem known as **total least squares**. We can represent a line as the collection of points where $ax+by+c=0$. Every line can be represented in this way, and we can think of a line as a triple of values (a, b, c). Notice that for $\lambda \neq 0$, the line given by $\lambda(a, b, c)$ is the same as the line represented by (a, b, c). In the exercises, you are asked to prove the simple, but extremely useful, result that the perpendicular distance from a point (u, v) to a line (a, b, c) is given by $\text{abs}(au + bv + c)$ *if* $a^2 + b^2 = 1$. In our experience, this fact is useful enough to be worth memorizing.

To minimize the sum of perpendicular distances between points and lines, we need to minimize

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

where $a^2 + b^2 = 1$ and C is some 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.

17.1.3 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*

Allocating points to lines with K-means

Assume that points carry no hints about which line they lie on (i.e. there is no colour, etc. information to help, and, crucially, the points are not linked). We can attempt to determine which point lies on which line is to use 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:

- allocate each point to the closest line;
- fit the best line to the points allocated to each line.

resulting in the algorithm shown in figure 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 by looking at the sum of perpendicular distances of points from their lines (which operates as a log likelihood).

```
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
```

Algorithm 17.2: *K-means line fitting by allocating points to the closest line and then refitting.*

17.2 Fitting Curves

In principle, fitting curves is not very different from fitting lines. We minimize the sum of squared 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). We sketch some solutions for the distance problem for the two main representations of curves.

17.2.1 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

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

at the point (u, v). We now have two equations in two unknowns, and *in principle* can solve them.

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

An alternative approximation is to use

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

which has the advantage of not requiring a 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

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**.

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.

17.2.2 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:

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

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

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

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.

17.3 Example: Finding Body Segments by Fitting

A body segment is, rather roughly, a cylinder. We should like to find near-cylinders reliably. It is surprisingly difficult to do this in a wholly satisfactory fashion (there remains no standard method). One approach is to study the relationship between a surface and its outline, and from this derive a model of the appearance of the boundary of a cylinder. We can then sift through edge points looking for collections that look like cylinders. Because the geometry is relatively simple, we will think of this problem as a search for the outline of a general surface of revolution (so the method can be recycled).

17.3.1 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 17.6). 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 17.6).

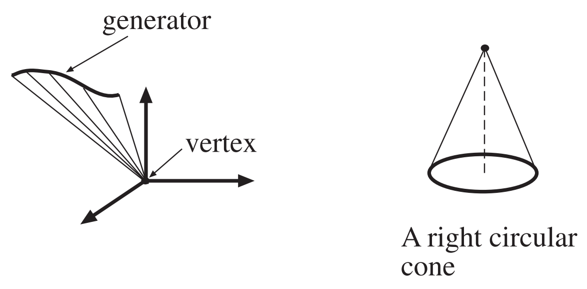


Figure 17.6. Cones are surfaces obtained by sweeping rays through a vertex along a generator. A right circular cone is a special cone, where the generator is a circle and the line joining the vertex with the center of the circle is normal to the circle's plane.

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 17.7). All this should be obvious (you are asked for a proof in the exercises), but is surprisingly useful.

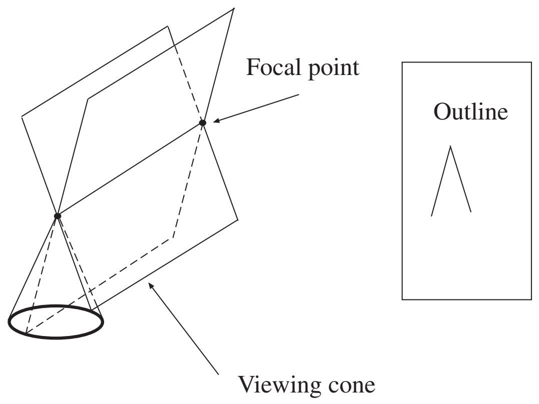


Figure 17.7. The viewing cone is a set of planes tangent to the cone, and passing through both the vertex of the cone and the focal point. The outline of a cone is obtained by slicing the viewing cone with a plane, and is a set of lines through a single point.

Surfaces of Revolution

A **surface of revolution** — or **SOR** for short — is a surface obtained by rotating a generating curve around some straight axis. Notice that, locally, a surface of revolution is a right circular cone — this means that we can think of an SOR as a set of thin sections of *different* right circular cones stacked up on top of one another. This is most easily demonstrated by approximating the generating curve by a polygon of short line segments tangent to the curve. An alternative is to notice that the tangent at one point of the generating curve is rotated around the axis, too, to form a right circular cone.

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 17.8). The effect is governed by the field of view of the camera, and for the vast majority of practical cameras, it is tiny.

This means that the outline of an SOR has two "sides", which correspond to one another under the symmetry. This symmetry has a line of fixed points, which is the projection of the axis of the SOR. Now if we take the tangents to the outline at two corresponding points, they intersect along the projection of the axis. There are several different proofs: you can observe that this configuration is basically a cone; alternatively, you could notice that the tangents are one anothers image under the symmetry, meaning that their intersection is a fixed point and so lies on the axis. These observations suggest a methods for finding the outlines of surfaces of revolution.

17.3.2 Using Constraints to Fit SOR Outlines

In principle, we could exploit the symmetry constraint directly by looking for pairs of edge curves that have a symmetry. This is usually impractical, because curves are usually broken, etc. meaning that it is unlikely that one will obtain a pair where more than short segments are symmetric. Instead, it is more usual to construct local symmetries and then assemble them into conforming groups.

Symmetries and Surfaces of Revolution

Two points on image curves where the tangent is at about the same angle to the line joining the points (figure 17.9) 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

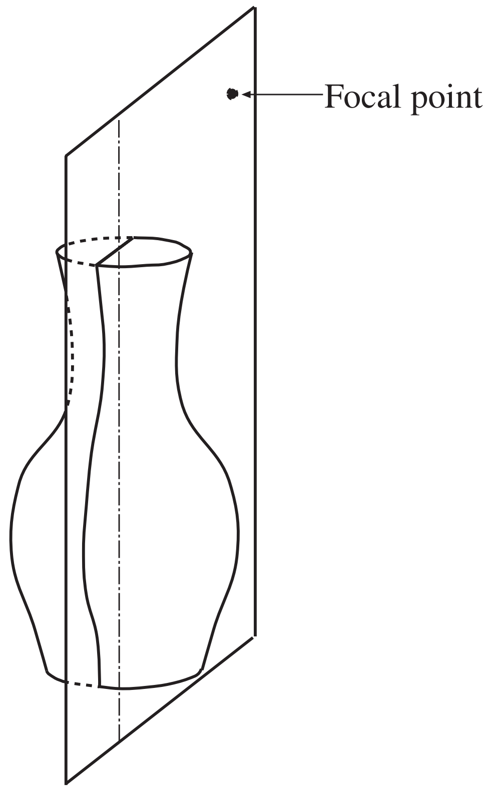


Figure 17.8. A surface of revolution and a focal point together give a plane of symmetry, that passes through the axis of the SOR and the focal point. The contour generator must have a mirror symmetry in this plane. This doesn't mean that the outline has an exact mirror symmetry, because the outline 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.

test. However, the strategy is helpful for cylinders, because it is easier to winnow out unsatisfactory groups of symmetries.

Cylinders and Body Segments

The local symmetries generated by a cylinder 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

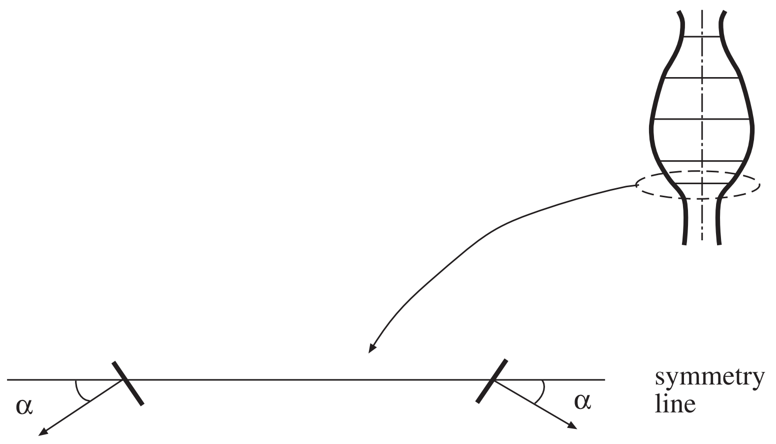


Figure 17.9. A local symmetry is a pair of contour points where the tangents to the contours are at roughly the same angle to the line joining the contours (the symmetry line). Such symmetries are often seen in the outlines of surfaces of revolution.

just about practical to winnow through them looking for groups that satisfy these constraints.

Cylinders can be found with a relatively crude algorithm (too crude to display!). We use a combination of agglomerative clustering and incremental line fitting. 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 then fit a line to the midpoints. If the line fits the midpoints sufficiently well, then we accept these new symmetries. We 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.

17.4 Fitting as a Probabilistic Inference Problem

Up to this point, our criteria for fitting to a model have been arbitrary. Total least squares seems like a reasonable criterion, but (of course!) the criterion should depend on the kind of error model that we expect — how did the tokens come to not lie on a line in the first place? We return to the problem of fitting a line to a set of points that are known to have come from the line. It turns out that total least squares is, quite naturally, a probabilistic criterion. We start with a model that indicates how image measurements arise.

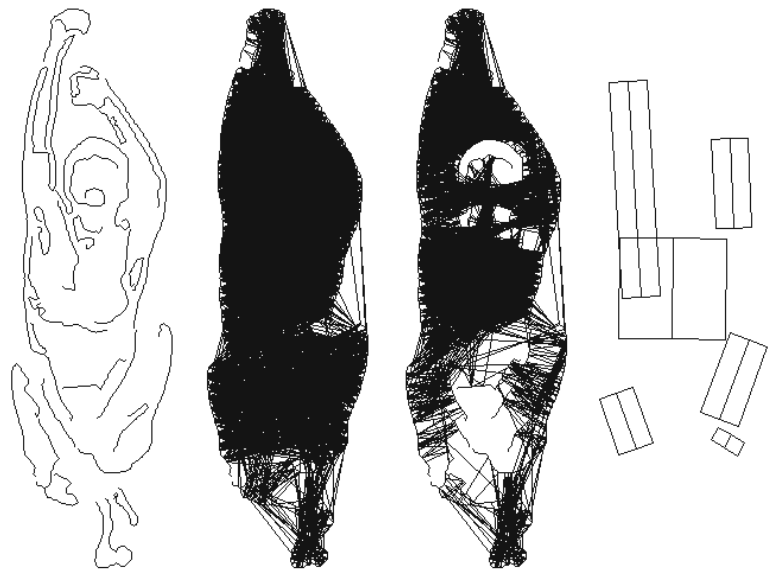


Figure 17.10. On the **far left**, the edges obtained from an image of a sitting human figure. The subject is wearing very little clothing; the doubled edges at the boundary of his limbs are characteristic of images of people, as there can be quite sharp variations in shading across a limb axis. On the **left**, all local symmetries superimposed on that image. Notice that there is a very large number of symmetries; this is a characteristic difficulty with the representation. It is possible to test symmetries to determine whether the shading pattern across a symmetry corresponds to that across a limb (see[Haddon and Forsyth, 1998b]), and the figure on the **right** shows symmetries that pass this test. Some, but not all, of the extended segments consisting of roughly parallel symmetries with roughly collinear centers are shown on the **far right**.

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 $a^2+b^2 = 1$. This model yields $P(\text{measurements}|a, b, c)$ (the likelihood) as

$$\prod_i P(x_i, y_i|a, b, c)$$

Now we could choose either the maximum likelihood or the maximum *a posteriori* line, given this model. Typically, we have no particular reason to prefer one line over another, and maximum likelihood is fine.

Now the log-likelihood is

$$-\frac{1}{2\sigma^2}\sum_i (ax_i + by_i + c)^2$$

given that $a^2 + b^2 = 1$. Maximising the likelihood boils down to minimizing the sum of perpendicular distances between points and lines, as in section 17.1.2. There are two significant phenomena that we must deal with in using this criterion:

- **Robustness:** the total least squares criterion places huge weight on large errors. This could become a serious problem; for example, if one data point lay a long way from a line that fits all others well (we discuss some mechanisms by which this can happen below), the resulting fitted line will be heavily biased by that data point. This phenomenon can become a very serious problem. For example, if we are fitting a fundamental matrix to a data set, we need correspondences between data points in left and right images; but if we get one correspondence wrong, we have a potentially huge error in our data set. We discuss this difficulty in detail in the following two sections.

- **Missing data:** we assumed that we knew which points belonged to the line; it is usually the case that we do not. For example, we might have a set of measured points, some of which come from a line and others of which are noise. If we knew which points came from a line, it would be easy to determine what the line was. Similarly, if we knew what line generated the points, it would be easy to determine which points had come from the line. The missing data — which point is noise and which is not — is a crucial component of the problem. Most segmentation problems can be seen as missing data problems; we devote most of chapter 18 to this view.

17.5 Robustness

All of the line-fitting methods we have described involve squared error terms. This can lead to very poor fits in practice, because a single wildly inappropriate data point can give errors that are dominate those due to many good data points; these errors can result in a substantial bias in the fitting process (figure 17.11). It appears to be very 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.

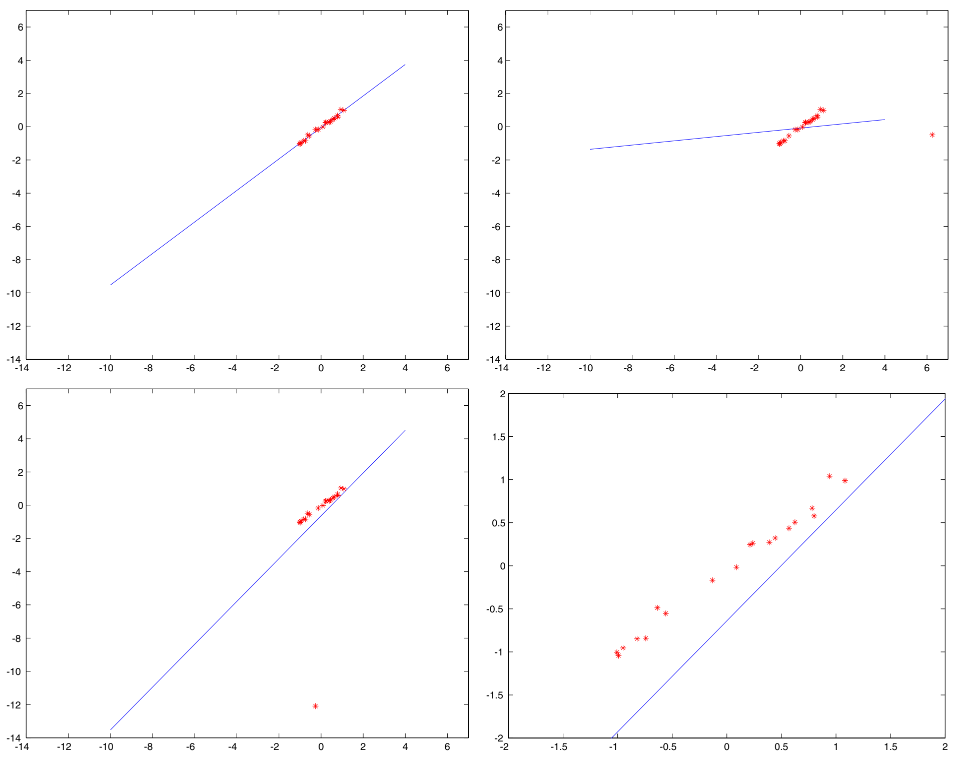


Figure 17.11. Least squares line fitting is extremely sensitive to outliers, both in x and y coordinates. At the top left, a good least-squares fit of a line to a set of points. Top-right shows the same set of points, but with the x-coordinate of one point corrupted. In this case, the slope of the fitted line has swung wildly. Bottom-left shows the same set of points, but with the y-coordinate of one point corrupted. In this particular case, the x-intercept has changed. These three figures are on the same set of axes for comparison, but this choice of axes does not clearly show how bad the fit is for the third case; Bottom-right shows a detail of this case — the line is clearly a very bad fit.

One approach to this problem puts the model at fault: the model predicts these outliers occuring perhaps once in the lifetime in the universe, and they clearly occur much more often than that. The natural response is to improve the model, either by giving the noise "heavier tails" (section 17.5.1) or by allowing an explicit outlier model. The second strategy requires a study of missing data problems — we don't know which point is an outlier and which isn't — and we defer discussion until section 18.2.4 in the following chapter. An alternative approach is to search for points that appear to be good (section 17.5.2).

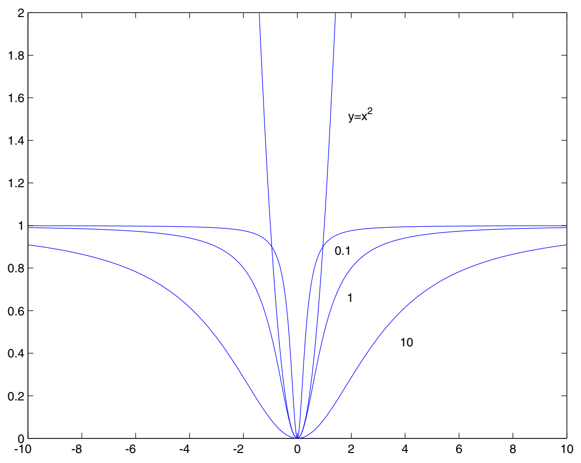


Figure 17.12. The function $\rho(x; \sigma) = x^2/(\sigma^2 + x^2)$, plotted for $\sigma^2 = 0.1$, 1 and 10, with a plot of $y = x^2$ for comparison. Replacing quadratic terms with ρ reduces the influence of outliers on a fit — a point which is several multiples of σ away from the fitted curve is going 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.

17.5.1 M-estimators

The difficulty with modelling the source of outliers is that the model might be wrong. Generally, the best we can hope for from a probabilistic model of a process is that it is quite close to the right model. Assume that we are guaranteed that our model of a process is close to the right model — say, the distance between the density functions in some appropriate sense is less than ϵ. We can use this guarantee to reason about the design of estimation procedures for the parameters of the model. In particular we can choose an estimation procedure by assuming that nature is malicious and well-informed about statistics[1]. In this line of reasoning, we assess the goodness of an estimator by assuming that somewhere in the collection of processes close to our model is the real process, and it just happens to be the one that makes the estimator produce the worst possible estimates. The best estimator is the one that behaves best on the worst distribution close to the parametric model. This is a criterion which can be used to produce a wide variety of estimators.

An **M-estimator** estimates parameters by minimizing 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 and $r_i(\boldsymbol{x}_i, \theta)$ is the residual

[1]Generally, sound assumptions for any enterprise; the world is full of opportunities for painful and expensive lessons in practical statistics

error of the model on the i'th data point. Generally, $\rho(u;\sigma)$ looks like u^2 for part of its range, and then flattens out. A common choice is

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

The parameter σ controls the point at which the function flattens out; we have plotted a variety of examples in figure 17.12. 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 criterion is

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

For the kind of problems we consider, we would expect a good influence function to be antisymmetric — there is no difference between a slight over prediction and a slight under prediction — 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. Firstly, the extremisation problem is non-linear and must be solved iteratively. The standard difficulties apply: there may be more than one local minimum; the method may diverge; and the behaviour of the method is likely to be quite dependent on the start point. A common strategy for dealing with this problem is to draw a subsample of the data set, fit to that subsample using least squares, and use this as a start point for the fitting process. We do this for a large number of different subsamples, enough to ensure that there is a very high probability that in that set there is at least one that consists entirely of good data points.

Secondly, as figures 17.13 and 17.14 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

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

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

```
For $s = 1$ to $s = k$
  draw a subset of $r$ distinct points, chosen uniformly at random

  Fit to this set of points using maximum likelihood
  (usually least squares) to obtain $\theta_s^0$

  estimate $\sigma_s^0$ using $\theta_s^0$

  Until convergence (usually $|\theta_s^n - \theta_s^{n-1}|$ is small):
    take a minimising step using $\theta_s^{n-1}$, $\sigma_s^{n-1}$
    to get $\theta_s^n$
    now compute $\sigma_s^n$

report the best fit of this set, using the median of the
residuals as a criterion
```

Algorithm 17.3: *Using an M-estimator to fit a probabilistic model*

17.5.2 RANSAC

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

For example, assume that we are fitting a line to a data set that consists of about 50% outliers. If we draw pairs of points uniformly and at random, then about 1/4 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 will lie close to the line fitted to such a pair. Of course, a better estimate of the line could then be obtained by fitting a line to the points that lie close to our current line.

This approach leads to an algorithm — search for a random sample that leads to a fit on which many of the data points agree. The algorithm is usually called `RANSAC`, for RANdom SAmple Consensus, and is displayed in algorithm 4.

To make this algorithm practical, we need to be able to choose three parameters.

How Many Samples are Necessary?

Our samples will consist of sets of points drawn uniformly and at random from the data set. Each sample will contain the minimum number of points required to fit the abstraction we wish to fit; for example, if we wish to fit lines, we will draw pairs of points; if we wish to fit circles, we will draw triples of points, etc. We assume that

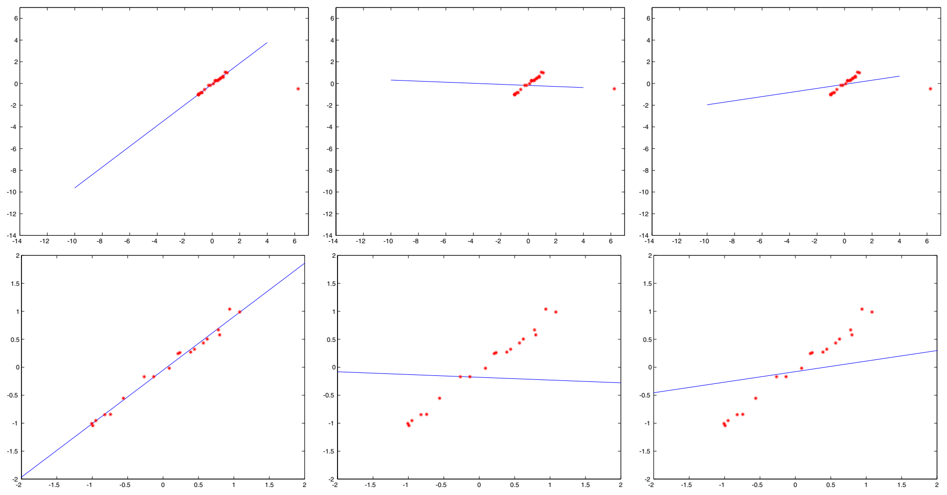


Figure 17.13. The top row shows lines fitted to the second dataset of figure 17.11 using a weighting function that de-emphasizes the contribution of distant points (the function ϕ of figure 17.12). On the left, μ has about the right value; the contribution of the outlier has been down-weighted, and the fit is good. In the center, the value of μ is too small, so that the fit is insensitive to the position of all the data points, meaning that its relationship to the data is obscure. On the right, the value of μ is too large, meaning that the outlier makes about the same contribution that it does in least-squares. The bottom row shows close-ups of the fitted line and the non-outlying data points, for the same cases.

we need to draw n data points, and that w is the fraction of these points that are good (we will need only a reasonable estimate of this number). Now the expected value of the number of draws k required to get one point is given by

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

(where the last step takes a little manipulation of algebraic series). Now we would like to be fairly confident that we have seen a good sample, so we would wish to draw rather more than w^{-n} samples; a natural thing to do is to add a few standard deviations to this number (see section 7.3.2 for an inequality that suggests why this is the case). 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 choose to look at a number of samples that guarantees a low probability z of seeing only bad samples. In this case, we

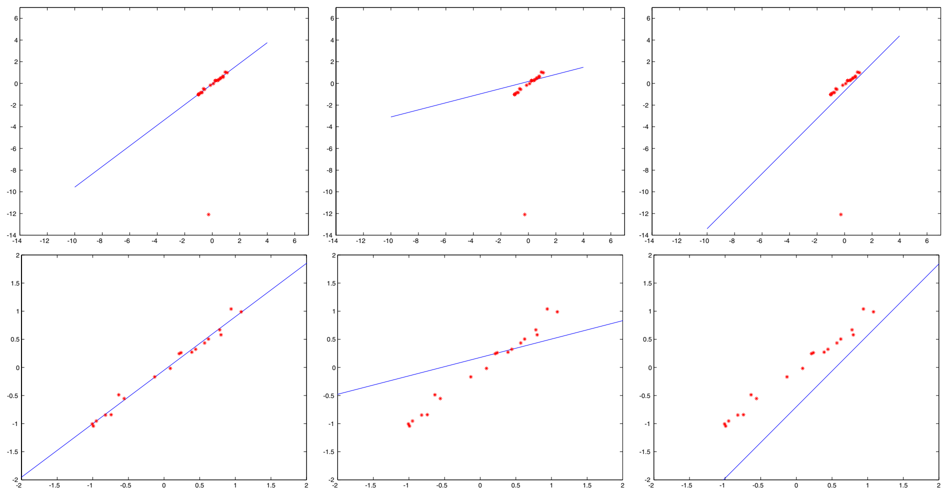


Figure 17.14. The top row shows lines fitted to the third dataset of figure 17.11 using a weighting function that de-emphasizes the contribution of distant points (the function ϕ of figure 17.12). On the left, μ has about the right value; the contribution of the outlier has been down-weighted, and the fit is good. In the center, the value of μ is too small, so that the fit is insensitive to the position of all the data points, meaning that its relationship to the data is obscure. On the right, the value of μ is too large, meaning that the outlier makes about the same contribution that it does in least-squares. The bottom row shows close-ups of the fitted line and the non-outlying data points, for the same cases.

have

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

which means that

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

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

How Far Away is Agreement?

We need to determine whether a point lies close to a line fitted to a sample. We will do this by determining the distance between the point and the fitted line, and testing

```
Determine:
    $n$ --- the smallest number of points required
    $k$ --- 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 there is a $k$ 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 line
      against $t$; if the distance from the point to the line
      is less than $t$, the point is close

    end
    if there are $d$ or more points close to the line
    then there is a good fit.  Refit the line using all
    these points.
end
Use the best fit from this collection, using the
fitting error as a criterion
```

Algorithm 17.4: *RANSAC: fitting lines using random sample consensus*

that distance against a threshold d; if the distance is below the threshold, then the point lies close. In general, specifying this parameter is part of the modelling process. For example, when we fitted lines using maximum likelihood, there was a term σ in the model (which disappeared in the manipulations to find an maximum). This term gives the average size of deviations from the model being fitted.

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

How Many Points Need to Agree?

Assume that we have fitted a line to some random sample of two data points. We need to know whether that line is good or not. We will do this by counting the number of points that lie within some distance of the line (the distance was determined in the previous section). In particular, assume that we know the probability that an outlier lies in this collection of points; write this probability as y. We should like to choose some number of points t such that y^t is small (say, less than 0.05).

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

17.6 Example: Using RANSAC to Fit Fundamental Matrices

A point in 3D generates two measurements, one in the left view and one in the right. We write the actual coordinates of the 3D point as $\boldsymbol{X}_i$, the coordinates in the left (respectively, right) image as $\boldsymbol{x}_{li}$ (resp. $\boldsymbol{x}_{ri}$), the measured coordinates in the left (resp., right) image as $\boldsymbol{m}_{ri}$ (resp. $\boldsymbol{m}_{ri}$). The fundamental matrix is an expression of the epipolar constraint. In particular, using the hat to indicate that we are employing homogenous coordinates, we have that $\hat{\boldsymbol{x}}_{ri}^T \mathcal{F} \hat{\boldsymbol{x}}_{li} = 0$ for every point. Here $\mathcal{F}$ is the fundamental matrix.

17.6.1 An Expression for Fitting Error

Notice that, if we avoid homogenous coordinates and write $\boldsymbol{x}_{ri} = (x_{ri}, y_{ri})^T$, $\boldsymbol{x}_{li} = (x_{li}, y_{li})^T$, the i, j'th element of $\mathcal{F}$ as f_{ij} and the expression

$$(f_{10}x_{ri} + f_{11}y_{ri} + f_{12})$$

as d_{ri} we can expand the equation to get

$$y_{li} = \frac{1}{d_{ri}}((f_{00}x_{li}x_{ri} + f_{01}x_{li}y_{ri} + f_{02}x_{li} + f_{20}x_{ri} + f_{21}y_{ri} + f_{22})$$

We make this substitution, and now write

$$\begin{aligned}
\boldsymbol{x}_{ri} &= (x_{ri}, y_{ri}) \\
\boldsymbol{x}_{li} &= (x_{li}, \frac{1}{d_{ri}}((f_{00}x_{li}x_{ri} + f_{01}x_{li}y_{ri} + f_{02}x_{li} + f_{20}x_{ri} + f_{21}y_{ri} + f_{22}))
\end{aligned}$$

This may not be the best set of coordinates in which to write the problem — if we're unlucky, the denominator of the fraction may go to zero, which will create difficulties. We ignore this issue as being secondary, and proceed.

The constraint will not generally hold for the measured values. We assume that measurements are subject to additive Gaussian noise of uniform rotationally symmetric covariance. We write $\boldsymbol{m}_{li}$ and $\boldsymbol{m}_{ri}$ in affine (conventional, or non-homogenous!) coordinates, too. This means that

$$P(\boldsymbol{m}_{li}, \boldsymbol{m}_{ri} | \boldsymbol{x}_{ri}, \boldsymbol{x}_{li}, \mathcal{F}) \propto \exp -\frac{1}{2\sigma^2} \left\{ \begin{array}{c} (\boldsymbol{x}_{ri} - \boldsymbol{m}_{ri})^T (\boldsymbol{x}_{ri} - \boldsymbol{m}_{ri}) + \\ (\boldsymbol{x}_{li} - \boldsymbol{m}_{li})^T (\boldsymbol{x}_{li} - \boldsymbol{m}_{li}) \end{array} \right\}$$

Now this is a complicated function of the data, with parameters $\mathcal{F}$, x_{ri}, y_{ri} and x_{li}. However, with sufficient data points we could, in principle, obtain an extremum. We can't do this currently, because we don't know which points correspond between left and right images. Furthermore, we should suspect that the sum-of-squares form of the log-likelihood will lead to robustness problems.

17.6.2 Correspondence as Noise

One way to deal with the correspondence problem is to assume that there is only a small camera motion between the two views. In turn, we can assert that feature points whose position in the second view is "close" to their position in the first view, correspond. This is a fairly dangerous assumption; it can lead to fits that are very bad, and look quite good. The difficulty is that *a correspondence error behaves like an outlier*. An alternative strategy is to search the correspondences for a set that is consistent with a good fundamental matrix.

This search can be simplified if we attach some representation of the local image neighbourhood to each point.This means that we can associate a neighbourhood in the right image with each point in the left image — only the points inside this neighbourhood could correspond to the relevant points in the left image. We can do this the other way, too (points in the right image, neighbourhoods in the left) and come up with a set of possible correspondences. We now apply RANSAC to this set of possible correspondences.

17.6.3 Applying RANSAC

As we shall see below, seven point correspondences yield a fundamental matrix. With this information, applying RANSAC to the set of possible correspondences is relatively straighforward. While we may not know what percentage of correspondences is good, it is possible to estimate this using fitting attempts (as above). The distance threshold to determine whether a point is an inlier or not is usually of the order of a pixel or so (this must depend on the quality of the cameras, etc.).

Obtaining a Fundamental Matrix from Seven Points

Each constraint $\hat{\boldsymbol{x}}_{ri}^T \mathcal{F} \hat{\boldsymbol{x}}_{li} = 0$ yields a single linear equation in the coefficients of the fundamental matrix for *known* $\boldsymbol{x}_{ri}$, $\boldsymbol{x}_{li}$. Furthemore, the equation $\hat{\boldsymbol{x}}_{ri}^T \mathcal{F} \hat{\boldsymbol{x}}_{li} = 0$ is homogenous in the elements of the fundamental matrix — that is, if $\mathcal{U}$ satisfies these constraints, then so does $\lambda \mathcal{U}$. Finally, recall from chapter 12 that the fundamental matrix has rank two, and so $\det(\mathcal{F}) = 0$.

This means that we need only seven point correspondences to estimate $\mathcal{F}$. Each point yields a single homogenous equation in the elements of $\mathcal{F}$. The solution of seven of these homogenous equations is a two dimensional linear space, which we can write as $\lambda\mathcal{F}_0 + \mu\mathcal{F}_1$, for known $\mathcal{F}_0$ and $\mathcal{F}_1$, and arbitrary λ, μ. But we need an element of this space with zero determinant; and the equation $\det(\lambda\mathcal{F}_0 + \mu\mathcal{F}_1) = 0$ is a homogenous cubic in λ, μ. We can divide both sides by μ, and solve for λ/μ. There is either one real root — and so only one solution — or three.

17.7 Discussion

We have covered a few important points from a very large body of technique here. Fitting is a problem that occurs in any number of contexts; it is almost always possible, and often helpful, to see a problem as a fitting problem. This means it is very difficult to supply a useful guide to the literature.

Usually the main difficulties one encounters in practice are: (1) determining distances (which can be very hard indeed); (2) ensuring that outliers do not overwhelm good data; and (3) deciding what to fit in the first place.

Approximations to the distance from a point to a curve or to a surface are discussed in numerous papers: we particularly recommend [Bookstein, 1979; Porrill, 1990; Cabrera and Meer, 1996; Agin, 1981; Sabin, 1994; Taubin *et al.*, 1994a; Sullivan *et al.*, 1994a; Sullivan and Ponce, 1998a].

Robustness hasn't had as much impact on practices within the vision community as it probably should have. Good starting points for reading include [Rousseeuw, 1987; Huber, 1981; Stewart, 1999; Meer *et al.*, 1991]. The ideas are genuinely useful, despite the subject's tendency to inspire zealotry. We haven't gone deeply into the topic, mainly because a superficial acquaintance with the topic is sufficient to deal with any issues likely to arise in practice. RANSAC is the method of choice for a number of problems and has had tremendous impact on the structure from motion community; we recommend reading [Fischler and Bolles, 1981; Torr and Murray, 1997; Hartley and Zisserman, 2000a]. We expect that in future it will be used to start EM-like methods; more on that later, once we have discussed EM.

We haven't really stressed fitting as inference. The advantage of thinking about fitting as inference is that anything one knows about fitting can be exchanged into knowledge about fitting, too, and the exchange rate seems to be favourable. The next chapter will show some of that, but you should also be aware that our comments of the probability chapter apply to pretty much everything in this chapter, too. In particular, one might use MAP inference — all this would require would be to attach a prior term to the fitting error. In years gone by, it was fashionable to construct a prior term that penalised models that wiggled too quickly (a phenomenon widely confused with a failure to be smooth). This practice seems to have diminished in the last few years; samples of this literature include [Horn and Schunck, 1981; Poggio *et al.*, 1985; Bertero *et al.*, 1988].

Fitting can also be used for image reconstruction. One fits a surface to image data — which is interpreted as a height map — tearing the surface at edges. It is then important to account correctly for the cost of tearing vs. the cost of a poor fit. This general strategy can also be applied to reconstruction of depth maps, stereo maps, etc. Various constraints are available. In [Grimson, 1981c], the absence of features is taken to imply smoothness ("No news is good news"). More recent work accounts for discontinuities in various forms, including tears and creases [Blake and Zisserman, 1987; Mumford and Shah, 1985; Mumford and Shah, 1988]

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 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 $\mathcal{T}^T\mathcal{M}_1\mathcal{T} = Id \textit{ and}$ $\mathcal{T}^T\mathcal{M}_2\mathcal{T}$ 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 viewing cone for a cone is a family of planes, all of which pass through the focal point and the vertex of the cone. Now show the outline of a cone consists of a set of lines passing through a vertex. You should be able to do this by a simple argument, without any need for calculations.

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).
- Implement a hough transform line finder.
- Count lines with an HT line finder - how well does it work?

Chapter 18

SEGMENTATION AND FITTING USING PROBABILISTIC METHODS

All the segmentation algorithms we described in the previous chapter involve essentially local models of similarity. Even though some algorithms attempt to build clusters that are good globally, the underlying *model* of similarity compares individual pixels. Furthermore, none of these algorithms involved an explicit probabilistic model of how measurements differed from the underlying abstraction that we are seeking.

We shall now look at explicitly probabilistic methods for segmentation. These methods attempt to explain data using models that are global. These models will attempt to explain a large collection of data with a small number of parameters. For example, we might take a set of tokens and fit a line to them; or take a pair of images and attempt to fit a parametric set of motion vectors that explain how pixels move from one to the other.

The key concept is that of a missing data problem. We introduce this problem by revisiting line fitting, which we now see as a quite general example of a probabilistic fitting problem. In many segmentation problems, there are several possible sources of data (for example, a token might come from a line, or from noise); if we knew from which source the data had come (i.e. whether it came from the line, or from noise), the segmentation problem would be easy. In section 18.1, we deal with a number of segmentation problems by phrasing them in this form, and then using a general algorithm for missing data problems. This leads us to situations where occasional data items are hugely misleading, and we discuss methods for making fitting algorithms robust (section 17.5). Finally, we discuss methods for determining how many elements (lines, curves, segments, etc.) to fit to a particular data set (section 18.3).

18.1 Missing Data Problems, Fitting and Segmentation

A number of important vision problems can be phrased as missing data problems. 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 very general view: segmenting an image into regions involves determining which source of colour and texture pixels generated the image pixels; segmenting a set of tokens into collinear groups involves determining which tokens lie on which line; segmenting a motion sequence into moving regions involves allocating moving pixels to motion models. There is a standard, quite simple, algorithm for this class of problem.

18.1.1 Missing Data Problems

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 other (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 very much simpler by rewriting it using some variables whose values are unknown. We will demonstrate this method and appropriate algorithms with two examples.

Example: Image Segmentation

At each pixel in an image, we compute a d-dimensional feature vector $\boldsymbol{x}$, which encapsulates position, colour and texture information. This feature vector could contain various colour representations, and the output of a series of filters centered at a particular pixel. Our image model is that each pixel is produced by a density associated with one of g image segments. Thus, to produce a pixel, we choose an image segment, and then generate the pixel from the density associated with that segment.

We assume that the l'th segment is chosen with probability π_l, and we model the density associated with the l'th segment as a Gaussian, with parameters $\theta_l = (\boldsymbol{\mu}_l, \Sigma_l)$ that depend on the particular segment. This means that we can write the probability of generating a pixel vector as

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

This form of model is known as a **mixture model** (because it is a weighted sum, or **mixture** of probability models; the π_l are usually called **mixing weights**). We shall encounter the form often.

One way to interpret such a mixture model is to think of it as a **generative model**. In this view, each pixel in the image is obtained by (a) selecting the l'th component of the model with probability π_l, and then (b) drawing a sample from $p(\boldsymbol{x}|\theta_l)$. One can visualize this model as a density in feature vector space that consists of a set of g "blobs", each of which is associated with an image segment. We should like to determine: (1) the parameters of each of these blobs; (2) the mixing weights and (3) from which component each pixel came (thereby segmenting the image).

We encapsulate these parameters into a parameter vector, writing the mixing weights as α_l and the parameters of each blob as $\theta_l = (\boldsymbol{\mu}_l, \Sigma_l)$, to get $\Theta = (\alpha_1, \ldots, \alpha_g, \theta_1, \ldots, \theta_g)$. The mixture model then has the form

$$p(\boldsymbol{x}|\Theta) = \sum_{l=1}^{g} \alpha_l p_l(\boldsymbol{x}|\theta_l)$$

Each component density is the usual Gaussian:

$$p_l(\boldsymbol{x}|\theta_l) = \frac{1}{(2\pi)^{d/2} \det(\Sigma_i)^{1/2}} \exp\left\{-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_i)^T \Sigma_i^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_i)\right\}$$

The likelihood function for an image is:

$$\prod_{j \in \text{observations}} \left(\sum_{l=1}^{g} \alpha_l p_l(\boldsymbol{x}_j|\theta_l) \right)$$

Each component is associated with a segment, and Θ is unknown.

The important point is this: if we knew the component from which each pixel came, then it would be simple to determine Θ. We could use maximum likelihood estimates for each θ_l, and then the fraction of the image in each component would give the α_l. Similarly, if we knew Θ, then for each pixel, we could determine the component that is most likely to have produced that pixel — this yields an image segmentation. The difficulty is that we know neither.

Example: Fitting Lines to Point Sets

There are g different lines in the plane. The l'th line is parametrised by $\boldsymbol{a}_l$ and generates tokens with probability π_l. Each token results in a measurement vector $\boldsymbol{W}$, and the value of the j'th measurement is $\boldsymbol{W}_j$. For the l'th line, there is a probability density function describing how it emits tokens, which we write as $p(\boldsymbol{W}|\boldsymbol{a}_l)$. This means that the probability density function for a set of measurements of a token is

$$p(\boldsymbol{W}) = \sum_{l} \pi_l p(\boldsymbol{W}|\boldsymbol{a}_l)$$

This is another mixture model. Under this model, the likelihood of a set of observations is:

$$\prod_{j \in \text{observations}} \left(\sum_{l=1}^{g} \pi_l p(\boldsymbol{W}_j|\boldsymbol{a}_l) \right)$$

We would like to infer $\boldsymbol{a}_l$ and π_l. As in the case of segmentation, if we knew which point was generated by which line, the problem would be easy. We could estimate $\boldsymbol{a}_l$ by line fitting and obtain π_l by counting the number of tokens generated by the l'th line then dividing by the total number of lines. The difficulty is that we have only the measurements of the tokens, not the association between tokens and lines.

Strategy

For each of these examples, if we know the missing data then we can estimate the parameters effectively. Similarly, if we know the parameters, the missing data will follow. This suggests an iterative algorithm:

1. Obtain some estimate of the missing data, using a guess at the parameters;
2. now form a maximum likelihood estimate of the free parameters using the estimate of the missing data.

and we would iterate this procedure until (hopefully!) it converged. For image segmentation, this would look like:

1. Obtain some estimate of the component from which each pixel's feature vector came, using an estimate of the θ_l.
2. Now update the θ_l, using this estimate.

In the case of the tokens and the lines, this would look like:

1. Obtain some estimate of the correspondence between tokens and lines, using a guess at $\boldsymbol{a}_l$;
2. now form a revised estimate of $\boldsymbol{a}_l$ using the estimated correspondence.

18.1.2 The EM Algorithm

Although it would be nice if the procedures given above 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 shown by showing that they are examples of a general algorithm, the **expectation-maximization** algorithm.

Assume we have two spaces, the **complete data space** $\mathcal{X}$ and the **incomplete data space** $\mathcal{Y}$. There is a map f, which takes $\mathcal{X}$ to $\mathcal{Y}$. This map "loses" the missing data; for example, it could be a projection. For the example of image segmentation, the complete data consists of the measurements at each pixel *and* a set of variables indicating from which component of the mixture the measurements came; the incomplete data is obtained by dropping this second set of variables. For the example of lines and tokens, the complete data space consists of the measurements of the tokens (position, certainly, but colour and shape could come into this) *and* a set of variables indicating from which line the token came; the incomplete data is obtained by dropping this second set of variables.

There is a parameter space $\mathcal{U}$. For the image segmentation example, the parameter space consists of the mixing weights and of the parameters of each mixture component; for the lines and tokens, the parameter space consists of the mixing weights and the parameters of each line. We wish to obtain a maximum likelihood estimate for these parameters, given only incomplete data. If we had complete data, we could use the probability density function for the complete data space, written $p_c(\boldsymbol{x};\boldsymbol{u})$. The complete data log-likelihood is

$$\begin{aligned} L_c(\boldsymbol{x};\boldsymbol{u}) &= \log\{\prod_j p_c(\boldsymbol{x}_j;\boldsymbol{u})\} \\ &= \sum_j \log\left(p_c(\boldsymbol{x}_j;\boldsymbol{u})\right) \end{aligned}$$

In either of our examples, this log-likelihood would be relatively easy to work with. In the case of image segmentation, the problem would be to estimate the parameters for each image segment, given the segment from which each pixel came.In the case of the lines and tokens, the problem would be to estimate the mixing weights and parameters, given the line from which each token came.

The problem is that we don't have the complete data. The probability density function for the *incomplete* data space is $p_i(\boldsymbol{y};\boldsymbol{u})$. Now the probability density function for the incomplete data space is obtained by integrating the probability density function for the complete data space over all values that give the same $\boldsymbol{y}$. That is

$$p_i(\boldsymbol{y};\boldsymbol{u}) = \int_{\{\boldsymbol{x}|f(\boldsymbol{x})=\boldsymbol{y}\}} p_c(\boldsymbol{x};\boldsymbol{u})d\eta$$

(where η measures volume on the space of $\boldsymbol{x}$ such that $f(\boldsymbol{x})=\boldsymbol{y}$). The incomplete data likelihood is

$$\prod_{j\in\text{observations}} p_i(\boldsymbol{y}_j;\boldsymbol{u})$$

We could form a maximum likelihood estimate for $\boldsymbol{u}$, given $\boldsymbol{y}$ by writing out the likelihood and maximising it. This isn't easy, because both the integral and the maximisation can be quite difficult to do. The usual strategy of taking logs doesn't make things easier, because of the integral *inside* the log. We have:

$$\begin{aligned} L_i(\boldsymbol{y};\boldsymbol{u}) &= \log\left\{\prod_j p_i(\boldsymbol{y}_j;\boldsymbol{u})\right\} \\ &= \sum_j \log\left(p_i(\boldsymbol{y}_j;\boldsymbol{u})\right) \\ &= \sum_j \log\left(\int_{\{\boldsymbol{x}|f(\boldsymbol{x})=\boldsymbol{y}_j\}} p_c(\boldsymbol{x};\boldsymbol{u})d\eta\right) \end{aligned}$$

This form of expression is difficult to deal with. The reason that we are stuck with the incomplete data likelihood is that we don't know which of the many possible $\boldsymbol{x}$'s that could correspond to the $\boldsymbol{y}$'s that we observe actually does correspond. Forming the incomplete data likelihood involves averaging over all such $\boldsymbol{x}$'s.

The key idea in E-M is to obtain a working values for the missing data (and so for $\boldsymbol{x}$) by substituting an expectation for each missing value. In particular, we will fix the parameters at some value, and then compute the expected value of $\boldsymbol{x}$, given the value of $\boldsymbol{y}$ and the parameter values. We then plug the expected value of $\boldsymbol{x}$ into the complete data log-likelihood, which is much easier to work with, and obtain a value of the parameters by maximising that. Now at this point, the expected value of $\boldsymbol{x}$ may have changed. We obtain an algorithm by alternating the expectation step with the maximisation step, and iterate until convergence.

More formally, given $\boldsymbol{u}^s$, we form $\boldsymbol{u}^{s+1}$ by:

1. Computing an expected value for the *complete* data using the incomplete data and the current value of the parameters. This estimate is given by:

$$\overline{\boldsymbol{x}}_j^s = \int_{\{\boldsymbol{x}|f(\boldsymbol{x})=\boldsymbol{y}_j\}} \boldsymbol{x} p_c(\boldsymbol{x}; \boldsymbol{u}^s) d\eta$$

   This is referred to as **the E-step**.

2. Maximizing the *complete* data log likelihood with respect to $\boldsymbol{u}$, using the expected value of the complete data computed in the E-step. That is, we compute

$$\boldsymbol{u}^{s+1} = \arg\max_{\boldsymbol{u}} L_c(\overline{\boldsymbol{x}}^s; \boldsymbol{u})$$

   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. []). Of course, there is no guarantee that this algorithm converges to the *right* local maximum, and in some of the examples below we will show that finding the right local maximum can be a nuisance.

18.2 The EM Algorithm in Practice

Missing data problems turn up all over computer vision. We have collected a variety of examples here to illustrate the general story. The calculations are usually straightforward; once we have shown a few, we will pass over the rest in silence.

18.2.1 Example: Image Segmentation, Revisited

Assume there are a total of n pixels. The missing data forms an n by g array of indicator variables $\mathcal{I}$. In each row there is a single one, and all other values are zero — this indicates the blob from which each pixel's feature vector came.

The E-step: The l, m'th element of $\mathcal{I}$ is one if the l'th pixel comes from the m'th blob, and zero otherwise. This means that

$$\begin{aligned} \mathrm{E}(I_{lm}) &= 1P(l\text{'th pixel comes from the } m\text{'th blob}) + \\ &\quad 0.P(l\text{'th pixel does not come from the } m\text{'th blob}) \\ &= P(l\text{'th pixel comes from the } m\text{'th blob}) \end{aligned}$$

Assuming that the parameters are for the s'th iteration are $\Theta^{(s)}$, we have:

$$\overline{I}_{lm} = \frac{\alpha_m^{(s)} p_m(\boldsymbol{x}_l|\theta_l^{(s)})}{\sum_{k=1}^{K} \alpha_k^{(s)} p_k(\boldsymbol{x}_l|\theta_l^{(s)})}$$

(keeping in mind that $\alpha_m^{(s)}$ means the value of α_m on the s'th iteration!).

M-step: Once we have an expected value of $\mathcal{I}$, the rest is easy. We are essentially forming maximum likelihood estimates of Θ^{s+1}. Again, the expected value of the indicator variables is not in general going to be zero or one; instead, they will take some value in that range. This should be interpreted as an observation of that particular case that occurs with that frequency, meaning that the term in the likelihood corresponding to a particular indicator variable is raised to the power of the expected value. The calculation yields expressions for a weighted mean and weighted standard deviation that should be familiar:

$$\begin{aligned} \alpha_m^{(s+1)} &= \frac{1}{r}\sum_{l=1}^{r} p(m|\boldsymbol{x}_l, \Theta^{(s)}) \\ \boldsymbol{\mu}_m^{(s+1)} &= \frac{\sum_{l=1}^{r} \boldsymbol{x}_l p(m|\boldsymbol{x}_l, \Theta^{(s)})}{\sum_{l=1}^{r} p(m|\boldsymbol{x}_l, \Theta^{(s)})} \\ \Sigma_m^{s+1} &= \frac{\sum_{l=1}^{r} p(m|\boldsymbol{x}_l, \Theta^{(s)}) \left\{(\boldsymbol{x}_l - \boldsymbol{\mu}_m^{(s)})(\boldsymbol{x}_l - \boldsymbol{\mu}_m^{(s)})^T\right\}}{\sum_{l=1}^{r} p(m|\boldsymbol{x}_l, \Theta^{(s)})} \end{aligned}$$

(again, keeping in mind that $\alpha_m^{(s)}$ means the value of α_m on the s'th iteration!).

It remains to specify appropriate feature vectors, and discuss such matters as starting the EM algorithm. The results shown in figures 18.1-18.2 use three colour features — the coordinates of the pixel in L*a*b*, after the image has been smoothed — and three texture features — which use filter outputs to estimate local scale, anisotropy and contrast (figure 18.1); other features may well be more effective — and the position of the pixel.

What should the segmenter report? One option is to choose for each pixel the value of m for which $p(m|\boldsymbol{x}_l, \Theta^s)$ is a maximum. Another is to report these probabilities, and build an inference process on top of them.

18.2.2 Example: Line Fitting with EM

An EM line fitting algorithm follows the lines of the example above; the missing data is an array of indicator variables $\mathcal{M}$ whose k, l'th element m_{kl} is one if point

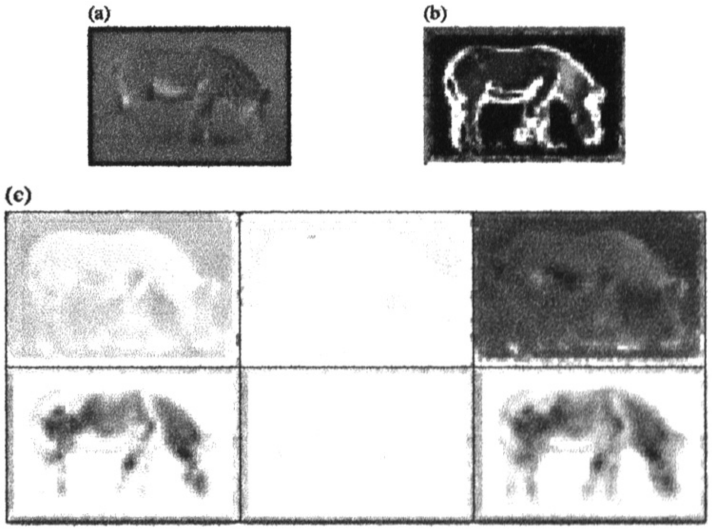


Figure 18.1. The image of the zebra in (a) gives local scale measurements shown in (b). These scale measurements essentially measure the scale of the change around a pixel; at edges, the scale is narrow, and in stripey regions it is broad, for example. The features that result are shown in (c); the top three images show the smoothed colour coordinates and the bottom three show the texture features (*ac*, *pc* and *c* — the scale and anisotropy features are weighted by contrast). *figure from Belongie et al, Color and Texture Based Image Segmentation Using EM and Its Application to Content Based Image Retrieval, ICCV98, p 5, in the fervent hope that permission will be granted*

k is drawn from line l, zero otherwise. As in that example, the expected value is given by determining $P(m_{kl} = 1|\text{point } k, \text{line } l\text{'s parameters})$, and this probability is proportional to

$$\exp\left(-\frac{\text{distance from point } k \text{ to line } l^2}{2\sigma^2}\right)$$

for σ as above. The constant of proportionality is most easily determined from the fact that

$$\sum_k P(m_{kl} = 1|\text{point } k, \text{line } l\text{'s parameters}) = \sum_l P(m_{kl} = 1|\text{point } k, \text{line } l\text{'s parameters}) = 1$$

The maximisation follows the form of that for fitting a single line to a set of points, only now it must be done g times and the point coordinates are weighted by the value of $\overline{l_{kl}}$. Convergence can be tested by looking at the size of the change in the lines, or by looking at the sum of perpendicular distances of points from their lines (which operates as a log likelihood, see question **??**).

```
Choose a number of segments

Construct a set of support maps, one per segment,
containing one element per pixel.  These support maps
will contain the weight associating a pixel with a segment

Initialize the support maps by either:

    Estimating segment parameters from small
    blocks of pixels, and then computing weights
    using the E-step;

    Or randomly allocating values to the support maps.

Until convergence

    Update the support maps with an E-Step

    Update the segment parameters with an M-Step

end
```

Algorithm 18.1: *Colour and texture segmentation with EM*

18.2.3 Example: Motion Segmentation and EM

For example, motion sequences quite often consist of large regions which have quite similar motion internally. Let us assume for the moment that we have a very short sequence — two frames — and wish to determine the motion field at each point on the first frame. We will assume that the motion field comes from a mixture model. Recall that a general mixture model is a weighted sum of densities — the components do not have to have the Gaussian form used in section 18.1.1 (missing data, the EM algorithm and general mixture models turn up rather naturally together in vision applications).

A generative model for a motion sequence would have the following form:

- At each pixel in each image, there is a motion vector connecting it to a pixel in the next image;
- there are a set of different parametric motion fields, each of which is given by a different probabilistic model;
- the overall motion is given by a mixture model, meaning that to determine

```
For each pixel location $l$

    For each segment $m$

      Insert $\alpha_m^{(s)} p_m(\boldsymbol{x}_l|\theta_l^{(s)})$
      in pixel location $l$ in the support map $m$

end
    Add the support map values to obtain
    $\sum_{k=1}^{K} \alpha_k^{(s)} p_k(\boldsymbol{x}_l|\theta_l^{(s)})$
    and divide the value in location $l$ in each support map by this term

end
```

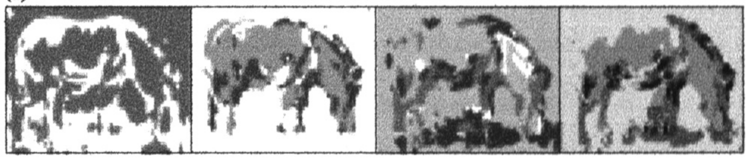
Algorithm 18.2: *Colour and texture segmentation with EM: - the E-step*

```
For each segment $m$

    Form new values of the segment parameters
    using the expressions:

$\alpha_m^{(s+1)} = \frac{1}{r} \sum_{l=1}^{r} p(m|\boldsymbol{x}_l, \Theta^{(s)})$

$\boldsymbol{\mu}_m^{(s+1)} = \frac{\sum_{l=1}^{r} \boldsymbol{x}_l p(m|\boldsymbol{x}_l, \Theta^{(s)})}{\sum_{l=1}^{r} p(m|\boldsymbol{x}_l, \Theta^{(s)})}$

$\Sigma_m^{s+1} = \frac{\sum_{l=1}^{r} p(m|\boldsymbol{x}_l, \Theta^{(s)}) \{(\boldsymbol{x}_l - \boldsymbol{\mu}_m^{(s)})(\boldsymbol{x}_l - \boldsymbol{\mu}_m^{(s)})^T\}}{\sum_{l=1}^{r} p(m|\boldsymbol{x}_l, \Theta^{(s)})}$

    Where $p(m|\boldsymbol{x}_l, \Theta_{(s)})$ is the value
    in the $m$'th support map for pixel location $l$

end
```

Algorithm 18.3: *Colour and texture segmentation with EM: - the M-step*

the image motion at a pixel, we firstly determine which component the motion comes from, and then secondly draw a sample from this component.

(d)



(e)

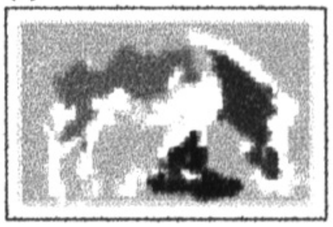


Figure 18.2. Each pixel of the zebra image of figure 18.1 is labelled with the value of m for which $p(m|\boldsymbol{x}_l, \Theta^s)$ is a maximum, to yield a segmentation. The images in (d) show the result of this process for $K = 2, 3, 4, 5$. Each image has K grey-level values corresponding to the segment indexes. *figure from Belongie et al, Color and Texture Based Image Segmentation Using EM and Its Application to Content Based Image Retrieval, ICCV98, p 5, in the fervent hope that permission will be granted*

This model encapsulates a a set of distinct, internally consistent motion fields — which might come from, say, a set of rigid objects at different depths and a moving camera (figure 18.5) — rather well. The separate motion fields are often referred to as **layers** and the model as a **layered motion model**.

Now assume that the motion fields have a parametric form, and that there are g different motion fields. Given a pair of images, we wish to determine (1) which motion field a pixel belongs to and (2) 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.

Assume that the pixel at (u, v) in the first image belongs to the l'th motion field, with parameters θ_l. This means that this pixel has moved to $(u, v) + \boldsymbol{m}(u, v; \theta_l)$ in the second frame, and so that the intensity at these two pixels is the same up to measurement noise. We will write $I_1(u, v)$ for the image intensity of the first image at the u, v'th pixel, and so on. The missing data is the motion field to which the pixel belongs. We can represent this by an indicator variable $V_{uv,l}$ where

$$V_{uv,l} = \left\{ \begin{array}{c} 1, \text{ if the } u, v\text{'th pixel belongs to the } l\text{'th motion field} \\ 0, \text{ otherwise} \end{array} \right\}$$

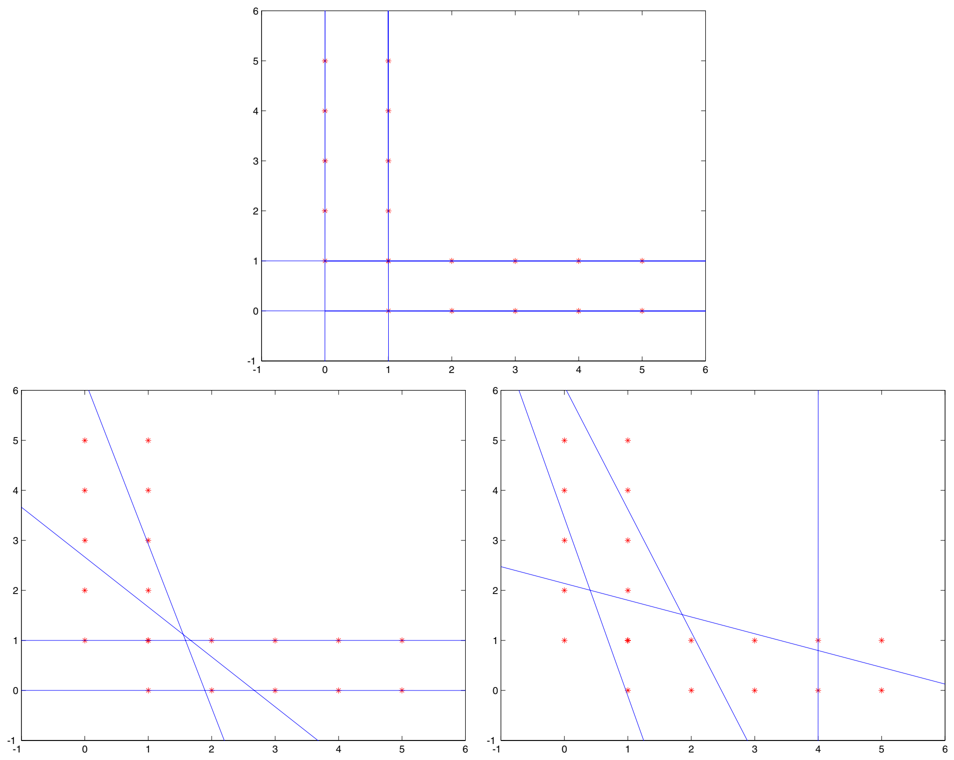


Figure 18.3. The top figure shows a good fit obtained using EM line fitting. The two bad examples in the bottom row were run with the right number of lines, but have converged to poor fits — which can be fairly good interpretations of the data, and are definitely local minima. This implementation adds a term to the mixture model that models the data point as arising uniformly and at random on the domain; a point that has a high probability of coming from this component has been identified as noise. Further examples of poor fits appear in figure 18.4.

We assume Gaussian noise with standard deviation σ in the image intensity values, so the complete data log-likelihood is

$$L(V,\Theta) = -\sum_{ij,l} V_{uv,l} \frac{(I_1(u,v) - I_2(u + m_1(u,v;\theta_l), v + m_2(u,v;\theta_l)))^2}{2\sigma^2} + C$$

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

$$P\left\{V_{uv,l} = 1 | I_1, I_2, \Theta\right\}$$

These probabilities are often represented as **support maps**, maps assigning a grey-level representing the maximum probability layer to each pixel (figure 18.6). The

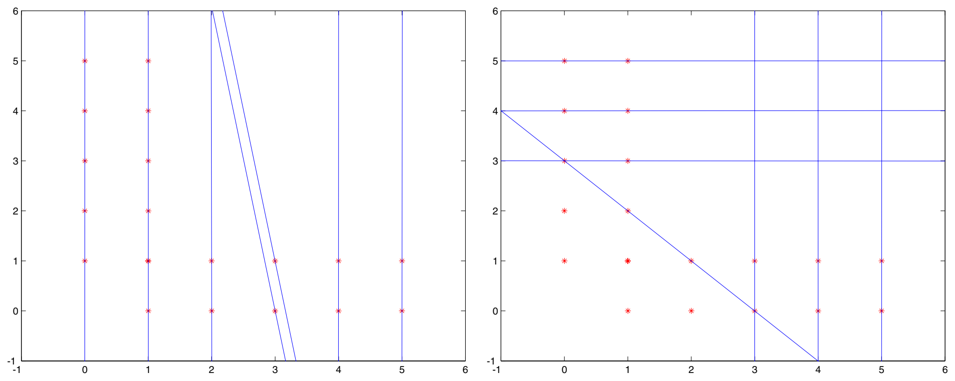


Figure 18.4. More poor fits to the data shown in figure 18.3; for these examples, we have tried to fit seven lines to this data set. Notice that these fits are fairly good interpretations of the data; they are local extrema of the likelihood. This implementation adds a term to the mixture model that models the data point as arising uniformly and at random on the domain; a point that has a high probability of coming from this component has been identified as noise. The fit on the bottom left has allocated some points to noise, and fits the others very well.

```
Choose $k$ lines (perhaps uniformly at random)
*or* choose $\overline{\mathcal{L}}$
Until convergence
  e-step:
    recompute $\overline{\mathcal{L}}$, from perpendicular distances
  m-step:
    refit lines using weights in $\overline{\mathcal{L}}$
```

Algorithm 18.4: *EM line fitting by weighting the allocation of points to each line, with the closest line getting the highest weight*

more interesting question is the appropriate choice of parametric motion model. A common choice is an **affine motion model**, where

$$\left\{ \begin{array}{c} m_1 \\ m_2 \end{array} \right\} (i, j; \theta_l) = \left\{ \begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right\} \left\{ \begin{array}{c} i \\ j \end{array} \right\} + \left\{ \begin{array}{c} a_{13} \\ a_{23} \end{array} \right\}$$

and $\theta_l = (a_{11}, \ldots, a_{23})$. Layered motion representations are useful for several reasons: firstly, they cluster together points moving "in the same way"; secondly, they expose motion boundaries; finally, new sequences can be reconstructed from the layers in interesting ways (figure 18.7).

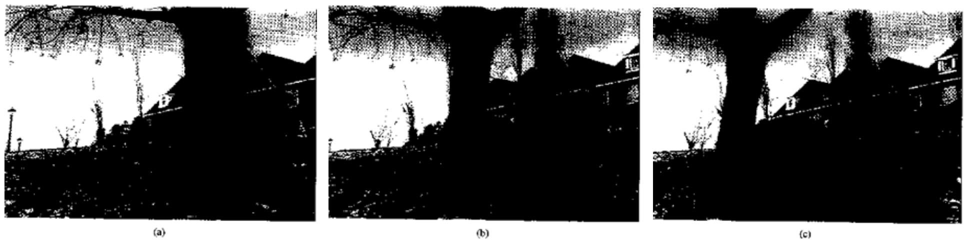


Figure 18.5. Frames 1, 15 and 30 of the MPEG flower garden sequence, which is often used to demonstrate motion segmentation algorithms. This sequence appears to be taken from a translating camera, with the tree very much closer to the camera than the house, and a flower garden on the ground plane. As a result, the tree appears to be translating quickly across the frame, and the house slowly; the plane generates an affine motion field.

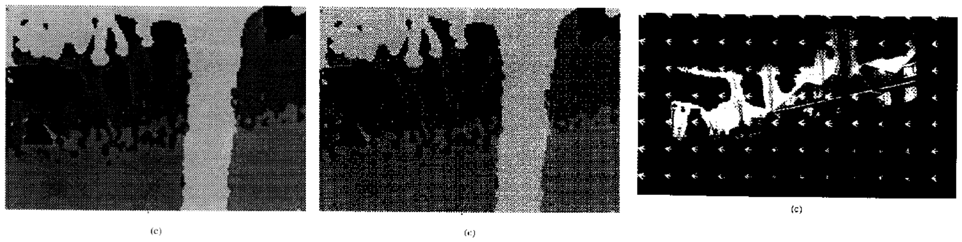


Figure 18.6. On the **left**, a map indicating to which layer pixels in a frame of the flower garden sequence belong, obtained by clustering local estimates of image motion. Each grey level corresponds to a layer, and each layer is moving with a different affine motion model. This map can be refined by checking the extent to which the motion of pixel neighbourhoods is consistent with neighbourhoods in future and past frames, resulting in the map on the **center**. One of the layers, and its motion model, is shown on the **right**.

18.2.4 Example: Using EM to Identify Outliers

The line fitters we have described have difficulty with outliers because they encounter outliers with a frequency that is wildly underpredicted by the model. Outliers are often referred to as being "in the **tails**" of a probability distribution. In probability distributions like the normal distribution, there is a large collection of values with very small probability; these values are the tails of the distribution (probably because these values are where the distribution *tails off*). A natural mechanism for dealing with outliers is to modify the model so that the distribution has **heavier tails** (i.e. that there is more probability in the tails).

One way to do this is to construct an explicit model of outliers, which is usually quite easy to do. We form a weighted sum of the likelihood $P(\text{measurements}|\text{model})$

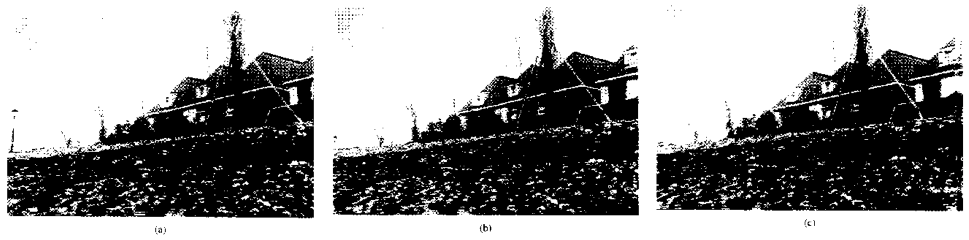


Figure 18.7. One feature of representing motion in terms of layers is that one can reconstruct a motion sequence *without* some of the layers. In this example, the MPEG garden sequence has been reconstructed with the tree layer omitted. The figure on the **left** shows frame 1; that in the **center** shows frame 15; and that on the **right** shows frame 30.

and a term for outliers $P(\text{outliers})$, to obtain:

$$(1-\lambda)P(\text{measurements}|\text{model}) + \lambda P(\text{outliers})$$

here $\lambda \in [0,1]$ models the frequency with which outliers occur, and $P(\text{outliers})$ is some probability model for outliers; failing anything better, it could be uniform over the possible range of the data.

The natural way to deal with this model is to construct a variable that indicates which component generated each point. With this variable, we have a complete data likelihood function with an easy form. Of course, we don't *know* this variable, but this is a missing data problem, and we know how to proceed here using EM (you provide the details in the exercises!). The usual difficulties with EM occur here, too. In particular, it is easy to get trapped in local minima, and we may need to be careful about the numerical representation adopted for very small probabilities.

18.2.5 Example: Background Subtraction using EM

As we saw in section 16.3.1, estimating the background can be difficult. Simply averaging over frames has the difficulty that an object that spends a lot of time in one place can bias the average quite seriously. We could see this as a missing variable problem: the image in each frame of video is the same (multiplied by some constant, to take account of various adjustments made by automatic gain control), with noise added. We model the noise as coming from some uniform source. This has the added benefit that every pixel that belongs to noise is not background; we can obtain the "subtraction" by simply looking at the expected values of the missing variables at the extremum. The calculations are straightforward; (d) and (e) in figure 16.10 and 16.11 are obtained in this fashion.

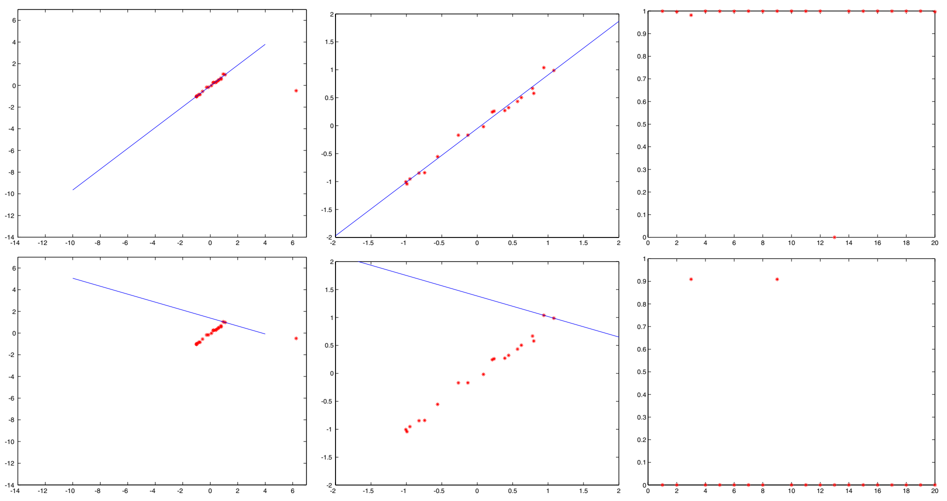


Figure 18.8. EM can be used to reject outliers; here we demonstrate a line fit to the second data set of figure 17.11. The top row shows the correct local minimum, and the bottom row shows another local minimum. The first column shows the line superimposed on the data points using the same axes as figure 17.11; 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.

18.2.6 Example: Finding Body Segments with EM

When we found image segments, we were somewhat vague about a spatial model — we remarked that the position of a pixel could be one of the features encapsulated in the mixture model, but didn't discuss how this could be used. Assume that each pixel has a feature vector containing position information, colour information and texture information. If we allow an arbitrary covariance in the Gaussians that emit pixels, there could be strong correlations between, say, position and colour; furthermore, there can be a lot of parameters to estimate, too many to hope obtain a reasonable estimate. Typically, one deals with this problem by imposing some form of parametric structure on the covariance — for example, we might set the covariance between any colour term and any position term to be zero. Now we can use this strategy to make an attack on the problem of finding body segments: we impose a structure on the spatial terms as well. For example, we might insist that an image segment be substantially extended in some direction — that is, that the covariance of the position terms be an ellipse with a fixed aspect ratio. This is another example for which we shall omit calculations.

18.2.7 Example: EM and the Fundamental Matrix

Fitting the fundamental matrix can also be seen as a missing data problem — the missing data is now the correspondence information. Assuming we have n points in left and right images, we could set up an $n \times n$ matrix $\mathcal{C}$ to represent the correspondence. In particular, c_{ij} is one if the ith point in the left image corresponds to the j'th point in the right image and zero otherwise. The detailed form of the E- and the M-steps are tedious to write out —we leave them to the exercises, and only sketch them here. The matrix $\mathcal{C}$ represents our missing data, and if we knew this matrix, we could compute a complete data log-likelihood and solve for the parameters — which are $\mathcal{A}$, and x_{ri}, y_{ri} and x_{li} for each point — using an extremisation method. Similarly, given an estimate of the parameters, it is relatively straightforward to compute the expected value of $\mathcal{C}$ (each point in the left image predicts a point in the right image — the exponential of the distances between these points and their measurements yields the expected values). Again, we expect that the expected value of $\mathcal{C}$ is generally not an integer, and, as above, we interpret this weight as a frequency and raise the probability to the relevant power in computing the likelihood.

This is a method that is unlikely to work well unless started very carefully. The problem is simple: EM may be able to alleviate the effects of the combinatorial search component of missing variable problems, but cannot make them disappear. The space of correspondences represented by $\mathcal{C}$ is huge ($n!$), and a substantial fraction of this space contains few or no good correspondences. Because the algorithm looks for the best available fundamental matrix given a set of correspondences, a poor initialisation is likely to lead to serious trouble. This is because the original set of correspondences may be wrong, leading to a poor estimate of the fundamental matrix, leading to a prediction of more incorrect correspondences, etc. One possibility is to start EM with RANSAC. The advantage of this approach over conventional RANSAC is that, by using EM in the final phase, we can weight the contribution of conforming correspondences appropriately.

18.2.8 Difficulties with the EM Algorithm

EM is inclined to get stuck in local minima. These local minima are typically associated with combinatorial aspects of the problem being studied, as in the line fitting example or in the fundamental matrix examples. These difficulties follow from the assumption that the points are interchangeable (see figure 18.3 and figure 18.4). This can be dodged by noticing that the final configuration of either fitter is a deterministic function of its start point, and using carefully chosen start points. One strategy is to start in many different (randomly chosen) configurations, and sift through the results looking for the best fit. Another is to preprocess the data using something like a Hough transform to guess good initial line fits. Neither is guaranteed. A cleaner approach is to notice that we are seldom, if ever, faced with a cloud of indistinguishable points and required to infer some structure on that cloud; usually, this is the result of posing a problem poorly. If points are not indistinguishable and have some form of linking structure, then a good start point should be much easier to choose.

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

18.3 How Many are There?

At each stage of our discussion of missing variable problems, we have assumed that the number of components in the mixture model is known. This is generally not the case in practice. Finding the number of components is, in essence, a model selection problem — we will search through the collection of models (where different models have different numbers of components) to determine which fits the data best (recall section 21). Generally, the value of the negative log-likelihood is a poor guide to the number of components because, in general, a model with more parameters will fit a dataset better than a model with fewer parameters. This means that simply minimizing the negative log-likelihood as a function of the number of components will tend to lead to too many components. For example, we can fit a set of lines extremely accurately by passing a line through each pair of points — there may be a lot of lines, but the fitting error is zero. We resolve this difficulty by adding a term that *increases* with the number of components — this penalty compensates for the decrease in negative log-likelihood caused by the increasing number of parameters.

It is important to understand that there is no *canonical* model selection process. Instead, we can choose from a variety of techniques, each of which uses a different discount corresponding to a different extremality principle (and different approximations to these criteria!).

18.3.1 Basic Ideas

Model selection is a general problem in fitting parametric models. The problem can be set up as follows: there is a data set, which is a sample from a parametric model which is itself a member of a family of models. We wish to determine (1) which model the data set was drawn from and (2) what the parameters of that model were. A proper choice of the parameters will predict future samples from the model — a **test set** — *as well as* the data set (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 data set is likely to be biased, because the parameters chosen ensure that the model fits the *training set* — rather than the entire set of possible data — optimally. 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, and 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.

The correct penalty to use comes from the **deviance**, given by

> twice (log-likelihood of the best model minus log-likelihood of the current model).

(from Ripley, [Ripley, 1996], p. 348); the best model should be the true model. Ideally, the deviance would be zero; the argument above suggests that the deviance on a training set will be larger than the deviance on a test set. A natural penalty to use is the difference between these deviances averaged over both test and training sets. This penalty is applied to twice the log-likelihood of the fit — the factor of two appears for reasons we cannot explain, but has no effect in practice. Let us write the best choice of parameters as Θ^* and the log-likelihood of the fit to the data set as $L(\boldsymbol{x};\Theta^*)$.

18.3.2 AIC — An Information Criterion

Akaike proposed a penalty, widely called **AIC**[1] which leads to minimizing

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

where p is the number of free parameters. 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 the deviance between a fitted model and the real model should go down as the number of data points goes up. Secondly, there is a body of experience that the AIC tends to **overfit** — that is, to choose a model with too many parameters which fits the training set well but doesn't perform as well on test sets.

18.3.3 Bayesian methods and Schwartz' BIC

We discussed Bayesian model selection in section 21, coming up with the expression:

$$\begin{aligned} P(\text{model}|\text{data}) &= \frac{P(\text{data}|\text{model})}{P(\text{data})} \\ &= \frac{\int P(\text{data}|\text{model},\text{parameters})P(\text{parameters})d\{\text{parameters}\}}{P(\text{data})} \\ &\propto \int P(\text{data}|\text{model},\text{parameters})P(\text{parameters})d\{\text{parameters}\} \end{aligned}$$

A number of possibilities now present themselves. We could give the posterior over the models, or choose the MAP model. The first difficulty is that we need to specify

[1]For "An information criterion," *not* "Akaike information criterion", []

a prior over the models. For example, in the case of EM based segmentation, we would need to specify a prior on the number of segments. The second difficulty is that we need to compute the integral over the parameters. This could be done using a sampling method ([Evans and Swartz, 2000]). An alternative is to construct some form of approximation to the integral, which yields another form of the penalty term.

For simplicity, let us write $\mathcal{D}$ for the data, $\mathcal{M}$ for the model, and θ for the parameters. The extent to which data support model i over model j is proportional to the **Bayes factor**

$$\frac{P(\mathcal{D}|\mathcal{M}_i)}{P(\mathcal{D}|\mathcal{M}_j)} = \frac{\int P(\mathcal{D}|\mathcal{M}_i,\theta)P(\theta)d\theta}{\int P(\mathcal{D}|\mathcal{M}_j,\theta)P(\theta)d\theta} = \frac{P(\mathcal{M}_i|\mathcal{D})}{P(\mathcal{M}_j|\mathcal{D})}\frac{P(\mathcal{M}_j)}{P(\mathcal{M}_i)} \propto \frac{P(\mathcal{M}_i|\mathcal{D})}{P(\mathcal{M}_j|\mathcal{D})}$$

Assume that $P(\mathcal{D},\theta|\mathcal{M})$ looks roughly normal — i.e. has a single mode, roughly elliptical level curves, and doesn't die off too fast. Now write the value of θ at the mode as θ^*. The **Hessian** at the mode is a matrix $\mathcal{H}$ whose i, j'th element is

$$\frac{\partial^2 \log p(\boldsymbol{x}_i;\Theta)}{\partial\theta_i\partial\theta_j}$$

evaluated at $\theta = \theta^*$.

If we take a Taylor series approximation to the log of $P(\mathcal{D},\theta|\mathcal{M})$ as a function of θ at the mode ($\theta = \theta^*$), we get

$$\begin{aligned}\log P(\mathcal{D},\theta|\mathcal{M}) &= L(\mathcal{D};\theta) + \log p(\theta) + \text{constant}\\ &= \Phi(\theta)\\ &= \Phi(\theta^*) + (\theta-\theta^*)^T\mathcal{H}(\theta-\theta^*) + O((\theta-\theta^*)^3)\\ &\approx \Phi(\theta^*) + (\theta-\theta^*)^T\mathcal{H}(\theta-\theta^*)\end{aligned}$$

(the linear term is missing because θ^* is the mode). Now

$$\begin{aligned}P(\mathcal{D}|\mathcal{M}) &= \int P(\mathcal{D},\theta|\mathcal{M})d\theta\\ &= \int \exp\Phi(\theta)d\theta\\ &\approx \exp\Phi(\theta^*)\int \exp(\theta-\theta^*)^T\mathcal{H}(\theta-\theta^*)d\theta\\ &= \{\exp\Phi(\theta^*)\}\,(2\pi)^{\frac{p}{2}}|\mathcal{H}^{-1}|^{\frac{1}{2}}\end{aligned}$$

All this implies that

$$\log p(\mathcal{D}|\mathcal{M}) \approx L(\mathcal{D};\theta^*) + \log p(\theta^*) + \frac{p}{2}\log(2\pi) + \frac{1}{2}\log|\mathcal{H}^{-1}|$$

Given a reasonable optimisation process and a bit of luck, we could find θ^* and $\mathcal{H}$, and evaluate this expression and so the Bayes factor. This analysis offers a criterion

$$-L(\mathcal{D};\theta^*) - \left\{\log p(\theta^*) + \frac{p}{2}\log(2\pi) + \frac{1}{2}\log|\mathcal{H}^{-1}|\right\}$$

(the signs are to allow comparison with the AIC in section 18.3.2). This might be difficult to evaluate. A series of approximations starting here leads to a criterion

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

called the **Bayes information criterion** or BIC.

18.3.4 Description Length

Models can be selected by criteria that are 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 data set most crisply. This **minimum description length** criterion chooses the model that allows the most efficient transmission of the data set. To transmit the data set, one codes and transmits the model parameters, and then codes and transmits the data given the model parameters. If the data fits the model very 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 [**?**]; similar ideas appear in [**?**; **?**]. Surprisingly, the BIC emerges from this analysis, yielding

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

18.3.5 Other Methods for Estimating Deviance

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, use one to fit the model and the other the test the fit. This is an approach known as **cross-validation**.

We can use cross-validation to determine the number of components in a model by splitting the data set, fitting a variety of different models to one side of the split, and then choosing the model that performs best on the other side. We expect this process to estimate the number of components, because a model that has too many parameters will fit the one data set well, but fit the other 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 to 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.

To our knowledge, this approach — which is standard for model selection in other kinds of problems — has not been used in fitting applications. It is certainly appropriate for estimating the number of components. First, assume that we compute the model error for a model with too few components to describe the image accurately. In this case, the model error will be large, because for many pixels the model will be insufficiently flexible to describe the pixel that was left out. Similarly, if we use too many components, the model will predict the left out pixel rather poorly.

18.4 Discussion

It should be obvious that we think missing variable models are important. EM was first formally described in the statistical literature by [Dempster *et al.*, 1977]. A very good summary reference is [McLachlan and Krishnan, 1996].

Missing variable models seem to crop up in all sorts of places. It is natural to use a missing variable model for segmentation [Belongie *et al.*, 1998a; Comer and Delp, 2000; Feng and Perona, 1998; Vasconcelos and Lippman, 1997; Adelson and Weiss, 1996; Wells *et al.*, 1995]. The model is in the process of reforming how we think about multiple images (i.e. both motion and stereo). The general idea is that the set is decomposed into different layers, where the elements of a layer share the same motion model [Dellaert *et al.*, 2000; Wang and Adelson, 1993; Wang and Adelson, 1994; Adelson and Weiss, 1996; Adelson and Weiss, 1995; Tao *et al.*, 2000; Weiss, 1997] or lie at the same depth [Brostow and Essa, 1999; Torr *et al.*, 1999a; Baker *et al.*, 1998], or have some other common property [Darrell and Pentland, 1995]. Other interesting cases include motions resulting from transparency, specularities, etc. [Darrell and Simoncelli, 1993; Black and Anandan, 1996; Jepson and Black, 1993; Hsu *et al.*, 1994; Szeliski *et al.*, 2000]. The resulting representation can be used for quite efficient image based rendering [Shade *et al.*, 1998]. This is a mixture model. While the problem isn't always seen as a hidden variable problem (the hidden variable is the layer to which a pixel belongs, or, equivalently, that generated it), it should probably be. We expect great things fairly shortly.

EM is an extremely successful inference algorithm, but it isn't magical. Problems with a nasty combinatorial structure are difficult, and remain difficult. Similarly, one can quite easily apply the formalism and end up with maximisation problems that are very difficult to do in practice.

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 [Torr, 1999; Torr, 1997; Kinoshita and Lindenbaum, 2000; 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.*, 1999b]. The standard problem in segmentation is how many segments are present [Raja *et al.*, 1998; Belongie *et al.*, 1998a; 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].

Assignments

Exercises

- Derive the expressions of section **??** for segmentation. One possible modification is to use the new mean in the estimate of the covariance matrices. Perform an experiment to determine whether this makes any difference in practice.
- Derive the expressions for EM for motion.
- Supply the details for the case of using EM for background subtraction. Would it help to have a more sophisticated foreground model than uniform random noise?
- Describe using leave-one-out cross-validation for selecting the number of segments

Programming Assignments

- Build an EM background subtraction program. Is it practical to insert a dither term to overcome the difficulty with high spatial frequencies illustrated in figure 16.11?
- Build an EM segmenter that uses colour, position (ideally, use texture too) to segment images; use a model selection term to determine how many segments there should be. How significant a phenomenon is the effect of local minima?
- Build an EM line fitter that works for a fixed number of lines. Investigate the effects of local minima. One way to avoid being distracted by local minima is to start from many different start points, and then look at the best fit obtained from that set. How successful is this? how many local minima do you have to search to obtain a good fit for a typical data set? Can you improve things using a Hough transform?
- Expand your EM line fitter to incorporate a model selection term, so that the fitter can determine how many lines fit a dataset. Compare the choice of AIC and BIC.
- Insert a noise term in your EM line fitter, so that it is able to perform robust fits. What is the effect on the number of local minima? Notice that, if there is a low probability of a point arising from noise, most points will be allocated to lines, but the fits will often be quite poor; if there is a high probability of a

point arising from noise, points will be allocated to lines only if they fit well. What is the effect of this parameter on the number of local minima?

- Construct a RANSAC fitter that can fit an arbitrary (but known) number of lines to a given data set. What is involved in extending your fitter to determine the best number of lines?