COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING 36, 139–161 (1986)

Fast Hough Transform: A Hierarchical Approach

HUNGWEN LI, MARK A. LAVIN, AND RONALD J. LE MASTER

Manufacturing Research, IBM Thomas J. Watson Research, P.O. Box 218, Yorktown Heights, New York 10598

Received November 22, 1985

We have developed a fast algorithm for the Hough transform that can be incorporated into the solutions to many problems in computer vision such as line detection, plane detection, segmentation, and motion estimation. The fast Hough transform (FHT) algorithm assumes that image space features "vote" for sets of points lying on hyperplanes in the parameter space. It recursively divides the parameter space into hypercubes from low to high resolution and performs the Hough transform only on the hypercubes with votes exceeding a selected threshold. The decision on whether a hypercube receives a vote from a hyperplane depends on whether the hyperplane intersects the hypercube. This hierarchical approach leads to a significant reduction of both computation and storage. Due to the hyperplane formulation of the problem and the hierarchical representation of the hypercube, the computation in the FHT is incremental and does not require multiplication, which further contributes to efficiency.

1.0. INTRODUCTION

The Hough transform is a powerful technique for multi-dimensional pattern detection and parameter extraction [1, 2]. It has advantages when little is known about the pattern and it is relatively unaffected by gaps in the pattern and by noise. The Hough transform has been successfully applied to line and curve detection [3, 4, 5] and motion estimation [6, 7].

The Hough transform is a mapping from an *image space* into a *parameter space*. One feature point in the image space is mapped into many points (e.g., a line) in the parameter space. An area in the parameter space containing many mapped points (e.g., the intersection of many lines) reveals the potential feature of interest. In the usual Hough transform approach [2], points of concentration are found by dividing the parameter space into an array of "accumulators" and identifying those accumulators receiving the largest number of votes. In that formulation, both the computational complexity of the voting and the storage for the votes grow exponentially with the quantization (q) and the dimensionality (k) of the parameter space (e.g., q^k). Therefore, the Hough transform may not be feasible for problems requiring high resolution (i.e., large q) and/or high dimensionality.

One promising approach to reduce the storage requirement is to represent the parameter space by a hierarchical data structure. Sloan [8] accomplished this by a dynamically quantized pyramid (DQP) in which the boundaries of cells that contain the votes are not fixed. The storage requirement is further reduced in [9] by using the $k - d$ tree structure [10].

The hierarchical concept can also be used to reduce computational complexity; that is the major theme of this paper. Instead of choosing a DQP or $k - d$ tree structure, we adopted the generalized quad- and oct-tree [11], called a *k-tree*, to simplify the computation in the FHT.

0734-189X/86 $3.00

Our contributions in this paper are (1) the hyperplane formulation of the Hough transform problem, and (2) the fast Hough transform (FHT) algorithm which reduces the computational complexity and the storage requirement simultaneously.

The FHT algorithm recursively divides the parameter space into hypercubes from low to high resolution; it performs the subdivision and subsequent "vote counting" only on hypercubes with votes exceeding a selected threshold. This hierarchical approach leads to a significant reduction in both computation and storage. To decide whether a hypercube receives a vote, the FHT algorithm tests whether a hyperplane intersects the hypercube. Due to the hyperplane formulation of the problem and the k-tree representation of the parameter space, the intersection testing can be done incrementally, without multiplication. This further contributes to efficiency.

The hyperplane formulation is given in the next section. In Section 3 we describe the incremental intersection testing. The FHT algorithm is given in Section 4. In Section 5, a line detection problem and a plane detection problem are given as examples of how problems can be formulated for FHT using the hyperplane approach. Properties of FHT are characterized by a series of experiments of these two examples. We then analyze the complexity of the algorithm in Section 6 and discuss the extension of FHT in Section 7.

2.0. HYPERPLANE FORMULATION

In the usual formulation of the Hough transform, each feature point in the image space generates "votes" for a set of parameter-space points. Our algorithm deals with the special case where these sets of parameter-space points are hyperplanes (see Sect. 5 for two examples of this special case.) More formally, we are given:

1. a k-dimensional parameter space denoted by $X_1, X_2, \ldots, X_k$, where the bound and the desired quantization of X_i are also given;

2. a set of feature points $\{F_j\}$ in the image space;

3. a transformation that maps each F_j into a hyperplane in parameter space; the resulting hyperplanes are represented by the equations

$$a_{0j} + \sum_{i=1}^{k} a_{ij} X_i = 0 \tag{1}$$

where each a_{ij} is a function of F_j and is normalized such that $\sum_{i=1}^{k} a_{ij}^2 = 1$;

4. a minimum vote count "**T**."

The goals of our algorithm are (1) to find one or more hypercubes with desired quantization in the parameter space that receive **T** or more votes; and (2) for each hypercube, list the feature points that voted for it. To count the votes for each hypercube, we test for the intersection between each hyperplane and the hypercube. The criterion for this test is discussed next.

3.0. HYPERCUBE / HYPERPLANE INTERSECTION TESTING

3.1. The Testing Criterion

To determine whether a hypercube receives a vote from a particular hyperplane, we test whether the hypercube intersects the hyperplane; Fig. 1 illustrates two criteria for testing intersection for the 2-dimensional case (squares and lines).

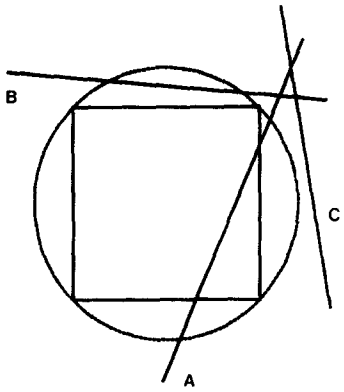


FIG. 1. Two tests for hypercube/hyperplane intersection.

One approach to testing for intersection is to evaluate the left-hand side expression in (1) for each vertex of the hypercube. If all the results have the same sign, the hyperplane and hypercube do not intersect. According to this criterion, line A intersects the square, while lines B and C do not.

A second approach to testing for intersection is to determine whether the hyperplane intersects the hypercube's circumscribing hypersphere, i.e., if

$$a_0 + \sum_{i=1}^{k} a_i C_i \leq r \tag{2}$$

where $[C_1, \ldots, C_k]$ are the coordinates of the hypercube's center and r is the radius of the hypersphere (one-half the diagonal length of the hypercube).

We call this the test for "the intersection in the radial sense." The criterion is not exact; for example, in Fig. 1, it indicates that line B also intersects the square. The imprecision caused by this approximation is balanced by the fact that the testing criterion in (2) can be computed incrementally with great efficiency; Section 3.3 will discuss this in greater detail.

3.2. K-tree Representation

For the FHT algorithm, we represent the parameter space as a nested hierarchy of hypercubes. As shown in Fig. 2, we can associate a *k-tree* with this representation as follows:

The root node of the tree corresponds to a hypercube with side-length S_0 having one vertex at the origin $[0, \ldots, 0]$ and the diagonally opposite vertex at $[S_0, \ldots, S_0]$.[1] Each node of the tree has 2^k sons arising when that node's hypercube is halved along each of its k dimensions. Each son has a *son index*, a vector $b = [b_1, \ldots, b_k]$, where each b_i is -1 or 1. The son index is interpreted as follows: if a node at level l

[1] We choose this position and size for the "root hypercube" without loss of generality, since we can always add additional scale and translation factors to the image space to parameter space transformation.

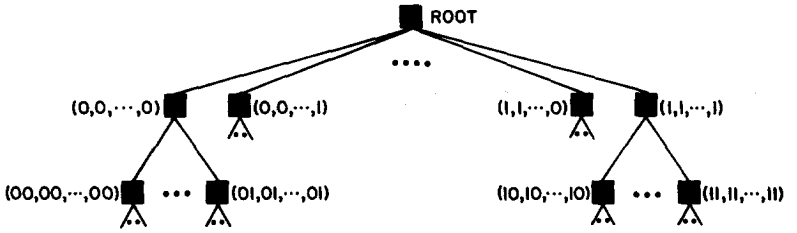


FIG. 2. Generalized oct-tree and k-tree representations.

of the tree has center C_l then the center of its son node with index $[b_1, \ldots, b_k]$ is

$$C_l + \frac{S_{l+1}}{2}[b_1, \ldots, b_k] \tag{3}$$

where S_{l+1} is the side length of the son at level $l + 1$; $S_{l+1} = S_l/2$.

We can also associate a *node index* with each hypercube, which encodes the path from the root to that hypercube's node. The node index for a hypercube at level l is a k-vector of l-bit strings $[s_{11}s_{12} \cdots s_{1l}, \ldots, s_{k1}s_{k2} \cdots s_{kl}]$ where each s_{ij} is 0 or 1. Given this node index for a hypercube at level l, the node index for its $l + 1$ level son with son index $[b_1, \ldots, b_k]$ is defined to be

$$[s_{11}s_{12} \cdots s_{1l}W(b_1), \ldots, s_{k1}s_{k2} \cdots s_{kl}W(b_k)]$$

where $W(-1) \equiv 0$ and $W(1) \equiv 1$. In Fig. 2, nodes are labeled with their node indices.

3.3. *Incremental Testing Formula*

In this section, we present an incremental formula for evaluating the testing criterion presented above in Section 3.1.

3.3.1. *The Formula*

We can define a normalized distance R_l between a hypercube at level l and hyperplane to be the perpendicular distance from the hypercube's center C_l to the plane, divided by the side length S_l of the hypercube. From (2),

$$R_l = a_0 + \sum_{i=1}^{k} c_{li}a_i/S_l. \tag{4}$$

The normalized distance can be computed incrementally for a son node at level $l + 1$ with son index $[b_1, \ldots, b_k]$ as

$$R_0 = \frac{a_0}{S_0} + \frac{1}{2}\sum_{i=1}^{k} a_i \tag{5}$$

$$R_{l+1} = 2R_l + \frac{1}{2}\sum_{i=1}^{k} a_i b_i. \tag{6}$$

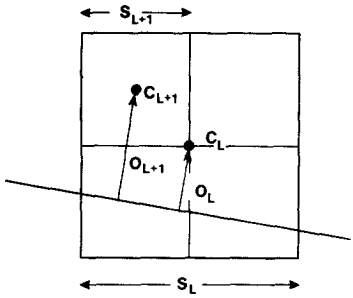


FIG. 3. Incremental construction of hyperplane/hypercube distance.

The test for intersection in the radial sense can be expressed as follows: A hyperplane intersects a hypercube in the radial sense iff

$$|R| \leq \sqrt{k}/2. \tag{7}$$

3.3.2. Proof

In this section, we prove that the incremental formula above correctly implements the test for intersection in the radial sense. We proceed in two steps: First, we validate the incremental computation of normalized hypercube/hyperplane distance; second, we show that the formula (7) is true iff the hypercube and hyperplane intersect in the radial sense.

The basis step for computing R_0 can be verified by substituting the center of the root hypercube $[S_0/2, \ldots, S_0/2]$ into the formula in (1) and dividing by S_0. The induction step can be verified as follows (see Fig. 3).

Let R_l be the normalized distance from the center C_l of a hypercube at level l to a hyperplane with coefficients $[a_0, a_1, \ldots, a_k]$. By definition,

$$R_l = O_l/S_l \tag{8}$$

$$R_{l+1} = O_{l+1}/S_{l+1} \tag{9}$$

where the O's are un-normalized distances from the hyperplane to the hypercube centers and the S's are the side-lengths of the hypercubes. As shown in Fig. 3, the un-normalized distance O_{l+1} can be constructed by adding the projection of the vector $C_{l+1} - C_l$ along the unit normal to the plane, namely $[a_1, \ldots, a_k]$, thus

$$O_{l+1} = O_l + [a_1, \ldots, a_k] \cdot (C_{l+1} - C_l). \tag{10}$$

We know from the definition in (3) that

$$C_{l+1} - C_l = \frac{S_{l+1}}{2}[b_1, \ldots, b_k]. \tag{11}$$

We can then substitute (11) into (10) to derive

$$O_{l+1} = O_l + \sum_{i=1}^{k} a_i \frac{S_{l+1}}{2} b_i. \tag{12}$$

Substituting (12) into (9), and moving $S_{l+1}/2$ outside the summation, we get

$$R_{l+1} = \frac{O_l + \frac{S_{l+1}}{2} \sum_{i=1}^{k} a_i b_i}{S_{l+1}}. \tag{13}$$

We can substitute $S_{l+1} = S_l/2$ from the definition above into (13) and rearrange terms to yield (6).

For the second part of the proof, we observe that the test for intersection in the radial sense is true iff the distance from the center of the hypercube to the hyperplane is less than or equal the distance from the center of the hypercube to any of its vertices ($= \frac{1}{2}$ the hypercube's diagonal length). We can normalize both distances by dividing them by the side length of the hypercube, yielding

$$\begin{aligned} |R| &\leq \tfrac{1}{2} \frac{\text{diagonal length}}{\text{side length}} \\ |R| &\leq \sqrt{k}/2 \end{aligned} \tag{14}$$

This completes the proof.

3.3.3. Discussion

The incremental formula above has several advantages:

1. Computation scales linearly with the number of dimensions; for an "exact" intersection, computation would scale exponentially with the number of dimensions.

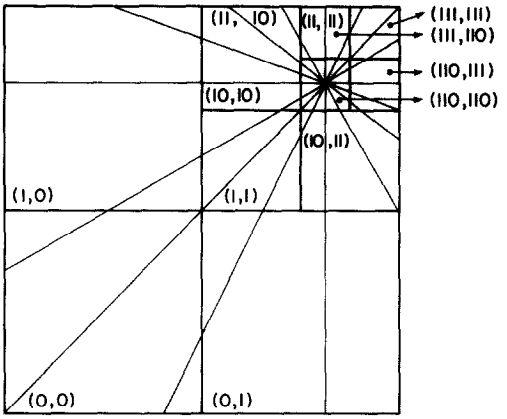
2. The computation can be implemented with shifts, adds, and subtracts, eliminating multiplications and divisions; this is desirable for compact representation in hardware.

3. Further reduction in computation time can be realized by trading space for time—precomputing all possible dot products of son indices with hyperplane parameters.

As noted previously, the main disadvantage is that the test for intersection in the radial sense prunes the search through the k-tree more slowly (because we are somewhat too generous in our "vote counting"). We believe that this problem is resolved by searching one level deeper, and that the savings in computation for each node justify the additional search cost.

4.0. FAST HOUGH TRANSFORM (FHT) ALGORITHM

For clarity, we introduce the FHT algorithm with an example first. The formal statement of the FHT algorithm then follows in Section 4.2.



VOTE	K-TREE ADDRESS			
SIZE 1/2 x 1/2	(0,0)	(0,1)	(1,0)	(1,1)
	3	3	4	8
SIZE 1/4 x 1/4	(10,10)	(10,11)	(11,10)	(11,11)
	3	5	6	8
SIZE 1/8 x 1/8	(110,110)	(110,111)	(111,110)	(111,111)
	8	4	4	3

FIG. 4. Example showing the FHT process.

4.1. Overview of the FHT Algorithm

As illustrated in Fig. 4, eight lines (2-dimensional "hyperplanes") intersect at a point in a 2-dimensional parameter space bounded by a unit square. The initial hypercube (i.e., a square in this 2D case) of size 1×1 receives eight votes and thus qualifies for the further consideration when the threshold (**T**) is chosen to be 7. We then divide both dimensions, which yields four hypercubes (of size $\frac{1}{2} \times \frac{1}{2}$) with node indices [0, 0], [0, 1], [1, 0] and [1, 1]. By choosing the threshold **T** = 7, only hypercube [1, 1] remains for further dividing.

By further dividing hypercube [1, 1], we obtain four smaller hypercubes (of size $\frac{1}{4} \times \frac{1}{4}$) with node indices [10, 10], [11, 10], [10, 11], and [11, 11]. After the testing and thresholidng (**T** = 7), only the hypercube [11, 11] survives and needs to be divided if higher resolution is desired.

One more division of the hypercube [11, 11] produces only one hypercube—[110, 110]—containing at least **T** votes, as shown in Fig 4. The resolution of the hypercube is $\frac{1}{8} \times \frac{1}{8}$. Such a dividing process can be repeated until the desired resolution (or quantization) is reached.

The storage required by this example is 4 vote registers at each level of dividing. A straightforward implementation of the Hough method (in which the parameter space is uniformly divided and each hypercube is tested against each hyperplane) would have required 64 vote registers for the chosen dimension ($k = 2$) and quantization ($q = 8$). The FHT algorithm performs the testing for 13 hypercubes (1 + 4 + 4 + 4) of different resolutions while a straightforward approach performs the testing for 64 hypercubes.

4.2. Formal Statement of the FHT Algorithm

Given:

1. a parameter space $X_1, \ldots, X_k$ where X_i is bounded by $[0, S_0]$;

2. a k-tree mapping of the parameter space with the root at $[S_0/2, S_0/2, \ldots, S_0/2]$;

3. a set of feature points $\{F_j\}$ in the image space;

4. a transformation that maps each F_j into a hyperplane in parameter space represented by the equation $a_{0j} + \sum_{i=1}^{k} a_{ij} X_i = 0$ where the a_{ij} is a function of F_j and is normalized so that $\sum_{i=1}^{k} a_{ij}^2 = 1$;

5. a minimum vote count "T" as threshold and a desired resolution "q," the Fast Hough transform (FHT) algorithm is

```
/* STEP 1: Compute normalized hypercube "radius" */

norm_radius = √k /2;

/* STEP 2: Compute the initial normalized distances from all
   hyperplanes to the root node and count its votes */

root = create node( );
root.level = 0;

FORALL feature points F_j, j = 1 to J DO BEGIN
    root.norm_dist[j] = a_0j/S_0 + ½Σ_{i=1}^{k} a_ij;
    IF abs(root.norm_dist[j]) ≤ norm_radius THEN BEGIN
       root.vote = root.vote + 1;
       root.in_cube[j] = true;
       END;
    ELSE root.in_cube[j] = false;
    END;

IF root.vote ≥ T THEN enqueue root on parent_list;

/* STEP 3: Search through k-tree by processing front node on
   parent_list and generating its sons */

WHILE non_empty (parent_list) DO BEGIN
    this node = extract front from parent_list;
    IF this_node.level ≥ log_2 q THEN place this_node on solution_list;
    ELSE FORALL sons [b_1, ..., b_k] of this_ node DO BEGIN
        new_node              = create_node( );
        new_node.level        = this_node.level + 1;
        new_node.parent       = this_node;
        new_node.son_index    = [b_1, ..., b_k];
        FORALL feature points F_j, j = 1 to J DO
            IF this_node.in_cube[j] THEN BEGIN
                new_node.norm_dist[j] =
                    2 * this_node.norm_dist[j] + ½Σ_{i=1}^{k} a_ij b_ij;
```

```
            IF abs(new_node.norm_dist[j]) ≤ norn_radius THEN BEGIN
                new_node.vote = new_node.vote + 1;
                new_node.in_cube[j] = true;
                END;
            ELSE new_node.in_cube[j] = false;
            END;
        ELSE new_node.in_cube[j] = false;

    IF new_node.vote ≥ T THEN
            add_node_to_list (new_node, parent_list);
    END; /* FORALL sons... */

END; /* WHILE */
/* STEP 4: Compute, average, etc., solutions */
. . .
```

Step 1 of the algorithm computes a constant *norm_radius* that will be used to test the intersection of all hyperplanes and hypercubes; *norm_radius* is the normalized half-diagonal of a k-dimensional hypercube.

In Step 2, the top-level hypercube *root* is tested for intersection with all parameter space hyperplanes. After allocating an instance of the hypercube node data structure with *create_node*( ), intersections are tested and the number of intersections is recorded in *root.vote*[j], while the boolean vector *root.in_cube*[j] records which hyperplanes voted for *root*. If *root* receives at least T votes, it is put on *parent_list* (a queue of hypercubes awaiting further subdivision). If *root* has fewer than T votes, the algorithm terminates without a solution.

Step 3 is where most of the computation for the algorithm occurs: a hypercube record *this_node* is taken from the front of *parent_list*, subdivided into sons, and they in turn are "voted for" by the parameter space hyperplanes; the number of votes and "yes-voters" are recorded in the *new_node.vote*[j] and *new_node.in_cube*[j]. By testing a node only against hyperplanes that voted for its parent (cf., *this_node.in_cube*[j]) the amount of computation per node can be reduced. In addition, the *in_cube* field allows for "backmapping" in Step 4.

In Step 3 the incremental testing formula developed in Section 3 is used to determine whether hypercubes and hyperplanes intersect. Each node *this_node* taken from the *parent_list* contains a list *this_node.norm_dist*[j] of the normalized distances to all hyperplanes that voted for it. These are used to incrementally compute *new_node.norm_dist*[j] for each son.

The control policy for the FHT algorithm's search through the k-tree is set by the subroutine *add_node_to_list*. Several policies are possible:

Depth-first. New nodes are added to the front of the *parent_list*.

Breadth-first. New nodes are added to the end of the *parent_list*.

Best-first. New nodes are sorted into the list in order of decreasing numbers of votes.

Other factors could also be considered to create an efficient control policy for the FHT, including sorting by hypercube size, bounding the length of the *parent_list*, or adaptively changing the minimum vote threshold. In the experiments performed

for the two examples in the next section, we adopted the breadth-first policy (with vote count thresholding).

The FHT algorithm's search through the k-tree in Step 3 can terminate in several ways:

1. The search reaches the desired quantization level $\log(q)$: any hypercube node that survives to this level is placed on the *solution_list* and is not subdivided.

2. A branch terminates before reaching level $\log(q)$ when a node having more that **T** votes is divided into 2^k sons, none of which has at least **T** votes. The extreme case occurs when the root receives fewer than **T** votes.

Case 2 raises several questions: Are we discarding good solutions that straddle the boundaries of several $\log(q)$-level hypercubes? Is the choice of threshold **T** appropriate? To some extent, these problems could be addressed by using a more complex best-first search policy; this is an area for further investigation.

Step 4 of the algorithm is only sketched in our current work. It takes all nodes on *solution_list* and derives a set of solution parameter vectors and "backmappings" (lists of image-space feature points that voted for each solution). In step 4, solutions that straddle several hypercubes must be identified and the contributions of each hypercube must be combined. This is also an area for further work.

5.0. EXAMPLES OF HYPERPLANE FORMULATION

5.1. Line Detection Example

Given a set of feature points (x_j, y_j), $j = 1$ to m, the line detection problem is to find one or more lines that best fit the feature points. The line detection problem can be formulated in a hyperplane for the FHT by observing that a point (x, y) in image space can lie on any line $y = mx + b$. Thus, we can map the point (x, y) into a line in the (m, b) parameter space

$$a_0 + a_1 m + a_2 b = 0 \tag{15}$$

where $a_0 = -y/\sqrt{x^2 + 1}$, $a_1 = x/\sqrt{x^2 + 1}$, $a_2 = 1/\sqrt{x^2 + 1}$.

The singularity problem (when m approaches ∞) can be resolved by adding a dual parameter space based on the equation $x = m'y + b'$ and only considering parts of the parameter spaces where $m, m' \in [-1, 1]$. In such a dual space formulation, we have two independent 2-parameter spaces.

We performed a series of experiments to characterize the behavior of the FHT algorithm under the effect of (1) statistical noise, (2) quantization error, and (3) minimum vote threshold. These experiments were performed by using a synthetic images consisting of feature points lying along a single line; in some cases, we added extraneous feature points, and in some cases we varied the "quantification noise" for the feature point coordinates. In all the examples in this section, we used the breadth-first control policy with minimum vote threshold described above. In the discussion below, the word "iteration," refers to the processing of all nodes on a given depth level. For the experiments, the FHT searching process is illustrated by the outlining each hypercube that was "voted into." In the figures, the hyperplanes are overlayed to show the location of the solution.

Effect of threshold. In this experiment, we investigated the behavior of the FHT under different minimum vote thresholds. Feature point coordinates were quantized

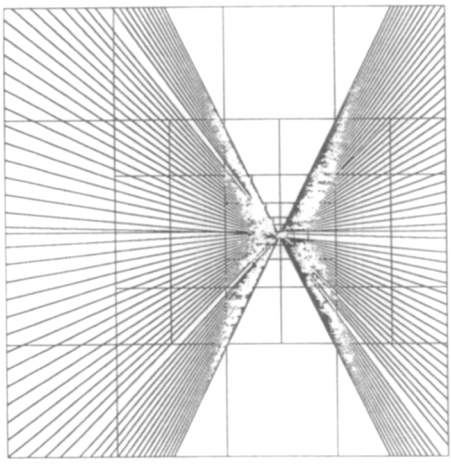


FIG. 5. Effect of threshold (75%).

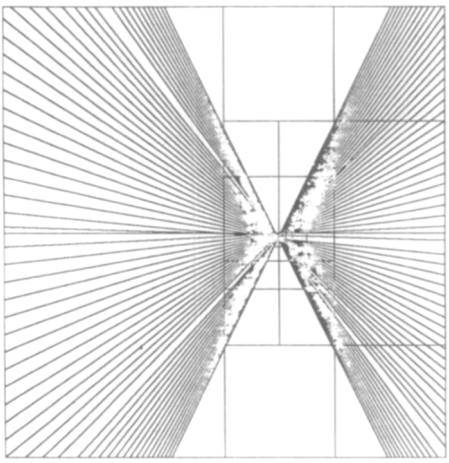


FIG. 6. Effect of threshold (50%).

to eight bits; 64 feature points were generated from a known line equation and no random points were added. Mapped to the parameter space, these feature points form 64 hyperplanes intersecting at the solution point in the parameter space. The results for thresholds of 75% (of 64 points), 50% and 25% are shown in Figs. 5, 6, and 7, respectively.

As expected, the higher the threshold, the fewer the hypercubes explored at each iteration of the algorithm. Ten iterations of the FHT were executed; the number of hypercube nodes in the *parent_list* after each iteration were (2, 2, 2, 2, 2, 2, 2, 2, 2, 2) for the **T** = 75% threshold, (4, 6, 4, 4, 4, 3, 6, 5, 3, 3) for **T** = 50%, and (4, 11, 22, 18, 15, 18, 21, 34, 39, 44) for **T** = 25%.

It is important to observe that the number of hypercubes per iteration tends to grow slowly (and often shrinks when a solution is approached). For example, in the

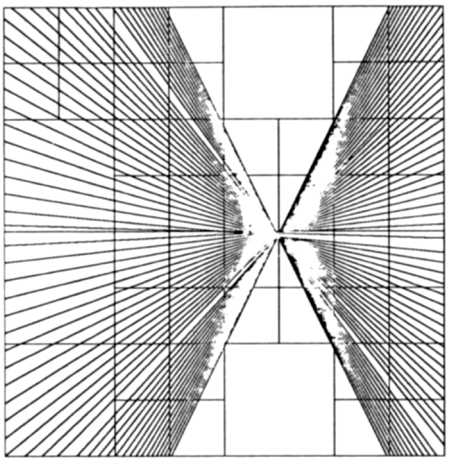


FIG. 7. Effect of threshold (25%).

worst case, there would be 4^{10} or 1,000,000 active hypercubes after 10 iterations; in this experiment, 2, 3, or 44 nodes were active at level 10. It appears that only a very "thin" portion of the k-tree needs to be searched to reach the solution. Consequently, the computational complexity and storage requirement do not grow exponentially with the quantization and the dimensionality of the parameter space. We call this the "thin tree" property; we discuss it further in Section 6.4.

The data on number of parent hypercubes above are for the parameter space where the solution resides. In each case, its dual space was rejected by the FHT due to insufficient vote count at the first iteration. The rejection of the dual space at the first iteration is true for the rest of the experiments in the line detection example.

Statistical noise. In this experiment, the 64 feature points from the previous experiment were corrupted by adding 48 noise points (Fig. 8), 32 noise points (Fig. 9), and 16 noise points (Fig. 10); noise points were generated from a uniform distribution over the image. A minimum vote threshold of 50% of the total points (object and noise) was for the three cases.

The overlay of the hyperplanes in the figures indicates that the hyperplanes mapped by the noise points distribute randomly in the parameter space. With random m and b, they do not intersect at one or more points with high concentration, therefore the statistical noise does not change the noise-free "vote" histogram when sufficient resolution is reached and does not affect the finding of the solution by the FHT. This is demonstrated by comparing Fig. 9 with Fig. 6. With statistical noise, Fig. 9 has 32 more hyperplanes in the parameter space. The hyperplanes contributed by the noise are "clustered" in upper-left, upper-right, and lower-right corners of the initial hypercube. They do increase the probability of these three hypercubes of size $\frac{1}{2}$ being "falsely" qualified. However, the effect of the noise hyperplanes fades away quickly as the FHT performs testing on hypercubes with higher resolution. This is because very few hyperplanes contributed by the noise can intersect the same high resolution hypercube. This demonstrates that the FHT is resistant to statistical noise and also confirms previous studies on Hough transform that it is highly resistant to the statistical noise [2].

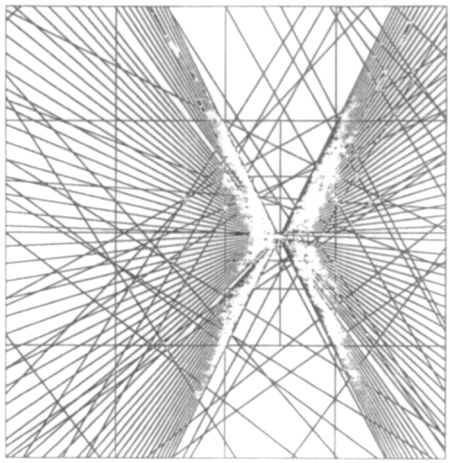


FIG. 8. Effect of statistical noise (75%, 48 points).

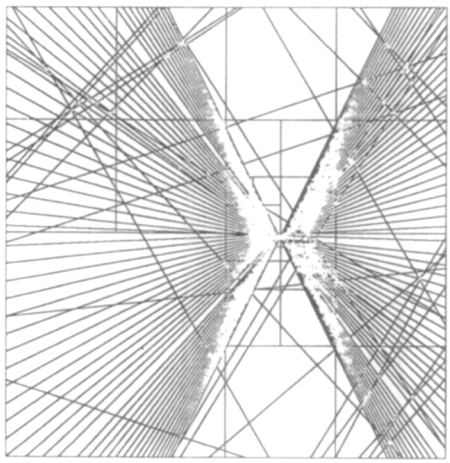


FIG. 9. Effect of statistical noise (50%, 32 points).

The FHT algorithm rejects the statistical noise similarly for all three noise levels. This can be seen from the empirical results of the number of parent hypercubes per iteration: these are (4, 2, 3, 2, 2, 2, 1, 0, 0, 0), (3, 3, 2, 2, 2, 2, 1, 0, 0, 0), and (4, 4, 3, 3, 2, 2, 3, 2, 2, 2) for the 48, 32, and 16 noise-point cases, respectively. For the first two, no hypercube has vote count exceeding the threshold in the last three iterations because that the threshold is based on the total number of points (object and noise), which is higher than that for the noise-free cases above.

Quantization error. In this experiment, we investigated the effect of different quantization of the input feature point coordinates: Fig. 11, Fig. 12, and Fig. 13 show the effects for 7-bit, 8-bit, and 9-bit quantization of image space feature coordinates. No noise was added and a 50% threshold was used.

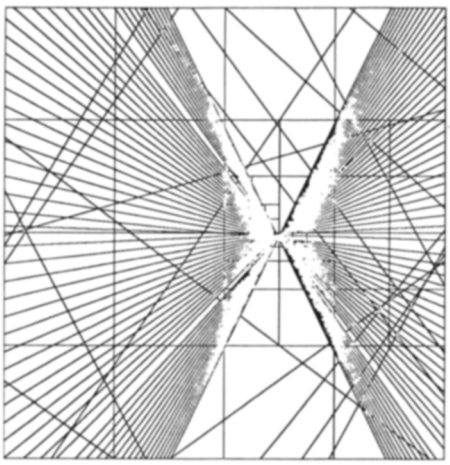


FIG. 10. Effect of statistical noise (25%, 16 points).

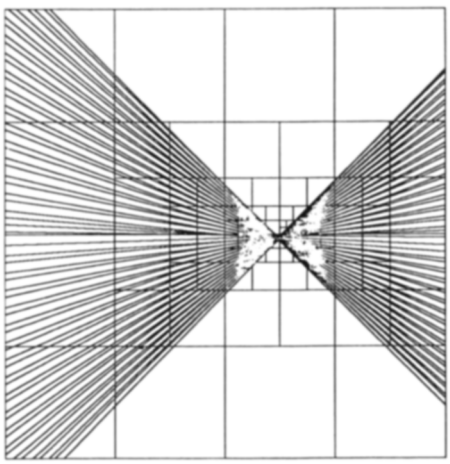


FIG. 11. Effect of quantization error (7-bit).

The effect of quantization on the FHT is caused indirectly by the fineness of the hyperplane intersection. Ideally, when infinite quantization is used, all hyperplanes generated from the same line model intersect at one point. When the number of quantization bits decreases, the point of intersection "diffuses" into an area. Instead of having one hypercube with vote count higher than its neighbors, the "diffusion" leads to the phenomena that a cluster of hypercubes in the vicinity of the solution will have similar but reduced vote count. This not only results in multiple solutions after the FHT searching but also implies that an enhanced control strategy (such as adaptive thresholding or best-first searching) may be needed.

The searching of Fig. 11 best illustrates the "diffusion" phenomena. The number of parent hypercubes for Fig. 11 is (4, 6, 8, 8, 6, 3, 6, 6, 0, 0), in which, until iteration 8, the vote count is maintained above the 50% threshold. When entering into the

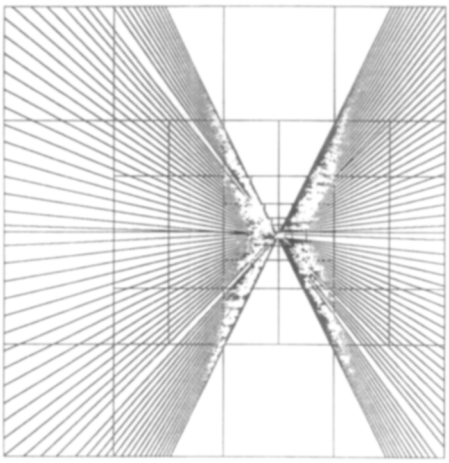


FIG. 12. Effect of quantization error (8-bit).

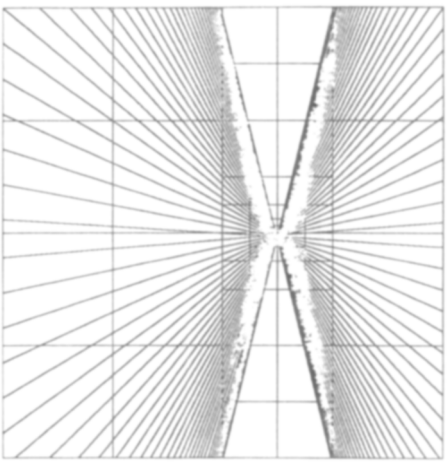


FIG. 13. Effect of quantization error (9-bit).

next resolution at iteration 9, the vote count in the vicinity drops due to the "diffusion" caused by the insufficient 7-bit representation.

The "diffusion" is reduced by increased image space quantization. For example, as shown in Fig. 12, the last two non-zero parent hypercubes in the searching process (4, 6, 4, 4, 4, 3, 6, 5, 3, 3) indicates that the diffusion area shrinks due to the extra bit in representation. With a 9-bit representation for Fig. 13, the searching process (4, 4, 4, 4, 4, 4, 4, 5, 5, 5) also does not suffer from the "diffusion."

Backmapping. As stated in Section 2, one of the goals of the FHT is to find the parameter space hyperplanes that voted for the qualified hypercubes. Figure 14 shows the image used for the example in Fig. 8, in which there are 64 feature points and 48 noise points, and the feature points were quantized in 8 bits. Figure 15 shows the feature points that voted for the hypercubes with 50% threshold after four

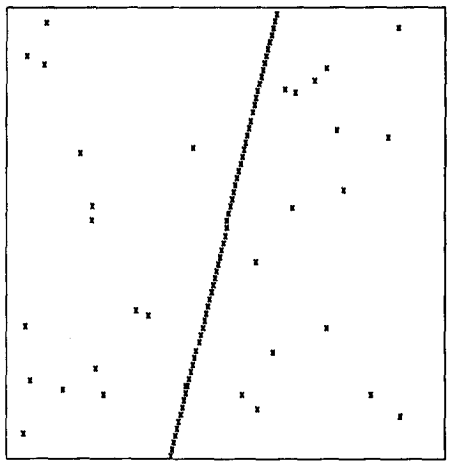


FIG. 14. Synthetic image used in Fig. 8.

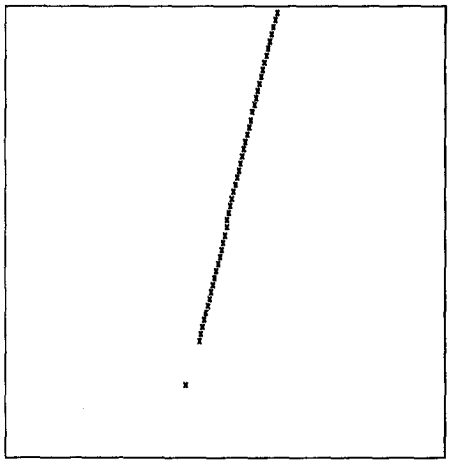


FIG. 15. Backmapping after 4 iterations.

iterations. All noise points were "filtered"; some feature points were "filtered" because of the quantization error. Figure 16 illustrates the result after processing the FHT for seven iterations under the same threshold.

We term this procedure of finding the hyperplanes that voted for the qualified hypercubes a "backmapping" process. Backmapping is important because it has a built-in "clustering" capability which is significantly important to any applications that require "classification." In computer vision applications, labelling, clustering, segmentation, and filtering can all benefit from the backmapping property. As shown in Figs. 15 and 16, backmapping can be used to cluster "pattern" points while rejecting noise points.

5.2. Plane Detection Example

Given M range data points, (x_j, y_j, z_j), $j = 1$ to M, the plane detection problem is to find one or more planes that best fit these feature points.

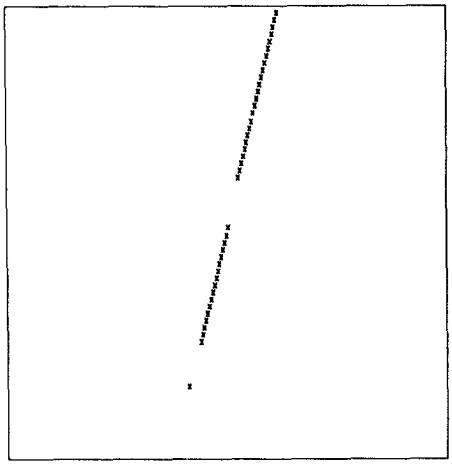


FIG. 16. Backmapping after 7 iterations.

We can apply the hyperplane formulation to this problem by observing that an image-space plane can be represented

$$\begin{aligned} x &= m_y y + m_z z + b_x \\ y &= m_x x + m_z z + b_y \\ z &= m_x x + m_y y + b_z \end{aligned} \tag{16}$$

where m_x, m_y and m_z are the directional normals of the plane and b_x, b_y and b_z are the intercepts of the x, y and z axes, respectively. By choosing (m_y, m_z, b_x), (m_x, m_z, b_y), and (m_x, m_y, b_z) as three sets of parameters and restricting m_x, m_y, and m_z to the interval $[-1, 1]$, we get three "dual" 3-parameter spaces. A feature point $[x_j, y_j, z_j]$ is transformed into three plane equations in the parameter spaces

$$\begin{aligned} a_{0j} + a_{1j} m_y + a_{2j} m_z + a_{3j} b_x &= 0 \\ a'_{0j} + a'_{1j} m_x + a'_{2j} m_z + a'_{3j} b_y &= 0 \\ a''_{0j} + a''_{1j} m_x + a''_{2j} m_y + a''_{3j} b_z &= 0 \end{aligned}$$

where

$$\begin{aligned} a_{0j} &= -x_j/c1_j, & a_{1j} &= y_j/c1_j, & a_{2j} &= z_j/c1_j, & a_{3j} &= 1/c1_j, \\ a'_{0j} &= -y_j/c2_j, & a'_{1j} &= x_j/c2_j, & a'_{2j} &= z_j/c2_j, & a'_{2j} &= 1/c2, \\ a''_{0j} &= -z_j/c3_j, & a''_{1j} &= x_j/c3_j, & a''_{2j} &= y_j/c3_j, & a''_{3j} &= 1/c3_j, \end{aligned}$$

and

$$c1_j = \sqrt{y_j^2 + z_j^2 + 1}, \qquad c2_j = \sqrt{x_j^2 + z_j^2 + 1}, \qquad c3_j = \sqrt{x_j^2 + y_j^2 + 1}.$$

A series of experiments similar to the line detection example were performed to characterize the behavior of the FHT in this 3D case under the influence of (1) quantization error, (2) the statistical noise, and (3) the threshold. The 3D image and noise were created in the same manner as described in the line detection example.

Effect of threshold. For this experiment, 64 range data points were generated according to the model in Eq. (16). They then were quantized in 8-bit representation. No noise points were added in this case. Three threshold levels (75, 50, and 25%) were used.

The number of hypercubes explored at each iteration was (8, 11, 8, 8, 8, 4, 8, 8, 8, 8) for 75% case, (8, 29, 24, 21, 21, 28, 26, 26, 26, 26) for 50% case, and (8, 64, 248, 271, 271, 154, 150, 203, 85, 85) for the 25% case, in the parameter space which contained the solution point. The hypercubes in the other two parameter spaces were rejected at the first iteration for the 75 and 50% cases, and at the second iteration for the 25% case due to insufficient vote count.

From these results, we see again that the "thin tree" property holds true for the 3D case: while we would expect to see 8^{10} or one billion active nodes in the worst case, we actually see 8, 26, and 85. This represents a significant reduction in computation time and storage. Compared with the line detection problem, the number of solutions found in the 3D case is larger. This is due to the fact that the solution lies in the boundary of neighbor hypercubes; and there are 8 neighbors for each hypercube in the 3D case while there are only four neighbors in the 2D case.

Obviously, having a large number of active solutions (e.g., 271 in 25% threshold case) costs a lot of storage. This implies that a very simple control strategy like the breadth-first policy adopted for the experiments may not be sufficient for some applications.

Statistical noise. In the next set of experiments, the effect of statistical noise was investigated. Three noise levels (75, 50 and 25%) were used in a 3D image having 64 feature points represented by an 8-bit quantization. We selected 50% threshold for all three cases.

The number of hypercubes remaining after each iteration are (8, 15, 8, 8, 4, 4, 8, 8, 8, 8) for 75% noise, (8, 14, 8, 8, 8, 4, 8, 8, 8, 8) for 50% noise, and (8, 23, 10, 10, 12, 12, 8, 8, 8) for 25% noise. The thin tree property holds for the noisy case, and the FHT algorithm is resistant to the statistical noise. In all three cases, the dual parameter spaces that didn't contain the solution point were rejected at the first iteration.

Quantization error. We observed the behavior of the FHT under 7-, 8-, and 9-bit quantizations. We used the same image as in the previous experiment, but without noise points. The minimum vote threshold was 50%. After each iteration, the numbers of hypercubes are (8, 26, 32, 32, 31, 40, 32, 32, 32, 32) for the 7-bit case, (8, 29, 24, 21, 21, 28, 26, 26, 26, 26) for the 8-bit case, and (8, 27, 24, 24, 22, 20, 24, 24, 24) for the 9-bit case. For the latter two cases, the non-solution dual parameter spaces were rejected at the first iteration, while for the 7-bit case, it remained active for three iterations. The effect of quantization error on the FHT is related to "diffusion" discussed before.

Backmapping. The backmapping property was investigated by an experiment using 75% noise, 50% threshold, and an 8-bit quantization. At the fourth iteration, 28 out of 48 noise points were filtered. At the seventh iteration, only two noise points were left; and at the ninth iteration, all noise points had been filtered.

The backmapping property has important applications in 3D vision. Based on range data, it can be used to segment the feature points into different planes each having different plane parameters. The resulting sets could be used in several ways:

- The sets constitute a kind of discrete approximation to the extended Gaussian image [14], which could be used as an index into an object data base for recognition.
- The sets could be further segmented and fitted with polygonal boundaries to construct a polyhedral model of the object(s) being imaged.

Conversely, constraints from pre-existing polyhedral models could be used to guide the FHT algorithm's search through the k-tree.

6.0. COMPLEXITY ANALYSIS

In this section we informally characterize the space and time complexity of the FHT algorithm by comparing the results in Section 5 against those expected from a hypothetical "flat method," in which the parameter space is uniformly divided and each hypercube is tested against each hyperplane. In Section 6.4 we present a theorem that explains our empirical observations about the "thin tree" phenomenon.

6.1. Hypercube Node Count

For the purposes of comparison, we will consider the testing of each hypercube against all hyperplanes as a single unit of computation for both the FHT and flat methods; we'll qualify this in later sections.

The flat method performs q^k hypercube tests when the k-dimensional parameter space is divided into $q \times q \times \cdots \times q$ hypercubes. This is illustrated by the solid lines in Figs. 17 and 18, which plot log(nodes) as a function of $\log(q)$ for $k = 2$ and $k = 3$, respectively. The number of tests performed by the FHT method, derived from the experiments described in Section 5, falls into the range between the dotted lines in the figures.

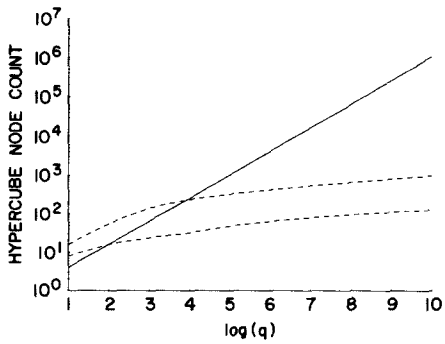


FIG. 17. Nodes tested vs q for 2-dimensional case.

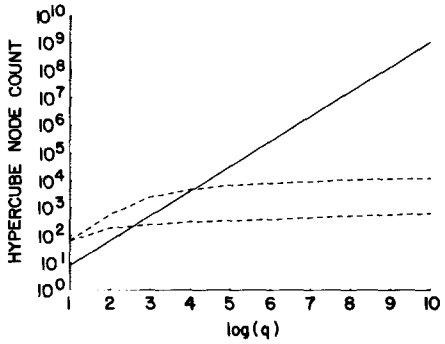


FIG. 18. Nodes tested vs q for 3-dimensional case.

The figures show a substantial reduction in the growth rate of the FHT with increasing q (factors of 1000 and 1,000,000 for the 2D and 3D cases when $q = 1000$); this is due to the "thin-tree" phenomenon described below.

6.2. Time Complexity

To a first approximation, the execution time of the FHT and flat algorithms is proportional to the number of hypercube tests performed.[2] For the flat method, each node test against M hyperplanes requires $M \times k$ multiplications and additions (assuming that we use the test for intersection in the radial sense). There is also a small amount of overhead per test (for indexing) and for each "yes vote."

For the FHT algorithm, each node test requires approximately M shifts and $M \times k$ sign tests and additions/subtractions (using the incremental testing method described in Sect. 3) plus some overhead for each node tested (to allocate and link the node data structure) and for each hyperplane tested (to retrieve and store the normalized distance). It is possible to reduce the node testing time by considering only the hyperplanes that voted for a node's parent (examining *this_node.in_cube*[j]); this may contribute substantially when multiple solutions and/or noise points are present.

It appears that the time cost per test is greater for the FHT than for the flat method, but probably by not more than a factor of two (assuming a large number M of feature points that "wash out" the overhead of node allocation). However, the additional per-node computation time is overcome by the substantial reduction in the number of nodes actually tested.

We observe that the time savings due to the incremental testing (converting multiplications to shifts) is modest for the hyperplane formulation of the FHT approach, when implemented on a general-purpose computer. We expect greater benefit when we use the FHT algorithm in problems where the image space features are transformed into higher-order surfaces in the parameter space, and for hardware implementations where shifting requires far less circuit complexity and/or execution time compared to multiplication.

[2] There is a second-order effect when the best-first control policy is used for the FHT algorithm, due to the need to keep the parent_list sorted. This can be made negligible by managing the list as a sorted tree.

6.3. Space Complexity

The storage cost for each hypercube tested in the flat method is one number (in the range $[0 \cdots M-1]$) for the vote accumulator plus an M-entry boolean vector for backmapping.[3]

The storage cost for each active hypercube node in the FHT algorithm includes some per-node overhead (pointers, vote count, etc.) plus, for each of the M features, one number (normalized distance) and one boolean value for backmapping.

Thus, the per-node storage cost for the FHT algorithm is considerably greater than for the flat method, but it is overbalanced by the substantial reduction in the number of active nodes in the FHT versus total nodes for the flat method.

6.4. The "Thin Tree" Observation and Theorem

We observed in previous sections that the number of hypercubes active during the FHT algorithm search of the k-tree does not grow exponentially with the quantization and dimensionality of the parameter space. We have called this the "thin tree" observation, and it is the basis for the FHT algorithm's reduction of time and storage cost compared with the other Hough transform approaches. In this section we state and prove the theorem that explaines the thin tree observation:

THEOREM. *Assume all M hyperplanes in the parameter space intersect at a single point C and that they are uniformly distributed in orientation. Given a minimum vote threshold* **T**, *the number of hypercubes of size q that can receive* **T** *or more votes is less than some number K that does not depend on q.*

Proof. (We prove the theorem in detail for the 2-dimensional case and leave the proof for higher dimensions to the reader.) Since we assume that the lines (2D hyperplanes) are uniformly distributed in orientation, the vote count of a square (2D hypercube) is proportional to the angle it subtends at C (see Fig. 19). For a square whose closest points is a distance r from C, the angle $\alpha(r)$ that it subtends can be bounded,

$$\alpha(r) \leq 2 \cdot \tan^{-1}\left(q\frac{\sqrt{2}}{2}\Big/ r\right). \tag{17}$$

A square will receive **T** votes (out of M) only if it subtends an angle

$$\alpha_{\min} = \pi T/M. \tag{18}$$

By equating the right-hand sides of (17) and (18) we can compute an $r_{\max}$ such that any square whose closest point is farther than $r_{\max}$ from C will receive fewer than **T** votes,

$$2 \cdot \tan^{-1}\left(q\frac{\sqrt{2}}{2}\Big/ r_{\max}\right) = \frac{\pi T}{M} \tag{19}$$

$$r_{\max} = q\frac{\sqrt{2}}{2}\Big/ \tan\left(\frac{\pi T}{2M}\right) \tag{20}$$

$$r_{\max} \equiv qK_0. \tag{21}$$

[3] We recognize that there are a number of approaches to reducing the storage cost for the flat method, including asscoiative caches for accumulator values that exceed the minimum vote threshold [12, 13]. We observe that such approaches imply additional computation time and/or the use of k-tree-like indexing structures.

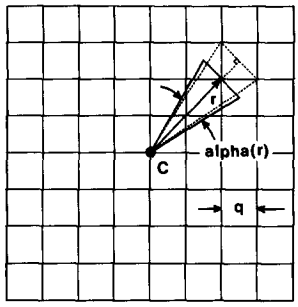


FIG. 19. Illustration for the thin-tree theorem.

Let $K_1 \equiv ceil(K_0) + 1$; all $q \times q$ squares outside the $(2K_1q) \times (2K_1q)$ square centered at C will have fewer than T votes. Thus, for any resolution q, there are at most $4K_1^2 \equiv K$ squares with side-length q having T votes, *independent of q*.

This bound constrains the breadth of the "thin tree" that we have observed.

7.0. CONCLUSION AND FUTURE EXTENSIONS

We have developed a fast Hough transform (FHT) algorithm, which provides significant reductions in computation and storage cost when applied to a class of problems in which feature space points are transformed into hyperplanes in the parameter space. We have presented two examples of this problem class: straight-line finding in 2D images and planar surface finding in range images.

The FHT algorithm represents the parameter space as a k-tree, which it can search in a variety of ways for hypercubes where large numbers of hyperplanes intersect. The performance improvements for the FHT algorithm appear due to the *thin tree property* that limits the (potentially) exponential growth in time and storage. We have demonstrated this property with a number of examples and proven a theorem that explains the property. The main implication of this is that the FHT can be applied to problems requiring high-dimension and/or finely-quantized parameter spaces.

We have developed a simplified test for hypercube/hyperplane intersection, which is repeatedly performed during the execution of the FHT algorithm. This method, while approximate, has the advantage that it can be computed incrementally without multiplication. This is important for future hardware implementation and for extensions of the method to handle higher order surfaces in the parameter space.

The FHT algorithm is highly parallel. As shown, the processing for the hypercubes are independent of each other. Furthermore, the intersection testing for a hyperplane does not depend on that of other hyperplanes. When parallel implementation is considered, synchronization becomes a nontrivial issue. The synchronization involves maintaining the partial order prescribed by the chosen control policy. For the policies that need sorting on the vote count (e.g., best-first search), the partial ordering is more difficult to maintain. Active research effort has been devoted to develop and implement algorithms that guarantee the partial ordering [12, 13].

Multiple instances were not treated in the paper and will be a future research topic. The problem involves the searching for multiple points of high vote con-

centration in the parameter space. The technical difficulty anticipated is the choice of a good control strategy. We expect that a combination of breadth-first and best-first policies with an adaptive threshold will deliver good performance. When a model is available, guidance from the model to control the execution of the FHT will help.

Not all problems can be naturally formulated by the hyperplane approach; consequently there is a need to extend the FHT to higher-order surfaces and/or to consider approaches that allow "nonplanar" problems to be recast as (possibly higher dimensional) "planar" problems.

REFERENCES

1. P. V. C. Hough, *Method and Means for Recognizing Complex Patterns*, U.S. Patent 3069654, 1962.
2. D. H. Ballard and C. M. Brown, *Computer Vision*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
3. R. O. Duda and P. E. Hart, Use of the Hough transform to detect lines and curves in pictures, *Comm. ACM* **15**, No. 1, 1972, 11–15.
4. D. H. Ballard, Generalizing the Hough transform to detect arbitrary shapes, *Pattern Recognit.* **13**, No. 2, 1981, 11–122.
5. S. D. Shapiro, Feature space transformation for curve detection, *Pattern Recognit.* **10**, 1978, 129–143.
6. D. H. Ballard and O. A. Kimball, Rigid body motion from depth and optical flow, *Comput. Vision Graphics Image Process.* **22**, 1983, 95–115.
7. J. O'Rourke, Motion detection using Hough techniques, in *Proc. Conf. on Pattern Recognition and Image Processing*, Dallas, Texas, 1981.
8. K. R. Sloan, Jr., Dynamically quantized pyramid, in *Proc. 7th Int. Joint Conf. Artif. Intell.*, pp. 734–736.
9. J. O'Rourke, Dynamically quantized spaces for focusing the Hough transform, in *Proc. 7th Int. Joint Conf. Artif. Intell.* pp. 737–739.
10. J. L. Bentley, Multidimensional binary search trees in data base applications, *IEEE Trans. Software Engineering* **SE-5**, No. 4, 1979.
11. S. N. Srihari, Hierarchical representations for serial section images, in *Proc. 5th Int. Conf. Pattern Recognit.*, Miami Beach, Fla., Dec. 1980, pp. 1075–1080.
12. C. M. Brown and D. B. Sher, Hough transformation into cache accumulator: Consideration and simulations, TR114, Computer Science Dept., University of Rochester, Rochester, N.Y., 1982.
13. D. Sher and A. Tevanian, The vote tallying chip: A custom integrated circuit, TR144, Computer Science Dept., University of Rochester, Rochester, N.Y., 1984.
14. K. Ikeuchi, Recognition of 3-D objects using the extended Gaussian image, in *Proc. 7th Int. Joint Conf. Artif. Intell., IJCAI-81*, Vancover, B.C., Canada, August 1981, pp. 595–600.