equations (10.9) forms a linear, overconstrained, homogeneous system in the entries of T that we write

$$A\mathbf{t} = 0, \tag{10.10}$$

with A the $2n \times 9$ matrix of coefficients and $\mathbf{t} = [T_{11}, T_{12}, \ldots, T_{33}]^\top$. The *compatibility* of the lines $\mathbf{l}_i$ and $\mathbf{L}_i$ can be checked by looking at the SVD of the matrix A, $A = UDV^\top$ (Appendix, section A.6). If the effective rank of A is 8 (that is, the least singular value of A is very small) all the lines are compatible and $\mathbf{t}$, as usual, is given by the column of V corresponding to the least singular value. Otherwise, the lines are not compatible.

In the assumption that the matrix T passed the compatibility test for lines, you are left with the problem of testing the compatibility for the conics c_i featuring in the definition of $\mathbf{g}_{f,1}, \ldots, \mathbf{g}_{f,H}$ (say $i = 1, \ldots, m$) with the corresponding model conics C_i. This can be done by looking at the ratio between each of the nine coefficients of c_i and $TC_iT^\top$. If the ratio k_i is approximately the same for all coefficients, the matrix T passed the global compatibility test and you may proceed to step 5. Otherwise, the model hypthesis O_f discarded.

In step 5, there are two points to make. First, how do we backproject lines and conics? For lines, we backproject each $\mathbf{l}_i$ by applying the transformation T to the model line $\mathbf{L}_i$ like in (10.9). For conics, we obtain the backprojection of each c_i by computing $TC_iT^\top$. Notice that the formal difference between this expression and the right hand side of (10.8) is due to the fact that T denotes a projective transformation of lines in the former and of points in the latter. In mathematical terms, T in the former is the inverse transpose of T in the latter. Second, "sufficiently close" (verification 2) means that the distance between backprojected features should never be larger than a few pixels; exact figures depend on resolution.

10.4 Appearance-Based Identification

Finally, we address Problem 2 of section 10.1: "Is this part of the image an instance of X?" And we address the question by *using images instead of features as basic components of object models*.

10.4.1 Images or Features?

The key idea behind appearance-based identification is simple: To store images of 3-D objects as their representation. Instead of representing object O through its geometric features and their spatial relations, *we represent O with the set of its possible appearances*; that is, the set of images taken, ideally, from all possible viewpoints and with all possible illumination directions.[4] In practice, we use a sufficiently large number of viewpoints and illuminations directions. As an example, Figure 10.5 shows a 12-image representation of a toy car. We can create a database of models for identification by building such a set

[4] We consider only direction for simplicity, but, as we know from Chapter 2, other illumination parameters play a role in determining pixel values.

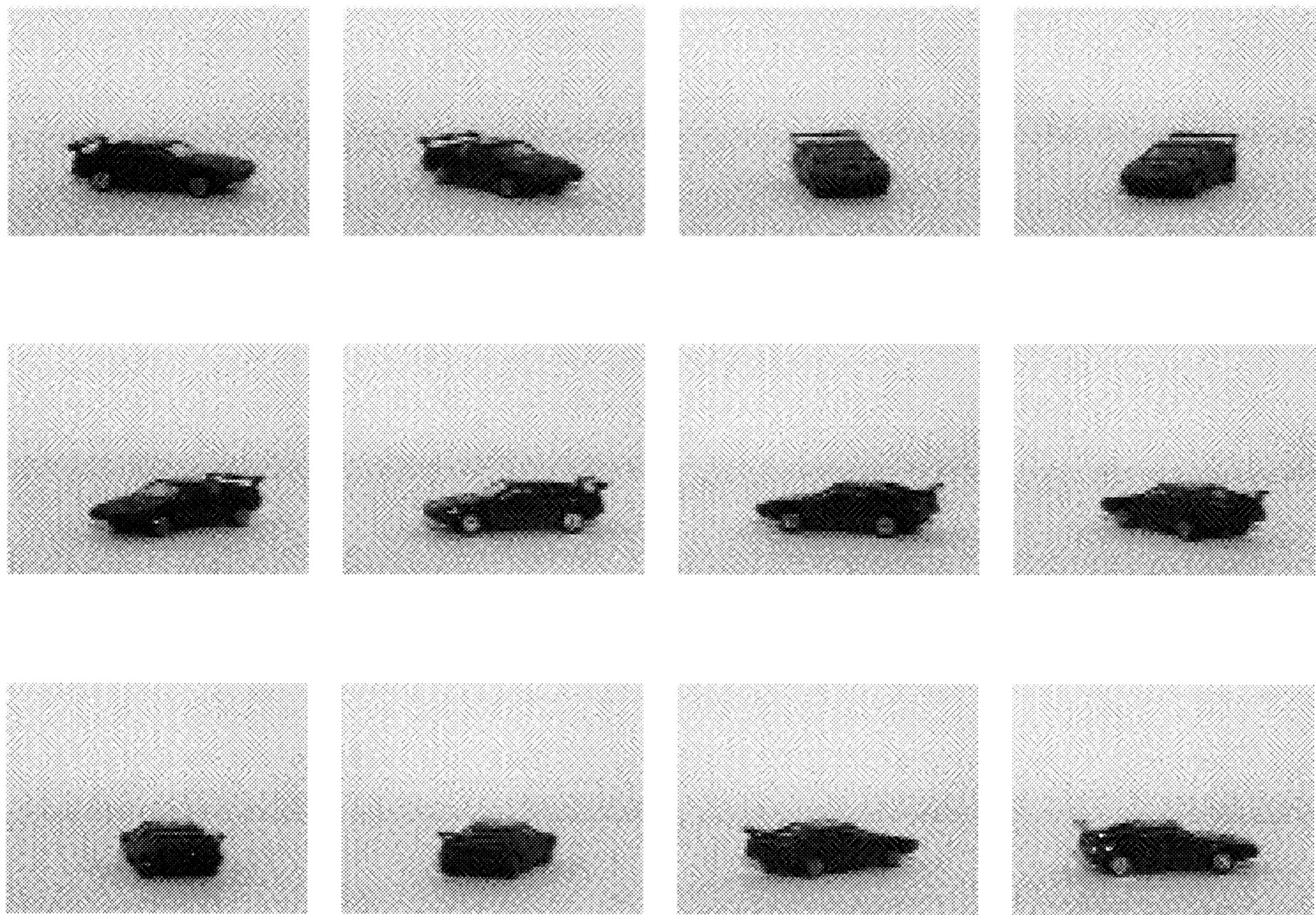


Figure 10.5 A simple database exemplifying appearance-based object representation. Only the viewpoint, not the illumination, was changed to obtain the views shown.

for all objects of interest. Identifying an object, then, means to find the set containing the image which is most similar to the one to be recognized.

Problem Statement

Given an image, I, containing an object to identify, and a database of object models, each one formed by a set of images showing the object under a large number of viewpoints and illumination conditions, find the set containing the image which is most similar to I.

A desirable characteristic of appearance-based identification, as presented, is that *object models can be compared directly with input data*, as both are images. Feature-based models (like the ones used with invariants and interpretation trees), instead, require that features be detected and described before data and model can be compared. Unfortunately there is a price to pay: the database may become extremely large even for limited numbers of objects and illumination conditions. For example, assuming 128×128 images, at one byte per pixel, 100 viewpoints per object, and 10 illumination directions, the representation of a single object would occupy about 64 megabytes of memory! Therefore the practical problem is, *can we devise a way to keep memory occupation within manageable limits while performing appearance-based recognition?*

10.4.2 Image Eigenspaces

We shall arrive at an appearance-based algorithm, the *parametric eigenspace* method in three steps:

1. We define a quantitative method to compare images and introduce some necessary assumptions.
2. We introduce an efficient, appearance-based object representation, which makes it feasible to search a large database of images.
3. We give algorithms to build the representation and to perform identification.

Comparing Images. A simple, quantitative way to compare two images, say I_1 and I_2, both $N \times N$ for simplicity, is to compute their *correlation*, c:

$$c = I_1 \circ I_2 = \frac{1}{K} \sum_{i=1}^{N} \sum_{j=1}^{N} I_1(i, j) I_2(i, j),$$

where K is a normalizing constant, and $\circ$ denotes image correlation. The larger c, the more similar I_1 and I_2.

This is simple enough, but we must take some precautions. As we are really interested in comparing 3-D objects, not just images, we need assumptions to guarantee that correlation is meaningful for our purposes.

Assumptions

1. Each image contains one object only.
2. The objects are imaged by a fixed camera under weak perspective.
3. The images are *normalized in size*; that is, the image frame is the minimum rectangle enclosing the largest appearance of the object.
4. The energy of the pixel values of each image is normalized to 1; that is, $\sum_{i=1}^{N} \sum_{j=1}^{N} I(i, j)^2 = 1$.
5. The object is completely visible and unoccluded in all images.

All these assumptions are consequences of the fact that we want to compare 3-D objects by comparing their images. You should be able to explain the reasons behind each assumption (see Review Questions for hints).

Efficient Image Comparison with Eigenspaces. Our next goal is to devise an efficient method to search a large image database in order to find the image most similar to a new one, in the correlation sense. The database contains the appearance-based representations of several objects; each object is represented by a set of images, taken by different viewpoints and with light coming from different directions. Such a database is suggested by Figure 10.6, which shows only one image per object for reasons of space. Clearly, if we store a full image for each view, and many views for

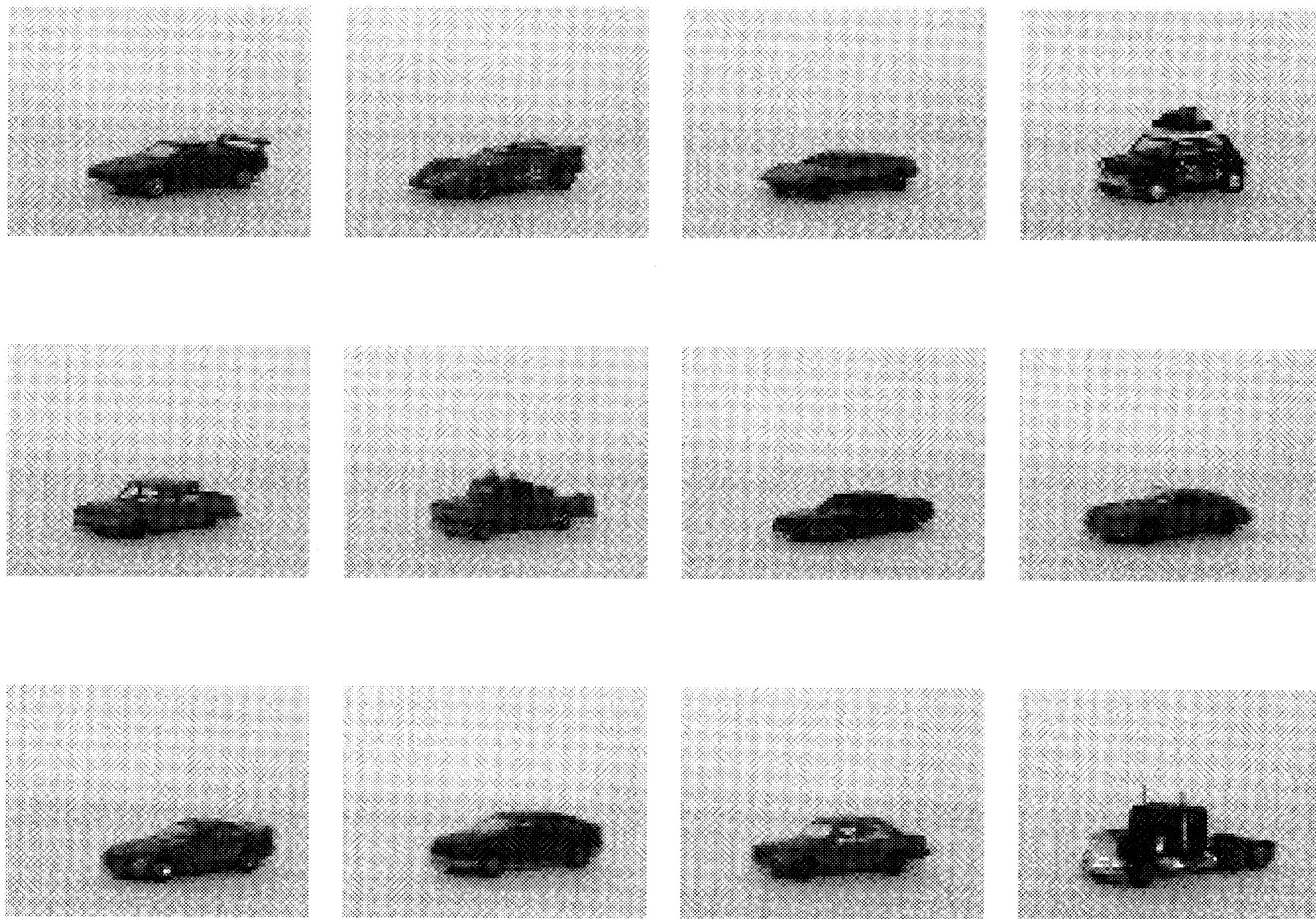


Figure 10.6 Images from a small appearance-based database composed of twelve toy cars. Only one image per object is shown.

each object, the size of such a database becomes prohibitively large, and search based on brute-force correlation unfeasible. Instead, we represent objects in *eigenspace*. To introduce eigenspaces and their advantages, we need to regard images as vector, and state a fundamental theorem.

To transform a 2-D image into a 1-D vector, we just scan the image top to bottom and left to right. In this way, a $N \times N$ image, X, is represented by a N^2-dimensional vector

$$\mathbf{x} = [X_{11}, X_{12} \dots X_{1N}, X_{21}, \dots X_{NN}]^\top .$$

Notice that *this representation allows us to write image correlation as the dot product of two vectors.*[5] For instance, the correlation of images X_1 and X_2, represented by vectors $\mathbf{x}_1, \mathbf{x}_2$ respectively, becomes

$$c = X_1 \circ X_2 = \mathbf{x}_1^\top \mathbf{x}_2.$$

From now on, we shall use vectors for images. And here is the fundamental theorem.

[5] We assume that the constant K in the correlation definition is 1.

Theorem: Eigenspace Representation

Let $\mathbf{x}_1, \ldots \mathbf{x}_n$ be N^2-dimensional vectors, and $\bar{\mathbf{x}} = \frac{1}{n}\sum_{j=1}^{n} \mathbf{x}_j$ their average. Given the $N^2 \times n$ matrix

$$X = [(\mathbf{x}_1 - \bar{\mathbf{x}}) | \ldots (\mathbf{x}_n - \bar{\mathbf{x}})],$$

we can write each $\mathbf{x}_j$ as

$$\mathbf{x}_j = \bar{\mathbf{x}} + \sum_{i=1}^{n} g_{ji}\mathbf{e}_i,$$

where $\mathbf{e}_1, \ldots \mathbf{e}_n$ are the eigenvectors of the *covariance matrix*, $Q = XX^\top$, corresponding to the n (nonzero) eigenvalues of Q, and $\mathbf{g}_j = [g_1, g_2, \ldots g_N]^\top$ is the vector of the *components* of $\mathbf{x}_j$ *in eigenspace.*

Now let us go back to our database of all images (all objects, all viewpoints, all illumination directions). Assuming O objects, with P viewpoints and L illumination directions for each object, the database contains OPL images. Using the procedure suggested by the theorem, we can build the covariance matrix, Q, of the whole database, and represent each image, $\mathbf{x}^o_{pl}$, with its vector of eigenspace coordinates, $\mathbf{g}^o_{pl}$. Q is clearly a very large matrix, and $\mathbf{g}^o_{pl}$ is the same size as $\mathbf{x}^o_{pl}$, but here comes the first advantage of eigenspaces: *only the components associated to the largest eigenvalues of Q are significant to represent the images*. In other words, assuming that the nonzero eigenvalues $\lambda_1 \ldots \lambda_n$ of Q are such that $\lambda_1 \geq \lambda_2 \geq \ldots \lambda_n$ and $\lambda_i \approx 0$ for $i > k$, we can write

$$\mathbf{x}_j \approx \sum_{i=1}^{k} g_{ji}\mathbf{e}_i + \bar{\mathbf{x}},$$

and ignore all the remaining $n - k$ components. If $k \ll n$ Each image, $\mathbf{x}_j$, is therefore represented by a point of coordinates $\mathbf{g}_j^\top$ in a k-dimensional eigenspace, a substantially smaller subspace of the original, n-dimensional eigenspace.

So far for one image, but how do we represent a *set of images*; that is, all the views in the representation of the o-th object? Imagine to move through the views of the representation; as pose and illumination change continuously, the point $\mathbf{g}^o_{pl}$ moves continuously in eigenspace, sweeping a so-called *manifold*,[6] $\mathbf{g}^o = \mathbf{g}^o(\mathbf{p}, \mathbf{l})$, where $\mathbf{p}$ and $\mathbf{l}$ are vectors defining the object pose and the illumination direction, respectively. The set of eigenspace points associated to the images of the o-th object is a sampling of the associated manifold.[7]

---

[6] Do not reel at the word "manifold"! You do not need any knowledge of manifolds to understand this section, and manifolds are replaced by curves in the algorithm boxes below. Notice that, if only one parameter is allowed to change between images, i.e., $\mathbf{g}^o(v)$, v a real number, the manifold can be viewed as a curve in eigenspace; if two parameters can change, a surface.

[7] If necessary, we can estimate the continuous manifold by interpolating between adjacent samples.

How do eigenspaces allow us to perform image correlation, and why do they make correlation more efficient? The key is that *the Euclidean distance in eigenspace is equivalent to image correlation*, and the advantage is in the fact that eigenspace points have only k coordinates. To show this, we first notice that the correlation of two image vectors, $\mathbf{x}_1$ and $\mathbf{x}_2$, with grey levels normalized as in our assumptions ($\|\mathbf{x}_1\|^2 = \|\mathbf{x}_2\|^2 = 1$), can be written through their Euclidean distance:

$$\|\mathbf{x}_1 - \mathbf{x}_2\|^2 = 2(1 - \|\mathbf{x}_1^\top \mathbf{x}_2\|),$$

so that maximizing correlation is equivalent to minimizing distance. The distance between image vectors, in turn, can be approximated by the distance in the k-dimensional eigenspace, $\|\mathbf{g}_1 - \mathbf{g}_2\|$:

$$\begin{aligned}
\|\mathbf{x}_1 - \mathbf{x}_2\|^2 &= \|\sum_{i=1}^{n} g_{1i}\mathbf{e}_i - \sum_{i=1}^{n} g_{2i}\mathbf{e}_i\|^2 \\
&\approx \|\sum_{i=1}^{k} g_{1i}\mathbf{e}_i - \sum_{i=1}^{k} g_{2i}\mathbf{e}_i\|^2 \\
&= \|\sum_{i=1}^{k} (g_{1i} - g_{2i})\mathbf{e}_i\|^2 \\
&= \sum_{i=1}^{k} (g_{1i} - g_{2i})^2 \\
&= \|\mathbf{g}_1 - \mathbf{g}_2\|^2.
\end{aligned}$$

And now we realize why correlation is computed more efficiently in eigenspace: Instead of $O(n)$ products needed by $\|\mathbf{x}_1 - \mathbf{x}_2\|^2$, we perform only $O(k)$ products for $\|\mathbf{g}_1 - \mathbf{g}_2\|^2$. As, in practice, n is usually larger than 100 and k smaller than, say, 20, we are reducing a minimum of $n^2 = 10000$ products to a maximum of $k^2 = 400$, saving two orders of magnitude!

Let us summarize it all. First, eigenspace points represent images with fewer numbers. Second, images are correlated efficiently by computing distances in eigenspace. Third, and most importantly, *eigenspaces suggest a way of learning object models automatically*: we acquire the complete set of all possible views for each object o,

$$\{\mathbf{x}_{11}^o, \mathbf{x}_{12}^o, \ldots \mathbf{x}_{1L}^o, \mathbf{x}_{21}^o \ldots \mathbf{x}_{PL}^o\},$$

reduce the dimensionality as described above, and compute the corresponding, discrete manifold in eigenspace, $\{\mathbf{g}_{11}^o, \mathbf{g}_{12}^o, \ldots \mathbf{g}_{1L}^o, \mathbf{g}_{21}^o \ldots \mathbf{g}_{PL}^o\}$. To identify an object from a new image $\mathbf{y}$, we project $\mathbf{y}$ in eigenspace (using the eigenvectors of the covariance matrix of *all* the OPL images in the database), obtaining a point $\mathbf{g}_y$, then look for the object manifold $\mathbf{g}^o(\mathbf{p}, \mathbf{l})$ closest to $\mathbf{g}_y$. So 3-D appearance-based identification is solved as a minimum-distance problem in eigenspace.

The Parametric Eigenspace Method. This section contains two algorithm boxes: one to learn appearance-based models, one to identify objects from new images. As the learning stage can be very expensive in terms of memory occupation, we suggest a small-scale version of the algorithm, which uses small images, assumes illumination fixed, and constrains pose changes to rotations around a fixed axis (which means the manifold becomes a curve in eigenspace). You can easily extend our version to consider illumination direction, full pose parameters, and larger images.[8]

Algorithm EIGENSPACE_LEARN

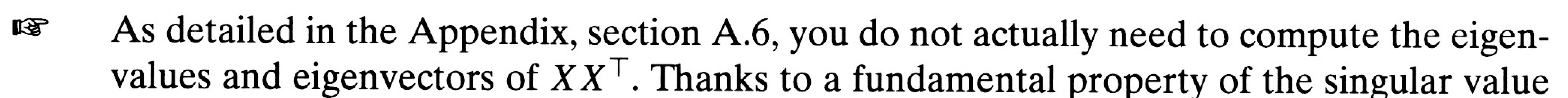
We consider the assumptions stated at the beginning of this section valid; moreover, we assume a fixed camera, fixed illumination conditions, and images of $N \times N$ pixels.

1. For each object o to be represented, $o = 1 \dots O$:
   (a) place the object on the turntable;
   (b) acquire a set of n images by rotating the turntable by $\frac{360°}{n}$ each time;
   (c) in all images, make sure to adjust the background so that the object can be easily segmented from the background;
   (d) segment the object from the background (see Exercise 10.9);
   (e) normalize the images in scale and energy as stated in the assumptions;
   (f) represent the normalized images as vectors, $\mathbf{x}_p^o$, where p is the rotation index, $p = 1, \dots, n$.
2. Compute the average image vector, $\bar{\mathbf{x}}$, of the complete database $\left\{\mathbf{x}_1^1, \dots \mathbf{x}_n^1, \mathbf{x}_1^2, \dots \mathbf{x}_n^2, \dots \mathbf{x}_n^O\right\}$.
3. Form the $N^2 \times N^2$ covariance matrix, $Q = XX^\top$, with $X = \left[\mathbf{x}_1^1 | \mathbf{x}_2^1 | \dots \mathbf{x}_n^1 | \mathbf{x}_1^2 | \dots \mathbf{x}_n^O\right]$.
4. Compute the eigenvalues of Q, keep the first k largest eigenvalues and the associated eigenvectors, $\mathbf{e}_1, \dots \mathbf{e}_k$.
5. for each object, o:
   (a) compute the k-dimensional eigenspace points corresponding to the n images:

$$\mathbf{g}_p^o = [\mathbf{e}_1 | \dots \mathbf{e}_k] \, (\mathbf{x}_p^o - \bar{\mathbf{x}});$$

   (b) store the discrete eigenspace curve $\left\{\mathbf{g}_1^o, \dots, \mathbf{g}_p^o, \dots, \mathbf{g}_n^o\right\}$, as the representation of object o.

The output is a set of O discrete curves in the k-dimensional eigenspace, each representing a 3-D object.



☞ As detailed in the Appendix, section A.6, you do not actually need to compute the eigenvalues and eigenvectors of $XX^\top$. Thanks to a fundamental property of the singular value

---

[8] If you have a few gigabytes to spare, that is.

decomposition, the eigenvalues of $XX^\top$ are the same as the eigenvalues of the $n \times n$ matrix $X^\top X$ (a matrix of much smaller size) and the eigenvectors of $XX^\top$ can be computed from the corresponding eigenvectors of $X^\top X$.

Notice that the dimensionality reduction is carried out on the global database. This ensures that the k important eigenvectors record visual information of all objects in the database. Conversely, as we see next, *recognition* is performed in the eigenspace of *individual* objects. To begin with, we suggest you try EIGENSPACE_LEARN with $N = 64$, $O = 5$, $n = 32$. This means that Q is 4096×4096, and X is 4096×32, all rather reasonable numbers.

☞ To implement EIGENSPACE_LEARN, you need a turntable to change the viewpoint by controlled rotations. The best would be to use a computer-controlled turntable, but the dish of an old record player, turning at constant speed, will do for the first attempts.

We now turn to the identification algorithm. Obviously enough, we assume that the learning and identification stage are run with the same illumination conditions and camera position.

Algorithm EIGENSPACE_IDENTIF

The input is a $N \times N$ image, I, of one of the objects in the database. The image I must satisfy the assumptions stated at the beginning of this section and acquired so that the object can be easily segmented from the background. We assume the same illumination conditions and camera position adopted in EIGENSPACE_LEARN.

1. Segment the object from the background.
2. Normalize I in scale and energy, and represent the normalized image as a vector, $\mathbf{i}$.
3. Compute the k-dimensional eigenspace point corresponding to $\mathbf{i}$:

$$\mathbf{g} = [\mathbf{e}_1 | \dots \mathbf{e}_k] \, (\mathbf{i} - \bar{\mathbf{x}}),$$

   where $\bar{\mathbf{x}}$ is the average image vector of the whole database.
4. Find the eigenspace point, $\hat{\mathbf{g}}$, created by EIGENSPACE_LEARN, closest to $\mathbf{g}$.

The output is the object associated to the curve on which $\hat{\mathbf{g}}$ lies; that is, the identity of the object in I.

Now for a few points of practical importance. First, finding the point of a curve or surface which is closest to a given point is not trivial; if the curve is represented by a high number of points (as in our case), brute force can prove too expensive. Second, it may not always be true that $k \ll n$. Third, finding the eigenvalues of very large matrices is computationally expensive, and special algorithms exist for this purpose. Finally, the

figure-ground segmentation necessary to zero out background pixels is not trivial, and, in general, is simple only for certain classes of objects and with controlled scenes.

10.5 Concluding Remarks on Object Identification

How do the methods presented compare with each other? Although simple, INT_TREE is a reasonable algorithm for real images. It copes with missing features, noisy features, and multiple object instances. The wild card inflates complexity, which becomes exponential in the number of model and data features; branch-and-bound and other methods (see Further Readings) alleviate this problem. INT_TREE performs grouping and identification *simultaneously* (see the four problems of section 10.1): It selects which image features are most likely to belong to the object, and performs the identity test. The inevitable price is a rather high complexity. *Alignment* or *hybrid methods* are another way to reduce the complexity of IT search. Such methods match only the number of data features strictly necessary for carrying out verification. More of this is discussed in the next chapter.

Invariants provide image measurements independent of viewpoint and intrinsic parameters, and suggest an easy strategy for model acquisition from real images. These are very valuable characteristics for practical recognition systems. However, their usability is subject to the possibility of defining invariants for the objects of interest, but not many invariants are known and easy to compute for 3-D shapes. Other points requiring attention include grouping, which is shared by all classifications methods based on local features, and the discriminational power of each invariant (how reliably can different objects be told apart given noisy images).

It can be more laborious to build models suitable for an interpretation-tree algorithm, as we presented it, than for invariants. Moreover, interpretation tree locate instances of a *given* object in an image, and require model search to recognize all objects present in an image; invariants, instead, support direct model indexing, and do not require model search. However, interpretation trees take care of feature grouping; invariant do not.

Invariants-based methods allow one to build model libraries from only one or two views per object, as opposed to the many views required to build a parametric eigenspace. However, eigenspaces can be built for *any* 3-D shape and do not require feature extraction, while invariants can cater only for special shape classes and depend on the performance of the feature extractor. Again, eigenspaces do not require feature grouping, invariants do. A disadvantage of parametric eigenspace methods is that they are vulnerable to occlusion and sensitive to segmentation.

10.6 3-D Object Modelling

As promised at the beginning of this chapter, we now bring together the hints to 3-D object modelling scattered throughout this chapter. By now you have certainly realized that designing adequate models is tremendously important, and, indeed, *3-D object modelling* is a much-investigated issue in computer vision. The aim of this section is *not* to list the many representations in existence (although some examples will be

Step 4: Classifying Local Shape. Finally, the shape classification given in Chapter 4 is achieved by defining two further quantities, the *mean curvature, H,* and the *Gaussian curvature, K*:

$$H = -\frac{k_1 + k_2}{2} \qquad K = k_1 k_2.$$

One can show that *the Gaussian curvature measures how fast the surface moves away from the tangent plane around P*, and in this sense is an extension of the 1-D curvature k. The formulae giving H and K for a range surface in r_{ij} form, $(x, y, h(x, y))$ are given in Chapter 4.

References

M.P. Do Carmo, *Differential Geometry of Curves and Surfaces*, Prentice-Hall, Englewood Cliffs (NJ) (1976).

A.6 Singular Value Decomposition

The aim of this section is to collect the basic information needed to understand the *Singular Value Decomposition* (SVD) as used throughout this book. We start giving the definition of SVD for a generic, rectangular matrix A and discussing some related concepts. We then illustrate three important applications of the SVD:

- solving systems of nonhomogeneous linear equations;
- solving rank-deficient systems of homogeneous linear equations;
- guaranteeing that the entries of a matrix estimated numerically satisfy some given constraints (e.g., orthogonality).

Definition

Singular Value Decomposition

Any $m \times n$ matrix A can be written as the product of three matrices:

$$A = UDV^\top. \tag{A.6}$$

The columns of the $m \times m$ matrix U are mutually orthogonal unit vectors, as are the columns of the $n \times n$ matrix V. The $m \times n$ matrix D is diagonal; its diagonal elements, σ_i, called *singular values*, are such that $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_n \geq 0$.

☞ While both U and V are not unique, the singular values σ_i are fully determined by A.

Some important properties now follow.

Properties of the SVD

Property 1. The singular values give you valuable information on the singularity of a square matrix, *A square matrix, A, is nonsingular if and only if all its singular values are different from zero.* Most importantly, the σ_i also tell you how close A is to be singular: the ratio

$$C = \frac{\sigma_1}{\sigma_n},$$

called *condition number*, measures the *degree of singularity of A*. When $1/C$ is comparable with the arithmetic precision of your machine, the matrix A is *ill-conditioned* and, for all practical purposes, can be considered singular.

Property 2. *If A is a rectangular matrix, the number of nonzero σ_i equals the rank of A.* Thus, given a fixed tolerance, ϵ (typically of the order of 10^{-6}), the number of singular values greater than ϵ equals the *effective rank* of A.

Property 3. *If A is a square, nonsingular matrix, its inverse can be written as*

$$A^{-1} = VD^{-1}U^{\top}.$$

Be A singular or not, the *pseudoinverse* of A, A^{+}, can be written as

$$A^{+} = VD_0^{-1}U^{\top},$$

with D_0^{-1} equal to D^{-1} for all nonzero singular values and zero otherwise. If A is nonsingular, then $D_0^{-1} = D^{-1}$ and $A^{+} = A^{-1}$.

Property 4. The columns of U corresponding to the nonzero singular values span the range of A, the columns of V corresponding to the zero singular value the null space of A.

Property 5. The squares of the nonzero singular values are the nonzero eigenvalues of both the $n \times n$ matrix $A^{\top}A$ and $m \times m$ matrix $AA^{\top}$. The columns of U are eigenvectors of $AA^{\top}$, the columns of V eigenvectors of $A^{\top}A$. Morevoer, $A\mathbf{u}_k = \sigma_k \mathbf{v}_k$ and $A^{\top}\mathbf{v}_k = \sigma_k \mathbf{u}_k$, where $\mathbf{u}_k$ and $\mathbf{v}_k$ are the columns of U and V corresponding to σ_k.

Property 6. One possible distance measure between matrices can use the *Frobenius norm*. The Frobenius norm of a matrix A is simply the sum of the squares of the entries a_{ij} of A, or

$$\|A\|_F = \sum_{i,j} a_{ij}^2. \tag{A.7}$$

By plugging (A.6) in (A.7), it follows that

$$\|A\|_F = \sum_i \sigma_i^2.$$

We are now ready to summarize the applications of the SVD used throughout this book.

Least Squares

Assume you have to solve a system of m linear equations,

$$A\mathbf{x} = \mathbf{b},$$

for the unknown n-dimensional vector $\mathbf{x}$. The $m \times n$ matrix A contains the coefficients of the equations, the m-dimensional vector $\mathbf{b}$ the data. If not all the components of $\mathbf{b}$ are null, the solution can be found by multiplying both sides of the above equation for $A^\top$ to obtain

$$A^\top A\mathbf{x} = A^\top \mathbf{b}.$$

It follows that the solution is given by

$$\mathbf{x} = (A^\top A)^+ A^\top \mathbf{b}.$$

This solution is known to be optimal in the least square sense.

It is usually a good idea to compute the pseudoinverse of $A^\top A$ through SVD. In the case of more equations than unknowns the pseudoinverse is more likely to coincide with the *inverse* of $A^\top A$, but keeping an eye on the condition number of $A^\top A$ (**Property 1**) won't hurt.

☞ Notice that linear fitting amounts to solve exactly the same equation. Consequently, you can use the same strategy!

Homogeneous Systems

Assume you are given the problem of solving a homogeneous system of m linear equations in n unknowns

$$A\mathbf{x} = 0,$$

with $m \geq n - 1$ and $rank(A) = n - 1$. Disregarding the trivial solution $\mathbf{x} = 0$, a solution unique up to a scale factor can easily be found through SVD. *This solution is simply proportional to the eigenvector corresponding to the only zero eigenvalue of* $A^\top A$ (all other eigenvalues being strictly positive because $rank(A) = n - 1$). This can be proven as follows.

Since the norm of the solution of a homogeneous system of equations is arbitrary, we look for a solution of unit norm in the least square sense. Therefore we want to minimize

$$\|A\mathbf{x}\|^2 = (A\mathbf{x})^\top A\mathbf{x} = \mathbf{x}^\top A^\top A\mathbf{x},$$

subject to the constraint

$$\mathbf{x}^\top \mathbf{x} = 1.$$

Introducing the Lagrange multiplier λ this is equivalent to minimize the *Lagrangian*

$$\mathcal{L}(\mathbf{x}) = \mathbf{x}^\top A^\top A \mathbf{x} - \lambda(\mathbf{x}^\top \mathbf{x} - 1).$$

Equating to zero the derivative of the Lagrangian with respect to $\mathbf{x}$ gives

$$A^\top A \mathbf{x} - \lambda \mathbf{x} = 0.$$

This equation tells you that λ is an eigenvalue of $A^\top A$, and the solution, $\mathbf{x} = \mathbf{e}_\lambda$, the corresponding eigenvector. Replacing $\mathbf{x}$ with $\mathbf{e}_\lambda$, and $A^\top A \mathbf{e}_\lambda$ with $\lambda \mathbf{e}_\lambda$ in the Lagrangian yields

$$\mathcal{L}(\mathbf{e}_\lambda) = \lambda.$$

Therefore, the minimum is reached at $\lambda = 0$, the least eigenvalue of $A^\top A$. But from **Properties 4** and **5**, it follows that this solution could have been equivalently established as *the column of V corresponding to the only null singular value of A* (the kernel of A). This is the reason why, throughout this book, we have not distinguished between these two *seemingly* different solutions of the same problem.

Enforcing Constraints

One often generates numerical estimates of a matrix, A, whose entries are not all independent, but satisfy some algebraic constraints. This is the case, for example, of orthogonal matrices, or the fundamental matrix we met in Chapter 7. What is bound to happen is that the errors introduced by noise and numerical computations alter the estimated matrix, call it $\hat{A}$, so that its entries no longer satisfy the given constraints. This may cause serious problems if subsequent algorithms assume that $\hat{A}$ satisfies *exactly* the constraints.

Once again, SVD comes to the rescue, and allows us to *find the closest matrix to $\hat{A}$, in the sense of the Frobenius norm (**Property 6**), which satisfies the constraints exactly*. This is achieved by computing the SVD of the estimated matrix, $\hat{A} = UDV^\top$, and estimating A as $UD'V^\top$, with D' obtained by changing the singular values of D to those expected when the constraints are satisfied exactly.[4] Then, the entries of $A = UD'V^\top$ satisfy the desired constraints by construction.

References

G. Strang, *Linear Algebra and its Applications*, Harcourt Brace Jovanovich, Orlando (FL) (1988).

[4] If $\hat{A}$ is a good numerical estimate, its singular values should not be too far from the expected ones.