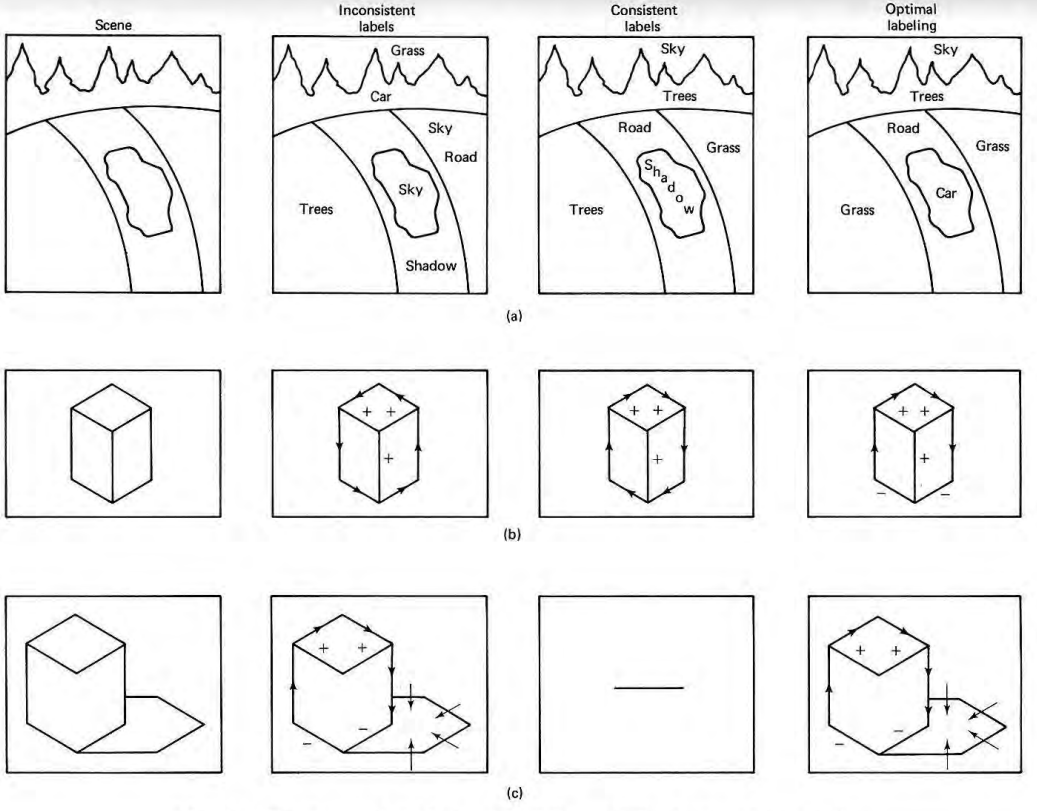


Fig. 12.7 Three scenes (A, B, C) and their labelings. Labelings are only "consistent," "inconsistent," or "optimal" with respect to a given relational structure of objects (an input scene) and a set of constraints. These examples are meant to be illustrative only.

plest way to find a consistent labeling of a relational structure (we shall often say "labeling of a scene") is to apply a depth-first *tree search* of the labeling possibilities, as in the backtracking algorithm (11.1).

Label an object in accordance with unary constraints.

Iterate until a globally consistent labeling is found:

Given the current labeling, label another object consistently—in accordance with all constraints.

If the object cannot be labeled consistently, backtrack and pick a new label for a previously labeled object.

This labeling algorithm can be computationally inefficient. First, it does not prune the search tree very effectively. Second, if it is used to generate all consistent labelings, it does not recognize important independences in the labels. That is, it does not notice that conclusions reached (labels assigned) in part of the tree search are usable in other parts without recomputation.

In a *serial relaxation*, the labels are changed one object at a time. After each such change, the new labeling is used to determine which object to process next. This technique has proved useful in some applications [Feldman and Yakimovsky 1974].

Assign all possible labels to each object in accordance with unary constraints.

Iterate until a globally consistent labeling is found:

Somehow pick an object to be processed.

Modify its labels to be consistent with the current labeling.

A *parallel iterative* algorithm adjusts all object labels at once; we have seen this approach in several places, notably in the "Waltz filtering algorithm" of Section 9.5.

Assign all possible labels to each object in accordance with unary constraints.

Iterate until a globally consistent labeling is found:

In parallel, eliminate from each object's label set those labels that are inconsistent with the current labels of the rest of the relational structure.

A less structured version of relaxation occurs when the iteration is replaced with an *asynchronous interaction* of labeled objects. Such interaction may be implemented with multiple cooperating processes or in a data base with "demons" (Ap-

pendix 2). This method of relaxation was used in MSYS [Barrow and Tenenbaum 1976]. Here imagine that each object is an active process that knows its own label set and also knows about the constraints, so that it knows about its relations with other objects. The program of each object might look like this.

> If I have just been activated, and my label set is not consistent with the labels of other objects in the relational structure, then I change my label set to be consistent, else I suspend myself.
>
> Whenever I change my label set, I activate other objects whose label set may be affected, then I suspend myself.

To use such a set of active objects, one can give each one all possible labels consistent with the unary constraints, establish the constraints so that the objects know where and when to pass on activity, and activate all objects.

Constraints involving arbitrarily many objects (i.e., constraints of arbitrarily high *order*) can efficiently be relaxed by recording acceptable labelings in a graph structure [Freuder 1978]. Each object to be labeled initially corresponds to a node in the graph, which contains all legal labels according to unary constraints. Higher order constraints involving more and more nodes are incorporated successively as new nodes in the graph. At each step the new node constraint is *propagated*; that is, the graph is checked to see if it is consistent with the new constraint. With the introduction of more constraints, node pairings that were previously consistent may be found to be inconsistent. As an example consider the following graph coloring problem: color the graph in Fig. 12.8 so that neighboring nodes have different colors. It is solved by building constraints of increasingly higher order and propagating them. The node constraints are given explicitly as shown in Fig. 12.8a, but the higher-order constraints are given in functional implicit form; prospective colorings must be tested to see if they satisfy the constraints. After the node constraints are given, order two constraints are synthesized as follows: (1) make a node for each node pairing; (2) add all labelings that satisfy the constraint. The result is shown in Fig. 12.8b. The single constraint of order three is synthesized in the same way, but now the graph is inconsistent: the match "Y,Z: Red,Green" is ruled out by the third order legal label set (RGY,GRY). To restore consistency the constraint is propagated through node (Y,Z) by deleting the inconsistent labelings. This means that the node constraint for node Z is now inconsistent. To remedy this, the constraint is propagated again by deleting the inconsistency, in this case the labeling $(Z{:}G)$. The change is propagated to node (X,Z) by deleting (X,Z: Red,Green) and finally the network is consistent.

In this example constraint propagation did not occur until constraints of order three were considered. Normally, some constraint propagation occurs after every order greater than one. Of course it may be impossible to find a consistent graph. This is the case when the labels for node Z in our example are changed from (G,Y) to (G,R). Inconsistency is then discovered at order three.

It is quite possible that a discrete labeling algorithm will not yield a unique label for each object. In this case, a consistent labeling exists using each label for the

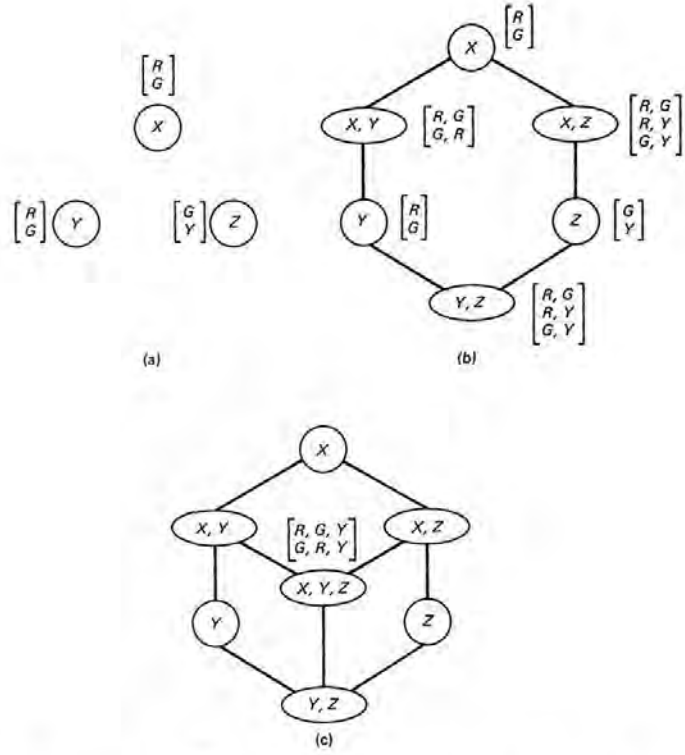


Fig. 12.8 Coloring a graph by building constraints of increasingly higher order.

object. However, which of an object's multiple labels goes with which of another object's multiple labels is not determined. The final enumeration of consistent labelings usually proceeds by tree search over the reduced set of possibilities remaining after the relaxation.

Convergence properties of relaxation algorithms are important; convergence means that in some finite time the labeling will "settle down" to a final value. In discrete labeling, constraints may often be written so that the label adjustment phase always reduces the number of labels for an object (inconsistent ones are eliminated). In this case the algorithm clearly must converge in finite time to a consistent labeling, since for each object the label set must either shrink or stay stable. In schemes where labels are added, or where labels have complex structure (such as real number "weights" or "probabilities"), convergence is often not guaranteed mathematically, though such schemes may still be quite useful. Some probabilistic labeling schemes (Section 12.4.3) have provably good convergence properties.

It is possible to use relaxation schemes without really considering their mathematical convergence properties, their semantics (What is the semantics of weights attached to labels—are they probabilities?), or a clear definition of what exactly the relaxation is to achieve (What is a good set of labels?). The fact that some schemes can be shown to have unpleasant properties (such as assigning nonzero weights to each of two inconsistent hypotheses, or not always converging to a solution), does not mean that they cannot be used. It only means that their behavior is not formally characterizable or possibly even predictable. As relaxation computations become more common, the less formalizable, less predictable, and less conceptually elegant forms of relaxation computations will be replaced by better behaved, more thoroughly understood schemes.

12.4.3 A Linear Relaxation Operator and a Line Labeling Example

The Formulation

We now move away from discrete labeling and into the realm of continuous *weights* or *supposition values* on labels. In Sections 12.4.3 and 12.4.4 we follow closely the development of [Rosenfeld et al. 1976]. Let us require that the sum of label weights for each object be constrained to sum to unity. Then the weights are reminiscent of probabilities, reflecting the "probability that the label is correct." When the labeling algorithm converges, a label emerges with a high weight if it occurs in a probable labeling of the scene. Weights, or supposition values, are in fact hard to interpret consistently as probabilities, but they are suggestive of likelihoods and often can be manipulated like them.

In what follows p refers to probability-like weights (supposition values) rather than to the value of a probability density function. Let a relational structure with n objects be given by a_i, $i = 1, \ldots, n$, each with m discrete labels $\lambda_1, \ldots, \lambda_m$. The shorthand $p_i(\lambda)$ denotes the weight, or (with the above caveats) the "probability" that the label λ (actually λ_k for some k) is correct for the object a_i. Then the probability axioms lead to the following constraints,

$$0 \leqslant p_i(\lambda) \leqslant 1 \tag{12.14}$$

$$\sum_{\lambda} p_i(\lambda) = 1 \tag{12.15}$$

The labeling process starts with an initial assignment of weights to all labels for all objects [consistent with Eqs. (12.14) and (12.15)]. The algorithm is parallel iterative: It transforms all weights at once into a new set conforming to Eqs. (12.14) and (12.15), and repeats this transformation until the weights converge to stable values.

Consider the transformation as the application of an operator to a vector of label weights. This operator is based on the *compatibilities* of labels, which serve as constraints in this labeling algorithm. A compatibility p_{ij} looks like a conditional probability.

$$\sum_{\lambda} p_{ij}(\lambda | \lambda') = 1 \qquad \text{for all } i, j, \lambda' \tag{12.16}$$

$$p_{ii}(\lambda|\lambda') = 1 \qquad iff\ \lambda = \lambda',\quad \text{else } 0. \tag{12.17}$$

The $p_{ij}(\lambda|\lambda')$ may be interpreted as the conditional probability that object a_i has label λ given that another object a_j has label λ'. These compatibilities may be gathered from statistics over a domain, or may reflect a priori belief or information.

The operator iteratively adjusts label weights in accordance with other weights and the compatibilities. A new weight $p_i(\lambda)$ is computed from old weights and compatibilities as follows.

$$p_i(\lambda) := \sum_j c_{ij} \left\{ \sum_{\lambda'} p_{ij}(\lambda|\lambda') p_j(\lambda') \right\} \tag{12.18}$$

The c_{ij} are coefficients such that

$$\sum_j c_{ij} = 1 \tag{12.19}$$

In Eq. (12.18), the inner sum is the expectation that object a_i has label λ, given the evidence provided by object a_j. $p_i(\lambda)$ is thus a weighted sum of these expectations, and the c_{ij} are the weights for the sum.

To run the algorithm, simply pick the p_{ij} and c_{ij}, and apply Eq. (12.18) repeatedly to the p_i until they stop changing. Equation (12.18) is in the form of a matrix multiplication on the vector of weights, as shown below; the matrix elements are weighted compatibilities, the $c_{ij}p_{ij}$. The relaxation operator is thus a matrix; if it is partitioned into several *component* matrices, one for each set of non-interacting weights, linear algebra yields proofs of convergence properties [Rosenfeld et al. 1976]. The iteration for the reduced matrix for each component does converge, and converges to the weight vector that is the eigenvector of the matrix with eigenvalue unity. This final weight vector is independent of the initial assignments of label weights; we shall say more about this later.

An Example

Let us consider the input line drawing scene of Fig. 12.9a used in [Rosenfeld et.al. 1976]. The line labels given in Section 9.5 allow several consistent labels as shown in Fig. 12.9b-e, each with a different physical interpretation.

In the discrete labelling "filtering" algorithm presented in Section 9.5 and outlined in the preceding section, the relational structure is imposed by the neighbor relation between vertices induced by their sharing a line. Unary constraints are imposed through a catalog of legal combinations of line labels at vertices, and the binary constraint is that a line must not change its label between vertices. The algorithm eliminates inconsistent labels.

Let us try to label the sides of the triangle a_1, a_2, and a_3 in Fig. 12.9 with the solid object edge labels $\{>, <, +, -\}$. To do this requires some "conditional probabilities" for compatibilities $p_{ij}(\lambda|\lambda')$, so let us use those that arise if all eight interpretations of Fig. 12.9 are equally likely. Remembering that

$$p(X|Y) = \frac{p(X,Y)}{p(Y)} \tag{12.20}$$

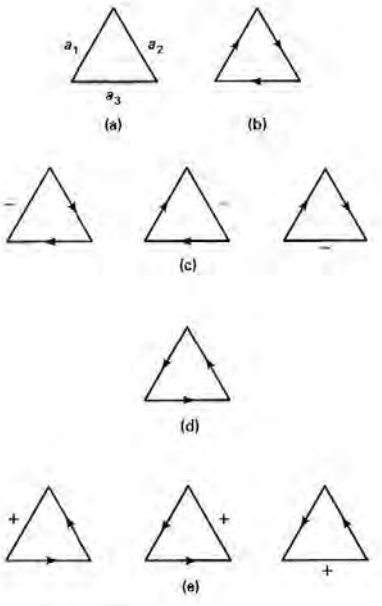


Fig. 12.9 A triangle and its possible labels. (a) Edge names. (b) Floating. (c) Flap folded up. (d) Triangular hole. (e) Flap folded down.

and taking $p(X, Y)$ to mean the probability that labels X and Y occur consecutively in clockwise order around the triangle, one can derive Table 12.2. Of course, we could choose other compatibilities based on any considerations whatever as long as Eqs. (12.16) and (12.17) are preserved.

Table 12.2 shows that there are two noninteracting components, $\{-,>\}$ and $\{+,<\}$. Consider the first component that consists of the weight vector

$$[p_1(>),\ p_1(-),\ p_2(>),\ p_2(-),\ p_3(>),\ p_3(-)] \tag{12.21}$$

The second is treated similarly. This vector describes weights for the subpopulation of labelings given by Fig. 12.9b and c. The matrix M of compatibilities has columns of weighted p_{ij}.

$$M = \begin{bmatrix} c_{11}p_{11}(>|>) & c_{21}p_{21}(>|>) & \cdots \\ c_{11}p_{11}(>|-) & c_{21}p_{21}(>|-) & \cdots \\ c_{12}p_{12}(>|>) & c_{22}p_{22}(>|>) & \cdots \\ c_{12}p_{12}(>|-) & c_{22}p_{22}(>|-) & \cdots \\ c_{13}p_{13}(>|>) & c_{23}p_{23}(>|>) & \cdots \\ c_{13}p_{13}(>|-) & c_{23}p_{23}(>|-) & \cdots \end{bmatrix} \tag{12.22}$$

Table 12.2

λ_1	λ_2	$p(\lambda_1, \lambda_2)$	$p(\lambda_1 \mid \lambda_2)$
>	>	1/4	2/3
>	−	1/8	1
−	>	1/8	1/3
−	−	0	0
>	<	0	0
>	+	0	0
−	<	0	0
−	+	0	0
<	>	0	0
+	>	0	0
<	−	0	0
+	−	0	0
<	<	1/4	2/3
<	+	1/8	1
+	<	1/8	1/3
+	+	0	0

If we let $c_{ij} = 1/3$ for all i, j, then

$$M = \frac{1}{3}\begin{bmatrix} 1 & 0 & 2/3 & 1/3 & 2/3 & 1/3 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 2/3 & 1/3 & 1 & 0 & 2/3 & 1/3 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 2/3 & 1/3 & 2/3 & 1/3 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{12.23}$$

An analytic eigenvector calculation (Appendix 1) shows that the M of Eq. (12.23) yields (for any initial weight vector) the final weight vector of

$$[3/4,\ 1/4,\ 3/4,\ 1/4,\ 3/4,\ 1/4] \tag{12.24}$$

Thus each line of the population in the component we chose (Fig. 12.9b and c) has label > with "probability" 3/4, −with "probability" 1/4. In other words, from an initial assumption that all labelings in Fig. 12.9b and c were equally likely, the system of constraints has "relaxed" to the state where the "most likely" labeling is that of Fig. 12.9b, the floating triangle.

This relaxation method is a crisp mathematical technique, but it has some drawbacks. It has good convergence properties, but it converges to a solution entirely determined by the compatibilities, leaving no room for preferences or local scene evidence to be incorporated and affect the final weights. Further, the algorithm perhaps does not exactly mirror the following intuitions about how relaxation should work.

1. Increase $p_i(\lambda)$ if high probability labels for other objects are compatible with assignment of λ to a_i.
2. Decrease $p_i(\lambda)$ if high probability labels are incompatible with the assignment of λ to a_i.
3. Labels with low probability, compatible or incompatible, should have little influence on $p_i(\lambda)$.

However, the operator of this section decreases $p_i(\lambda)$ the most when other labels have both low compatibility and low probability. Thus it accords with (1) above, but not with (2) or (3). Some of these difficulties are addressed in the next section.

12.4.4 A Nonlinear Operator

The Formulation

If compatibilities are allowed to take on both positive and negative values, then we can express strong incompatibility better and obtain behavior more like (1), (2), and (3) just above. Denote the compatibility of the event "label λ on a_i" with the event "label λ on a_j" by $r_{ij}(\lambda, \lambda')$. If the two events occur together often, r_{ij} should be positive. If they occur together rarely, r_{ij} should be negative. If they are independent, r_{ij} should be 0. The *correlation coefficient* behaves like this, and the compatibilities of this section are based on correlations (hence the the notation r_{ij} for compatibilities). The correlation is defined using the covariance.

$$\operatorname{cov}(X, Y) = p(X, Y) - p(X)p(Y)$$

Now define a quantity σ which is like the standard deviation

$$\sigma(X) = [p(X) - (p(X))^2]^{1/2} \tag{12.25}$$

then the correlation is the normalized covariance

$$\operatorname{cor}(X, Y) = \frac{\operatorname{cov}(X, Y)}{\sigma(X)\sigma(Y)} \tag{12.26}$$

This allows the formulation of an expression precisely analogous to Eq. (12.18), only that r_{ij} instead of p_{ij} is used to obtain a means of calculating the positive or negative change in weights.

$$q_i^{(k)}(\lambda) = \sum_j c_{ij} \left[\sum_{\lambda'} r_{ij}(\lambda, \lambda') p_j^{(k)}(\lambda')\right] \tag{12.27}$$

In Eqs. (12.27)–(12.29) the superscripts indicate iteration numbers. The weight change (Eq. 12.27) could be applied as follows,

$$p_i^{(k+1)}(\lambda) = p_i^{(k)}(\lambda) + q_i^{(k)}(\lambda) \tag{12.28}$$

but then the resultant label weights might not remain nonnegative. Fixing this in a straightforward way yields the iteration equation

$$p_i^{(k+1)}(\lambda) = \frac{p_i^{(k)}(\lambda)[1 + q_i^{(k)}(\lambda)]}{\sum_{\lambda} p_i^{(k)}(\lambda)[1 + q_i^{(k)}(\lambda)]} \tag{12.29}$$

The convergence properties of this operator seem to be unknown, and like the linear operator it can assign nonzero weights to maximally incompatible labelings. However, its behavior can accord with intuition, as the following example shows.

An Example

Computing the covariances and correlations for the set of labels of Fig. 12.9b-e yields Table 12.3.

Figure 12.10 shows the nonlinear operator of Eq. (12.29) operating on the example of Fig. 12.9. Figure 12.10 shows several cases.

1. Equal initial weights: convergence to apriori probabilities ($\frac{3}{8}$, $\frac{3}{8}$, $\frac{1}{8}$, $\frac{1}{8}$).
2. Equal weights in the component $\{>,-\}$: convergence to "most probable" floating triangle labeling.
3. Slight bias toward a flap labeling is not enough to overcome convergence to the "most probable" labeling, as in (2).
4. Like (3), but greater bias elicits the "improbable" labeling.
5. Contradicatory biases toward "improbable" labelings: convergence to "most probable" labeling instead.
6. Like (5), but stronger bias toward one "improbable" labeling elicits it.
7. Bias toward one of the components $\{>,-\}$, $\{<,+\}$ converges to most probable labeling in that component.
8. Like (7), only biased to less probable labelling in a component.

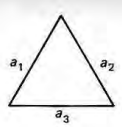
12.4.5 Relaxation as Linear Programming

The Idea

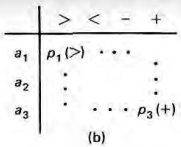
Linear programming (LP) provides some useful metaphors for thinking about relaxation computations, as well as actual algorithms and a rigorous basis [Hummel and Zucker 1980]. In this section we follow the development of [Hinton 1979].

Table 12.3

λ_1	λ_2	$\text{cov}(\lambda_1, \lambda_2)$	$\text{cor}(\lambda_1, \lambda_2)$
$>$	$>$	$\frac{7}{64}$	$\frac{7}{15}$
$>$	$-$	$\frac{5}{64}$	$5/\sqrt{105}$
$-$	$>$	$\frac{5}{64}$	$5/\sqrt{105}$
$-$	$-$	$-\frac{1}{64}$	$-\frac{1}{7}$
$>$	$<$	$-\frac{9}{64}$	$-\frac{1}{5}$
.	.	.	.
.	.	.	.
.	.	.	.



(a)



(b)

Case	Initial weights				After 2 to 3 iterations				After 20 to 30 iterations				Limit			
(1)	0.25	0.25	0.25	0.25	0.3	0.3	0.2	0.2	0.33	0.33	0.17	0.17	0.37	0.37	0.13	0.13
	0.25	0.25	0.25	0.25	0.3	0.3	0.2	0.2	0.33	0.33	0.17	0.17	0.37	0.37	0.13	0.13
	0.25	0.25	0.25	0.25	0.3	0.3	0.2	0.2	0.33	0.33	0.17	0.17	0.37	0.37	0.13	0.13
(2)	0.5	0	0.5	0	0.8	0	0.2	0	0.98	0	0.2	0	1	0	0	0
	0.5	0	0.5	0	0.8	0	0.2	0	0.98	0	0.2	0	1	0	0	0
	0.5	0	0.5	0	0.8	0	0.2	0	0.98	0	0.2	0	1	0	0	0
(3)	0.5	0	0.5	0	0.62	0	0.37	0	1	0	0	0	1	0	0	0
	0.4	0	0.6	0	0.49	0	0.51	0	0.97	0	0.03	0	1	0	0	0
	0.5	0	0.5	0	0.62	0	0.37	0	1	0	0	0	1	0	0	0
(4)	0.5	0	0.5	0	0.64	0	0.36	0	1	0	0	0	1	0	0	0
	0.3	0	0.7	0	0.36	0	0.64	0	0.07	0	0.93	0	0	0	1	0
	0.5	0	0.5	0	0.64	0	0.36	0	1	0	0	0	1	0	0	0
(5)	0.3	0	0.7	0	0.5	0	0.5	0	0.95	0	0.05	0	1	0	0	0
	0.3	0	0.7	0	0.5	0	0.5	0	0.95	0	0.05	0	1	0	0	0
	0.5	0	0.5	0	0.84	0	0.16	0	1	0	0	0	1	0	0	0
(6)	0.2	0	0.8	0	0.3	0	0.7	0	0.06	0	0.94	0	0	0	1	0
	0.3	0	0.7	0	0.51	0	0.49	0	1	0	0	0	1	0	0	0
	0.5	0	0.5	0	0.83	0	0.17	0	1	0	0	0	1	0	0	0
(7)	0.3	0.2	0.3	0.2	0.41	0.13	0.32	0.14	0.98	0	0.02	0	1	0	0	0
	0.3	0.2	0.3	0.2	0.41	0.13	0.32	0.14	0.98	0	0.02	0	1	0	0	0
	0.3	0.2	0.3	0.2	0.41	0.13	0.32	0.14	0.98	0	0.02	0	1	0	0	0
(8)	0.3	0.2	0.3	0.2	0.38	0.17	0.29	0.16	1	0	0	0	1	0	0	0
	0.25	0.25	0.25	0.25	0.35	0.20	0.25	0.20	1	0	0	0	1	0	0	0
	0.2	0.2	0.4	0.2	0.23	0.16	0.45	0.16	0.2	0	0.8	0	0	0	1	0

(c)

Fig. 12.10 The nonlinear operator produces labelings for the triangle in (a). (b) shows how the label weights are displayed, and (c) shows a number of cases (see text).

To put relaxation in terms of linear programming, we use the following translations.

- LABEL WEIGHT VECTORS $\Longrightarrow$ POINTS IN EUCLIDEAN N-SPACE. Each possible assignment of a label to an object is a *hypothesis*, to which a weight (supposition value) is to be attached. With N hypotheses, an N-vector of weights describes a labeling. We shall call this vector a (hypothesis or label) *weight vector*. For m labels and n objects, we need at most Euclidean nm-space.
- CONSTRAINTS $\Longrightarrow$ INEQUALITIES. *Constraints* are mapped into *linear inequalities* in hypothesis weights, by way of various identities like those of "fuzzy logic" [Zadeh 1965]. Each inequality determines an infinite half-space. The weight vectors within this half-space satisfy the constraint. Those outside do not. The convex solid that is the set intersection of all the half-spaces includes those weight vectors that satisfy all the constraints: each represents a "consistent" labeling. In linear programming terms, each such weight vector is a *feasible solution*. We thus have the usual geometric interpretation of the linear programming problem, which is to find the best (optimal) consistent (feasible) labeling (solution, or weight vector). Solutions should have *integer*-valued (1- or 0-valued) weights indicating convergence to actual labelings, not probabilistic ones such as those of Section 12.4.3, or the one shown in Fig. 12.10c, case 1.
- HYPOTHESIS PREFERENCES $\Longrightarrow$ PREFERENCE VECTOR. Often some hypotheses (label assignments) are preferred to others, on the basis of a priori knowledge, image evidence, and so on. To express this preference, make an N-dimensional *preference vector*, which expresses the relative importance (preference) of the hypotheses. Then
    - The *preference of a labeling* is the dot product of the preference vector and the weight vector (it is the sum for all hypotheses of the weight of each hypothesis times its preference).
    - The preference vector defines a *preference direction* in N-space. The optimal feasible solution is that one "farthest" in the preference direction. Let $\mathbf{x}$ and $\mathbf{y}$ be feasible solutions; they are N-dimensional weight vectors satisfying all constraints. If $\mathbf{z} = \mathbf{x} - \mathbf{y}$ has a component in the positive preference direction, then $\mathbf{x}$ is a better solution than $\mathbf{y}$, by the definition of the preference of a labeling.

  It is helpful for our intuition to let the preference direction define a "downward" direction in N-space as gravity does in our three-space. Then we wish to pick the lowest (most preferred) feasible solution vector.
- LABELING $\Longrightarrow$ OPTIMAL SOLUTION. The relaxation algorithm must solve the linear programming problem—find the best consistent labeling. Under the conditions we have outlined, the best solution vector occurs generally at a vertex of the N-space solid. This is so because usually a vertex will be the "lowest" part of the convex solid in the preference direction. It is a rare coincidence that the solid "rests on a face or edge," but when it does a whole edge or face of the solid contains equally preferred solutions (the preference direction is normal to

the edge or face). For integer solutions, the solid should be the convex hull of integer solutions and not have any vertices at noninteger supposition values.

The "simplex algorithm" is the best known solution method in linear programming. It proceeds from vertex to vertex, seeking the one that gives the optimal solution. The simplex algorithm is not suited to parallel computation, however, so here we describe another approach with the flavor of hill-climbing optimization. Basically, any such algorithm moves the weight vector around in N-space, iteratively adjusting weights. If they are adjusted one at a time, serial relaxation is taking place; if they are all adjusted at once, the relaxation is parallel iterative. The feasible solution solid and the preference vector define a "cost function" over all N-space, which acts like a potential function in physics. The algorithm tries to reach an optimum (minimum) value for this cost function. As with many optimization algorithms, we can think of the algorithm as trying to simulate (in N-space) a ball bearing (the weight vector) rolling along some path down to a point of minimum gravitational (cost) potential. Physics helps the ball bearing find the minimum; computer optimization techniques are sometimes less reliable.

Translating Constraints to Inequalities

The supposition values, or hypothesis weights, may be encoded into the interval [0, 1], with 0 meaning "false," 1 meaning "true." The extension of weights to the whole interval is reminiscent of "fuzzy logic," in which truth values may be continuous over some range [Zadeh 1965]. As in Section 12.4.3, we denote supposition values by $p(\cdot)$; H, A, B, and C are label assignment events, which may be considered as hypotheses that the labels are correctly assigned. $\bar{}$, $\vee$, $\wedge$, $\Longrightarrow$ and $\Longleftrightarrow$ are the usual logical connectives relating hypotheses. The connectives allow the expression of complex constraints. For instance, a constraint might be "Label x as 'y' if and only if z is labeled 'w' or q is labelled 'v'." This constraint relates three hypotheses: h_1: (x is "y"), h_2: (z is "w"), h_3: (q is "v"). The constraint is then $h_1 \Longleftrightarrow (h_2 \vee h_3)$.

Inequalities may be derived from constraints this way.

1. *Negation.* $p(H) = 1 - p(\bar{}(H))$.
2. *Disjunction.* The sums of weights of the disjunct are greater than or equal to one. $p(A \vee B \vee \ldots \vee C)$ gives the inequality $p(A) + p(B) + \ldots + p(C) \geqslant 1$.
3. *Conjunction.* These are simply separate inequalities, one per conjunct. In particular, a conjunction of disjunctions may be dealt with conjunct by conjunct, producing one disjunctive inequality per conjunct.
4. *Arbitrary expressions.* These must be put into conjunctive normal form (Chapter 10) by rewriting all connectives as $\wedge$'s and $\vee$'s. Then (3) applies.

As an example, consider the simple case of two hypotheses A and B, with the single constraint that $A \Longrightarrow B$. Applying rules 1 through 4 results in the following five inequalities in $p(A)$ and $p(B)$; the first four assure weights in [0, 1]. The fifth arises from the logical constraint, since $A \Longrightarrow B$ is the same as $B \vee \bar{}(A)$.

$$0 \leqslant p(A)$$
$$p(A) \leqslant 1$$
$$0 \leqslant p(B)$$
$$p(B) \leqslant 1$$
$$p(B) + (1 - p(A)) \geqslant 1 \quad \text{or} \quad p(B) \geqslant p(A)$$

These inequalities are shown in Fig. 12.11. As expected from the $\Longrightarrow$ constraint, optimal feasible solutions exist at: (1,1) or (A,B); (0,1) or $(\bar{}(A),B)$; (0,0) or $(\bar{}(A), \bar{}(B))$. Which of these is preferred depends on the preference vector. If both its components are positive, (A,B) is preferred. If both are negative, $(\bar{}(A), \bar{}(B))$ is preferred, and so on.

A Solution Method

Here we describe (in prose) a search algorithm that can find the optimal feasible solution to the linear programming problem as described above. The description makes use of the mechanical analogy of an N-dimensional solid of feasible solutions, oriented in N-space so that the preference vector induces a "downward" direction in space. The algorithm attempts to move the vector of hypothesis weights to the point in space representing the feasible solution of maximum preference. It should be clear that this is a point on the surface of the solid, and unless the preference vector is normal to a face or edge of the solid, the point is a unique "lowest" vertex.

To establish a potential that leads to feasible solutions, one needs a measure of the infeasibility of a weight vector for each constraint. Define the amount a vector violates a constraint to be zero if it is on the feasible side of the constraint hyperplane. Otherwise the violation is the normal distance of the vector to the hyperplane. If $\mathbf{h}_i$ is the coefficient vector of the i^{th} hyperplane (Appendix 1) and $\mathbf{w}$ the weight vector, this distance is

$$d_i = \mathbf{w} \cdot \mathbf{h}_i \tag{12.30}$$

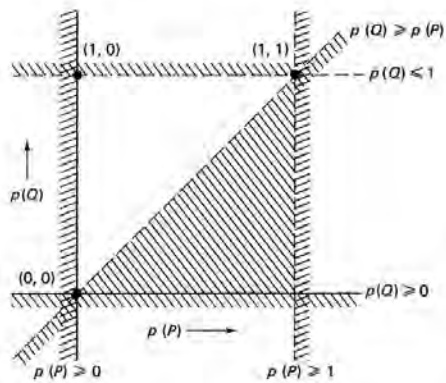


Fig. 12.11 The feasible region for two hypotheses A and B and the constraint A B. Optimal solutions may occur at the three vertices. The preferred vertex will be that one farthest in the direction of the preference vector, or lowest if the preference vector defines "down."

If we then define the infeasibility as

$$I = \sum_i \frac{d_i^2}{2} \tag{12.31}$$

then $\partial I / \partial d_i = d_i$ is the rate the infeasibility changes for changes in the violation. The force exerted by each constraint is proportional to the normal distance from the weight vector to the feasible region defined by that constraint, and tends to pull the weight vector onto the surface of the solid.

Now add a weak "gravity-like" force in the preference direction to make the weight vector drift to the optimal vertex. At this point an optimization program might perform as shown in Fig. 12.12.

Figure 12.12 illustrates a problem: The forces of preference and constraints will usually dictate a minimum potential outside the solid (in the preference direction). Fixes must be applied to force the weight vector back to the closest (presumably the optimum) vertex. One might round high weights to 1 and low ones to 0, or add another local force to draw vectors toward vertices.

Examples

An algorithm based on the principles outlined in the preceeding section was successfully used to label scenes of "puppets" such as Fig. 12.13 with body parts [Hinton 1979].

The discrete, consistency-oriented version of line labeling may be extended to incorporate the notion of optimal labelings. Such a system can cope with the explosive increase in consistent labelings that occurs if vertex labels are included for cases of missing lines, accidental alignment, or "two-dimensional" objects such as folded paper. It allows modeling of the fact that human beings do not "see" all possible interpretations of scenes with accidental alignments. If labelings are given

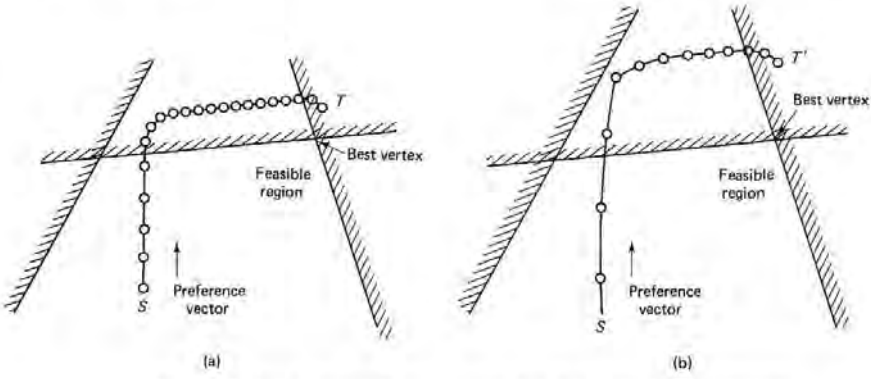
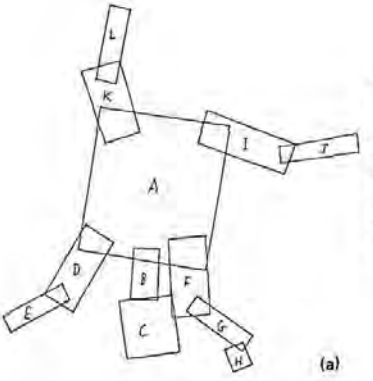


Fig. 12.12 In (a), the weight vector moves from S to rest at T, under the combined influence of the preferences and the violated constraints. In (b), convergence is speeded by making stronger preferences, but the equilibrium is farther away from the optimal vertex.



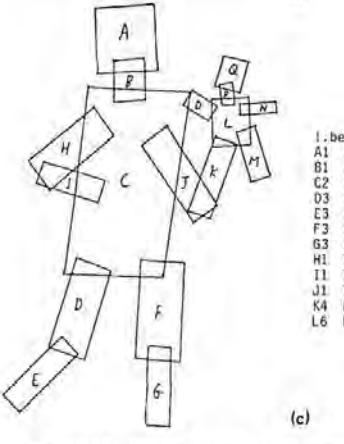

```
!.bestset;
A1  BOT TRUNK  NECK B1  UPPERARM D2 F2  THIGH I3 K2
B1  BOT NECK  HEAD C1  TRUNK A1
C1  BOT HEAD  NECK B1
D2  TOP UPPERARM  TRUNK A1  LOWERARM E4
E4  TOP LOWERARM  UPPERARM D2  HAND -
F2  TOP UPPERARM  TRUNK A1  LOWERARM G2
G2  TOP LOWERARM  UPPERARM F2  HAND H2
H2  TOP HAND  LOWERARM G2
I3  TOP THIGH  TRUNK A1  CALF J4
J4  BOT CALF  THIGH I3  FOOT -
K2  BOT THIGH  TRUNK A1  CALF L4
L4  BOT CALF  THIGH K2  FOOT -
```

(a)

```
!trytointerpret [trunk as upright importance=1];
!trytointerpret [thigh as upright importance=1];

!.bestset;
A2  TOP TRUNK  NECK -  UPPERARM I2 K1  THIGH D3 F3
B1  BOT NECK  HEAD C1  TRUNK -
C1  BOT HEAD  NECK B1
D3  TOP THIGH  TRUNK A2  CALF E3
E3  TOP CALF  THIGH D3  FOOT -
F3  TOP THIGH  TRUNK A2  CALF E3
G3  TOP CALF  THIGH F3  FOOT H1
H1  TOP FOOT  CALF G3
I2  TOP UPPERARM  TRUNK A2  LOWERARM J3
J3  BOT LOWERARM  UPPERARM I2  HAND -
K1  BOT UPPERARM  TRUNK A2  LOWERARM L3
L3  BOT LOWERARM  UPPERARM K1  HAND -
```

(b)




```
!.bestset;
A1  TOP HEAD  NECK B1
B1  TOP NECK  HEAD A1  TRUNK C2
C2  TOP TRUNK  NECK B1  UPPERARM H1 J1  THIGH D3 F3
D3  TOP THIGH  TRUNK C2  CALF E3
E3  TOP CALF  THIGH D3  FOOT -
F3  TOP THIGH  TRUNK C2  CALF G3
G3  TOP CALF  THIGH F3  FOOT-
H1  TOP UPPERARM  TRUNK C2  LOWERARM I1
I1  TOP LOWERARM  UPPERARM H1  HAND -
J1  TOP LOWERARM  TRUNK C2  LOWERARM K4
K4  BOT LOWERARM  UPPERARM J1  HAND L6
L6  BOT HAND  LOWERARM K4
```

(c)

Fig. 12.13 Puppet scenes interpreted by linear programming relaxation. (a) shows an upside down puppet. (b) is the same input along with preferences to interpret the trunk and thighs as upright; these result in an interpretation with trunk and neck not connected. In (c), the program finds only the "best" puppet, since it was only expecting one.

costs, then one can include labels for missing lines and accidental alignment as high-cost labels, rendering them usable but undesirable. Also, in a scene-analysis system using real data, local evidence for edge appearance can enhance the a priori likelihood that a line should bear a particular label. If such preferences can be extracted along with the lines in a scene, the evidence can be used by the line labeling algorithm.

The inconsistency constraints for line labels may be formalized as follows. Each line and vertex has one label in a consistent labeling; thus for each line L and vertex J,

$$\sum_{\text{all line labels}} p(L \text{ has label LLABEL}) = 1 \tag{12.32}$$

$$\sum_{\text{all vertex labels}} p(J \text{ has label VLABEL}) = 1 \tag{12.33}$$

Of course, the VLABELS and LLABELS in the above constraints must be forced to be compatible (if L has LLABEL, JLABEL must agree with it). For a line L and a vertex J at its end,

$$p(L \text{ has LLABEL}) = \sum_{\substack{\text{all VLABELS} \\ \text{giving LLABEL to } L}} p(J \text{ has label VLABEL}) \tag{12.34}$$

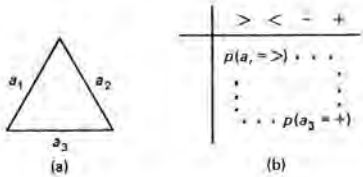
This constraint also enforces the coherence rule (a line may not change its label betwen vertices).

Using these constraints, linear programming relaxation labeled the triangle example of Fig. 12.7 as shown in Fig. 12.14, which shows three cases.

1. Preference 0.5 for each of the three junction label assignments (hypotheses) corresponding to the floating triangle, 0 preference for all other junction and line label hypotheses: converges to floating triangle.
2. Like (1), but with equal preferences given to the junction labels for the triangular hole interpretation, 0 to all other preferences.
3. Preference 3 to the convex edge label for a 2 overrides the three preferences of 1/2 for the floating triangle of case (1). All preferences but these four were 0.

Some Extensions

The translation of constraints to inequalities described above does not guarantee that they produce a set of half-spaces whose intersection is the convex hull of the feasible integer solutions. They can produce "noninteger optima," for which supposition values are not forced to 1 or 0. This is reminiscent of the behavior of the linear relaxation operator of Section 12.4.3, and may not be objectionable. If it is, some effort must be expended to cope with it. Here is an example



Case	After 10 iterations				After 20 iterations				After 30 to 40 iterations			
(1)	0.65	0.22	0.01	0.14	0.90	0.07	0	0.04	0.99	0	0	0
	0.65	0.22	0.01	0.14	0.90	0.07	0	0.04	0.99	0	0	0
	0.65	0.22	0.01	0.14	0.90	0.07	0	0.04	0.99	0	0	0
(2)	0.39	0.89	0	0	0.14	0.95	0	0	0	0.99	0	0
	0.39	0.89	0	0	0.14	0.95	0	0	0	0.99	0	0
	0.39	0.89	0	0	0.14	0.95	0	0	0	0.99	0	0
(3)	0.56	0.48	0	0.05	0.81	0.23	0	0	0.99	0	0	0
	0	0.34	0	0.99	0	0.15	0	0.99	0	0	0	0.99
	0.56	0.48	0	0.05	0.81	0.23	0	0	0.99	0	0	0

(c)

Fig. 12.14 As in Fig. 12.10, the triangle of (a) is to be assigned labels, and the changing label weights are shown for three cases in (c) using the format of (b). Supposition values for junction labels were used as well, but are not shown. All initial supposition values were 0.

of the problem. Assume three logical constraints, $\neg(A \wedge B)$, $\neg(B \wedge C)$, and $\neg(C \wedge A)$. Suppose A, B, and C have equal preferences of unity (the preference vector is $(1, 1, 1)$). Translating the constraints yields

$$\begin{aligned} p(A) + p(B) &\leqslant 1 \\ p(B) + p(C) &\leqslant 1 \\ p(C) + p(A) &\leqslant 1 \end{aligned} \tag{12.35}$$

The best feasible solution has a total preference of 1½, and is

$$p(A) = p(B) = p(C) = \tfrac{1}{2} \tag{12.36}$$

Here the "best" solution is outside the convex hull of the integer solutions (Fig. 12.15).

The basic way to ensure integer solutions is to use stronger constraints than those arising from the simple rules given above. These may be introduced at first, or when some noninteger optimum has been reached. These stronger constraints are called *cutting planes*, since they cut off the noninteger optima vertices. In the example above, the obvious stronger constraint is

$$p(A) + p(B) + p(C) \leqslant 1 \tag{12.37}$$

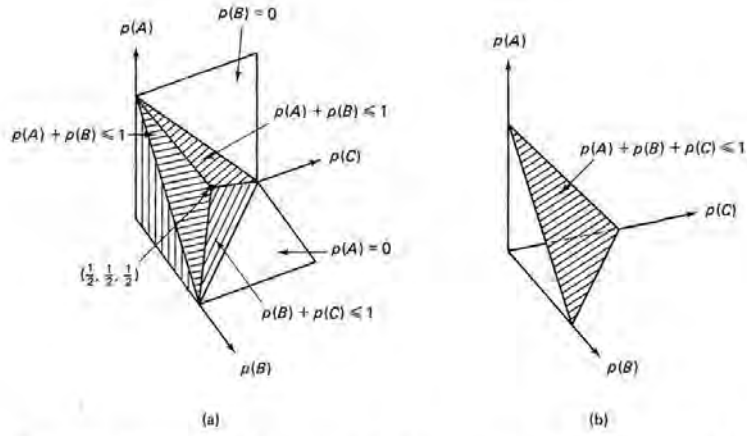


Fig. 12.15 (a) shows part of the surface of the feasible solid with constraints $\neg(A \,\&\, B)$, $\neg(B \,\&\, C)$, $\neg(C \,\&\, A)$, and the non-integer vertex where the three halfspaces intersect. (b) shows a cutting plane corresponding to the constraint "at most one of A, B, or C" that removes the non-integer vertex.

which says that at most one of A, B, and C is true (this is a logical consequence of the logical constraints). Such cutting planes can be derived as needed, and can be guaranteed to eliminate all noninteger optimal vertices in a finite number of cuts [Gomory 1968; Garfinkel and Nemhauser 1972]. Equality constraints may be introduced as two inequality constraints in the obvious way: This will constrain the feasible region to a plane.

Suppose that one desires "weak rules," which are usually true but which can be broken if evidence demands it? For each constraint arising from such a rule, add a hypothesis to represent the situation where the rule is broken. This hypothesis is given a negative preference depending on the strength of the rule, and the constraint enhanced to include the possibility of the broken rule. For example, if a weak rule gives the constraint $P \vee Q$, create a hypothesis H equivalent to $\neg(P \vee Q) = (\neg(P) \wedge \neg(Q))$, and replace the constraint with $P \vee Q \vee H$. Then by "paying the cost" of the negative preference for H, we can have neither P nor Q true.

Hypotheses can be created as the algorithm proceeds by having demon-like "generator hypotheses." The demon watches the supposition value of the generator, and when it becomes high enough, runs a program that generates explicit hypotheses. This is clearly useful; it means that all possible hypotheses do not need to be generated in advance of any scene investigation. The generator can be given a preference equal to that of the best hypotheses that it can generate.

Relaxation sometimes should determine a real number (such as the slope of a line) instead of a truth value. A generator-like technique can allow the method to refine the value of real-valued hypotheses. Basically, the idea is to assign a (Boolean-valued) generator hypothesis to a range of values for the real value to be

determined. When this generator triggers, more hypotheses are generated to get a finer partition of the range, and so on.

The enhancements to the linear programming paradigm of relaxation give some idea of the flexibility of the basic idea, but also reveal that the method is not at all cut-and-dried, and is still open to basic investigation. One of the questions about the method is exactly how to take advantage of parallel computation capabilities. Each constraint and hypothesis can be given its own processor, but how should they communicate? Also, there seems little reason to suppose that the optimization problems for this form of relaxation are any easier than they are for any other multidimensional search, so the method will encounter the usual problems inherent in such optimization. However, despite all these technical details and problems of implementation, the linear programming paradigm for the relaxation computation is a coherent formalization of the process. It provides a relatively "classical" context of results and taxonomy of problems [Hummel and Zucker 1980].

12.5 ACTIVE KNOWLEDGE

Active knowledge systems [Freuder 1975] are characterized by the use of procedures as the elementary units of knowledge (as opposed to propositions or data base items, for instance). We describe how active knowledge might work, because it is a logical extreme of the procedural implementation of propositions. In fact, this style of control has not proven influential; some reasons are given below.

Active knowledge is notionally parallel and heterarchical. Many different procedures can be active at the same time depending on the input. For this reason active knowledge is more easily applied to belief maintenance than to planning; it is very difficult to organize sequential activity within this discipline. Basically, each procedure is responsible for a "chunk" of knowledge, and knows how to manage it with respect to different visual inputs. Control in an active knowledge system is completely distributed. Active knowledge can also be viewed as an extension of the constraint relaxation problem; powerful procedures can make arbitrary detailed tests of the consistency between constraints.

Each piece of active knowledge (program module) knows which other modules it depends on, which depend on it, which it can complain to, and so forth. Thus the choice of "what to do next" is contained in the modules and is not made by an exterior executive.

We describe HYPER, a particular active knowledge system design which illustrates typical properties of active knowledge [Brown 1975]. HYPER provides a less structured mechanism for construction and exploration of hypotheses than does LP-relaxation. Using primitive control functions of the system, the user may write programs for establishing hypotheses and for using the conclusions so reached. The programs are "procedurally embedded" knowledge about a problem domain (e.g. how events relate one to another, what may be conjectured or inferred from a clue, or how one might verify a hypothesis).

When HYPER is in use on a particular task in a domain, hypotheses are created, or instantiated, on the basis of low-level input, high-level beliefs, or any

reason in between. The process of establishing the initial hypotheses leads to a propagation of activity (creation, verification, and disconfirmation of hypotheses). Activation patterns will generally vary with the particular task, in heterarchical fashion. A priority mechanism can rank hypotheses in importance depending on the data that contribute to them. Generally, the actions that occur are conditioned by previous assumptions, the data, the success of methods, and other factors. HYPER can be used for planning applications and for multistep vision processing as well as inference (procedures then should generate parallel activity only under tight control). We shall thus allow HYPER to make use of a context-oriented data base (Section 13.1.1). It will use the context mechanism to implement "alternative worlds" in which to reason.

12.5.1 Hypotheses

A HYPER hypothesis is the attribution of a predicate to some arguments; its name is always of the form (PREDICATE ARGUMENTS). Sample hypothesis names could be (HEAD-SHAPED REGION1), (ABOVE A B), (TRIANGLE (X1,Y1) (X2,Y2) (X3,Y3)). A hypothesis is represented as a data structure with four components; the *status*, *contents*, *context*, and *links* of the hypothesis.

The *status* represents the state of the HYPER's knowledge of the truth of the hypothesis; it may be T(rue), F(alse), (in either case the hypothesis has been *established*) or P(ending). The *contents* are arbitrary; hypotheses are not just truth-valued assertions. The hypothesis was asserted in the data-base context given in *context*. The *links* of a hypothesis H are pointers to other hypotheses that have asked that H be established because they need H's contents to complete their own computations.

12.5.2 HOW-TO and SO-WHAT Processes

Two processes are associated with every predicate P which appears as the predicate of a hypothesis. Their names are (HOW-TO P) and (SO-WHAT P). In them is embedded the procedural knowledge of the system which remains compiled in from one particular task to another in a problem domain. (HOW-TO P) expresses how to establish the hypothesis (P arguments). It knows what other hypotheses must be established first, the computations needed to establish (P arguments), and so forth. It has a backward-chaining flavor. Similarly, (SO-WHAT P) expresses the consequences of knowing P: what hypotheses could possibly now be established using the contents of (P arguments), what alternative hypotheses should be explored if the status of (P arguments) is *F*, and so on. The feeling here is of forward chaining.

12.5.3 Control Primitives

HYPER hypotheses interact through *primitive control statements*, which affect the investigation of hypotheses and the ramification of their consequences. The primi-

tives are used in HOW-TO and SO-WHAT programs together with other general computations. Most primitives have an argument called priority, which expresses the reliability, urgency, or importance of the action they produce, and is used to schedule processes in a nonparallel computing environment (implemented as a priority job queue [Appendix 2]). The primitives are GET, AFFIRM, DENY, RETRACT, FAIL, WONDERIF, and NUDGE.

GET is to ascertain or establish the status and contents of a hypothesis. It takes a hypothesis H and priority PRI as arguments and returns the status and contents of the hypothesis. If H's status is T or F at the time of execution of the statement, the status and contents are returned immediately. If the status is P (pending), or if H has not been created yet, the current HOW-TO or SO-WHAT program calling GET (call it CURPROG) is exited, the proper HOW-TO job (i.e., the one that deals with H's predicate) is run at priority PRI with argument H, and a link is planted in H back to CURPROG. When H is established, CURPROG will be reactivated through the link mechanism.

AFFIRM is to assert a hypothesis as true with some contents. AFFIRM(H,CONT,PRI) sets H's status to T, its contents to CONT, activates its linked programs and then executes the proper SO-WHAT program on it. The newly activated SO-WHAT programs are performed with priority PRI.

DENY is to assert that a hypothesis with some contents is false. DENY(H,CONT,PRI) is like AFFIRM except that no activation though links occurs, and the status of H is of course set to F.

ASSUME is to assert a hypothesis as true hypothetically. ASSUME(H,CONT,PRI) uses the data base context mechanism to create a new context in which H is AFFIRMED; the original context in which the ASSUME command is given is preserved in the context field of H. H itself is stored into a context-dependent item named LASTASSUMED; this corresponds to remembering a decision point in PLANNER. By using the information in LASTASSUMED and the primitive FAIL (see below), simple backtracking can take place in a tree of contexts.

RETRACT(H) establishes as false a hypothesis that was previously ASSUMEd. RETRACT is always carried out at highest priority, on the principle that it is good to leave the context of a mistaken assumption as quickly as possible. Information (including the name of the context being exited) is transmitted back to the original context in which H was ASSUMEd by passing it back in the fields of H.

FAIL just RETRACTs the hypothesis that is the value of the item LASTASSUMED in the present context.

WONDERIF is to pass suggested contents to HOW-TO processes for verification. It can be useful if verifying a value is easier than computing it from scratch, and is the primitive that passes substantive suggestions. WONDERIF(H1, CONT, H2, PRI) approximates the notion "H2 wonders if H1 has contents CONT."

NUDGE is to wake up HOW-TO programs. NUDGE(H,PRI) runs the HOW-TO program on H with priority PRI. It is used to awaken hypotheses that might be able to use information just computed. Typically it is a SO-WHAT pro-

gram that NUDGEs others, since the SO-WHAT program is responsible for using the fact that a hypothesis is known.

12.5.4 Aspects of Active Knowledge

The active knowledge style of computation raises a number of questions or problems for its users.

A hypothesis whose contents may attain a large range can be established for some contents and thus express a perfectly good fact (e.g., that a given location of an x-ray does not contain evidence for a tumor) but such a fact is usually of little help when we want to reason about the predicate (about the location of tumors). The SO-WHAT program for a predicate should be written so as to draw conclusions from such negative facts if possible, and from the conclusions endeavor to establish the hypothesis as true for some contents. Usually, therefore, it would set the status of the hypothesis back to P and initiate a new line of attack, or at its discretion abandon the effort and start an entirely new line of reasoning.

Priorities

A major worry with the scheme as described is that priorities are used to schedule running of HOW-TO and SO-WHAT processes, not to express the importance (or supposition value) of the hypotheses. The hypothesis being investigated has no way to communicate how important it is to the program that operates on it, so it is impossible to accumulate importance through time. A very significant fact may lie ignored because it was given to a self-effacing process that had no way of knowing it had been handed something out of the ordinary.

The obvious answer is to make a supposition value a field of the hypothesis, like its status or contents—a hypothesis should be given a measure of its importance. This value may be used to compute execution priorities for jobs involving it. This solution is used in some successful systems [Turner 1974].

Structuring Knowledge

One has a wide choice in how to structure the "theory" of a complex problem in terms of HYPER primitives, predicates, arguments, and HOW-TO and SO-WHAT processes. The set of HOW-TO and SO-WHAT processes specify the complete theory of the tasks to be performed; HYPER encourages one to consider the interrelations between widely separated and distinct-sounding facts and conjectures about a problem, and the structure it imposes on a problem is minimal.

Since HOW-TO and SO-WHAT processes make explicit references to one another via the primitives, they are not "modular" in the sense that they can easily be plugged in and unplugged. If HOW-TO and SO-WHAT processes are invoked by patterns, instead of by names, some of the edge is taken off this criticism. Removing a primitive from a program could modify drastically the avenues of activation, and the consequences of such a modification are sometimes hard to foresee in a program that logically could be running in parallel.

Writing a large and effective program for one domain may not help to write a program for another domain. New problems of segmenting the theory into predicates, and quantifying their interactions via the primitives, setting up a priority

structure, and so forth will occur in the new domain, and it seems quite likely that little more than basic utility programs will carry over between domains.

EXERCISES

12.1 In the production system example, write a production that specifies that blue regions are sky using the opponents color notation. How would you now deal with blue regions that are lakes (a) in the existing color-only system; (b) in a system which has surface orientation information?

12.2 This theorem was posed as a challenge for a clausal automatic theorem prover [Henschen et al. 1980]. It is obviously true: what problems does it present?

$$\begin{aligned}&\{[(\exists x)(\forall y)(P(x) \iff P(y))]\\ \iff &[[(\exists x)Q(x)] \iff [(\forall y)(P(y))]]] \iff\\ &\{[(\exists x)(\forall y)(Q(x) \iff Q(y))]\\ \iff &[[(\exists x)P(x)] \iff [(\forall y)(Q(y))]]\end{aligned}$$

12.3 Prove that the operator of Eq. (12.18) takes probability vectors into probability vectors, thus deriving the reason for Eq. (12.19).

12.4 Verify (12.23).

12.5 How do the c_{ij} of (12.18) affect the labeling? What is their semantics?

12.6 If events X and Y always co-occur, then $p(X, Y) = p(X) = p(Y)$. What is the correlation in this case? If X and Y never co-occur, what values of $p(X)$ and $p(Y)$ produce a minimum correlation? If X and Y are independent, how is $p(X, Y)$ related to $p(X)$ and $p(Y)$? What is the value of the correlation of independent X and Y?

12.7 Complete Table 12.3.

12.8 Use only the labels of Fig. 12.9b and c to compute covariances in the manner of Table 12.3. What do you conclude?

12.9 Show that Eq. (12.29) preserves the important properties of the weight vectors.

12.10 Think of some rival normalization schemes to Eq. (12.29) and describe their properties.

12.11 Implement the linear and nonlinear operators of Section 12.4.3 and 12.4.4 and investigate their properties. Include your ideas from Exercise 12.10.

12.12 Show a case that the nonlinear operator of Eq. (12.29) assigns nonzero weights to maximally incompatible labels (those with $r_{ij} = -1$).

12.13 How can a linear programming relaxation such as the one outlined in sec. 12.4.5 cope with faces or edges of the feasible solution solid that are normal to the preference direction, yielding several solutions of equal preference?

12.14 In Fig. 12.11, what (P, Q) solution is optimal if the preference vector is $(1, 4)$? $(4, 1)$? $(-1, 1)$? $(1, -1)$?

REFERENCES

AIKINS, J. S. "Prototypes and production rules: a knowledge representation for computer consultations." Ph.D. dissertation, Computer Science Dept., Stanford Univ., 1980.

BAJCSY, R. and A. K. JOSHI. "A partially ordered world model and natural outdoor scenes." In *CVS*, 1978.

BARROW, H. G. and J. M. TENENBAUM. "MSYS: a system for reasoning about scenes." Technical Note 121, AI Center, SRI International, March 1976.

BRACHMAN, R. J. "On the epistemological status of semantic networks." In *Associative Networks: Representation and Use of Knowledge by Computers*, N. V. Findler (Ed.). New York: Academic Press, 1979, 3–50.

BROWN, C. M. "The HYPER system." DAI Working Paper 9, Dept. of Artificial Intelligence, Univ. Edinburgh, July 1975.

BUCHANAN, B. G. and E. A. FEIGENBAUM. "DENDRAL and meta-DENDRAL: their applications dimensions." *Artificial Intelligence 11*, 2, 1978, 5–24.

BUCHANAN, B. G. and T. M. MITCHELL. "Model-directed learning of production rules." In *Pattern Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (Eds.). New York: Academic Press, 1978.

COLLINS, A. "Fragments of a theory of human plausible reasoning." *Theoretical Issues in Natural Language Processing-2*, Univ. Illinois at Urbana-Champaign, July 1978, 194–201.

DAVIS, R. and J. KING. "An overview of production systems." AIM-271, Stanford AI Lab, October 1975.

DAVIS, L. S. and A. ROSENFELD. "Applications of relaxation labelling 2. Spring-loaded template matching." Technical Report 440, Computer Science Center, Univ. Maryland, 1976.

DELIYANNI, A. and R. A. KOWALSKI. "Logic and semanatic networks." *Comm. ACM 22*, 3, March 1979, 184–192.

ERMAN, L. D. and V. R. LESSER. "A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge." *Proc.*, 4th IJCAI, September 1975, 483–490.

FELDMAN, J. A. and Y. YAKIMOVSKY. "Decision theory and artificial intelligence: I. A semantics-based region analyser." *Artificial Intelligence 5*, 4, 1974, 349–371.

FIKES, R. E. "Knowledge representation in automatic planning systems." In *Perspectives on Computer Science*, A. Jones (Ed). New York: Academic Press, 1977.

FIKES, R. E. and N. J. NILSSON. "STRIPS: a new approach to the application of theorem proving to problem solving." *Artificial Intelligence 2*, 3/4, 1971, 189–208.

FREUDER, E. C. "A computer system for visual recognition using active knowledge." Ph.D. dissertation, MIT, 1975.

FREUDER, E. C. "Synthesizing constraint expressions." *Comm. ACM 21*, 11, November 1978, 958–965.

GARFINKEL, R. S. and G. L. NEMHAUSER. *Integer Programming*. New York: Wiley, 1972.

GOMORY, R. E. "An algorithm for integer solutions to linear programs." *Bull. American Mathematical Society 64*, 1968, 275–278.

HARALICK, R. M. "The characterization of binary relation homomorphisms." *International J. General Systems 4*, 1978, 113–121.

HARALICK, R. M. and J. S. KARTUS. "Arrangements, homomorphisms, and discrete relaxation." *IEEE Trans. SMC 8*, 8, August 1978, 600–612.

HARALICK, R. M. and L. G. SHAPIRO. "The consistent labeling problem: Part I." *IEEE Trans. PAMI 1*, 2, April 1979, 173–184.

HARALICK, R. M., L. S. DAVIS, and A. ROSENFELD. "Reduction operations for constraint satisfaction." *Information Sciences 14*, 1978, 199–219.

HAYES, P. J. "In defense of logic." *Proc.*, 5th IJCAI, August 1977, 559–565.

HAYES, P. J. "Naive physics: ontology for liquids." Working paper, Institute for Semantic and Cognitive Studies, Geneva, 1978a.

HAYES, P. J. "The naive physics manifesto." Working paper, Institute for Semantic and Cognitive Studies, Geneva, 1978b.

HAYES, P. J. "The logic of frames." *The Frame Reader.* Berlin: DeGruyter, in press, 1981.

HENDRIX, G. G. "Encoding knowledge in partitioned networks." In *Associative Networks: Representation and Use of Knowledge by Computers*, N. V. Findler (Ed.). New York: Academic Press, 1979, 51–92.

HENSCHEN, L., E. LUSK, R. OVERBEEK, B. SMITH, R. VEROFF, S. WINKER, and L. WOS. "Challenge Problem 1." *SIGART Newsletter 72*, July 1980, 30–31.

HERBRAND, J. "Recherches sur la théorie de la démonstration." *Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III, Sciences Mathématiques et Physiques, 33*, 1930.

HEWITT, C. "Description and theoretical analysis (using schemata) of PLANNER" (Ph.D. dissertation). AI-TR-258, AI Lab, MIT, 1972.

HINTON, G. E. "Relaxation and its role in vision." Ph.D. dissertation, Univ. Edinburgh, December 1979.

HUMMEL, R. A. and S. W. ZUCKER. "On the foundations of relaxation labelling processes." TR-80-7, Computer Vision and Graphics Lab, Dept. of Electrical Engineering, McGill Univ., July 1980.

KOWALSKI, R. A. "Predicate logic as a programming language." *Information Processing 74.* Amsterdam: North-Holland, 1974, 569–574.

KOWALSKI, R. A. *Logic for Problem Solving.* New York: ElsevierNorth-Holland (AI Series), 1979.

LINDSAY, R. K., B. G. BUCHANAN, E. A. FEIGENBAUM, and J. LEDERBERG. *Applications of Artificial Intelligence to Chemistry: The DENDRAL Project.* New York: McGraw-Hill, 1980.

LOVELAND, D. "A linear format for resolution." *Proc.*, IRIA 1968 Symp. on Automatic Demonstration, Versailles, France. New York: Springer-Verlag, 1970.

LOVELAND, D. *Automated Theorem Proving: A Logical Basis.* Amsterdam: North-Holland, 1978.

MCCARTHY, J. "Circumscription induction—a way of jumping to conclusions." Unpublished report, Stanford AI Lab, 1978.

MCCARTHY, J. and P. J. HAYES. "Some philosophical problems from the standpoint of artificial intelligence." In *MI4*, 1969.

MCDERMOTT, D. "The PROLOG phenomenon." *SIGART Newsletter 72*, July 1980, 16–20.

MENDELSON, E. *Introduction to Mathematical Logic.* Princeton, NJ: D. Van Nostrand, 1964.

MINSKY, M. L. "A framework for representing knowledge." In *PCV*, 1975.

NEWELL, A., J. SHAW, and H. SIMON. "Empirical explorations of the logic theory machine." In *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.). New York: McGraw-Hill, 1963.

NILSSON, N. J. *Problem-Solving Methods in Artificial Intelligence.* New York: McGraw-Hill, 1971.

NILSSON, N. J. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga, 1980.

REITER, R. "On reasoning by default." *Theoretical Issues in Natural Language Processing-2*, Univ. Illinois at Urbana-Champaign, July 1978, 210–218.

ROBINSON, J. A. "A machine-oriented logic based on the resolution principle." *J. ACM 12*, 1, January 1965, 23–41.

ROSENFELD, A., R. A. HUMMEL and S. W. ZUCKER. "Scene labelling by relaxation operations." *IEEE Trans. SMC 6*, 1976, 420.

RYCHNER, M. "An instructable production system: basic design issues." In *Pattern Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (Eds.). New York: Academic Press, 1978.

SHORTLIFFE, E. H. *Computer-Based Medical Consultations: MYCIN.* New York: American Elsevier, 1976.

SLOAN, K. R. "World model driven recognition of natural scenes." Ph.D. dissertation, Moore School of Electrical Engineering, Univ. Pennsylvania, June 1977.

SLOAN, K. R. and R. BAJCSY. "World model driven recognition of outdoor scenes." TR40, Computer Science Dept., Univ. Rochester, September 1979.

SUSSMAN, G. J. and D. MCDERMOTT. "Why conniving is better than planning." AI Memo 255, AI Lab, MIT, 1972.

TURNER, K. J. "Computer perception of curved objects using a television camera." Ph.D. dissertation, School of Artificial Intelligence, Univ. Edinburgh, 1974.

WARREN, H. D., L. PEREIRA, and F. PEREIRA. "PROLOG: The language and its implementation compared with LISP." *Proc.*, Symp. on Artificial Intelligence and Programming Languages, SIGPLAN/SIGART, 1977; *SIGPLAN Notices 12*, 8, August 1977, 109–115.

WATERMAN, D. A. and F. HAYES-ROTH (Eds.). *Pattern-Directed Inference Systems.* New York: Academic Press, 1978.

WINOGRAD, T. "Extended inference modes in reasoning by computer systems." *Proc.*, Conf. on Inductive Logic, Oxford Univ., August 1978.

ZADEH, L. "Fuzzy sets." *Information and Control 8*, 1965, 338–353.

ZUCKER, S. W. "Relaxation labelling and the reduction of local ambiguities." Technical Report 451, Computer Science Dept., Univ. Maryland, 1976.

Goal Achievement 13

Goal Achievement and Vision

Goals and plans are important for visual processing.

- Some skilled vision actually is like problem solving.
- Vision for information gathering can be part of a planned sequence of actions.
- Planning can be a useful and efficient way to guide many visual computations, even those that are not meant to imply "conscious" cognitive activity.

The artificial intelligence activity often called *planning* traditionally has dealt with "robots" (real or modeled) performing actions in the real world. Planning has several aspects.

- Avoid nasty "subgoal interactions" such as getting painted into a corner.
- Find the plan with optimal properties (least risk, least cost, maximized "goodness" of some variety).
- Derive a sequence of steps that will achieve the goal from the starting situation.
- Remember effective action sequences so that they may be applied in new situations.
- Apply planning techniques to giving advice, presumably by simulating the advisee's actions and making the next step from the point they left off.
- Recover from errors or changes in conditions that occur in the middle of a plan.

Traditional planning research has not concentrated on plans with information gathering steps, such as vision. The main interest in planning research has been the expensive and sometimes irrevocable nature of actions in the world. Our goal is to give a flavor of the issues that are pursued in much more detail in the planning

literature [Nilsson 1980; Tate 1977; Fahlman 1974; Fikes and Nilsson 1971; Fikes et al. 1972a; 1972b; Warren 1974; Sacerdoti 1974; 1977; Sussman 1975].

Planning concerns an active agent and its interaction with the world. This conception does not fit with the idea of vision as a passive activity. However, one claim of this book is that much of vision is a constructive, active, goal-oriented process, replete with uncertainty. Then a model of vision as a sequence of decisions punctuated by more or less costly information gathering steps becomes more compelling. Vision often is a sequential (recursive, cyclical) process of alternating information gathering and decision making. This paradigm is quite common in computer vision [Shirai 1975; Ballard 1978; Mackworth 1978; Ambler et al. 1975]. However, the formalization of the process in terms of minimizing cost or maximizing utility is not so common [Feldman and Sproull 1977; Ballard 1978; Garvey 1976]. This section examines the paradigms of planning, evaluating plans with costs and utilities, and how plans may be applied to vision processing.

13.1 SYMBOLIC PLANNING

In artificial intelligence, planning is usually a form of problem-solving activity involving a formal "simulation" of a physical world. (Planning, theorem proving, and state-space problem solving are all closely related.) There is an agent (the "robot") who can perform actions that transform the state of the simulated world. The robot planner is confronted with an initial world state and a set of goals to be achieved. Planning explores world states resulting from actions, and tries to find a sequence of actions that achieves the goals. The states can be arranged in a tree with initial state as the root, and branches resulting from applying different actions in a state. Planning is a search through this tree, resulting in a path or sequence of actions, from the root to a state in which the goals are achieved. Usually there is a metric over action sequences; the simplest is that there be as few actions as possible. More generally (Section 13.2), actions may be assigned some cost which the planner should minimize.

13.1.1 Representing the World

This section illustrates planning briefly with a classical example—block stacking. In one simple form there are three blocks initially stacked as shown on the left in Fig. 13.1, to be stacked as shown.

This task may be "formalized" [Bundy 1978] using only the symbolic objects Floor, A, B, and C. (A formalization suitable for a real automated planner must be much more careful about details than we shall be). Assume that only a single block can be picked up at a time. Necessary predicates are CLEAR(X) which is true if a block may be put directly on X and which must be true before X may be picked up, and ON(X, Y), which is true if X is resting directly on Y. Let us stipulate that the Floor is always CLEAR, but otherwise if ON(X, Y) is true, Y is not CLEAR. Then the initial situation in Fig. 13.1 is characterized by the following assertions.

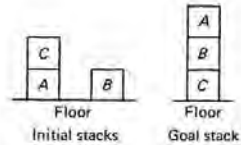


Fig. 13.1 A simple block stacking task.

INITIAL STATE: ON(C,A), ON(A, Floor), ON(B, Floor),
CLEAR(C), CLEAR(B), CLEAR(Floor)

The goal state is one in which the following two assertions are true.

GOAL ASSERTIONS: ON(A,B), ON(B,C)

With only these rules, the formalization of the block stacking world yields a very "loose" semantics. (The task easily translates to sorting integers with some restrictions on operations, or to the "seriation" task of arranging blocks horizontally in order of size, or a host of others.)

Actions transform the set of assertions describing the world. For problems of realistic scale, the representation of the tree of world states is a practical problem. The issue is one of maintaining several coexisting "hypothetical worlds" and reasoning about them. This is another version of the frame problem discussed in sec. 12.1.6. One way to solve this problem is to give each assertion an extra argument, naming the hypothetical world (usually called a situation [Nilsson 1980; McCarthy and Hayes 1969]) in which the assertion holds. Then actions map situations to situations as well as introducing and changing assertions.

An equivalent way to think about (and implement) multiple, dependent, hypothetical worlds is with a tree-structured *context-oriented data base*. This idea is a general one that is useful in many artificial intelligence applications, not just symbolic planning. Such data bases are included in many artificial intelligence languages and appear in other more traditional environments as well. A context-oriented data base *acts* like a tree of data bases; at any node of the tree is a set of assertions that makes up the data base. A new data base (context) may be spawned from any context (data base) in the tree. All assertions that are true in the spawning (ancestor) context are initially true in the spawned (descendant) context. However, new assertions added in any context or deleted from it do not affect its ancestor. Thus by going back to the ancestor, all data base changes performed in the descendent context disappear.

Implementing such a data base is an interesting exercise. Copying all assertions to each new context is possible, but very wasteful if only a few changes are made in each context. The following mechanism is much more efficient. The root or initial context has some set of assertions in it, and each descendant context is merely an *add list* of assertions to add to the data base and a *delete list* of assertions to delete. Then to see if an assertion is true in a context, do the following.

1. If the context is the root context, look up "as usual."
2. Otherwise, if the assertion is on the *add list* of this context, return *true*. If the assertion is on the *delete list* of this context, return *false*.

3. Otherwise, recursively apply this procedure to the ancestor of this context.

In a general programming environment, contexts have names, and there is the facility of executing procedures "in" particular contexts, moving around the context tree, and so forth. However, in what follows, only the ability to look up assertions in contexts is relevant.

13.1.2 Representing Actions

Represent an action as a triple.

ACTION ::= [PATTERN, PRECONDITIONS, POSTCONDITIONS].

Here the pattern gives the name of the action and names for the objects with which it deals—its "formal parameters." Preconditions and postconditions may use the formal variables of the pattern. In a sense, the preconditions and postconditions are the "body" of the action, with subroutine-like "variable bindings" taking place when the action is to be performed. The preconditions give the world states in which the action may be applied. Here the preconditions are assumed simply to be a list of assertions all of which must be true. The postconditions describe the world state that results from performing the action. The context-oriented data base of hypothetical worlds can be used to implement the postconditions.

POSTCONDITIONS ::= [ADD-LIST, DELETE-LIST].

An action is then performed as follows.

1. Bind the pattern variables to entities in the world, thus binding the associated variables in the preconditions and postconditions.
2. If the preconditions are met (the bound assertions exist in the data base), do the next step, else exit reporting failure.
3. Delete the assertions in the delete list, add those in the add list, and exit reporting success.

Here is the *Move* action for our block-stacking example.

Move Object X from Y to Z			
PATTERN	*PRECONDITIONS*	*DELETE-LIST*	*ADD-LIST*
Move(X,Y,Z)	CLEAR(X) CLEAR(Z) ON(X,Y)	ON(X,Y) CLEAR(Z)	ON(X,Z) CLEAR(Y)

Here X, Y, and Z are all variables bound to world entities. In the initial state of Fig. 13.1, Move(C,A,Floor) binds X to C, Y to A, Z to Floor, and the preconditions are satisfied; the action may proceed.

However, notice two things.

1. The action given above deletes the CLEAR(Floor) assertion that always should be true. One must fix this somehow; putting CLEAR(Floor) in the add-list does the job, but is a little inelegant.
2. What about an action like Move(C,A,C)? It meets the preconditions, but causes trouble when the add and delete lists are applied. One fix here is to keep in the data base ("world model") a set of assertions such as Different (A,B), Different(A,Floor), . . . , and to add assertions such as Different (X,Z) to the preconditions of Move.

Such housekeeping chores and details of axiomatization are inherent in applying basically syntactic, formal solution methods to problem solving. For now, let us assume that CLEAR(Floor) is never deleted, and that Move(X,Y,Z) is applied only if Z is different from X and Y.

13.1.3 Stacking Blocks

In the block-stacking example, the goal is two simultaneous assertions, ON(A,B) and ON(B,C). One solution method proceeds by repeatedly picking a goal to work on, finding an operator that moves closer to the goal, and applying it. In this case of only one action the question is how to apply it—what to move where. This is answered by looking at the postconditions of the action in the light of the goal. The reasoning might go like this: ON(B,C) can be made true if X is B and Z is C. That is possible in this state if Y is A; all preconditions are satisfied, and the goal ON(B,C) can be achieved with one action.

Part of the world state (or context) tree the planner must search is shown in Fig. 13.2, where states are shown diagrammatically instead of through sets of assertions. Notice the following things in Fig. 13.2.

1. Trying to achieve ON(B,C) first is a mistake (Branch 1).
2. Trying to achieve ON(A,B) first is also a mistake for less obvious reasons (Branch 2).
3. Branches 1 and 2 show "subgoal interaction." The goals as stated are not independent. Branch 3 must be generated somehow, either through backtracking or some intelligent way of coping with interaction. It will never be found by the single-minded approach of (1) and (2). However, if ON(C,Floor) were one of the goal assertions, Branch 3 could be found.

Clearly, representing world and actions is not the whole story in planning. Intelligent search of the context is also necessary. This search involves subgoal selection, action selection, and action argument selection. Bad choices anywhere can mean inefficient or looping action sequences, or the generation of impossible subgoals. "Intelligent" search implies a meta-level capability: the ability of a program to reason about its own plans. "Plan critics" are often a part of sophisticated planners; one of their main jobs is to isolate and rectify unwanted subgoal interaction [Sussman 1975].

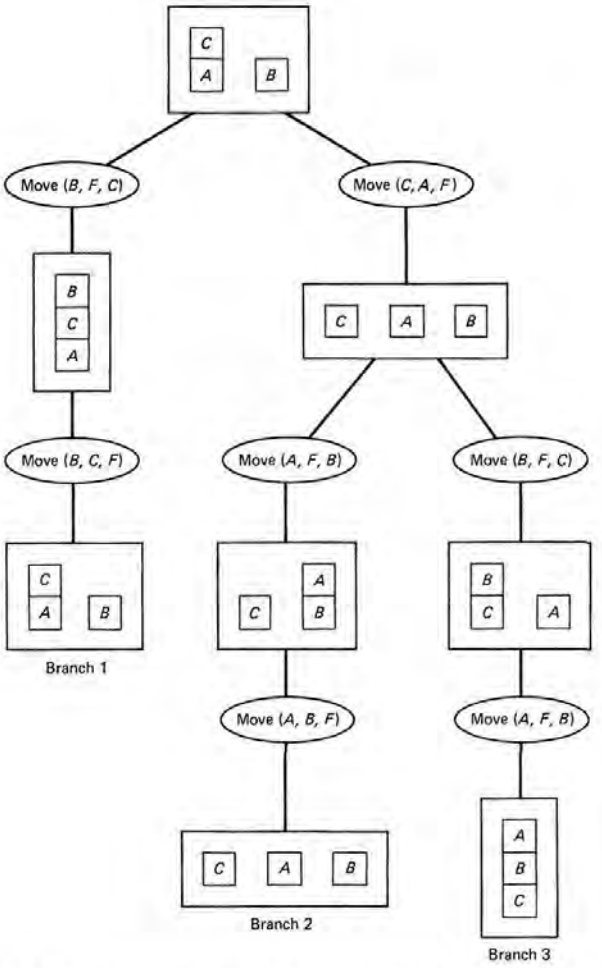


Fig. 13.2 A state tree generated in planning how to stack three blocks.

Intelligent choice of actions is the crux of planning, and is a major research issue. Several avenues have been and are being tried. Perhaps subgoals may be ordered by difficulty and achieved in that order. Perhaps planning should proceed at various levels of detail (like multiresolution image understanding), where the strategic skeleton of a plan is derived without details, then the details are filled in by applying the planner in more detail to the subgoals in the low-resolution plan.

13.1.4 The Frame Problem

All planning is plagued by aspects of the *frame problem* (introduced in Section 12.1.6).

1. It is impractical (and boring) to write down in an action all the things that stay the same when an action is applied.
2. Similarly, it is impractical to reassert in the data base all the things that remain true when an action is implied.
3. Often an action has effects that cannot be represented with simple add and delete lists.

The add and delete list mechanism and the context-oriented data base mechanism addressed the first two problems. The last problem is more troublesome.

Add and delete lists are simple ideas, whereas the world is a complex place. In many interesting cases, the add and delete lists depend on the current state of the world when the action is applied. Think of actions TURNBY(X) and MOVEBY(Z) in a world where orientation and location are important. The orientation and location after an action depend not just on the action but on the state of the world just before the action.

Again, the action may have very complex effects if there are complex dependencies between world objects. Consider the problem of the "monkey and bananas," where the monkey plans to push the box under the bananas and climb on it to reach them (Fig. 13.3). Implementation of realistically powerful add and delete lists may in fact require arbitrary amounts of deduction and computation.

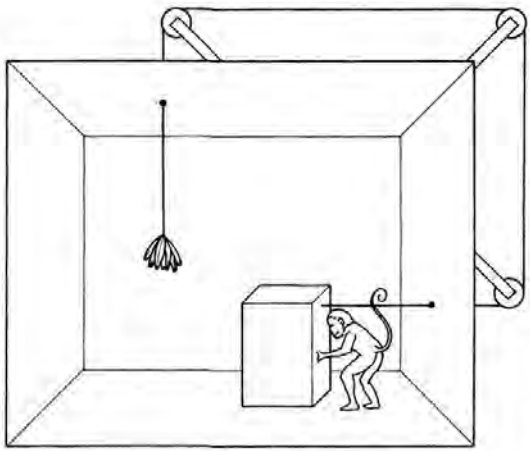


Fig. 13.3 Actions may have complex effects.

This quick précis of symbolic planning does not address many "classical" topics, such as learning or remembering useful plans. Also not discussed are: planning at varying levels of abstraction, plans with uncertain information, or plans with costs. The interested reader should consult the References for more information. The next section addresses plans with costs since they are particularly relevant to vision; some of the other issues appear in the Exercises.

13.2 PLANNING WITH COSTS

Decision making under uncertainty is an important topic in its own right, being of interest to policymakers and managers [Raiffa 1968]. Analytic techniques that can derive the strategy with the "optimal expected outcome" or "maximal expected utility" can be based on Bayesian models of probability.

In [Feldman and Sproull 1977] these techniques are explored in the context of action planning for real-world actions and vision. As an example of the techniques, they are used to model an extended version of the "monkey and bananas" problem of the last section, with multiple boxes but without the maddening pulley arrangement. In the extended problem, there are boxes of different weights which may or may not support the monkey, and he can apply tests (e.g., vision) at some cost to determine whether they are usable. Pushing weighted boxes costs some effort, and the gratification of eating the bananas is "worth" only some finite amount of effort. This extended set of considerations is more like everyday decision making in the number of factors that need balancing, in the uncertainty inherent in the universe, and in the richness of applicable tests. In fact, one might make the claim that human beings always "maximize their expected utility," and if one knew a person's utility functions, his behavior would become predictable. The more intuitive claim that humans beings plan only as far as "sufficient expected utility" can be cast as a maximization operation with nonzero "cost of planning."

The sequential decision-making model of planning with the goal of maximizing the goodness of the expected outcome was used in a travel planner [Sproull 1977]. Knowledge of schedules and costs of various modes of transportation and the attendant risks could be combined with personal prejudices and preferences to produce an itinerary with the maximum expected utility. If unexpected situations (canceled flights, say) arose *en route,* replanning could be initiated; this incremental plan ramification is a natural extension of sequential decision making.

This section is concerned with measuring the expected performance of plans using a single number. Although one might expect one number to be inadequate, the central theorem of decision theory [DeGroot 1970] shows essentially that one number is enough. Using a numerical measure of goodness allows comparisons between normally incomparable concepts to be made easily. Quite frequently numerical scores are directly relevant to the issues at stake in planning, so they are not obnoxiously reductionistic. Decision theory can also help in the process of applying a plan—the basic plan may be simple, but its application to the world may be complex, in terms of when to declare a result established or an action unsuccessful. The decision-theoretic approach has been used in several artificial intelligence and

vision programs [Feldman and Yakimovsky 1974; Bolles 1977; Garvey 1976; Ballard 1978; Sproull 1977].

13.2.1 Planning, Scoring, and Their Interaction

For didactic purposes, the processes of plan generation and plan scoring are considered separately. In fact, these processes may cooperate more or less intimately. The planner produces "sequences" of *actions* for evaluation by the scorer. Each action (computation, information gathering, performing a real-world action) has a *cost*, expressing expenditure of resources, or associated unhappiness. An action has a set of possible *outcomes*, of which only one will really occur when the action is performed. A *goal* is a state of the world with an associated "happiness" or *utility*. For the purposes of uniformity and formal manipulation, goals are treated as (null) actions with no outcomes, and negative utilities are used to express costs. Then the plan has only actions in it; they may be arranged in a strict sequence, or be in loops, be conditional on outcomes of other actions, and so forth.

The *scoring* process evalutes the *expected utility* of a plan. In an uncertain world, a plan prior to execution has only an expected goodness—something might go wrong. Such a scoring process typically is not of interest to those who would use planners to solve puzzles or do proofs; what is interesting is the result, not the effort. But plans that are "optimal" in some sense are decidedly of interest in real-world decision making. In a vision context, plans are usually useful only if they can be evaluated for efficiency and efficacy.

Scoring can take place on "complete" plans, but it can also be used to guide plan generation. The usual artificial intelligence problem-solving techniques of progressive deepening search and branch-and-bound pruning may be applied to planning if scoring happens as the plan is generated [Nilsson 1980]. Scoring can be used to assess the cost of planning and to monitor planning horizons (how far ahead to look and how detailed to make the plan). Scoring will penalize plans that loop without producing results. Plan improvements, such as replanning upon failure, can be assessed with scores, and the contribution of additional steps (say for extra information gathering) can be assessed dynamically by scoring. Scoring can be arbitrarily complex utility functions, thus reflecting such concepts as "risk aversion" and nonlinear value of resources [Raiffa 1968].

13.2.2 Scoring Simple Plans

Scoring and an Example

A *simple plan* is a tree of nodes (there are no loops). The nodes represent actions (and goals). Outcomes are represented by labeled arcs in the tree. A probability of occurrence is associated with each possible outcome; since exactly one outcome actually occurs per action, the probabilities for the possible outcomes of any action sum to unity.

The *score* of a plan is its *expected utility*. The expected utility of any node is recursively defined as its utility times the probability of reaching that node in the

plan, plus the expected utilities of the actions at its (possible) outcomes. The probability of reaching any "goal state" in the plan is the product of probabilities of outcomes forming a path from the root of the plan to the goal state.

As an example, consider the plan shown in Fig. 13.4. If the plan of Fig. 13.4

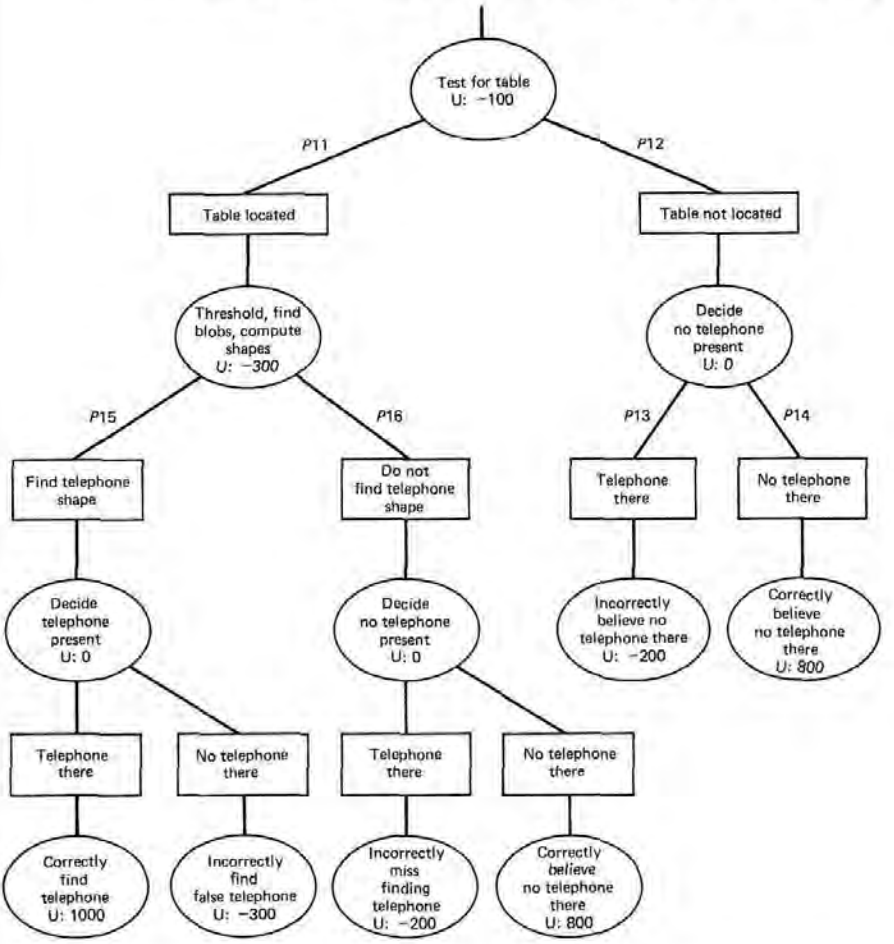


Fig. 13.4 This plan to find a telephone in an office scene involves finding a table first and looking there in more detail. The actions and outcomes are shown. The probabilities of outcomes are assigned symbols (P10, etc.). Utilities (denoted by U:) are given for the individual actions. Note that negative utilities may be considered costs. In this example, decision-making takes no effort, image processing costs vary, and there are various penalties and rewards for correct and incorrect finding of the telephone.

has probabilities assigned to its outcomes, we may compute its expected utility. Figure 13.5 shows the calculation. The probability of correctly finding the telephone is 0.34, and the expected utility of the plan is 433.

Although the generation of a plan may not be easy, scoring a plan is a trivial exercise once the probabilities and utilities are known. In practice, the assignment of probabilities is usually a source of difficulty. The following is an example using

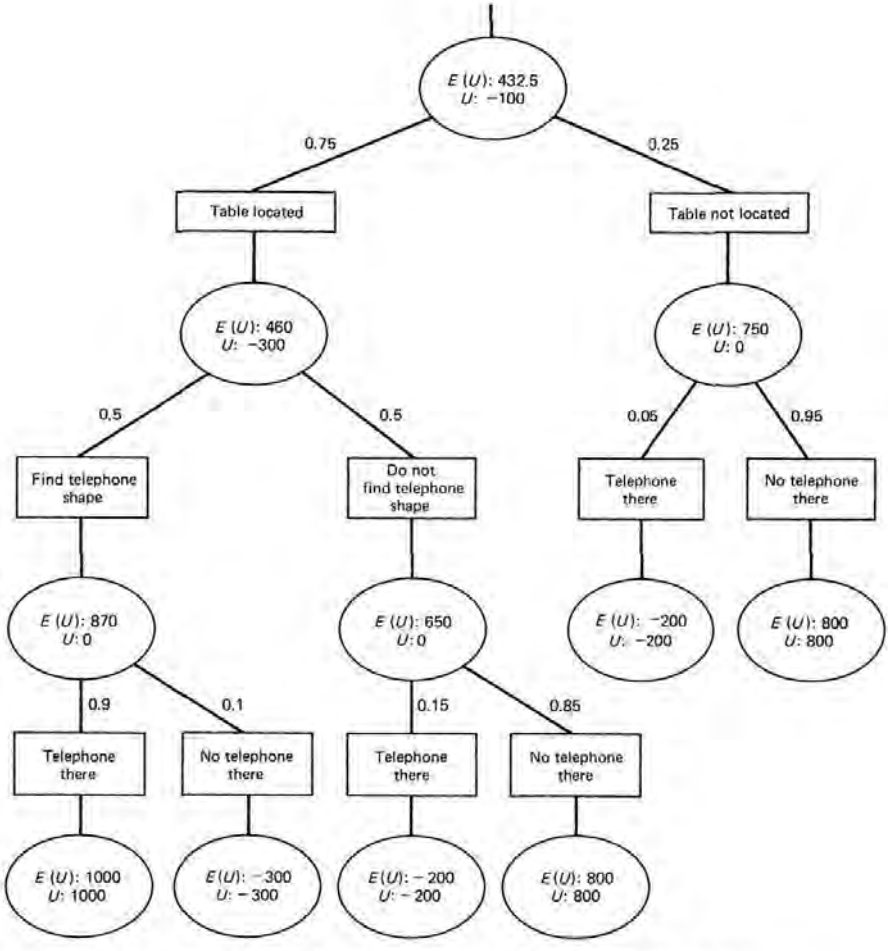


Fig. 13.5 As for Fig. 13.4. U: gives the utility of each action. E(U): gives the expected utillity of the action, which depends on the outcomes below it. Values for outcome probabilities are given on the outcome arcs.

the telephone-finding plan and some assumptions about the tests. Different assumptions yield different scores.

Computing Outcome Probabilities: An Example

This example relies heavily on Bayes' rule:

$$P(B|A)P(A) = P(A \wedge B) = P(A|B)P(B). \tag{13.1}$$

Let us assume a specific a priori probability that the scene contains a telephone.

$$P_1 = \text{apriori probability of Telephone} \tag{13.2}$$

Also assume that something is known about the behavior of the various tests in the presence of what they are looking for. This knowledge may accrue from experiments to see how often the table test found tables when telephones (or tables) were and were not present. Let us assume that the following are known probabilities.

$$P_3 = \text{P(table located|telephone in scene)} \tag{13.3}$$

$$P_5 = \text{P(table located|no telephone in scene)} \tag{13.4}$$

Either there is a telephone or there is not, and a table is located or it is not, so

$$P_2 = \text{a priori probability of no telephone} = 1 - P_1 \tag{13.5}$$

$$P_4 = \text{P(no table located|telephone in scene)} = 1 - P_3 \tag{13.6}$$

$$P_6 = \text{P(no table located|no telephone in scene)} = 1 - P_5 \tag{13.7}$$

Similarly with the "shape test" for telephones: assume probabilities

$$P_7 = \text{P(telephone shape located|telephone)} \tag{13.8}$$

$$P_9 = \text{P(telephone shape located|no telephone)} \tag{13.9}$$

with

$$P_8 = 1 - P_7, \qquad P_{10} = 1 - P_9 \tag{13.10}$$

as above.

There are a few points to make: First, it is not necessary to know exactly these probabilities in order to score the plan; one could use related probabilities and Bayes' rule. Other useful probabilities are of the form

$$\text{P(telephone|telephone shape located)}.$$

In some systems [Garvey 1976] these are assumed to be available directly. This section shows how to derive them from known conditional probabilities that describe the behavior of detectors given certain scene phenomena.

Second, notice the assumption that although both the outcome of the table test and the shape test depend on the presence of telephones, they are taken to be independent of each other. That is, having found a table tells us nothing about the likelihood of finding a telephone shape. Independence assumptions such as this are

useful to limit computations and data gathering, but can be somewhat unrealistic. To account for the dependence, one would have to measure such quantities as

$$\mathrm{P}(\text{telephone shape found} \mid \text{table located}).$$

Now to compute some outcome probabilities: Consider the probability

$$P_{11} = \mathrm{P}(\text{table located}) \tag{13.11}$$

Let us write

TL for Table Located
TNL for Table Not Located.

A table may be located whether or not a telephone is in the scene. In terms of known probabilities, Bayes' rule yields

$$P_{11} = P_3\, P_1 + P_5\, P_2 \tag{13.12}$$

Then

$$P_{12} = P(\mathrm{TNL}) = 1 - P_{11} \tag{13.13}$$

Calculating P_{13} shows a neat trick using Bayes' Rule:

$$P_{13} = P(\text{telephone} \mid \mathrm{TNL}) \tag{13.14}$$

That is, P_{13} is the probability that there is a telephone in the scene given that search for a table was unsuccessful. This probability is not known directly, but

$$\begin{aligned} P_{13} &= \frac{P(\text{telephone and TNL})}{P(\mathrm{TNL})} \\ &= \frac{P(\mathrm{TNL} \text{ and telephone})}{P_{12}} \\ &= \frac{[P(\mathrm{TNL} \mid \text{telephone})\,\mathrm{P}(\text{telephone})]}{P_{12}} \\ &= \frac{[P_4\, P_1]}{P_{12}} \end{aligned} \tag{13.15}$$

Then, of course

$$P_{14} = 1 - P_{13} \tag{13.16}$$

Reasoning in this way using the conditional probabilities and assumptions about their independence allows the completion of the calculation of outcome probabilities (see the Exercises). One possibly confusing point occurs in calculation of P_{15}, which is

$$P_{15} = P(\text{telephone shape found} \mid \text{table located}) \tag{13.17}$$

By assumption, these events are only indirectly related. By the simplifying assumptions of independence, the shape operator and the table operator are independent in their operation. (Such assumptions might be false if they used common image processing subroutines, for example.) Of course, the probability of success of each

depends on the presence of a telephone in the scene. Therefore their performance is linked in the following way (see the Exercises). (Write TSL for Telephone Shape Located.)

$$P_{15} = P(\text{TSL}|\text{TL})P(\text{TSL}|\text{telephone})P(\text{telephone}|\text{TL}) + P(\text{TSL}|\text{no telephone})P(\text{no telephone}|\text{TL}) \quad (13.18)$$

13.2.3 Scoring Enhanced Plans

The plans of Section 13.2.2 were called "simple" because of their tree structure, complete ordering of actions, and the simple actions of their nodes. With a richer output from the symbolic planner, the plans may have different structure. For example, there may be *OR* nodes, any one of whose sons will achieve the action at the node; *AND* nodes, all of which must be satisfied (in any order) for the action to be satisfactorily completed; *SEQUENCE* nodes, which specify a set of actions and a particular order in which to achieve them. The plan may have loops, shared subgoal structure, or goals that depend on each other. How enhanced plans are interpreted and executed depends on the scoring algorithms, the possibilities of parallel execution, whether execution and scoring are interleaved, and so forth. This treatment ignores parallelism and limits discussion to expanding enhanced plans into simple ones.

It should be clear how to go about converting many of these enhanced plans to simple plans. For instance, sequence nodes simply go to a unique path of actions. Alternatively, depending on assumptions about outcomes of such actions (say whether they can fail), they may be coalesced into one action, as was the "threshold, find blobs, and compute shapes" action in the telephone-finding plan.

Rather more interesting are the OR and AND nodes, the order of whose subgoals is unspecified. Each such node yields many simple plans, depending on the order in which the subgoals are attacked. One way to score such a plan is to generate all possible simple plans and score each one, but perhaps it is possible to do better. For example, loops and mutual dependencies in plans can be dealt with in various ways. A loop can be analyzed to make sure that it contains an exit (such as a branch of an OR node that can be executed). One can make ad hoc assumptions that the cost of execution is always more than the cost of planning [Garvey 1976], and score the loop by its executable branch. Another idea is to plan incrementally with a finite horizon, expanding the plan through some progressive deepening, heuristic search, or pruning strategy. The accumulated cost of going around a loop will soon remove it from further consideration.

Recall (Figs. 13.4 and 13.5) that the expected utility of a plan was defined as the sum of the utility of each leaf node times the probability of reaching that node. However, the utilities need not combine linearly in scoring. Different monotonic functions of utility express such different conceptions as "aversion to risk" or "gambling addiction." These considerations are real ones, and nonlinear utilities are the rule rather than the exception. For instance, the value of money is notoriously nonlinear. Many people would pay $5 for an even chance to win $15; not so many people would pay $5,000 for an even chance to win $15,000.

One common way to compute scores based on utilities is the "cost/benefit" ratio. This, in the form "cost/confidence" ratio, is used by Garvey in his planning vision system. This measure is examined in Section 13.2.5; roughly, his "cost" was the effort in machine cycles to achieve goals, and his "confidence" approximated the probability of a goal achieving the correct outcome. The utility of correct outcomes was not explicitly encoded in his planner.

Sequential plan elaboration or partial plan elaboration can be interleaved with execution and scoring. Most practical planning is done in interaction with the world, and the plan scoring approach lends itself well to assessing such interactions. In Section 13.2.5 considers a planning vision system that uses enhanced plans and a limited replanning capability.

A thorny problem for decision making is to assess the cost of planning itself. The planning process is given its own utility (cost), and is carried only out as far as is indicated. Of course, the problem is in general infinitely recursive, since there is also the cost of assessing the cost of planning, etc. If, however, there is a known upper bound on the utility of the best achievable plan, then it is known that infinite planning could not improve it. This sort of reasoning is weaker than that needed to give the expected benefits of planning; it measures only the cost and maximum value of planning.

Another more advanced consideration is that the results of actions can be continuous and multidimensional, and discrete probabilities can be extended to probability distribution functions. Such techniques can reflect the precision of measurements.

An obviously desirable extension to a planner is a "learner," that can abstract rules for action applicability and remember successful plans. One approach would be to derive and remember ranges of planning parameters arising during execution; a range could be associated with a rule specifying appropriate action. This problem is difficult and the subject of current research.

13.2.4 Practical Simplifications

The expected utility calculations allow plans to be evaluated in a more or less "realistic" manner. However, in order to complete the calculations certain probabilities are necessary, and many of these reflect detailed knowledge about the interaction of phenomena in the world. It is thus often impractical to go about a full-blown treatment of scoring in the style of Section 13.2.2. This section presents some possible simplifications.

Of course, in many planning problems, such as those whose costs are nil or irrelevant, or all of whose goals are equally valuable, there is no need to address utility of plans at all. Such plans are typically not concerned with expenditure of real-world or planning resources.

Independence of various probabilities is one of the most helpful and pervasive assumptions in the calculation of probabilities. An example appeared in Section 13.2.2 with the table and telephone shape detectors.

Certain information can be ignored. Garvey [Garvey 1976] ignores failure information. His planning parameters include the "cost" of an action (strictly nega-

tive utilities reflecting effort), the probability of the action "succeeding," and the conditional probability that the state of the world is correctly indicated, given success. Related to ignoring some information is the assumption that certain outcomes are more reliable than others. For instance, the decision not to plan past "failure" reports means that they are assumed reliable.

Non-Bayesian rules of inference abound in planners [Shortliffe 1976]; the idea of assigning a single numerical utility score to plans is by no means the only way to make decisions.

13.2.5 A Vision System Based on Planning

Overview

This section outlines some features of a working vision system whose actions are controlled by the planning paradigm [Garvey 1976]. As with all large vision systems, more issues are addressed in this work than with the planning paradigm as a control mechanism. For one thing, the system uses multisensory input, including range and color information. An interactive facility aids in developing and testing low-level operators and "strategies" for object location. The machine-usable representation of knowledge about the objects in the scene domains and how they could be located is of course a central component.

The domain is office scenes (Fig. 13.6). For the task of locating different objects in such scenes, a "uniform strategy" is adopted. That is, the vision task is always broken down into a sequence of major goals to be performed in order. Such uniform strategies, if they are imposed on a system at all, tend to vary with different tasks, with different sensors or domain, or with different research goals.

Garvey's uniform strategy consists of the following steps.

1. *Acquire* some pixels thought to be in the desired region (the area of scene making up the image of the desired object).

Fig. 13.6 The planning vision system uses input scenes such as these, imaged in different wavelengths and with a rangefinder.

2. *Verify* to some confidence that indeed the region was the desired one.
3. *Bound* the region accurately.

The outline the plan generation, scoring, and execution used in the system are described in the following paragraphs. The plans generated by the system are typically enhanced versions of plans like the telephone finder. Plan scoring proceeds as expected for such plans; allowances are made for the enhanced semantics of plan nodes. A "cost/confidence" scoring function is used, and various practical simplifications are made that do not affect the planning paradigm itself.

An Example Plan and Its Execution

The system's plans are enhanced plans, in the sense of Section 13.2.3. Actions can be *AND*, *OR* or *SEQUENCE* actions, and shared plan structure and loops are permitted. Loops that contain only internal, planning actions would never terminate. However, a loop with an OR node can terminate (has an exit) if one of the subactions of the OR is executable. A plan for locating a chair in an office scene is shown in Fig. 13.7. In Fig. 13.7, the acquire–validate–bound strategy is evident in the two SEQUENCE subgoals of the Find Chair main goal, which is an AND goal. The loop in the plan is evident, and makes sense here because often planning is done for information gathering, not for real world actions.

As noted in Section 13.2.3, an enhanced plan may not be completely specified. If it is to be executed one subgoal at a time (no parallelism is allowed), sequences of subactions must be determined for its AND and OR actions. In Garvey's planner, these sequences are determined initially on the basis of apriori information, but the partial results of actions are "fed back," so that dynamic rescoring and hence dynamic reordering of goal sequences is possible. For example, if one subgoal of an AND action fails, the AND action is abandoned. Thus this planner is to some degree incremental.

In execution, Fig. 13.7 might result in the sequence of actions depicted in Fig. 13.8. The acquisition phase of object location has the most alternatives, so plan generation effort is mainly spent there. Acquisition proceeds either directly or indirectly. Direct acquisition is the classification of input data gathered from a random sampling of a window in the image; the input data are rich enough to allow basic pattern recognition techniques to identify the source of individual pixels.

Indirect acquisition is the use of the location of other "objects" (really identified regions) in the scene to locate the desired region. The desired region might be found by "scanning" vertically or horizontally from the already identified region, for instance. The idea is a planning version of a common one (e.g., the geometric location networks of Section 10.3.2): use something already located to limit and direct search for something else.

Plan Generation

A plan such as Fig. 13.7 is "elaborated" from the basic Find Chair goal by recursively expanding goals. Some goals (such as to find a chair) are not directly executable; they need further elaboration. Elaboration continues until all the subgoals are executable. Executable subgoals are those that analyze the image, run filters and detectors over parts of it, and generate decisions about the presence or absence

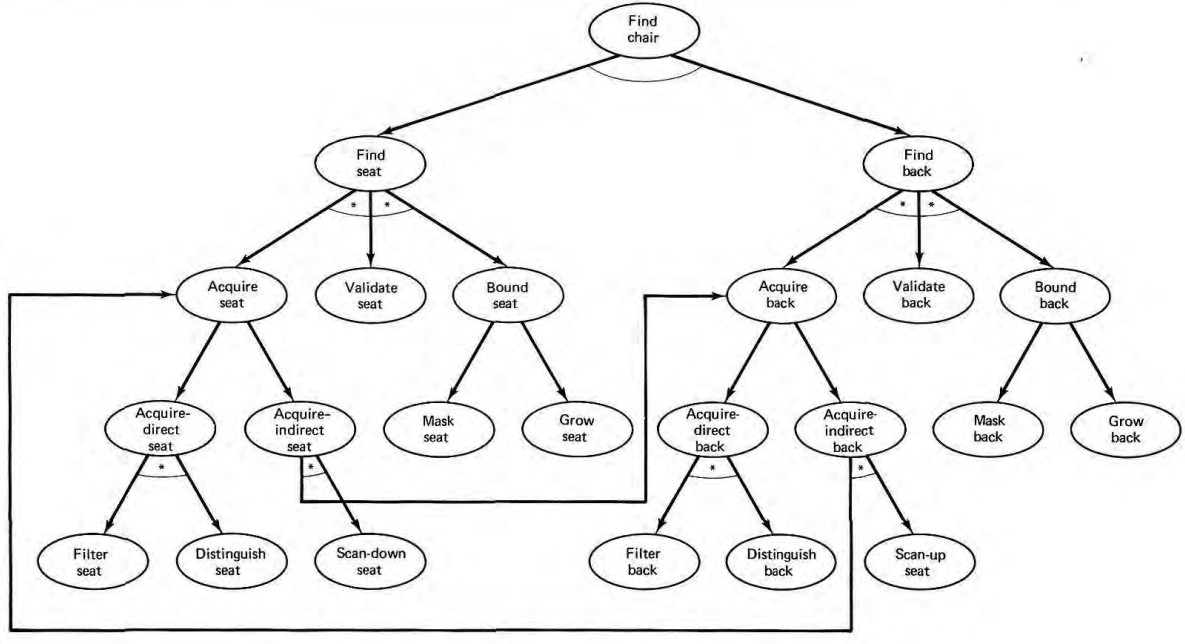


Fig. 13.7 An enhanced plan to locate a chair in an office scene. Untied multiple arcs denote OR actions, arcs tied together denote AND actions, those with *'s denote SEQUENCE actions. The loop in the plan has executable exits.

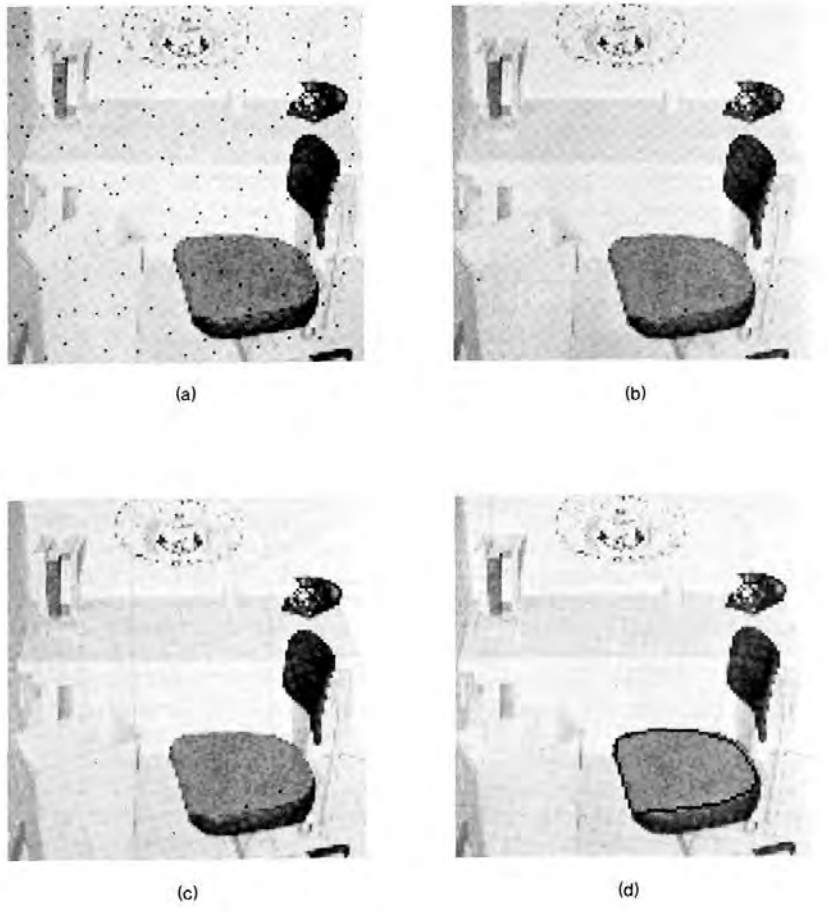


Fig. 13.8 The plan of Fig. 13.7 finds the most promising execution sequence for finding the chair in the scene of Fig. 13.6: find the seat first, then scan upwards from the seat looking for the back. Acquisition of the seat proceeds by sampling (a), followed by classification (b). The Validation procedure eliminates non-chair points (c), and the Bounding procedure produces the seat region (d). To find the back, scanning proceeds in the manner indicated by (e) (actually fewer points are examined in each scan). The back is acquired and bounded, leading to the final location of the chair regions (f).

(e) (f)

Fig. 13.8 (cont.)

of image phenomena. This straightforward elaboration is akin to macro expansion, and is not a very sophisticated planning mechanism (the program cannot criticize and manipulate the plan, only score it). A fully elaborated plan is presented for scoring and execution.

The elaboration process, or planner, has at its disposal several sorts of knowledge embodied as modules that can generate subgoals for a goal. Some are general (to find something, find all its parts); some are less general (a chair has a back and a seat); some are quite specific, being perhaps programs arising from an earlier interactive method-generation phase. The elaborator is guided by information stored about objects, for instance this about a tabletop:

OBJECT	PROPERTIES	RELATIONS
Table TOP	Hue:26–58	Supports Telephone 0.6
	Sat.: 0.23–0.32	Supports Book 0.4
	Bright.: 18–26	Occludes Wall 1
	Height: 26–28	
	Orient.: −7–7	

Here the orientation information indicates a vertical surface normal. The planner knows that it has a method of locating horizontal surfaces, and the plan elaborator can thus create a goal of direct acquisition by first locating a horizontal plane. The relational information allows for indirect acquisition plans. The elaborator puts direct and indirect alternatives under an *OR* node in the plan. Information not used for acquisition (height, color) may be used for validation.

Loops may occur in an elaborated plan because each newly generated goal is checked against goals already existing. Should it or an equivalent goal already exist, the existing goal is substituted for the newly generated one. Goals may thus have more than one ancestor, and may depend on one another.

At this stage, the planner does not use any planning parameters (cost, utilities, etc.); it is strictly symbolic. As mentioned above, important information about execution sequences in an enhanced plan is provided by scoring.

Plan Scoring and Execution

The scoring in the vision plan is a version of that explained in Sections 13.2.2 through 13.2.4. Each action in a plan is assumed either to succeed (S) in locating an object or to fail. Each action may report either success ("S") or failure. An action is assumed to report failure correctly, but possibly to be in error in reporting success. Each action has three "planning parameters" associated with it. They are C, its "cost" (in machine cycles), $P(\text{"}S\text{"})$ the probability of it reporting success, and $P(S|\text{"}S\text{"})$, the probability of success given a report of success.

As shown earlier, the product

$$P(S|\text{"}S\text{"})P(\text{"}S\text{"}) \tag{13.19}$$

is the probability that the action has correctly located an object and reported success. This product is called the "confidence" of the action. An action has structure as shown in Fig. 13.9.

The score of an action is computed as

$$\text{score} = \frac{\text{cost}}{\text{confidence}} \tag{13.20}$$

The planner thus must minimize the score.

The initial planning parameters of an executable action typically are determined by experimentation. The parameters of internal (AND, OR, SEQUENCE) actions by scoring methods alluded to in Sections 13.2.2, 13.2.3, and the Exercises (there are a few idiosyncratic ad hoc adjustments.).

It may bear repeating that planning, scoring, and execution are not separated temporally in this sytem. Scoring is used after the enhanced plan is generated to derive a simple plan (with ordered subgoals). Execution can affect the scores of nodes, and so execution can alternate with "replanning" (really rescoring resulting in a reordering). Recall the example of failure of an *AND* or *SEQUENCE* subgoal, which can immediately fail the entire goal. More generally, the entire goal and ultimately the plan may be rescored. For instance, the parameters of a successful action are modified by setting the cost of the executed action to 0 and its confidence to its second parameter, $P(S|\text{"}S\text{"})$.

Given a scored plan, execution is then easy; the execution program starts at the top goal of the plan, working its way down the best path as defined by the scores of nodes it encounters. When an executable subgoal is found (e.g. "look for a green region"), it is passed to an evaluation function that "runs" the action associated with the subgoal.

The subgoal is either achieved or not; in either case, information about its outcome is propagated back up the plan. Failure is easy; a failed subgoal of an AND or SEQUENCE goal fails the goal, and this failure is propagated. A failed subgoal of an OR goal is removed from the plan. The use of success information is more complex, involving the adjustment of confidences and planning parameters illustrated above.

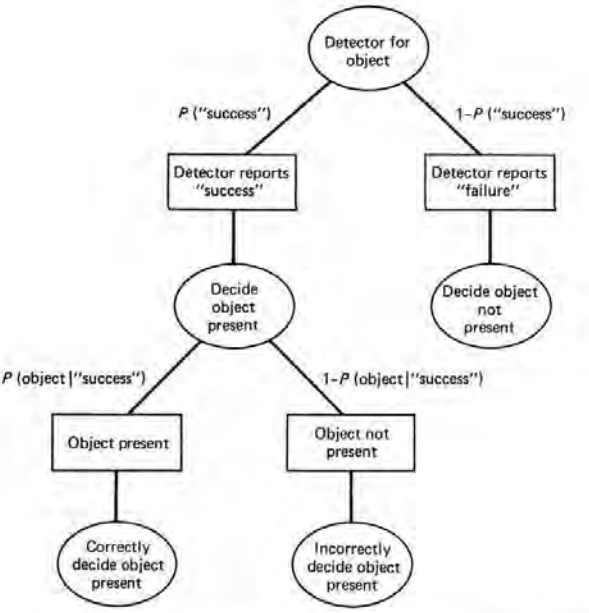


Fig. 13.9 This is the microstructure of a node ("action") of Garvey's planning system in terms of simple plans. Think of actions as being object detectors which announce "Found" or "Not Found." Garvey's planning parameters are P("Found") and P(Object is there|"Found"). Confidence in the action is their product; it is the probability of correctly detecting the object. All other outcomes are lumped together and not used for planning.

After the outcome of a goal is used to adjust the parameters of other goals, the plan is rescored and another cycle of execution performed. The execution can use knowledge about the image picked up along the way by prior execution. This is how results (such as acquired pixels) are passed to later processing stages (such as the validation process). Such a mechanism can even be used to remember successful subplans for later use.

EXERCISES

13.1 Complete the computation of outcome probabilities in the style of Section 13.2.2, using the assumptions given there. Check your work by showing (symbolically) that the probabilities of getting to the terminal actions ("goal states") of the plan sum to 1.

13.2 Assume in Section 13.2.2 that the results of the "table" and "telephone shape" detectors are not independent. Formulate your assumptions and compute the new outcome probabilities for Fig. 13.4.

13.3 Show that

$$P(A|(B \wedge C)) = \frac{P(B|(A \wedge C))P(A|C)}{P(B|C)}$$

13.4 B and C are independent if $P(B \wedge C) = P(B)\,P(C)$. Assuming that B and C are independent, show that

$$P(B|C) = P(B)$$
$$P((B \wedge C)|A) = P(B|A)P(C|A)$$
$$P(B|(A \wedge C)) = P(B|A)$$

13.5 Starting from the fact that

$$P(A \wedge B) = P(A \wedge B \wedge C) + P(A \wedge B \wedge (\neg C))$$

show how P_{15} was computed in Section 13.2.2.

13.6 A sequence $D(N)$ of N detectors is used to detect an object; the detectors either succeed or fail. Detector outputs are assumed independent of each other, being conditioned only on the object. Using previous results, show that the probability of an object being detected by applying a sequence of N detectors $D(N)$ is recursively rewritable in terms of the output of the first detector $D1$ and the remaining sequence $D(N-1)$ as

$$P(O|D(N)) = \frac{P(D1|O)P(O|D(N-1))}{P(D1|D(N-1))}$$

13.7 Consider scoring a plan containing an OR node (action). Presumably, each subgoal of the OR has an expected utility. The OR action is achieved as soon as one of the subgoals is achieved. Is it possible to order the subgoals for trial so as to maximize the expected utility of the plan? (This amounts to a unique "best" rewriting of the plan to make it a simple plan.)

13.8 Answer question 13.7 for an AND node; remember that the AND will fail as soon as any of its subgoals fails.

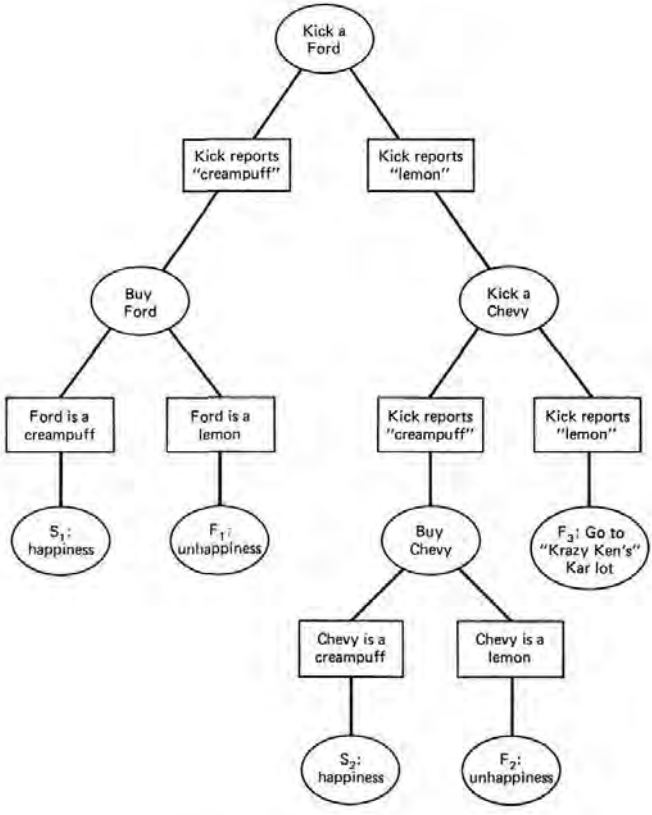
13.9 What can you say about how the cost/confidence ratio of Garvey's planner is related to the expected utility calculations of Section 13.2.2?

13.10 You are at Dandy Dan's used car lot. *Consumer Reports* says that the a priori probability that any car at Dandy Dan's is a lemon is high. You know, though, that to test a car you kick its tire. In fact, with probability:

$P(\text{"}C\text{"}|C)$: a kick correctly announces "creampuff" when the car actually is a creampuff
$P(\text{"}C\text{"}|L)$: a kick incorrectly announces "creampuff" when the car is actually a lemon
$P(L)$: the a priori probability that the car is a lemon

Your plan for dealing with Dandy Dan is shown below; give expressions for the probabilities of arriving at the nodes labeled S_1, S_2, F_1, F_2, and F_3. Give numeric answers using the following values

$$P(\text{"}C\text{"}|C) = 0.5,\ P(\text{"}C\text{"}|L) = 0.5,\ P(L) = 0.75$$



Ex. 13.10

13.11 Two bunches of bananas are in a room with a monkey and a box. One of the bunches is lying on the floor, the other is hanging from the ceiling. One of the bunches is made of wax. The box may be made of flimsy cardboard. Given that:

$P(\text{WH}) = 0.2$: probability that the hanging bananas are wax
$P(\text{WL}) = 0.8$: probability that the lying bananas are wax
$P(\text{C}) = 0.5$: probability that the box is cardboard
$U(\text{eat}) = 200$: utility of eating a bunch of bananas
$C(\text{walk}) = -10$: cost of walking a unit distance
$C(\text{push}) = -20$: cost of pushing the box a unit distance
$C(\text{climb}) = -20$: cost of climbing up on box

(a) Analyze two different plans for the monkey, showing all paths and calculations. Give criteria (based upon extra information not given here) that would allow the monkey to choose between these plans.

(b) Suppose the monkey knows that the probability that the box will collapse is inversely proportional to the cost of pushing the box a unit distance (and that he can sense this cost after pushing the box 1 unit distance). For example,

$$\begin{aligned} P(C) &= 1.0 - [C(\text{push}) \times 0.01] \\ P(C(\text{push}) &= 10) = 0.1 \\ P(C(\text{push}) &= 20) = 0.1 \\ P(C(\text{push}) &= 100) = 0.1 \end{aligned}$$

Repeat part(a) (in detail).

REFERENCES

AMBLER, A. P., H. G. BARROW, C. M. BROWN, R. M. BURSTALL, and R. J. POPPLESTONE. "A versatile system for computer controlled assembly." *Artificial Intelligence 6*, 2, 1975, 129–156.

BALLARD, D. H. "Model-directed detection of ribs in chest radiographs." TR11, Computer Science Dept., U. Rochester, March 1978.

BOLLES, R. C. "Verification vision for programmable assembly." *Proc.*, 5th IJCAI, August 1977, 569–575.

BUNDY, A. *Artificial Intelligence: An introductory course.* New York: North Holland, 1978.

DEGROOT, M. H. *Optimal Statistical Decisions.* New York: McGraw-Hill, 1970.

FAHLMAN, S. E. "A planning system for robot construction tasks." *Artificial Intelligence 5*, 1, Spring 1974, 1–49.

FELDMAN, J. A. and R. F. SPROULL. "Decision theory and artificial intelligence: II. The hungry monkey." *Cognitive Science 1*, 2, 1977, 158–192.

FELDMAN, J. A. and Y. YAKIMOVSKY. "Decision theory and artificial intelligence: I. A semantics-based region analyser." *Artificial Intelligence 5*, 4, 1974, 349–371.

FIKES, R. E. and N. J. NILSSON. "STRIPS: A new approach to the application of theorem proving to problem solving." *Artificial Intelligence 2*, 3/4, 1971, 189–208.

FIKES, R. E., P. E. HART, and N. J. NILSSON. "New Directions in robot problem solving." In *MI7*, 1972a.

FIKES, R. E., P. E. HART, and N. J. NILSSON. "Learning and executing generalized robot plans." *Artificial Intelligence 3*, 4, 1972b, 251–288.

GARVEY, J. D. "Perceptual strategies for purposive vision." Technical Note 117, AI Center, SRI Int'l, 1976.

MACKWORTH, A. K. "Vision research strategy: Black magic, metaphors, mechanisms, mini-worlds, and maps." In *CVS*, 1978.

MCCARTHY, J. and P. J. HAYES. "Some philosophical problems from the standpoint of artificial intelligence." In *MI4*, 1969.

NILSSON, N. J. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga Publishing Company, 1980.

RAIFFA, H. *Decision Analysis.* Reading, MA: Addison-Wesley, 1968.

SACERDOTI, E. D. "Planning in a hierarchy of abstraction spaces." *Artificial Intelligence 5*, 2, 1974, 115–135.

SACERDOTI, E. D. *A Structure for Plans and Behavior.* New York: Elsevier, 1977.

SHIRAI, Y. "Analyzing intensity arrays using knowledge about scenes." In *PCV*, 1975.

SHORTLIFFE, E. H. *Computer-Based Medical Consultations: MYCIN.* New York: American Elsevier, 1976.

SPROULL, R. F. "Strategy construction using a synthesis of heuristic and decision-theoretic methods." Ph.D. dissertation, Dept. of Computer Science, Stanford U., May 1977.

SUSSMAN, G. J. *A Computer Model of Skill Acquisition.* New York: American Elsevier, 1975.

TATE, A. "Generating project networks." *Proc.*, 5th IJCAI, August 1977, 888–983.

WARREN, D. H. D. "WARPLAN: A system for generating plans." Memo 76, Dept. of Computational Logic, U. Edinburgh, June 1974.

Some Mathematical Tools

Appendix 1

A1.1 COORDINATE SYSTEMS

A1.1.1 Cartesian

The familiar two- and three-dimensional rectangular (Cartesian) coordinate systems are the most generally useful ones in describing geometry for computer vision. Most common is a right-handed three-dimensional system (Fig. A1.1.). The coordinates of a point are the perpendicular projections of its location onto the coordinate axes. The two-dimensional coordinate system divides two-dimensional space into quadrants, the three-dimensional system divides three-space into octants.

A1.1.2 Polar and Polar Space

Coordinate systems that measure locations partially in terms of angles are in many cases more natural than Cartesian coordinates. For instance, locations with respect

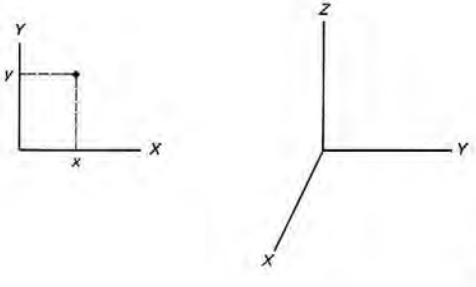


Fig. A1.1 Cartesian coordinate systems.

to the pan-tilt head of a camera or a robot arm may most naturally be described using angles. Two- and three-dimensional polar coordinate systems are shown in Fig. A1.2.

Cartesian Coordinates	*Polar Coordinates*
x	$\rho \cos \theta$
y	$\rho \sin \theta$
$(x^2 + y^2)^{1/2}$	ρ
$\tan^{-1}\left(\frac{y}{x}\right)$	θ

Cartesian Coordinates	*Polar Space Coordinates*
(x, y, z)	$(\rho \cos \xi, \rho \cos \eta, \rho \cos \zeta)$
$(x^2 + y^2 + z^2)^{1/2}$	ρ
$\cos^{-1}\left(\frac{x}{\rho}\right)$	ξ
$\cos^{-1}\left(\frac{y}{\rho}\right)$	η
$\cos^{-1}\left(\frac{z}{\rho}\right)$	ζ

In these coordinate systems, the Cartesian quadrants or octants in which points fall are often of interest because many trigonometric functions determine only an angle modulo $\pi/2$ or π (one or two quadrants) and more information is necessery to determine the quadrant. Familiar examples are the inverse angle functions (such as arctangent), whose results are ambiguous between two angles.

A1.1.3 Spherical and Cylindrical

The spherical and cylindrical systems are shown in Fig. A1.3.

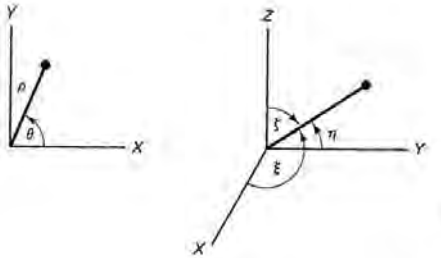


Fig. A1.2 Polar and polar space coordinate systems.

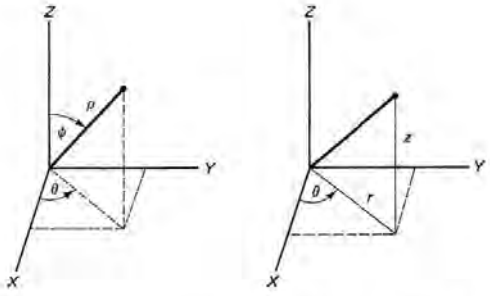


Fig. A1.3 Spherical and cylindrical coordinate systems.

Cartesian Coordinates	*Spherical Coordinates*
x	$\rho \sin \phi \cos \theta$,
y	$\rho \sin \phi \sin \theta = x \tan \theta$
z	$\rho \cos \theta$
$(x^2 + y^2 + z^2)^{1/2}$	ρ
$\tan^{-1}\left(\frac{y}{x}\right)$	θ
$\cos^{-1}\left(\frac{z}{\rho}\right)$	ϕ

Cartesian Coordinates	*Cylindrical Coordinates*
x	$r \cos \theta$
y	$r \sin \theta$
z	z
$(x^2 + y^2)^{1/2}$	r
$\tan^{-1}\left(\frac{y}{x}\right)$	θ

A1.1.4 Homogeneous Coordinates

Homogeneous coordinates are a very useful tool in computer vision (and computer graphics) because they allow many important geometric transformations to be represented uniformly and elegantly (see Section A1.7). Homogeneous coordinates are redundant: a point in Cartesian n-space is represented by a line in homogeneous $(n + 1)$-space. Thus each (unique) Cartesian coordinate point corresponds to infinitely many homogeneous coordinates.

Cartesian Coordinates	*Homogeneous Coordinates*
(x, y, z)	(wx, wy, wz, w)
$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$	(x, y, z, w)

Here x, y, z, and w are real numbers, wx, wy, and wz are the products of the two reals, and x/w and so on are the indicated quotients.

A1.2. TRIGONOMETRY

A1.2.1 Plane Trigonometry

Referring to Fig. A1.4, define

$$\text{sine:} \quad \sin(A) \text{ (sometimes } \sin A) = \frac{a}{c}$$

$$\text{cosine:} \quad \cos(A) \text{ (or } \cos A) = \frac{b}{c}$$

$$\text{tangent:} \quad \tan(A) \text{ (or } \tan A) = \frac{a}{b}$$

The inverse functions arcsin, arccos, and arctan (also written $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$) map a value into an angle. There are many useful trigonometric identities; some of the most common are the following.

$$\tan(x) = \frac{\sin(x)}{\cos(x)} = -\tan(-x)$$

$$\sin(x + y) = \sin(x)\cos(y) + \cos(x)\sin(y)$$

$$\cos(x + y) = \cos(x)\cos(y) - \sin(x)\sin(y)$$

$$\tan(x \pm y) = \frac{\tan(x) \mp \tan(y)}{1 \mp \tan(x)\tan(y)}$$

In any triangle with angles A, B, C opposite sides a, b, c, the Law of Sines holds:

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

as does the Law of Cosines:

$$a^2 = b^2 + c^2 - 2bc\cos A$$

$$a = b\cos C + c\cos B$$

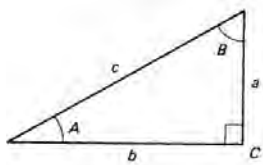


Fig. A1.4 Plane right triangle.

A1.2.2. Spherical Trigonometry

The sides of a spherical triangle (Fig. A1.5) are measured by the angle they subtend at the sphere center; its angles by the angle they subtend on the face of the sphere.

Some useful spherical trigonometric identities are the following.

$$\frac{\sin A}{\sin a} = \frac{\sin B}{\sin b} = \frac{\sin C}{\sin c}$$

$$\cos a = \cos b \cos c + \sin b \sin c \cos A = \frac{\cos b \cos (c \pm \theta)}{\cos \theta}$$

$$\text{Where } \tan \theta = \tan b \cos A,$$

$$\cos A = -\cos B \cos C + \sin B \sin C \cos a$$

A1.3. VECTORS

Vectors are both a notational convenience and a representation of a geometric concept. The familiar interpretation of a vector $\mathbf{v}$ as a directed line segment allows for a geometrical interpretation of many useful vector operations and properties. A more general notion of an n-dimensional vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ is that of an n-tuple abiding by mathematical laws of composition and transformation. A vector may be written horizontally (a row vector) or vertically (a column vector).

A point in n-space is characterized by its n coordinates, which are often written as a vector. A point at X, Y, Z coordinates x, y, and z is written as a vector $\mathbf{x}$ whose three components are (x, y, z). Such a vector may be visualized as a directed line segment, or arrow, with its tail at the origin of coordinates and its head at the point at (x, y, z). The same vector may represent instead the direction in which it points—toward the point (x, y, z) starting from the origin. An important type of direction vector is the normal vector, which is a vector in a direction perpendicular to a surface, plane, or line.

Vectors of equal dimension are equal if they are equal componentwise. Vectors may be multiplied by scalars. This corresponds to stretching or shrinking the vector arrow along its original direction.

$$\lambda \mathbf{x} = (\lambda x_1, \lambda x_2, \ldots, \lambda x_n)$$

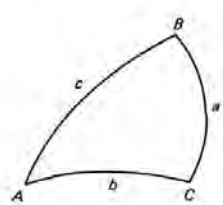


Fig. A1.5 Spherical triangle.

Vector addition and subtraction is defined componentwise, only between vectors of equal dimension. Geometrically, to add two vectors **x** and **y**, put **y**'s tail at **x**'s head and the sum is the vector from **x**'s tail to **y**'s head. To subtract **y** from **x**, put **y**'s head at **x**'s head; the difference is the vector from **x**'s tail to **y**'s tail.

$$\mathbf{x} \pm \mathbf{y} = (x_1 \pm y_1, x_2 \pm y_2, \ldots, x_n \pm y_n)$$

The length (or magnitude) of a vector is computed by an n-dimensional version of Euclidean distance.

$$|\mathbf{x}| = (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2}$$

A vector of unit length is a unit vector. The unit vectors in the three usual Cartesian coordinate directions have special names.

$$\mathbf{i} = (1, 0, 0)$$
$$\mathbf{j} = (0, 1, 0)$$
$$\mathbf{k} = (0, 0, 1)$$

The inner (or scalar, or dot) product of two vectors is defined as follows.

$$\mathbf{x} \cdot \mathbf{y} = |\mathbf{x}||\mathbf{y}| \cos\theta = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

Here θ is the angle between the two vectors. The dot product of two nonzero numbers is 0 if and only if they are orthogonal (perpendicular). The projection of **x** onto **y** (the component of vector **x** in the direction **y**) is

$$|\mathbf{x}| \cos\theta = \frac{\mathbf{x} \cdot \mathbf{y}}{|\mathbf{y}|}$$

Other identities of interest:

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{y} \cdot \mathbf{x}$$
$$\mathbf{x} \cdot (\mathbf{y} + \mathbf{z}) = \mathbf{x} \cdot \mathbf{y} + \mathbf{x} \cdot \mathbf{z}$$
$$\lambda(\mathbf{x} \cdot \mathbf{y}) = (\lambda \mathbf{x}) \cdot \mathbf{y} = \mathbf{x} \cdot (\lambda \mathbf{y})$$
$$\mathbf{x} \cdot \mathbf{x} = |\mathbf{x}|^2$$

The cross (or vector) product of two three-dimensional vectors is defined as follows.

$$\mathbf{x} \times \mathbf{y} = (x_2 y_3 - x_3 y_2,\ x_3 y_1 - x_1 y_3,\ x_1 y_2 - x_2 y_1)$$

Generally, the cross product of **x** and **y** is a vector perpendicular to both **x** and **y**. The magnitude of the cross product depends on the angle θ between the two vectors.

$$|\mathbf{x} \times \mathbf{y}| = |\mathbf{x}||\mathbf{y}| \sin\theta$$

Thus the magnitude of the product is zero for two nonzero vectors if and only if they are parallel.

Vectors and matrices allow for the short formal expression of many symbolic

expressions. One such example is the formal determinant (Section A1.4) which expresses the definition of the cross product given above in a more easily remembered form.

$$\mathbf{x} \times \mathbf{y} = \det \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{bmatrix}$$

Also,

$$\begin{aligned} \mathbf{x} \times \mathbf{y} &= -\mathbf{y} \times \mathbf{x} \\ \mathbf{x} \times (\mathbf{y} \pm \mathbf{z}) &= \mathbf{x} \times \mathbf{y} \pm \mathbf{x} \times \mathbf{z} \\ \lambda(\mathbf{x} \times \mathbf{y}) &= \lambda\mathbf{x} \times \mathbf{y} = \mathbf{x} \times \lambda\mathbf{y} \\ \mathbf{i} \times \mathbf{j} &= \mathbf{k} \\ \mathbf{j} \times \mathbf{k} &= \mathbf{i} \\ \mathbf{k} \times \mathbf{i} &= \mathbf{j} \end{aligned}$$

The triple scalar product is $\mathbf{x} \cdot (\mathbf{y} \times \mathbf{z})$, and is equivalent to the value of the determinant

$$\det \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix}$$

The triple vector product is

$$\mathbf{x} \times (\mathbf{y} \times \mathbf{z}) = (\mathbf{x} \cdot \mathbf{z})\mathbf{y} - (\mathbf{x} \cdot \mathbf{y})\mathbf{z}$$

A1.4. MATRICES

A matrix A is a two-dimensional array of elements; if it has m rows and n columns it is of dimension $m \times n$, and the element in the ith row and jth column may be named a_{ij}. If m or $n = 1$, a row matrix or column matrix results, which is often called a vector. There is considerable punning among scalar, vector and matrix representations and operations when the same dimensionality is involved (the 1×1 matrix may sometimes be treated as a scalar, for instance). Usually, this practice is harmless, but occasionally the difference is important.

A matrix is sometimes most naturally treated as a collection of vectors, and sometimes an $m \times n$ matrix M is written as

$$M = [\mathbf{a}_1 \quad \mathbf{a}_2 \quad \cdots \quad \mathbf{a}_n]$$

or

$$M = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{b}_m \end{bmatrix}$$

where the **a**'s are column vectors and the **b**'s are row vectors.

Two matrices A and B are equal if their dimensionality is the same and they are equal elementwise. Like a vector, a matrix may be multiplied (elementwise) by a scalar. Matrix addition and subtraction proceeds elementwise between matrices of like dimensionality. For a scalar k and matrices A, B, and C of like dimensionality the following is true.

$$A = B \pm C \quad \text{if } a_{ij} = b_{ij} \pm c_{ij} \quad 1 \leqslant i \leqslant m, \quad 1 \leqslant j \leqslant n$$

Two matrices A and B are conformable for multiplication if the number of columns of A equals the number of rows of B. The product is defined as

$$C = AB \quad \text{where an element } c_{ij} \text{ is defined by } \quad c_{ij} = \sum_k a_{ik} b_{kj}$$

Thus each element of C is computed as an inner product of a row of A with a column of B. Matrix multiplication is associative but not commutative in general. The multiplicative identity in matrix algebra is called the identity matrix I. I is all zeros except that all elements in its main diagonal have value 1 ($a_{ij} = 1$ if $i = j$, else $a_{ij} = 0$). Sometimes the $n \times n$ identity matrix is written I_n.

The transpose of an $m \times n$ matrix A is the $n \times m$ matrix A^T such that the i,jth element of A is the j,ith element of A^T. If $A^T = A$, A is symmetric.

The inverse matrix of an $n \times n$ matrix A is written A^{-1}. If it exists, then

$$AA^{-1} = A^{-1}A = I$$

If its inverse does not exist, an $n \times n$ matrix is called singular.

With k and p scalars, and A, B, and C $m \times n$ matrices, the following are some laws of matrix algebra (operations are matrix operations):

$$\begin{aligned} A + B &= B + A \\ (A + B) + C &= A + (B + C) \\ k(A + B) &= kA + kB \\ (k + p)A &= kA + pA \\ AB &\neq BA \quad \text{in general} \\ (AB)C &= A(BC) \\ A(B + C) &= AB + AC \\ (A + B)C &= AC + BC \end{aligned}$$

$$A(kB) = k(AB) = (kA)B$$
$$I_m A = AI_n = A$$
$$(A + B^T) = A^T + B^T$$
$$(AB)^T = B^T A^T$$
$$(AB)^{-1} = B^{-1} A^{-1}$$

The determinant of an $n \times n$ matrix is an important quantity; among other things, a matrix with zero determinant is singular. Let A_{ij} be the $(n-1) \times (n-1)$ matrix resulting from deleting the ith row and jth column from an $n \times n$ matrix A. The determinant of a 1×1 matrix is the value of its single element. For $n > 1$,

$$\det A = \sum_{i=1}^{n} a_{ij} (-1)^{i+j} \det A_{ij}$$

for any j between 1 and n. Given the definition of determinant, the inverse of a matrix may be defined as

$$(a^{-1})_{ij} = \frac{(-1)^{i+j} \det A_{ji}}{\det A}$$

In practice, matrix inversion may be a difficult computational problem, but this important algorithm has received much attention, and robust and efficient methods exist in the literature, many of which may also be used to compute the determinant. Many of the matrices arising in computer vision have to do with geometric transformations, and have well-behaved inverses corresponding to the inverse transformations. Matrices of small dimensionality are usually quite computationally tractable.

Matrices are often used to denote linear transformations; if a row (column) matrix X of dimension n is post (pre) multiplied by an $n \times n$ matrix A, the result $X' = XA$ ($X' = AX$) is another row (column) matrix, each of whose elements is a linear combination of the elements of X, the weights being supplied by the values of A. By employing the common pun between row matrices and vectors, $\mathbf{x}' = \mathbf{x}A$ ($\mathbf{x}' = A\mathbf{x}$) is often written for a linear transformation of a vector $\mathbf{x}$.

An eigenvector of an $n \times n$ matrix A is a vector $\mathbf{v}$ such that for some scalar λ (called an eigenvalue),

$$\mathbf{v}A = \lambda \mathbf{v}$$

That is, the linear transformation A operates on $\mathbf{v}$ just as a scaling operation. A matrix has n eigenvalues, but in general they may be complex and of repeated values. The computation of eigenvalues and eigenvectors of matrices is another computational problem of major importance, with good algorithms for general matrices being complicated. The n eigenvalues are roots of the so-called characteristic polynomial resulting from setting a formal determinant to zero:

$$\det (A - \lambda I) = 0.$$

Eigenvalues of matrices up to 4 × 4 may be found in closed form by solving the characteristic equation exactly. Often, the matrices whose eigenvalues are of interest are symmetric, and luckily in this case the eigenvalues are all real. Many algorithms exist in the literature which compute eigenvalues and eigenvectors both for symmetric and general matrices.

A1.5. LINES

An infinite line may be represented by several methods, each with its own advantages and limitations. An example of a representation which is not often very useful is two planes that intersect to form the line. The representations below have proven generally useful.

A1.5.1 Two Points

A two-dimensional or three-dimensional line (throughout Appendix 1 this shorthand is used for "line in two-space" and "line in three-space"; similarly for "two (three) dimensional point") is determined by two points on it, **x1** and **x2**. This representation can serve as well for a half-line or a line segment. The two points can be kept as the rows of a (2 × n) matrix.

A1.5.2 Point and Direction

A two-dimensional or three-dimensional line (or half-line) is determined by a point **x** on it (its endpoint) and a direction vector **v** along it. This representation is essentially the same as that of Section A1.5.1, but the interpretation of the vectors is different.

A1.5.3 Slope and Intercept

A two-dimensional line can often be represented by the Y value b where the line intersects the Y axis, and the slope m of the line (the tangent of its inclination with the x axis). This representation fails for vertical lines (those with infinite slope). The representation is in the form of an equation making explicit the dependence of y on x:

$$y = mx + b$$

A similar representation may of course be based on the X intercept.

A1.5.4 Ratios

A two-dimensional or three-dimensional line may be represented as an equation of ratios arising from two points $\mathbf{x1} = (x_1, y_1, z_1)$ and $\mathbf{x2} = (x_2, y_2, z_2)$ on the line.

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} = \frac{z - z_1}{z_2 - z_1}$$

A1.5.5 Normal and Distance from Origin (Line Equation)

This representation for two-dimensional lines is elegant in that its parts have useful geometric significance which extends to planes (not to three-dimensional lines). The coefficients of the general two-dimensional linear equation represent a two-dimensional line and incidentally give its normal (perpendicular) vector and its (perpendicular) distance from the origin (Fig. A1.6).

From the ratio representation above, it is easy to derive (in two dimensions) that

$$(x - x_1) \sin\theta \quad - (y - y_1) \cos\theta = 0$$

so for

$$d = -(x_1 \sin\theta - y_1 \cos\theta),$$
$$x \sin\theta - y \cos\theta + d = 0$$

This equation has the form of a dot product with a formal homogeneous vector $(x, y, 1)$:

$$(x, y, 1) \cdot (\sin\theta, -\cos\theta, d) = 0$$

Here the two-dimensional vector $(\sin\theta, -\cos\theta)$ is perpendicular to the line (it is a unit normal vector, in fact), and d is the signed distance in the direction of the normal vector from the line to the origin. Multiplying both sides of the equation by a constant leaves the line invariant, but destroys the interpretation of d as the distance to the origin.

This form of line representation has several advantages besides the interpretations of its parameters. The parameters never go to infinity (this is useful in the Hough algorithm described in Chapter 4). The representation extends naturally to representing n-dimensional planes. Least squared error line fitting (Section A1.9) with this form of line equation (as opposed to slope-intercept) minimizes errors perpendicular to the line (as opposed to those perpendicular to one of the coordinate axes).

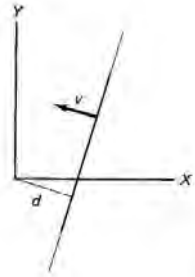


Fig. A1.6 Two-dimensional line with normal vector and distance to origin.

A1.5.6 Parametric

It is sometimes useful to be able mathematically to "walk along" a line by varying some parameter t. The basic parametric representation here follows from the two-point representation. If **x1** and **x2** are two particular points on the line, a general point on the line may be written as

$$\mathbf{x} = \mathbf{x1} + t(\mathbf{x2} - \mathbf{x1})$$

In matrix terms this is

$$\mathbf{x} = [t \quad 1]L$$

where L is the $2 \times n$ matrix whose first row is $(\mathbf{x2} - \mathbf{x1})$ and whose second is **x1**. Parametric representations based on points on the lines may be transformed by the geometric point transformations (Section A1.7).

A1.6. PLANES

The most common representation of planes is to use the coordinates of the plane equation. This representation is an extension of the line-equation representation of Section A1.5.5. The plane equation may be written

$$ax + by + cz + d = 0$$

which is in the form of a dot product $\mathbf{x} \cdot \mathbf{p} = 0$. Four numbers given by $\mathbf{p} = (a, b, c, d)$ characterize a plane, and any homogeneous point $\mathbf{x} = (x, y, z, w)$ satisfying the foregoing equation lies in the plane. In **p**, the first three numbers (a, b, c) form a normal vector to the plane. If this normal vector is made to be a unit vector by scaling **p**, then d is the signed distance to the origin from the plane. Thus the dot product of the plane coefficient vector and any point (in homogeneous coordinates) gives the distance of the point to the plane (Fig. A1.7).

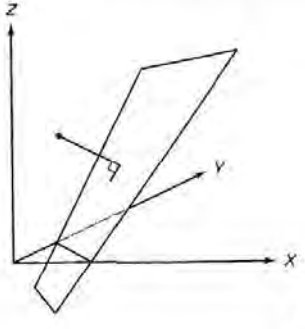


Fig. A1.7 Distance from a point to a plane.

Three noncollinear points **x1**, **x2**, **x3** determine a plane **p**. To find it, write

$$\begin{bmatrix} \mathbf{x1} & & & \\ \mathbf{x2} & & & \\ \mathbf{x3} & & & \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{p} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

If the matrix containing the point vectors can be inverted, the desired vector **p** is thus proportional to the fourth column of the inverse.

Three planes **p1**, **p2**, **p3** may intersect in a point **x**. To find it, write

$$\mathbf{x} \begin{bmatrix} \mathbf{p1} & \mathbf{p2} & \mathbf{p3} & 0 \\ & & & 0 \\ & & & 0 \\ & & & 1 \end{bmatrix} = [0 \quad 0 \quad 0 \quad 1]$$

If the matrix containing the plane vectors can be inverted, the desired point **p** is given by the fourth row of the inverse. If the planes do not intersect in a point, the inverse does not exist.

A1.7 GEOMETRIC TRANSFORMATIONS

This section contains some results that are well known through their central place in the computer graphics literature, and illustrated in greater detail there. The idea is to use homogeneous coordinates to allow the writing of important transformations (including affine and projective) as linear transformations. The transformations of interest here map points or point sets onto other points or point sets. They include rotation, scaling, skewing, translation, and perspective distortion (point projection) (Fig. A1.8).

A point x in three-space is written as the homogeneous row four-vector (x, y, z, w), and postmultiplication by the following transformation matrices accomplishes point transformation. A set of m points may be represented as an $m \times 4$ matrix of row point vectors, and the matrix multiplication transforms all points at once.

A1.7.1 Rotation

Rotation is measured clockwise about the named axis while looking along the axis toward the origin.

Rotation by θ about the X axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

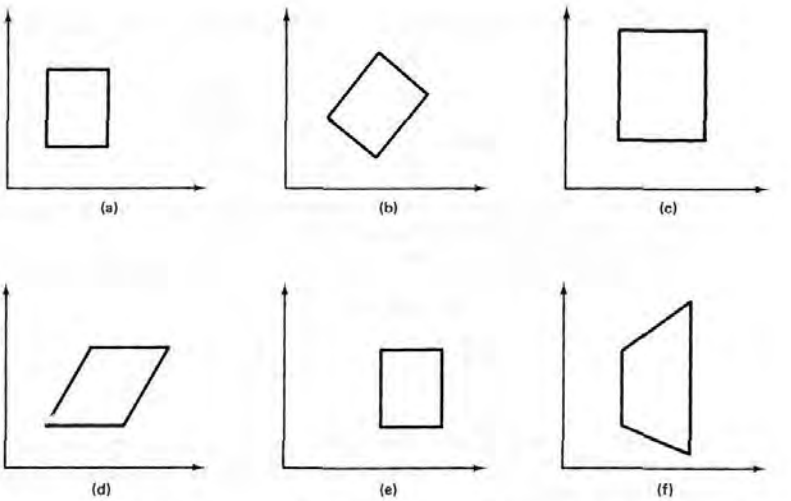


Fig. A1.8 Transformations: (a) original, (b) rotation, (c) scaling, (d) skewing, (e) translation, and (f) perspective.

Rotation by θ about the Y axis:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation by θ about the Z axis:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A1.7.2 Scaling

Scaling is stretching points out along the coordinate directions. Scaling can transform a cube to an arbitrary rectangular parallelepiped.

Scale by S_x, S_y, and S_z in the X, Y, and Z directions:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A1.7.3 Skewing

Skewing is a linear change in the coordinates of a point based on certain of its other coordinates. Skewing can transform a square into a parallelogram in a simple case:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ d & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In general, skewing is quite powerful:

$$\begin{bmatrix} 1 & k & n & 0 \\ d & 1 & p & 0 \\ e & m & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation is a composition of scaling and skewing (Section A1.7.7).

A1.7.4 Translation

Translate a point by (t, u, v):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t & u & v & 1 \end{bmatrix}$$

With a three-dimensional Cartesian point representation, this transformation is accomplished through vector addition, not matrix multiplication.

A1.7.5 Perspective

The properties of point projection, which model perspective distortion, were derived in Chapter 2. In this formulation the viewpoint is on the positive Z axis at $(0, 0, f, 1)$ looking toward the origin: f acts like a "focal length". The visible world is projected through the viewpoint onto the $Z = 0$ image plane (Fig. A1.9).

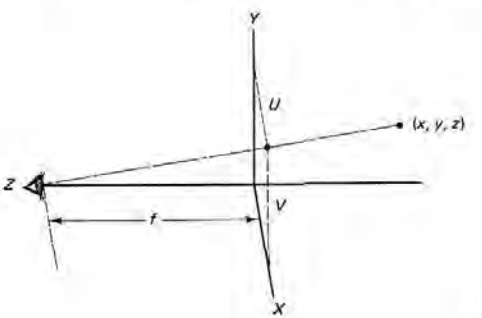


Fig. A1.9 Geometry of image formation.

Similar triangles arguments show that the image plane point for any world point (x, y, z) is given by

$$(U, V) = \left(\frac{fx}{f-z}, \frac{fy}{f-z} \right)$$

Using homogeneous coordinates, a "perspective distortion" transformation can be written which distorts three-dimensional space so that after orthographic projection onto the image plane, the result looks like that required above for perspective distortion. Roughly, the transformation shrinks the size of things as they get more distant in Z. Although the transformation is of course linear in homogeneous coordinates, the final step of changing to Cartesian coordinates by dividing through by the fourth vector element accomplishes the nonlinear shrinking necessary.

Perspective distortion (situation of Fig. A1.9):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \frac{-1}{f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective from a general viewpoint has nonzero elements in the entire fourth column, but this is just equivalent to a rotated coordinate system and the perspective distortion above (Section A1.7).

A1.7.6 Transforming Lines and Planes

Line and plane equations may be operated on by linear transformations, just as points can. Point-based parametric representations of lines and planes transform as do points, but the line and plane equation representations act differently. They have an elegant relation to the point transformation. If T is a transformation matrix (3×3 for two dimensions, 4×4 for three dimensions) as defined in Sections A1.7.1 to A1.7.5, then a point represented as a row vector is transformed as

$$\mathbf{x}' = \mathbf{x}T$$

and the linear equation (line or plane) when represented as a column vector $\mathbf{v}$ is transformed by

$$\mathbf{v}' = T^{-1}\mathbf{v}$$

A1.7.7 Summary

The 4×4 matrix formulation is a way to unify the representation and calculation of useful geometric transformations, rigid (rotation and translation), and nonrigid

(scaling and skewing), including the projective. The semantics of the matrix are summarized in Fig. A1.10.

Since the results of applying a transformation to a row vector is another row vector, transformations may be concatenated by repeated matrix multiplication. Such composition of transformations follows the rules of matrix algebra (it is associative but not commutative, for instance). The semantics of

$$\mathbf{x}' = \mathbf{x}ABC$$

is that $\mathbf{x}'$ is the vector resulting from applying transformation A to $\mathbf{x}$, then B to the transformed $\mathbf{x}$, then C to the twice-transformed $\mathbf{x}$. The single 4×4 matrix $D = ABC$ would do the same job. The inverses of geometric transformation matrices are just the matrices expressing the inverse transformations, and are easy to derive.

A1.8. CAMERA CALIBRATION AND INVERSE PERSPECTIVE

The aim of this section is to explore the correspondence between world and image points. A (half) line of sight in the world corresponds to each image point. Camera calibration permits prediction of where in the image a world point will appear. Inverse perspective transformation determines the line of sight corresponding to an image point. Given an inverse perspective transform and the knowledge that a visible point lies on a particular world plane (say the floor, or in a planar beam of light), then its precise three-dimensional coordinates may be found, since the line of sight generally intersects the world plane in just one point.

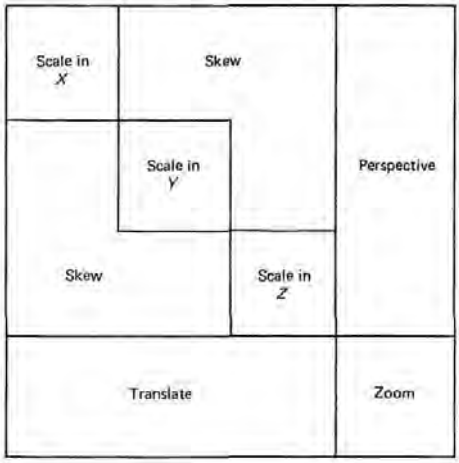


Fig. A1.10 The 4×4 homogeneous transformation matrix.

A1.8.1 Camera Calibration

This section is concerned with the "camera model"; the model takes the form of a 4×3 matrix mapping three-dimensional world points to two-dimensional image points. There are many ways to derive a camera model. The one given here is easy to state mathematically; in practice, a more general optimization technique such as hill climbing can be most effective in finding the camera parameters, since it can take advantage of any that are already known and can reflect dependencies between them.

Let the image plane coordinates be U and V; in homogeneous coordinates an image plane point is (u,v,t). Thus

$$U = \frac{u}{t}$$

$$V = \frac{v}{t}$$

Call the desired camera model matrix C, with elements C_{ij} and column four-vectors C_j. Then for any world point (x, y, z) a C is needed such that

$$(x, y, z, 1)C = (u, v, t)$$

So

$$u = (x, y, z, 1)C_1$$
$$v = (x, y, z, 1)C_2$$
$$t = (x, y, z, 1)C_3$$

Expanding the inner products and rewriting $u - Ut = 0$ and $v - Vt = 0$,

$$xC_{11} + yC_{21} + zC_{31} + C_{41} - UxC_{13} - UyC_{23} - UzC_{33} - UC_{43} = 0$$
$$xC_{12} + yC_{22} + zC_{32} + C_{42} - VxC_{13} - VyC_{23} - VzC_{33} - VC_{43} = 0$$

The overall scaling of C is irrelevant, thanks to the homogeneous formulation, so C_{43} may be arbitrarily set to 1. Then equations such as those above can be written in matrix form:

$$\begin{bmatrix} x^1 & y^1 & z^1 & 1 & 0 & 0 & 0 & 0 & -U^1x^1 & -U^1y^1 & -U^1z^1 \\ 0 & 0 & 0 & 0 & x^1 & y^1 & z^1 & 1 & -V^1x^1 & -V^1y^1 & -V^1z^1 \\ x^2 & y^2 & z^2 & 1 & \cdot & \cdot & \cdot & & & & \cdot \\ \cdot & & & & & & & & & & \cdot \\ \cdot & & & & & & & & & & \cdot \\ \cdot & & & & & & & & & & \cdot \\ 0 & 0 & 0 & 0 & x^n & y^n & z^n & 1 & -V^nx^n & -V^ny^n & -V^nz^n \end{bmatrix} \begin{bmatrix} C_{11} \\ C_{21} \\ \cdot \\ \cdot \\ \cdot \\ C_{34} \end{bmatrix} = \begin{bmatrix} U^1 \\ V^1 \\ \cdot \\ \cdot \\ U^n \\ V^n \end{bmatrix}$$

Eleven such equations allow a solution for C. Two equations result for every association of an (x, y, z) point with a (U, V) point. Such an association must be established using visible objects of known location (often placed for the purpose). If more than 5½ such observations are used, a least-squared-error solution to the overdetermined system may be obtained by using a pseudo-inverse to solve the resulting matrix equation (Section A1.9).

A1.8.2 Inverse Perspective

Finding the world line corresponding to an image point relies on the fact that the perspective transformation matrix also affects the z component of a world point. This information is lost when the z component is projected away orthographically, but it encodes the relation between the focal point and the z position of the point. Varying this third component references points whose world positions vary in z but which project onto the same position in the image. The line can be parameterized by a variable p that formally occupies the position of that z coordinate in three-space that has no physical meaning in imaging.

Write the inverse perspective transform P^{-1} as

$$(x', y', p, 1)P^{-1} = (x', y', p, 1 + \frac{p}{f})$$

Rewriting this in the usual way gives these relations between the (x, y, z) points on the line.

$$(x, y, z, 1) = \left(\frac{fx'}{f+p}, \frac{fy'}{f+p}, \frac{fp'}{f+p}, 1\right)$$

Eliminating the parameter p between the expressions for z and x and those for z and y leaves

$$x = \frac{x'}{y'}\, y = \frac{-x'}{f}\,(z - f)$$

Thus x, y, and z are linearly related; as expected, all points on the inverse perspective transform of an image point lie in a line, and unsurprisingly both the viewpoint $(0, 0, f)$ and the image point $(x', y', 0)$ lie on it.

A camera matrix C determines the three-dimensional line that is the inverse perspective transform of any image point. Scale C so that $C_{43} = 1$, and let world points be written $\mathbf{x} = (x, y, z, 1)$ and image points $\mathbf{u} = (u, v, t)$. The actual image points are then

$$U = \frac{u}{t},\ V + \frac{v}{t}, \qquad \text{so } u = Ut, \quad v + Vt$$

Since

$$\begin{aligned} \mathbf{u} &= \mathbf{x}C, \\ u &= Ut = \mathbf{x}C_1 \\ v &= Vt = \mathbf{x}C_2 \\ t &= \mathbf{x}C_3 \end{aligned}$$

Substituting the expression for t into that for u and v gives

$$U\mathbf{x}C_3 = \mathbf{x}C_1$$
$$V\mathbf{x}C_3 = \mathbf{x}C_2$$

which may be written

$$\mathbf{x}(C_1 - UC_3) = 0$$
$$\mathbf{x}(C_2 - VC_3) = 0$$

These two equations are in the form of plane equations. For any U, V in the image and camera model C, there are determined two planes whose intersection gives the desired line. Writing the plane equations as

$$a_1x + b_1y + c_1z + d_1 = 0$$
$$a_2x + b_2y + c_2z + d_2 = 0$$

then

$$a_1 = C_{11} - C_{13}U \qquad a_2 = C_{12} - C_{13}V$$

and so on. The direction (λ, μ, ν) of the intersection of two planes is given by the cross product of their normal vectors, which may now be written as

$$(\lambda, \mu, \nu) = (a_1, b_1, c_1) \times (a_2, b_2, c_2)$$
$$= (b_1c_2 - b_2c_1,\ c_1a_2 - c_2a_1,\ a_1b_2 - a_2b_1)$$

Then if $\nu \neq 0$, for any particular z_0,

$$x_0 = \frac{b_1(c_2z_0 + d_2) - b_2(c_1z_0 - d_1)}{a_1b_2 - b_1a_2}$$

$$y_0 = \frac{a_2(c_1z_0 + d_1) - a_1(c_2z_0 - d_2)}{a_1b_2 - b_1a_2}$$

and the line may be written

$$\frac{x - x_0}{\lambda} = \frac{y - y_0}{\mu} = \frac{z - z_0}{\nu}$$

A1.9. LEAST-SQUARED-ERROR FITTING

The problem of fitting a simple functional model to a set of data points is a common one, and is the concern of this section. The subproblem of fitting a straight line to a set of (x, y) points ("linear regression") is the first topic. In computer vision, this line-fitting problem is encountered relatively often. Model-fitting methods try to find the "best" fit; that is, they minimize some error. Methods which yield closed-form, analytical solutions for such best fits are at issue here.

The relevant "error" to minimize is determined partly by assumptions of dependence between variables. If x is independent, the line may be represented as $y = mx + b$ and the error defined as the vertical displacement of a point from the line. Symmetrically, if x is dependent, horizontal error should be minimized. If neither variable is dependent, a reasonable error to minimize is the perpendicular distance from points to the line. In this case the line equation $ax + by + 1 = 0$ can be used with the method shown here, or the eigenvector approach of Section A1.9.2 may be used.

A1.9.1 Pseudo-Inverse Method

In fitting an $n \times 1$ observations matrix y by some linear model of p parameters, the prediction is that the linear model will approximate the actual data. Then

$$Y = XB + E$$

where X is an $n \times p$ formal independent variable matrix, B is a $p \times 1$ parameter matrix whose values are to be determined, and E represents the difference between the prediction and the actuality: it is an $n \times 1$ error matrix.

For example, to fit a straight line $y = mx + b$ to some data (x_i, y_i) points, form Y as a column matrix of the y_i.

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ & \cdot \\ & \cdot \\ & \cdot \end{bmatrix}$$

$$B = \begin{bmatrix} b \\ m \end{bmatrix}$$

Now the task is to find the parameter B (above, the b and m that determine the straight line) that minimizes the error. The error is the sum of squared difference from the prediction, or the sum of the elements of E squared, or E^TE (if we do not mind conflating the one-element matrix with a scalar). The mathematically attractive properties of the squared-error definition are almost universally taken to compensate for whatever disadvantages it has over what is really meant by error (the absolute value is much harder to calculate with, for example).

To minimize the error, simply differentiate it with respect to the elements of B and set the derivative to 0. The second derivative is positive: this is indeed a minimum. These elementwise derivatives are written tersely in matrix form. First rewrite the error terms:

$$\begin{aligned} E^TE &= (Y - XB)^T(Y - XB) \\ &= Y^TY - B^TX^TY - Y^TXB + B^TX^TXB \\ &= Y^TY - 2B^TX^TY + B^TX^TXB \end{aligned}$$

(here, the combined terms were 1×1 matrices.) Now differentiate: setting the derivative to 0 yields

$$0 = X^T XB - X^T Y$$

and thus

$$B = (X^T X)^{-1} X^T Y = X^\dagger Y$$

where $X^\dagger$ is called the pseudo-inverse of X.

The pseudo-inverse method generalizes to fitting any parametrized model to data (Section A1.9.3). The model should be chosen with some care. For example, Fig. A1.11 shows a disturbing case in which the model above (minimize vertical errors) is used to fit a relatively vertical swarm of points. The "best fit" line in this case is not the intuitive one.

A1.9.2 Principal Axis Method

The principal axes and moments of a swarm of points determine the direction and amount of its dispersion in space. These concepts are familiar in physics as the principal axes and moments of inertia. If a swarm of (possibly weighted) points is translated so that its center of mass (average location) is at the origin, a symmetric matrix M may be easily calculated whose eigenvectors determine the best-fit line or plane in a least-squared-perpendicular-error sense, and whose eigenvalues tell how good the resulting fit is.

Given a set $\{\mathbf{x}^i\}$ row of vectors with weights w^i, define their "scatter matrix" to be the symmetric matrix M, where $\mathbf{x}^i = (x_1^i, x_2^i, x_3^i)$:

$$M = \sum_i \mathbf{x}^{iT} \mathbf{x}^i$$

$$M_{kp} = \sum_i x_k^i x_p^i \qquad 1 \leqslant k, p \leqslant 3$$

Define the dispersion of the $\mathbf{x}^i$ in a direction $\mathbf{v}$ (i.e., "dispersion around the plane whose normal is $\mathbf{v}$") to be the sum of weighted squared lengths of the $\mathbf{x}^i$ in the direction $\mathbf{v}$. This squared error E^2 is

$$E^2 = \sum_i w^i (\mathbf{x}^i \cdot \mathbf{v})^2 = \mathbf{v} \left(\sum_i w^i \mathbf{x}^{iT} \mathbf{x}^i\right) \mathbf{v}^T = \mathbf{v} M \mathbf{v}^T$$

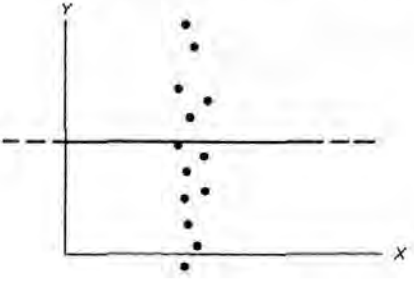


Fig. A1.11 A set of points and the "best fit" line minimizing error in Y.

To find the direction of minimum dispersion (the normal to the best-fit line or plane), note that the minimum of $\mathbf{v}M\mathbf{v}^T$ over all unit vectors $\mathbf{v}$ is the minimum eigenvalue λ_1 of M. If $\mathbf{v}_1$ is the corresponding eigenvector, the minimum dispersion is attained at $\mathbf{v} = \mathbf{v}_1$. The best fit line or plane of the points goes through the center of mass, which is at the origin; inverting the translation that brought the centroid to the origin yields the best fit line or plane for the original point swarm.

The eigenvectors correspond to dispersions in orthogonal directions, and the eigenvalues tell how much dispersion there is. Thus with a three-dimensional point swarm, two large eigenvalues and one small one indicate a planar swarm whose normal is the smallest eigenvector. Two small eigenvalues and one large one indicate a line in the direction of the normal to the "worst fit plane", or eigenvector of largest eigenvalue. (It can be proved that in fact this is the best-fit line in a least squared perpendicular error sense). Three equal eigenvalues indicate a "spherical" swarm.

A1.9.3 Fitting Curves by the Pseudo-Inverse Method

Given a function $f(\mathbf{x})$ whose value is known on n points $\mathbf{x}_1, \ldots, \mathbf{x}_n$, it may be useful is to fit it with a function $g(\mathbf{x})$ of m parameters $(b_1, \ldots, b_m)$. If the squared error at a point x_i is defined as

$$(e_i)^2 = [f(\mathbf{x}_i) - g(\mathbf{x}_i)]^2$$

a sequence of steps similar to that of Section A1.9.1 leads to setting a derivative to zero and obtaining

$$0 = G^T G\mathbf{b} - G^T\mathbf{f}$$

where $\mathbf{b}$ is the vector of parameters, $\mathbf{f}$ the vector of n values of $f(\mathbf{x})$, and

$$G = \frac{\partial \mathbf{g}}{\partial \mathbf{b}} = \begin{bmatrix} \dfrac{\partial g(x_1)}{\partial b_1} & \dfrac{\partial g(x_2)}{\partial b_2} & \cdots \\ \vdots & & \\ \cdots & \cdots & \dfrac{\partial g(x_n)}{\partial b_m} \end{bmatrix}$$

As before, this yields

$$\mathbf{b} = (G^T G)^{-1} G^T\mathbf{f}$$

Explicit least-squares solutions for curves can have nonintuitive behavior. In particular, say that a general circle is represented

$$\mathcal{G}(x, y) = x^2 + y^2 + 2Dx + 2Ey + F$$

this yields values of D, E, and F which minimize

$$e^2 = \sum_{i=1}^{n} \mathcal{G}(x_i, y_i)^2$$

for n input points. The error term being minimized does not turn out to accord with our intuitive one. It gives the intuitive distance of a point to the curve, but weighted by a factor roughly proportional to the radius of the curve (probably not desirable). The best fit criterion thus favors curves with high average curvature, resulting in smaller circles than expected. In fitting ellipses, this error criterion favors more eccentric ones.

The most successful conic fitters abandon the luxury of a closed-form solution and go to iterative minimization techniques, in which the error measure is adjusted to compensate for the unwanted weighting, as follows.

$$e^2 = \sum_{i=1}^{n} \left(\frac{f(x_i, y_i)}{|\nabla f(x_i, y_i)|} \right)^2$$

A1.10 CONICS

The conic sections are useful because they provide closed two-dimensional curves, they occur in many images, and they are well-behaved and familiar polynomials of low degree. This section gives their equations in standard form, illustrates how the general conic equation may be put into standard form, and presents some sample specific results for characterizing ellipses.

All the standard form conics may be subjected to rotation, translation, and scaling to move them around on the plane. These operations on points affect the conic equation in a predictable way.

Circle: r = radius $\quad x^2 + y^2 = r^2$

Ellipse: a, b = major, minor axes $\quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$

Parabola: $(p, 0)$ = focus, p = directrix $\quad y^2 = 4px$

Hyperbola: vertices $(\pm a, 0)$, asymptotes $y = \pm\left(\frac{b}{a}\right)x \quad \frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$

The general conic equation is

$$Ax^2 + 2Bxy + Cy^2 + 2Dx + 2Ey + F = 0$$

This equation may be written formally as

$$\begin{pmatrix} x & y & 1 \end{pmatrix} \begin{bmatrix} A & B & D \\ B & C & E \\ D & E & F \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \mathbf{x} M \mathbf{x}^T = 0$$

Putting the general conic equation into one of the standard forms is a common analytic geometry exercise. The symmetric 3×3 matrix M may be diagonalized, thus eliminating the coefficients B, D, and E from the equation and reducing it to be close to standard form. The diagonalization amounts to a rigid motion that puts the conic in a symmetric position at the origin. The transformation is in fact the 3×3 matrix E whose rows are eigenvectors of M. Recall that if $\mathbf{v}$ is an eigenvector of M,

$$\mathbf{v}M = \lambda \mathbf{v}$$

Then if D is a diagonal matrix of the three eigenvalues, $\lambda_1, \lambda_2, \lambda_3$,

$$EM = DE$$

but then

$$EME^{-1} = DEE^{-1} = D$$

and M has been transformed by a similarity transformation into a diagonal matrix such that

$$\mathbf{x}D\mathbf{x}^T = 0$$

This general idea is of course related to the principal axis calculation given in Section A1.9.2, and extends to three-dimensional quadric surfaces such as the ellipsoid, cone, hyperbolic paraboloid, and so forth. The general result given above has particular consequences illustrated by the following facts about the ellipse. Given a general conic equation representing an ellipse, its center (x_c, y_c) is given by

$$x_c = \frac{BE - 2CD}{B^2 - 4AC}$$

$$y_c = \frac{2EA - BD}{B^2 - 4AC}$$

The orientation is

$$\theta = \tfrac{1}{2}\tan^{-1}\left(\frac{B}{A-C}\right)$$

The major and minor axes are

$$\frac{-2G}{(A + C) \pm [B^2 + (A - C)^2]^{1/2}}$$

where

$$G = F - (Ax_c^2 + B_{x_c y_c} + Cy_c^2)$$

A1.11 INTERPOLATION

Interpolation fits data by giving values between known data points. Usually, the interpolating function passes through each given data point. Many interpolation methods are known; one of the simplest is Lagrangean interpolation.

A1.11.1 One-Dimensional

Given $n + 1$ points (x_j, y_j), $x_0 < x_1 < \cdots < x_n$, the idea is to produce an nth-degree polynomial involving $n + 1$ so-called Lagrangean coefficients. It is

$$f(x) = \sum_{j=0}^{n} L_j(x) y_j$$

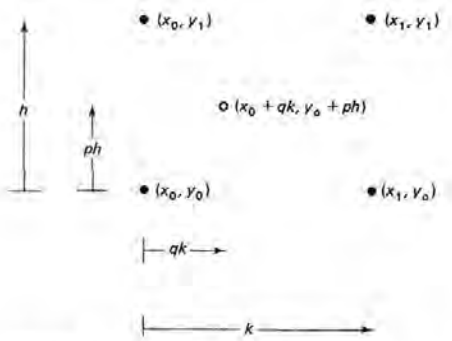


Fig. A1.12 Four point lagrangean interpolation on rectangular grid.

where $L_j(x)$ is the jth coefficient;

$$L_j(x) = \frac{(x-x_0)\ (x-x_1)\ \cdots\ (x-x_{j-1})\ (x-x_{j+1})\ \cdots\ (x-x_n)}{(x_j-x_0)\ (x_j-x_i)\ \cdots\ (x_j-x_{j-1})\ (x_j-x_{j+1})\ \cdots\ (x_j-x_n)}$$

Other interpolative schemes include divided differences, Hermite interpolation for use when function derivatives are also known, and splines. The use of a polynomial interpolation rule can always produce surprising results if the function being interpolated does not behave locally like a polynomial.

A1.11.2 Two-Dimensional

The four-point Lagrangean method is for the situation shown in Fig. A1.12. Let $f_{ij} = f(x_i, y_j)$. Then

$$f(x_0 + qk, y_0 + ph) = (1-p)\ (1-q)\ f_{00} + q(1-p)\ f_{10} + p(1-q)\ f_{01} + pqf_{11}$$

A1.12 THE FAST FOURIER TRANSFORM

The following routine computes the discrete Fourier transform of a one-dimensional complex array XIn of length N = $2^{\log N}$ and produces the one-dimensional complex array XOut. It uses an array W of the N complex Nth roots of unity, computed as shown, and an array Bits containing a bit-reversal table of length N. N, LogN, W, and Bits are all global to the subroutine as written. If the logical variable Forward is TRUE, the FFT is performed; if Forward is FALSE, the inverse FFT is performed.

```
SUBROUTINE FFT(XIn, KOut, Forward)
GLOBAL W, Bits, N, LogN
LOGICAL Forward
COMPLEX XIn, Xout, W, A, B
INTEGER Bits
ARRAY(0:N) W, Bits, XIn, XOut
```

```
DO (I = 0, N − 1) XOut(I) = XIn(Bits(I))
JOff = N/2
JPnt = N/2
JBk = 2
IOFF = 1
DO (I = 1, LogN)
.    DO (IStart = 0, N − 1, JBk)
.    .    JWPnt = 0
.    .    DO (K = IStart, IStart + IOff − 1)
.    .    .    WHEN (Forward)
.    .    .    .    A = XOut(K + IOff) * W(JWPnt) + XOut(K)
.    .    .    .    B = XOut(K + IOff) * W(JWPnt + JOff) + XOut(K)
.    .    .    ...  FIN
.    .    .    ELSE
.    .    .    .    A = XOut (K + IOff) * CONJG(W(JWPnt)) + XOut(K)
.    .    .    .    B = XOut(K + IOff) * CONJG(W(JWPnt + JOff)) + XOu
.    .    .    ...  FIN
.    .    .    XOut(K) = A
.    .    .    XOut(K + IOff) = B
.    .    .    JWPnt = JWPnt + JPnt
.    .    ...  FIN
.    ...  FIN
.    JPnt = JPnt/2
.    IOff = JBk
.    JBk = JBk * 2
...  FIN
UNLESS (Forward)
.    DO (I = 0, N − 1) XOut(I) = XOut(I)/N
...  FIN
END
```

```
TO INIT-W
.    Pi = 3.14159265
.    DO (K = 0, N − 1)
.    .    Theta = 2 * Pi/N
.    .    W(K) = CMPLX(COS(Theta * K), SIN(Theta * K))
.    ...  FIN
...  FIN
```

```
TO BIT-REV
.    Bits(0) = 0
.    M = 1
.    DO (I = 0, LogN − 1)
.    .    DO (J = 0, M − 1)
.    .    .    Bits(J) = Bits(J) * 2
```

```
.   .   .   Bits(J + M) = Bits(J) + 1
.   .   ... FIN
.   .   M = M * 2
.   ... FIN
... FIN
```

A1.13 THE ICOSAHEDRON

Geodesic dome constructions provide a useful way to partition the sphere (hence the three-dimensional directions) into relatively uniform patches. The resulting polyhedra look like those of Fig. A1.13.

The icosahedron has 12 vertices, 20 faces, and 30 edges. Let its center be at the origin of Cartesian coordinates and let each vertex be a unit distance from the center. Define

$$t, \text{ the golden ratio} = \frac{1+\sqrt{5}}{2}$$

$$a = \frac{\sqrt{t}}{5^{1/4}}$$

$$b = \frac{1}{(\sqrt{t}\ 5^{1/4})}$$

$$c = a + 2b = \frac{1}{b}$$

$$d = a + b = \frac{t^{3/2}}{5^{1/4}}$$

$$A = \text{angle subtended by edge at origin} = \arccos\left(\frac{\sqrt{5}}{5}\right)$$

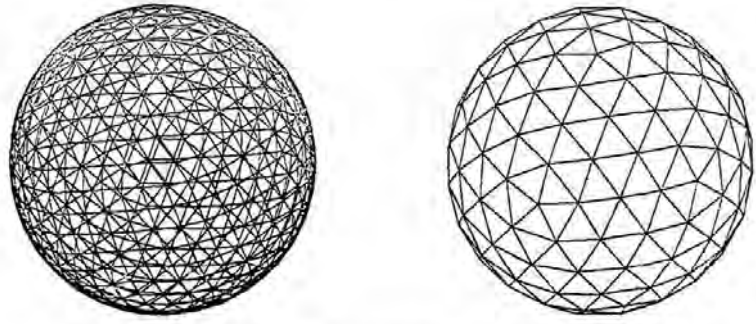


Fig. A1.13 Multifaceted polyhedra from the icosahedron.

Then

angle between radius and an edge $= b = \arccos(b)$
edge length $= 2b$
distance from origin to center of edge $= a$
distance from origin to center of face $= \frac{ta}{\sqrt{3}}$

The 12 vertices may be placed at

$$(0, \pm a, \pm b)$$
$$(\pm b, 0, \pm a)$$
$$(\pm a, \pm b, 0)$$

Then midpoints of the 20 faces are given by

$$\tfrac{1}{3}(\pm d, \pm d, \pm d)$$
$$\tfrac{1}{3}(0, \pm a, \pm c)$$
$$\tfrac{1}{3}(\pm c, 0, \pm a)$$
$$\tfrac{1}{3}(\pm a, \pm c, 0)$$

To subdivide icosahedral faces further, several methods suggest themselves, the simplest being to divide each edge into n equal lengths and then construct n^2 congruent equilateral triangles on each face, pushing them out to the radius of the sphere for their final position. (There are better methods than this if more uniform face sizes are desired.)

A1.14 ROOT FINDING

Since polynomials of fifth and higher degree are not soluble in closed form, numerical (approximate) solutions are useful for them as well as for nonpolynomial functions. The Newton–Raphson method produces successive approximations to a real root of a differentiable function of one variable.

$$x^{i+1} = x^i - \frac{f(x^i)}{f'(x^i)}$$

Here x^i is the ith approximation to the root, and $f(x^i)$ and $f'(x^i)$ are the function and its derivative evaluated at x^i. The new approximation to the root is x^{i+1}. The successive generation of approximations can stop when they converge to a single value. The convergence to a root is governed by the choice of initial approximation to the root and by the behavior of the function in the vicinity of the root. For instance, several roots close together can cause problems.

The one-dimensional form of this method extends in a natural way to solving systems of simultaneous nonlinear equations. Given n functions F_i, each of n parameters, the problem is to find the set of parameters that drives all the functions to zero. Write the parameter vector $\mathbf{x}$.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

Form the function column vector **F** such that

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} F_1(\mathbf{x}) \\ F_2(\mathbf{x}) \\ \cdot \\ \cdot \\ \cdot \\ F_n(\mathbf{x}) \end{bmatrix}$$

The Jacobean matrix J is defined as

$$J = \begin{bmatrix} \dfrac{\partial F_1}{\partial x_1} & \dfrac{\partial F_1}{\partial x_2} & \cdots & \dfrac{\partial F_1}{\partial x_n} \\ \cdot & & & \\ \cdot & & & \\ \dfrac{\partial F_n}{\partial x_1} & \cdots & & \dfrac{\partial F_n}{\partial x_n} \end{bmatrix}$$

Then the extension of the Newton–Raphson formula is

$$\mathbf{x}^{i+1} = \mathbf{x}^i - J^{-1}(\mathbf{x}^i)F(\mathbf{x}^i)$$

which requires one matrix inversion per iteration.

EXERCISES

A1.1 **x** and **y** are two two-dimensional vectors placed tail to tail. Prove that the area of the triangle they define is $|\mathbf{x} \times \mathbf{y}|/2$.

A1.2 Show that points **q** in a plane defined by the three points **x**, **y**, and **z** are given by

$$\boldsymbol{q} \cdot \Big[(\mathbf{y}-\mathbf{x}) \times (\mathbf{z}-\mathbf{x})\Big] = \mathbf{x} \cdot (\mathbf{y} \times \mathbf{z})$$

A1.3 Verify that the vector triple product may be written as claimed in its definition.

A1.4 Given an arctangent routine, write an arcsine routine.

A1.5 Show that the closed form for the inverse of a 2 × 2 A matrix is

$$\frac{1}{\det A}\begin{bmatrix} a_{22} & -a_{21} \\ -a_{12} & a_{11} \end{bmatrix}$$

A1.6 Prove by trigonometry that the matrix transformations for rotation are correct.

A1.7 What geometric transformation is accomplished when a_{44} of a geometric transformation matrix A varies from unity?

A1.8 Establish conversions between the given line representations.

A1.9 Write a geometric transform to mirror points about a given plane.

A1.10 What is the line-equation representation of a line $L1$ through a point $\mathbf{x}$ and perpendicular to a line $L2$ (similarly represented)? Parallel to $L2$?

A1.11 Derive the ellipse results given in Section A1.10.

A1.12 Explicitly derive the values of D, E, and F minimizing the error term

$$\sum_{i=1}^{n} [f(x_i, y_i)]^2$$

in the general equation for a circle

$$x^2 + y^2 + 2Dx + 2Ey + F = 0$$

A1.13 Show that if points and lines are transformed as shown in Section A1.7.6, the transformed points indeed lie on the transformed lines.

A1.14 Explicitly derive the least-squared-error solution for lines represented as $ax + by + 1 = 0$.

A1.15 If three planes intersect in a point, is the inverse of

$$\begin{bmatrix} \mathbf{p1} & \mathbf{p2} & \mathbf{p3} & 0 \\ & & & 0 \\ & & & 0 \\ & & & 1 \end{bmatrix}$$

guaranteed to exist?

A1.16 What is the angle between two three-space lines?

A1.17 In two dimensions, show that two lines $\mathbf{u}$ and $\mathbf{v}$ intersect at a point $\mathbf{x}$ given by $\mathbf{x} = \mathbf{u} \times \mathbf{v}$.

A1.18 How can you tell if two line segments (defined by their end points) intersect in the plane?

A1.19 Find a 4×4 matrix that transforms an arbitrary direction (or point) to lie on the Z axis.

A1.20 Derive a parametric representation for planes based on three points lying in the plane.

A1.21 Devise a scheme for interpolation on a triangular grid.

A1.22 What does the homogeneous point $(x, y, z, 0)$ represent?

REFERENCES AND FURTHER READING

Computer Graphics

1. NEWMAN, W. M., and R. F. SPROULL. *Principles of Interactive Computer Graphics*, 2nd Ed. New York: McGraw-Hill, 1979.
2. CHASEN, S. H. *Geometric Principles and Procedures for Computer Graphic Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

3. FAUX, I. D., and M. J. PRATT. *Computational Geometry for Design and Manufacture*, Chichester, UK: Ellis Horwood Ltd, 1979.
4. ROGERS, D. F., and J. A. ADAMS. *Mathematical Elements for Computer Graphics*. New York: McGraw-Hill 1976.

Computer Vision

5. HORN, B. K. P. "VISMEM: Vision Flash 34." AI Lab, MIT, December 1972.
6. SOBEL, I. "Camera models and machine perception." AIM-21, Stanford AI Lab, May 1970.
7. DUDA, R. O. and P. E. HART. *Pattern Classification and Scene Analysis*. New York: Wiley, 1973.
8. ROSENFELD, A., and A. C. KAK. *Digital Picture Processing*. New York: Academic Press, 1976.
9. PAVLIDIS, T. *Structural Pattern Recognition*. New York: Springer-Verlag, 1977.

Geometry, Calculus, Numerical Analysis

10. WEXLER, C. *Analytic Geometry: A Vector Approach*. Reading, MA: Addison Wesley, 1962.
11. APOSTOL, T. M. *Calculus.*, Vol. 2, Waltham, MA: Blaisdell, 1962.
12. CONTE, S. D. and C. DEBOOR. *Elementary Numerical Analysis: An Algorithmic Approach*. New York: McGraw-Hill, 1972.
13. RALSTON, A. *A First Course in Numerical Analysis*. New York: McGraw-Hill, 1965.
14. ABRAMOWITZ, M, and I. A. STEGUN. *Handbook of Mathematical Functions*. New York: Dover, 1964.
15. HODGEMAN, C. D. (Ed.). *CRC Standard Mathematical Tables*. West Palm Beach, FL: CRC Press.

Geodesic Tesselations

16. BROWN, C. "Fast display of well-tesselated surfaces." *Computers and Graphics 4*, 1979, 77–85.
17. CLINTON, J. D. "Advanced structural geometry studies, part I: polyhedral subdivision concepts for structural applications," NASA CR-1734/35, September 1971.

Fast Transformations

18. PRATT, W. K. *Digital Image Processing*. New York: Wiley-Interscience, 1978.
19. ANDREWS, H. C., and J. KANE. "Kronecker matrices, computer implementation, and generalized spectra." *J. ACM*, April 1970.

Advanced Control Mechanisms

Appendix 2

This appendix is concerned with specific control mechanisms that are provided by programming languages or that may be implemented on top of existing languages as aids to doing computer vision. The treatment here is brief; our aim is to expose the reader to several ideas for control of computer programs that have been developed in the artificial intelligence context, and to indicate how they relate to the main computational goals of computer vision.

A2.1 STANDARD CONTROL STRUCTURES

For completeness, we mention the control mechanisms that are provided as a matter of course by conventional research programming languages, such as Pascal, Algol, POP-2, SAIL, and PL/1. The influential language LISP, which provides a base language for many of the most advanced control mechanisms in computer vision, ironically is itself missing (in its pure form) a substantial number of these more standard constructs. Another common language missing some standard control mechanisms is SNOBOL. These standard constructions are so basic to the current conception of a serial von Neumann computer that they are often realized in the instruction set of the machine. In this sense we are almost talking here of computer hardware.

The standard mechanisms are the following:

1. *Sequence.* Advance the program counter to the next intruction.
2. *Branch instruction.* Go to a specific address.
3. *Conditional branch.* Go to a specific address if a condition is true, otherwise, go to the next instruction.
4. *Iteration.* Repeat a sequence of instructions until a condition is met.

5. *Subroutines.* Go to a certain location; execute a set of instructions using a set of supplied parameters; then return to the next instruction after the subroutine call.

All the standard control structures should be in the toolkit of a programmer. They will be used, together with the data structures and data types supplied in the working language, to implement other control mechanisms. The remainder of this appendix deals with "nonstandard" control mechanisms; those not typically provided in commercial programming languages and which have no close correlates in primitive machine instructions. Nonstandard control mechanisms, although not at all domain-specific, have developed to meet needs that are not the "lowest common denominator" of computer programming. They impose their own view of problem decomposition just as do the standard structures.

Less standard mechanisms are *recursion* and *co-routining.* Co-routining can be thought of as a form of recursion.

A2.1.1 Recursion

Recursion obeys all the constraints of subroutining, except that a routine may call upon "itself." The user sees no difference between recursive and nonrecursive subroutines, but internally recursion requires slightly more bookkeeping to be performed in the language software, since typically the hardware of a computer does not extend to managing recursion (although some machines have instructions that are quite useful here).

A typical use of a recursive control paradigm in computer vision might be:

```
To Understand-Scene (X);
(
If Immediately-Apparent(X)
then Report-Understanding-Of(X);
else
   (SimplerParts ← Decompose(X);
   ForEach Part in SimplerParts
      Understand-Scene(Part);
   )
[ )
```

Recursion is an elegant way to specify many important algorithms (such as tree traversals), but in a way it has no conceptual differences from subroutining. A routine is broken up into subroutines (some of which may involve smaller versions of the original task); these are attacked sequentially, and they must finish before they return control to the routine that invokes them.

A2.1.2 Co-Routining

Co-routines are simply programs that can call (invoke) each other. Most high-level languages do not directly provide co-routines, and thus they are a nonstandard control structure. However, co-routining is a fundamental concept [Knuth 1973]

and serves here as a bridge between standard and nonstandard control mechanisms.

Subroutines and their calling programs have a "slave–master" aspect: control is always returned to the master calling program after the subroutine has carried out its job. This mechanism not only leads to efficiencies by reducing the amount of executable code, but is considered to be so useful that it is built into the instruction set of most computers. The pervasiveness of subroutining has subtle effects on the approach to problem decomposition, encouraging a hierarchical subproblem structure. The co-routine relationship is more egalitarian than the subroutine relationship. If co-routine *A* needs the services of co-routine *B*, it can call *B*, and (here is the difference) conversely, *B* can call *A* if *B* needs *A*'s services.

Here is a simple (sounding) problem [Floyd 1979]: "Read lines of text, until a completely blank line is found. Eliminate redundant blanks between the words. Print the text, 30 characters to a line, without breaking words between lines." This problem is hard to program elegantly in most languages because the iterations involved do not nest well (try it!). However, an elegant solution exists if the job is decomposed into three co-routines, calling each other to perform input, formatting, and output of a character stream.

A useful paradigm for problem solving, besides the strictly hierarchical, is that of a "heterarchical" community of experts, each performing a job and when necessary calling on other experts. A heterarchy can be implemented by co-routines. Many of the nonstandard mechanisms discussed below are in the spirit of co-routines.

A2.2 INHERENTLY SEQUENTIAL MECHANISMS

A2.2.1 Automatic Backtracking

The PLANNER language [Hewitt 1972] implicitly implemented the feature of "automatic backtracking." The advisability of uniformly using this technique, which is equivalent to depth-first search, was questioned by those who wished to give the programmer greater freedom to choose which task to activate next [Sussman and McDermott 1972].

A basic backtracking discipline may be provided by recursive calls, in which a return to a higher level is a "backtrack." The features of automatic backtracking are predicated on an ability to save and reinstate the computational state of a process automatically, without explicit specification by the programmer.

Automatic backtracking has its problems. One basic problem occurs in systems that perform inferences while following a particular line of reasoning which may ultimately be unsuccessful. The problem is that along the way, perhaps many perfectly valid and useful computations were performed and many facts were added to the internal model. Mixed in with these, of course, are wrong deductions which ultimately cause the line of reasoning to fail. The problem: After having restored control to a higher decision point after a failure is noticed, how is the system

to know which deductions were valid and which invalid? One expensive way suggested by automatic backtracking is to keep track of all hypotheses that contributed to deriving each fact. Then one can remove all results of failed deduction paths. This is generally the wrong thing to do; modern trends have abandoned the automatic backtracking idea and allow the programmer some control over what is restored upon failure-driven backtracking. Typically, a compromise is implemented in which the programmer may mark certain hypotheses for deletion upon backtracking.

A2.2.2 Context Switching

Context switching is a general term that is used to mean switching of general process state (a control primitive) or switching a data base context (a data access primitive). The two ideas are not independent, because it could be confusing for a process to put itself to sleep and be reawakened in a totally different data context.

Backtracking is one use of general control context switching. The most general capability is a "general GO TO." A regular GO TO allows one to go only to a particular location defined in a static program. After the GO TO, all bindings and returnpoints are still determined by the current state of processing. In contrast, a general GO TO allows a transfer not only across program "space," but through program "time" as well. Just as a regular GO TO can go to a predefined program label, a general GO TO can go to a "tag" which is created to save the entire state of a process. To GO TO such a tag is to go back in time and recreate the local binding, access, control, and process state of the process that made the tag.

A good example of the use of such power is given in a problem-solving program that constructs complex structures of blocks [Fahlman 1974].

A2.3 SEQUENTIAL OR PARALLEL MECHANISMS

Some language constructs explicity designate parallel computing. They may actually reflect a parallel computing environment, but more often they control a simulated version in which several control paths are maintained and multi-processed under system control. Examples here are module and message primitives given below and statements such as the CO-BEGIN, CO-END pairs which can bracket notionally parallel blocks of code in some Algol-like language extensions.

A2.3.1 Modules and Messages

Modules and messages form a useful, versatile control paradigm that is relatively noncommittal. That is, it forces no particular problem decomposition or methodological style on its user, as does a pure subroutine paradigm, for example. Message passing is a general and elegant model of control which can be used to subsume others, such as subroutining, recursion, co-routining, and parallelism [Feldman 1979].

There are many antecedents to the mechanism of modules communicating by messages described here. They include [Feldman and Sproull 1971; Hewitt and

Smith 1975; Goldberg and Kay 1976; Birtwhistle et al. 1973]. In the formulation presented by Hewitt, the message-passing paradigm can be extended down into the lowest level of machine architecture. The construction outlined here [Feldman 1979] is more moderate, since in it the base programming language may be used with its full power, and itself is not module and message based.

A program is made up of *modules*. A module is a piece of code with associated local data. The crucial point is that the internal state of a module (e.g. its data) is not accessible to other modules. Within a module, the base programming language, such as Algol, may be used to its full power (subroutine calls, recursion, iteration, and so forth are allowed). However, modules may not in any sense "call upon" each other. Modules communicate only by means of *messages*. A module may send a message to another module; the message may be a request for service, an informational message, a signal, or whatever. The module to whom the message is sent may, when it is ready, receive the message and process it, and may then itself send messages either to the original module, or indeed to any combination of other modules.

The module–message paradigm has several advantages over subroutine (or co-routine) calls.

1. If subroutines are in different languages, the subroutine call mechanisms must be made compatible.
2. Any sophisticated lockout mechanism for resource access requires the internal coding of queues equivalent to that which a message switcher provides.
3. A subroutine that tries to execute a locked subroutine is unable to proceed with other computation.
4. Having a resource always allocated by a single controlling module greatly simplifies all the common exclusion problems.
5. For inherently distributed resources, message communication is natural. Module-valued slots provide a very flexible but safe discipline for control transfers.

Another view of messages is as a generalization of parameter lists in subroutine or coroutine calls. The idea of explicitly naming parameters is common in assembly languages, where the total number of parameters to a routine may be very large. More important, the message discipline presents to a module a collection of suggested parameters rather than automatically filling in the values of parameters. This leads naturally to the use of semantic checks on the consistency of parameters and to the use of default values for unspecified ones, which can be a substantial improvement on type checking. The use of return messages allows multiple-valued functions; an answer message may have several slots. Messages solve the so-called "uniform reference problem"—one need not be concerned with whether an answer (say an array element) is computed by a procedure or a table.

There is yet another useful view of messages. One can view a message as a partially specified relation (or pattern), with some slot values filled in and some unbound. This is common in relational data bases [Astrahan et al. 1976] and artificial intelligence languages [Bobrow and Raphael 1974]. In this view, a mes-

sage is a task specification with some recipient and some complaint departments to talk to about it. Various modules can attempt to satisfy or contract out parts of the task of filling in the remaining slots. A module may handle messages containing slots unknown to it. This allows several modules to work together on a task while maintaining locality. For example, an executive module could route messages (on the basis of a few slots that it understands) to modules that deal with special aspects of a problem using different slots in the message.

There is no apparent conflict between these varying views of messages. It is too early in their development to be sure, but the combined power of these paradigms seems to provide a qualitative improvement in our ability to develop vision programs.

A2.3.2 Priority Job Queue

In any system of independent processes on a serial computer, there must be a mechanism for scheduling activation. One general mechanism for accomplishing scheduling is the priority job queue. Priority queues are a well-known abstraction [Aho et al. 1974]. Informally, a priority job queue is just an ordered list of processes to be activated. A monitor program is responsible for dequeueing processes and executing them; processes do not give control directly to other processes, but only to the monitor. The only way for a process to initiate another is to enqueue it in the job queue. It is easiest to implement a priority job queue if processes are definable entities in the programming language being used; in other words, programs should be manipulable datatypes. This is possible in LISP and POP-2, for example.

If a process needs another job performed by another process, it enqueues the sub job on the job queue and *suspends* itself (it is *deactivated*, or put to sleep). The sub job, when it is dequeued and executed by the monitor, must explicitly enqueue the "calling" process if a subroutining effect is desired. Thus along with usual arguments telling a job what data to work on, a job queue discipline implies passing of control information.

Job queues are a general implementational technique useful for simulating other types of control mechanisms, such as active knowledge (Chapter 12). Also, a job queue can be used to switch between jobs which are notionally executing in parallel, as is common in multiprocessing systems. In this case sufficient information must be maintained to start the job at arbitrary points in its execution.

An example of a priority job queue is a program [Ballard 1978] that locates ribs in chest radiographs. The program maintains a relational model of the ribcage including geometric and procedural knowledge. Uninstantiated model nodes corresponding to ribs might be called hypotheses that those ribs exist. Associated with each hypothesis is a set of procedures that may, under various conditions, be used to verify it (i.e., to find a rib). Procedures carry information about preconditions that must be true in order that they may be executed, and about how to compute estimates of their utility once executed. These descriptive components allow an executive program to rank the procedures by expected usefulness at a given time.

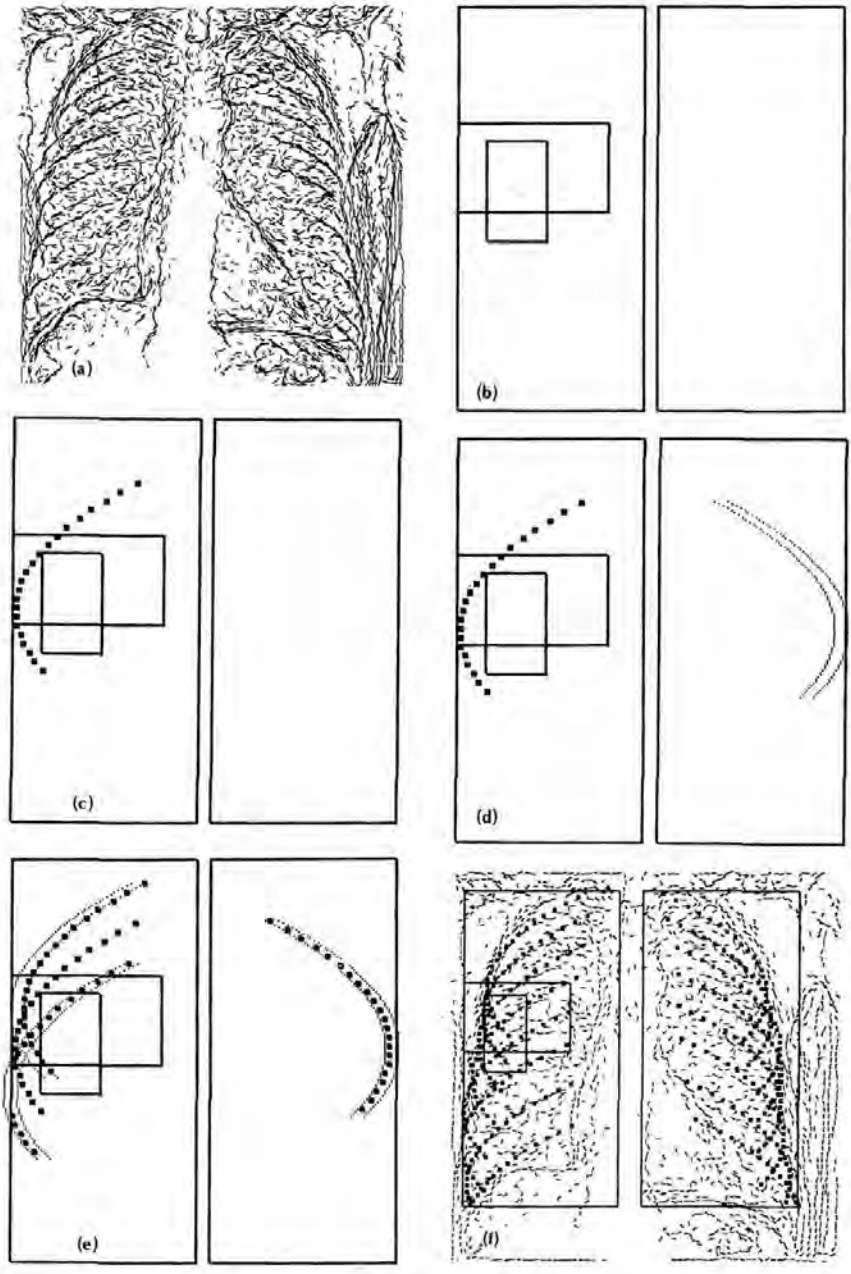


Fig. A2.1 The rib-finding process in action (see text).

There is an initial action that is likely to succeed (locating a particular rib that is usually obvious in the x-ray). In heterarchical fashion, further actions use the results of previous actions. Once the initial rib has been found, its neighbors (both above and below and directly across the body midline) become eligible for consideration.

Eligible rib-finding procedures correspond to short-term plans; they are all put on a job queue to be considered by an *executive* program that must compute the expected utility of expending computational energy on verifying one of the hypotheses by running one of the jobs. The executive computes a priority on the jobs based on how likely they are to succeed, using the utility functions and parameters associated with the individual nodes in the rib model (the individual hypotheses) and the current state of knowledge. The executive not only picks a hypothesis but also the procedure that should be able to verify it with least effort.

The hypothesis is either "verified," "not-verified," or "some evidence is found." Verifying a hypothesis results in related hypotheses (about the neighboring ribs) becoming eligible for consideration. The information found during the verification process is used in several ways that can affect the utility of other procedures.

The position of the rib with respect to instantiated neighbors is used to adjust horizontal and vertical scale factors governing the predicted size of the ribcage. The position of the rib affects the predicted range of locations for other unfound ribs. The shape of the rib also affects the search region for uninstantiated rib neighbors.

If some evidence is found for the rib, but not enough to warrant an instantiation, the rib hypothesis is left on the active list and the rib model node is not instantiated. Rib hypotheses left on the active list will be reconsidered by the executive, which may try them again on the basis of new evidence.

The sequence of figures (Fig. A2.1, p. 503) shows a few steps in the finding of ribs using this program. Figure A2.1a shows the input data. A2.1b shows rectangles enclosing the lung field and the initial area to be searched for a particular rib which is usually findable. Only one rib-finding procedure is applicable for ribs with no neighbors found, so it is invoked and the rib shown by dark boxes in Fig. A2.1b is found. Predicted locations for neighboring ribs are generated and are used in order by the executive which invokes the rib-finding procedures in order of expected utility (A2.1c-e). Predicted locations are shown by dots, actual locations by crosses; in Fig. A2.1f, all modelled ribs are found. The type of procedure that found the rib is denoted by the symbol used to draw in the rib. Figure A2.1f shows the final rib borders superimposed on the data.

A2.3.3 Pattern Directed Invocation

Considerable attention has been focused recently on pattern directed systems (see, e.g., [Waterman and Hayes-Roth 1978]). Another common example of a pattern directed system is the production system, discussed in Section 12.3. The idea behind a pattern directed system is that a procedure will be activated not when its

name is invoked, but when a key situation occurs. These systems have in common that their activity is guided by the appearance of "patterns" of data in either input or memory. Broadly construed, all data forms patterns, and hence patterns guide any computation. This section is concerned with a definition of patterns as something very much smaller than the entire data set, together with the specification of control mechanisms that make use of them.

Pattern directed systems have three components.

1. A data structure or data base containing modifiable items whose structure may be defined in terms of patterns
2. Pattern-directed modules that match patterns in the data structure
3. A controlling executive that selects modules that match patterns and activates them

A popular name for a pattern-directed procedure is a *demon.* Demons were named originally by Selfridge [Selfridge 1959]. They are used successfully in many AI programs, notably in a natural language understanding system [Charniak 1972]. Generally, a demon is a program which is associated with a *pattern* that describes part of the knowledge base (usually the pattern is closely related to the form of "items" in a data base). When a part of the knowledge base matching the pattern is added, modified, or deleted, the demon runs "automatically." It is as if the demon were constantly watching the data base waiting for information associated with certain patterns to change. Of course, in most implementations on conventional computers, demons are not always actively watching. Equivalent behavior is simulated by having the demons register their interests with the system routines that access the data base. Then upon access, the system can check for demon activation conditions and arrange for the interested demons to be run when the data base changes.

Advanced languages that support a sophisticated data base often provide demon facilities, which are variously known as if-added and if-removed procedures, antecedent theorems, traps, or triggers.

A2.3.4 Blackboard Systems

In artificial intelligence literature, a "blackboard" is a special kind of globally accessible data base. The term first became prominent in the context of a large pattern directed system to understand human speech [Erman and Lesser 1975; Erman et al. 1980]. More recently, blackboards have been used as a vision control system [Hanson and Riseman 1978]. Blackboards often have mechanisms associated with them for invoking demons and synchronizing their activities. One can appreciate that programming with demons can be difficult. Since general patterns are being used, one can never be sure exactly when a pattern directed procedure will be activated; often they can be activated in incorrect or bizarre sequences not anticipated by their designer. Blackboards attempt to alleviate this uncertainly by controlling the matching process in two ways:

1. Blackboards represent the current part of the model that is being associated with image data;

2. Blackboards incorporate rules that determine which specialized subsystems of demons are likely to be needed for the current job. This structuring of the data base of procedures increases efficiency and loosely corresponds to a "mental set."

These two ideas are illustrated by Figs. A2.2 and A2.3 [Hanson and Riseman 1978]. Figure A2.2 shows the concept of a blackboard as a repository for only model-image bindings. Figure A2.3 shows transformations between model entities that are used to select appropriate groups of demons.

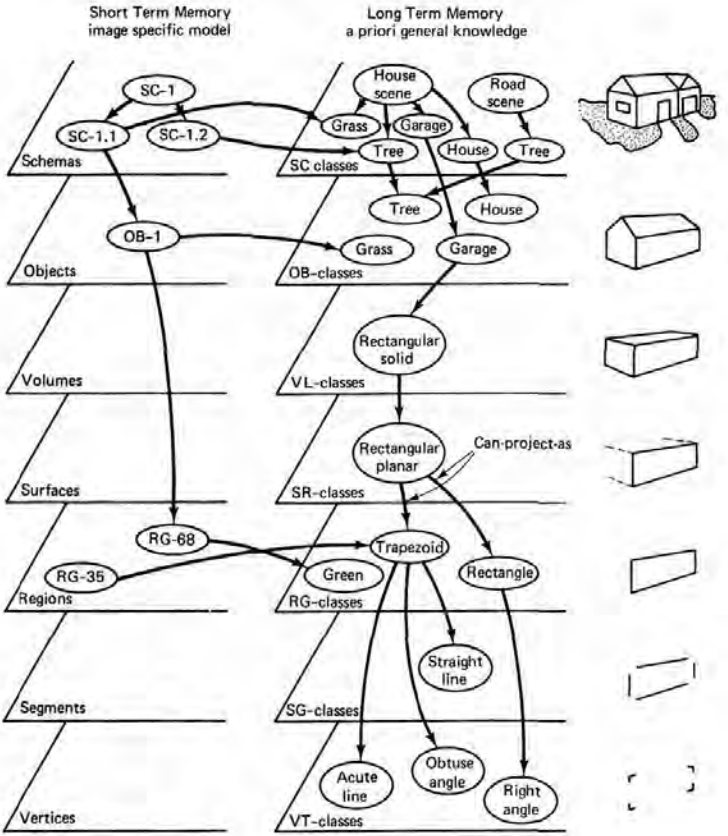


Fig. A2.2 An implementation of the blackboard concept. Here the blackboard is called Short Term Memory; it holds a partial interpretation of a specific image.

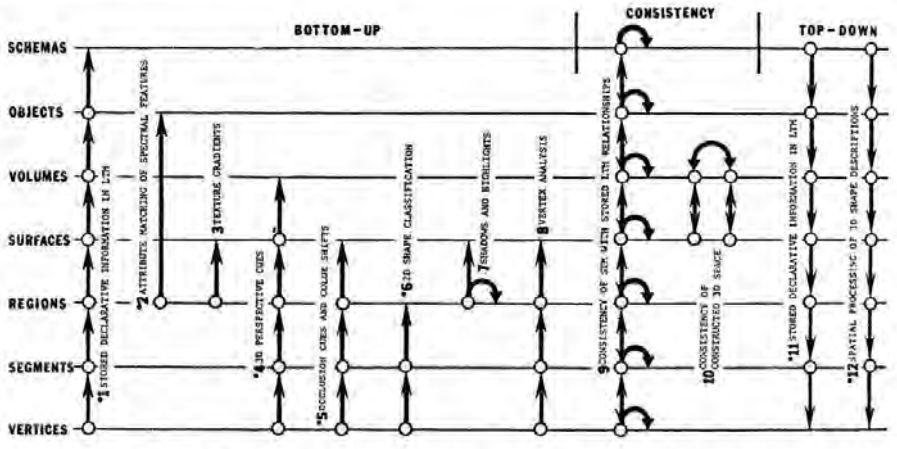


Fig. A2.3 Paths for hypothesis flow, showing transformations between model entities and the sorts of knowledge needed for the transformations.

REFERENCES

AHO, A. V., J. E. HOPCROFT and J. D. ULLMAN. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

ASTRAHAN, M. M. et al. "System R: A relational approach to data base management." IBM Research Lab, February 1976.

BALLARD, D. H. "Model-directed detection of ribs in chest radiographs." Proceedings, Fourth IJCPR, Kyoto, Japan, 1978.

BIRTWHISTLE, G. et al. *Simula Begin*. Philadelphia: Auerbach, 1973.

BOBROW, D. G. and B. RAPHAEL. "New programming languages for artificial intelligence." *Computing Surveys 6*, 3, September 1974, 155–174.

CHARNIAK, E. "Towards a model of children's story comprehension." AI-TR-266, AI Lab, MIT, 1972.

ERMAN, L. D. and V. R. LESSER. "A multi-level organization for problem solving using many diverse cooperating sources of knowledge." *Proc.*, 4th IJCAI, September 1975, 483–490.

ERMAN, L. D., F. HAYES-ROTH, V. R. LESSER, and D. R. REDDY. "The HEARSAY-II speech-understanding system: Integrating knowledge to resolve uncertainty." *Computing Surveys 12*, 2, June 1980, 213–253.

FAHLMAN, S. E. "A planning system for robot construction tasks." *Artificial Intelligence 5*, 1, Spring 1974, 1–49.

FELDMAN, J. A. "High-level programming for distributed computing." *Comm. ACM 22*, 6, July 1979, 363–368.

FELDMAN, J. A. and R. F. SPROULL. "System support for the Stanford hand-eye system." *Proc.*, 2nd IJCAI, September 1971, 183–189.

FLOYD, R. W. "The paradigms of programming." *Comm. ACM 22*, 8, August 1979, 455–460.

GOLDBERG, A. and A. KAY (Eds). "SMALLTALK-72 Instruction Manual." SSL 76-6, Xerox PARC, Palo Alto, CA, 1976.

HANSON, A. R. and E. M. RISEMAN. "Visions: A computer system for interpreting scenes." In *CVS*, 1978.

HEWITT, C. "Description and theoretical analysis (using schemata) of PLANNER" (Ph.D. dissertation). AI-TR-258, AI Lab, MIT, 1972.

HEWITT, C. and B. SMITH. "Towards a programming apprentice." *IEEE Trans. Software Engineering. 1*, 1, March 1975, 26–45.

KNUTH, D. E. *The Art of Computer Programming*, Vol. 1. Reading, MA: Addison-Wesley, 1973.

SELFRIDGE, O. "Pandemonium, a paradigm for learning." In *Proc.*, Symp. on the Mechanisation of Thought Processes, National Physical Laboratory, Teddington, England, 1959.

SUSSMAN, G. J. and D. MCDERMOTT. "Why conniving is better than planning." AI Memo 255A, AI Lab, MIT, 1972.

WATERMAN, D. A. and F. HAYES-ROTH (Eds.). *Pattern-Directed Inference Systems*. New York: Academic Press, 1978.

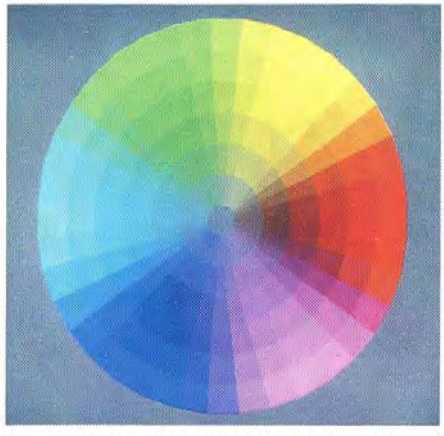
Author Index

Subject Index

FIG. 2-7a

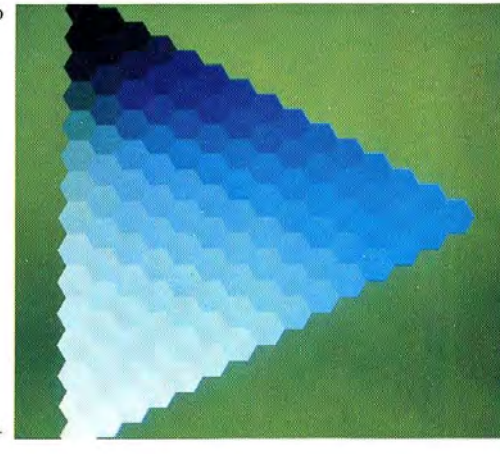
A painting by Louis Condax; courtesy of Eastman Kodak Company and the Optical Society of America.

FIG. 2-8a



Courtesy of D. Greenberg and G. Joblove, Cornell Program of Computer Graphics.

FIG. 2-8b



Courtesy of Tom Check.

FIG. 5-4a

Courtesy of Sam Kapilivsky.

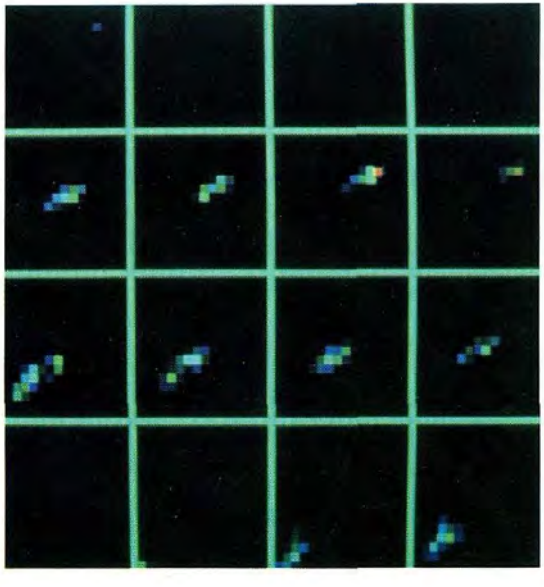


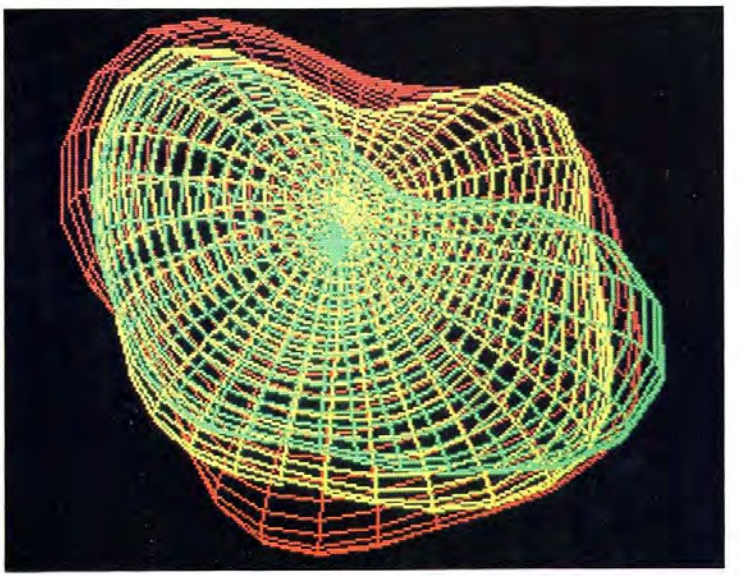
FIG. 5-4b

Courtesy of Sam Kapilivsky.

FIG. 5-4c

Courtesy of Sam Kapilivsky.

FIG. 9-10



Courtesy of Robert Schudy.

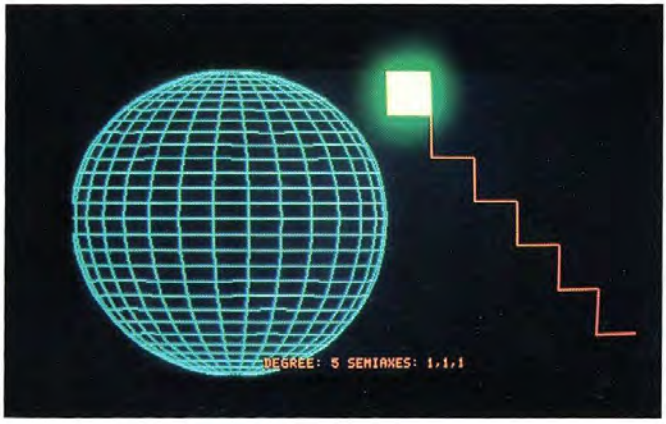
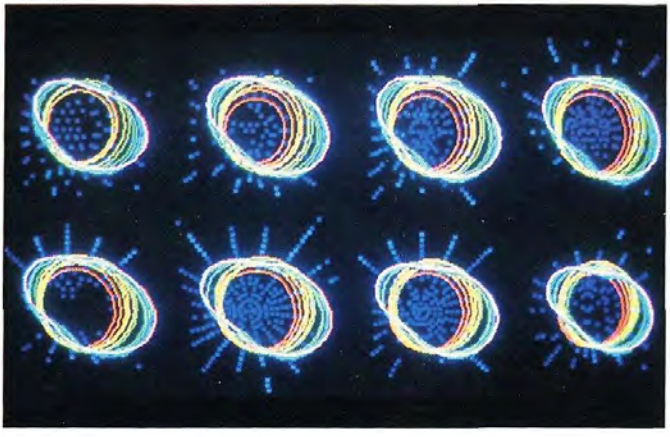


FIG. 11-3a

Courtesy of Robert Schudy.

FIG. 11-3b



Courtesy of Robert Schudy.

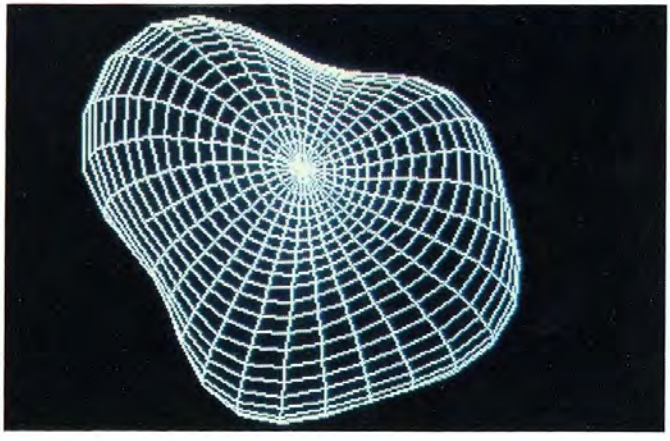


FIG. 11-3c

Courtesy of Robert Schudy.

COMPUTER VISION

DANA H. BALLARD ▪ CHRISTOPHER M. BROWN

What information about scenes can be extracted from an image using only basic assumptions about physics and optics?

How are images segmented into meaningful parts?

At what stage must domain-dependent, prior knowledge about the world be incorporated into the understanding process?

How are world models and conceptual knowledge represented and used?

These and many other questions, inherent in this relatively new and fast-growing field, are explored and answered in **Computer Vision** by **Dana H. Ballard** and **Christopher M. Brown**. The authors assemble crucial material from many disciplines including artificial intelligence, psychology, computer graphics, and image processing to form a practical text and reference for anyone involved in building vision systems.

Ballard and Brown write in their preface, "**Computer Vision** has a strong artificial intelligence flavor, and we hope this will provoke thought. The text shows how both intrinsic image information and internal models of the world are important in successful vision systems."

Divided into four parts, **Computer Vision** offers descriptions of objects at four levels of abstraction:

- Generalized images—images and image-like entities;
- Segmented images—images organized into sub-images that are likely to correspond to "interesting objects";
- Geometrical structures—quantitative models of image and world structures;
- Relational structures—complex symbolic descriptions of image and world structures.

PRENTICE-HALL, INC., Englewood Cliffs, N.J. 07632

ISBN 0-13-165316-4