Real-Time Feature Extraction: A Fast Line Finder For Vision-Guided Robot Navigation

Philip Kahn
Leslie Kitchen
Edward M. Riseman

COINS Technical Report 87-57

July 1987

Real-Time Feature Extraction: a Fast Line Finder for Vision-Guided Robot Navigation[1]

Philip Kahn
Leslie Kitchen
Edward M. Riseman

Computer Vision Research Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, MA 01003

Abstract:

The sheer amount of data contained within an image can potentially make pixel-based algorithms computationally infeasible. There are two ways to improve the speed of these algorithms: the amount of effort expended on each pixel can be reduced by careful code optimization, and, the number of pixels fully processed by the algorithm can be reduced. Reducing the effort expended on each pixel reduces the time required to process an image by a constant factor. Selective processing, such as focus of attention, can decrease overall computation by several orders of magnitude by excluding irrelevant pixels which do not significantly contribute to the final result.

This paper develops a fast pixel-based algorithm which uses these principles to achieve real-time feature extraction of lines for use in vision-guided mobile robot navigation. It is based upon a line extraction algorithm first developed by Burns *et al.* [3], though it differs significantly in the way pixels are processed, lines are fitted, and its inherent time performance. At the expense of robustness and reliability, the algorithm is modified and simplified so that it is significantly faster. The resulting Fast Line Finder (FLF) program allows parametric control of computational resources required to extract lines with particular characteristics. As a rough comparison, the FLF program ran in about two seconds for a 256×256 image on a *DEC VAX-11/750* versus several minutes required for the original Burns *et al.* algorithm. A modified version of the FLF which supports table-based image segmentation displayed similar speed and parametric control of computation and image segmentation. The results obtained from an image of an outdoor scene are presented.

[1]This research was supported by the DARPA Autonomous Land Vehicle Project under contract DACA76-85-C-0008, monitored by U.S. Army ETL.

I. Introduction

The sheer amount of data contained within an image can potentially make pixel-based algorithms computationally infeasible. As a result, it can often be difficult to implement fast pixel-based algorithms. The problem is increased by the relative lack of parallelism forced upon algorithms by conventional sequential machines (upon which the vast majority of computer vision applications are still implemented). These potential problems require careful consideration of performance issues.

There are two ways to improve the speed of a pixel-based algorithm: the amount of effort expended on each pixel can be reduced by careful code optimization, and, the number of pixels fully processed by the algorithm can be reduced.

Reducing the effort expended on each pixel reduces the time required to process an image by a constant factor. An algorithm is only as efficient as its parts, and efficient processing of pixels is at the heart of fast programs. Sophisticated code optimization often requires the use of fast languages, meticulous control-flow analysis, a detailed understanding of the speed of language constructs generated by the compiler, and a great deal of time for program development. Unfortunately, due to the limitations of compiler code optimizers, cost of multiplication/division, language overhead and other factors, extreme attempts to reduce the effort expended on each pixel can often produce code that is obscure and undecipherable.

Selective processing, such as focus of attention, can be used to dramatically reduce the amount of overall computation. Pixels which do not significantly contribute to the final result are irrelevant and can be left unprocessed. If the number of relevant pixels is some small fraction of the image and the cost of determining the relevance of a given pixel is small compared to the cost of fully processing that pixel, substantial reductions in computation can be realized. The extent to which this approach is successful depends upon the cost of determining relevant pixels, the cost of processing pixels found relevant, and the sparseness of relevant pixels in the image. Many pixel-based algorithms can simply determine the relevance of pixels, and thus, large increases in speed are possible. For many applications, this approach can produce far greater speedups than obtained by detailed optimization of the code which processes each pixel.

This paper develops a fast pixel-based algorithm which uses the principles mentioned above to achieve "real-time" extraction of lines for use in vision-guided mobile robot navigation [1]. It is based upon a *gradient-based* and *region-based* line extraction algorithm first developed by Burns *et al.* [3]. These line extraction algorithms group adjacent pixels which share similar gradient direction of image intensity into *line support regions,* and then they fit a line to each region. The algorithms have the following basic steps:

1. Compute the direction and magnitude of the image intensity gradient at each pixel.
2. Coarsely quantize the gradient direction of the pixel into one of a set of ranges ("buckets") which will serve as labels.
3. Apply a connected-components algorithm (CCA) to group pixels into line support regions (i.e., group adjacent pixels which have identical bucket labels).
4. Fit lines to line support regions.

The first stage in the line extraction algorithm computes the direction and magnitude of the image intensity gradient at each pixel; figure 1a shows an intensity surface of a dark corner on a light background; gradient direction and magnitude at each pixel corresponds to the orientation and steepness of the intensity surface at that pixel. Once gradients have been computed, pixels are coarsely quantized into one of a fixed number of "buckets" based upon gradient direction. For example, figure 1b shows that all pixel gradient directions from $15\pi/8$ to $\pi/8$ are classified into bucket 1, gradient directions from $\pi/8$ to $3\pi/8$ are classified into bucket 2, etc. A connected-components algorithm (CCA) is then used to group adjacent pixels which have identical bucket labels into line support regions. Figure 1c shows the line support regions (solid white regions) and the underlying gradient vectors shown as arrows (most of which appear as a dot since the gradient magnitude is not large at most points in the intensity surface shown in figure 1a); note the similarity of gradient directions within each line support region. As shown in figure 1d, each resulting line support region is then fit with a line. A similar algorithm was developed by Peleg in [7].

Our algorithm remains in the spirit of [3], but it differs significantly in the way pixels are processed, lines are fitted, and its inherent time performance. The algorithm is modified and simplified so that it runs much faster, although the algorithm will tend to fragment or miss lines that are extracted in the original algorithm. It restricts processing to pixels considered relevant to the final result, more efficient methods are used to process each pixel, and it allows parametric control of computational resources required to extract lines with particular characteristics. The resulting program, which we call the Fast Line Finder (FLF), is being effectively used by the UMass AuRA autonomous mobile robot project to extract lines for use in path recognition and robot localization [1]. As a rough comparison, the FLF program ran in about two seconds for a 256×256 image on a *DEC VAX-11/750* versus several minutes required for the original Burns *et al.* algorithm. A modified version which supports table-based image segmentation, called the Fast Region Finder (FRF) displayed similar speed and parametric control of computation and image segmentation.

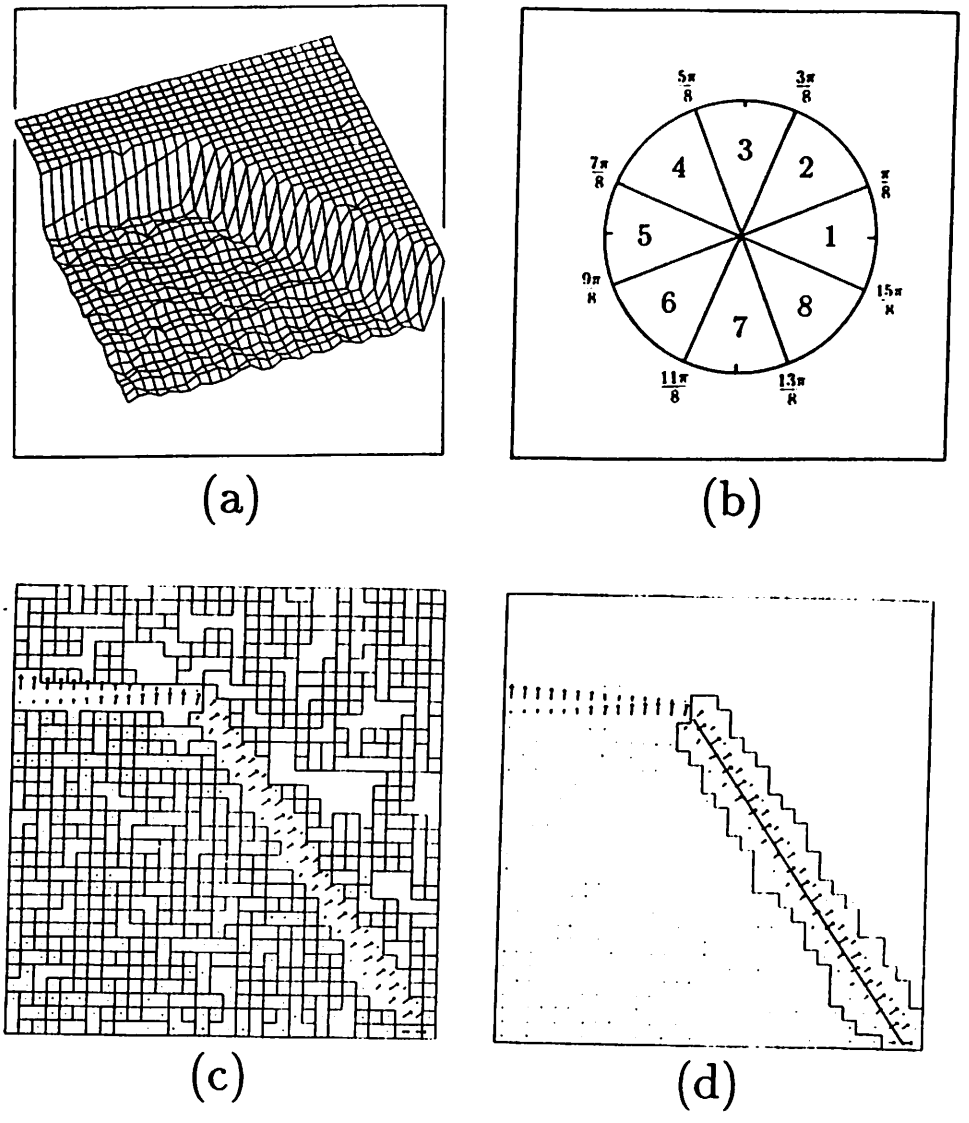

Figure 1: (a) An intensity surface of a corner in which gradient direction and magnitude correspond to the orientation and steepness of the intensity surface; (b) a coarse quantization of gradient direction space for assigning bucket labels to pixels; (c) connected components obtained by grouping adjacent pixels which have identical bucket labels; (d) fitting a line to a line support region. (Figures provided courtesy of Burns *et al.* from [3].)

The next section discusses how the order and use of information can reduce the the overall computation required to extract lines from an image. Efficient algorithms for subcomputations are then examined. We describe the effect of program parameters upon line extraction and overall computational effort. Since relatively inexpensive specialized image processing hardware (e.g., pipeline processors, look-up tables) is sometimes available to speed up computation, we briefly discuss processing initial stages of the algorithm on these devices. The results obtained from an image of an outdoor scene are then presented and a discussion follows.

II. Ordering Computations and Filtering to Reduce Processing

Pixels which do not significantly contribute to the final result are irrelevant and can be filtered out without deleterious effects. Generally, there are several points in a computation at which filtering may be performed based upon differing criteria. The order of the computation determines when these filters may be applied, and this order greatly affects the efficiency of the overall computation.

Several important issues affect the ordering of computational steps. Clearly, steps which require data from other steps impose a partial order. This is usually not a complete order, and there is quite a bit of flexibility in deciding when to perform computations. There is a tradeoff between step complexity, the number of pixels which the step may exclude from further processing, and the cost of further processing. In general, the most dramatic reductions in computational effort can be realized by first performing less expensive steps which can exclude pixels from later, more expensive, processing steps.

The four basic line-extraction steps that were described in the last section differ in the amount of computation required for each pixel and the extent to which they can exclude pixels from further processing. This section discusses how the order of computational steps can exclude most pixels from complete processing.

At the first stage of processing, no information about the "relevance" of pixels is available to the line extraction process. Image gradients must therefore be computed for every pixel in the image. These operations are straightforward and they require relatively little computation [2,5,10]. The gradient direction is used to group pixels into line support regions, and the gradient magnitude is used to weight the placement of each line in its support region.

Gradients measured in flat areas of the intensity surface are extremely prone to noise, so pixels with small gradient magnitudes should be excluded from further processing. Most images contain a large number of pixels with fairly small gradient magnitudes. Figure 2 shows the original image of an outdoor scene; figure 3 shows a histogram of the gradient magnitudes computed from this

image. It is clear from figure 3 that even a low threshold on gradient magnitude can eliminate a major portion of an image from subsequent grouping and line fitting. Because grouping and line fitting occupy most of the computation required to fully process a pixel, avoiding these operations for a large number of pixels results in dramatic gains in speed.

The use of the FLF for real-time mobile robot path recognition and localization allows the orientation of desired lines to be constrained [1]. The robot's last position and *a priori* knowledge of path location and geometry can be used to generate *expectations* about the orientation and position of lines delineating the path in the current input image. The number of lines at expected orientations is generally far less than the total number of lines available from the image.

Final line orientations can be restricted by classifying pixels into directional buckets only when their gradient direction falls within a range of desired line orientations; gradient directions outside the range of desired line orientations should be marked "irrelevant" to exclude them from further processing. The strong relationship between pixel gradient direction and line orientation makes this a highly effective way to significantly reduce the number of pixels fully processed by the algorithm. Figure 4 shows a histogram of gradient directions computed from the image shown in figure 2; gradient direction was computed as $\arctan(I_y/I_x)$ and the histogram does not distinguish between gradients with exactly opposite directions (i.e., $\pi/2$ and $3\pi/2$). The spikes in figure 4 result from the large relative effect of quantization error on the many small gradients in the image; the low frequency variation which peaks at $\pm\pi/4$ reflects a predominance of lines in figure 2 at those orientations. It is clear from figure 4 that restricting the range of allowed gradient directions can exclude a large portion of an image from further processing.

The third processing step groups adjacent (relevant) pixels which share identical buckets into line support regions. A connected components algorithm (CCA) is used to group pixels into these regions (see [9] for a general discussion of CCAs). Each line support region is a two-dimensional, contiguous group of pixels in the image which will later be fitted with a line that "best" describes it.

Just as expectations about line orientation can dramatically reduce computational effort, expectations about line length can also be used to exclude pixels from further processing. Very short lines are prone to noise, often they are not structurally significant, and they usually occur in large numbers. (However, in other applications, short lines may be useful for texture description [8].) Additionally, *expectation-driven* use of a line finder (such as mobile robot path following [1]) can often constrain the length of desired lines.

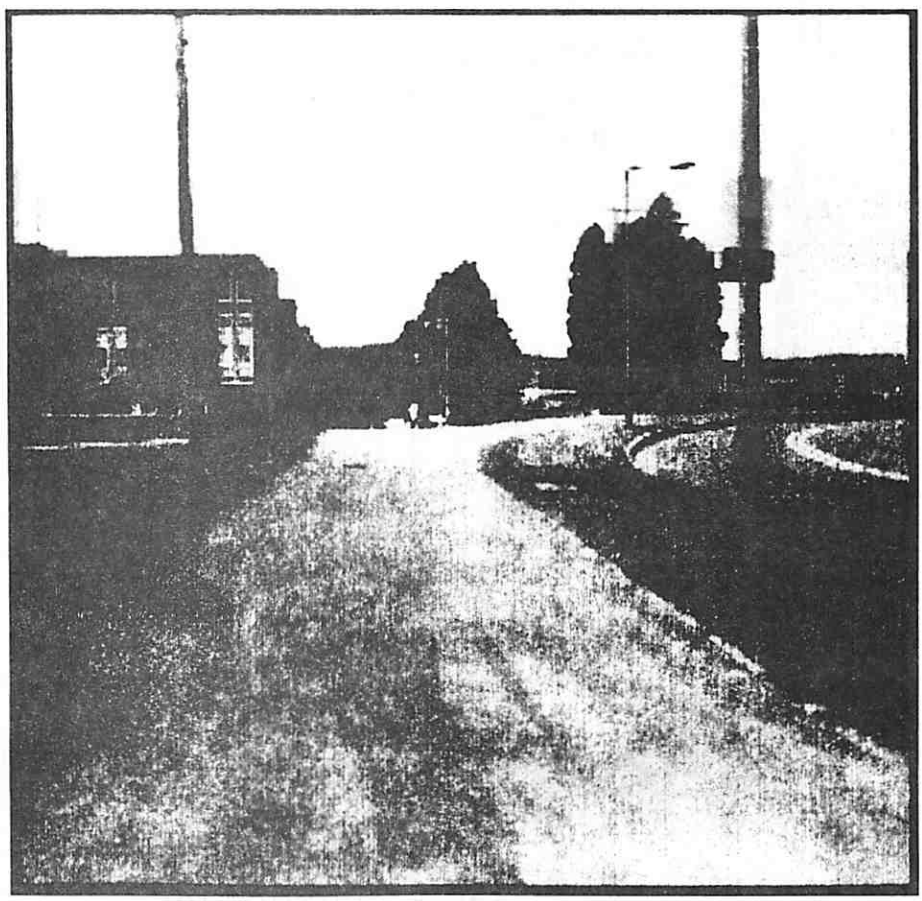

Figure 2: Original image of an outdoor scene in which the robot is currently positioned in the center of the path.

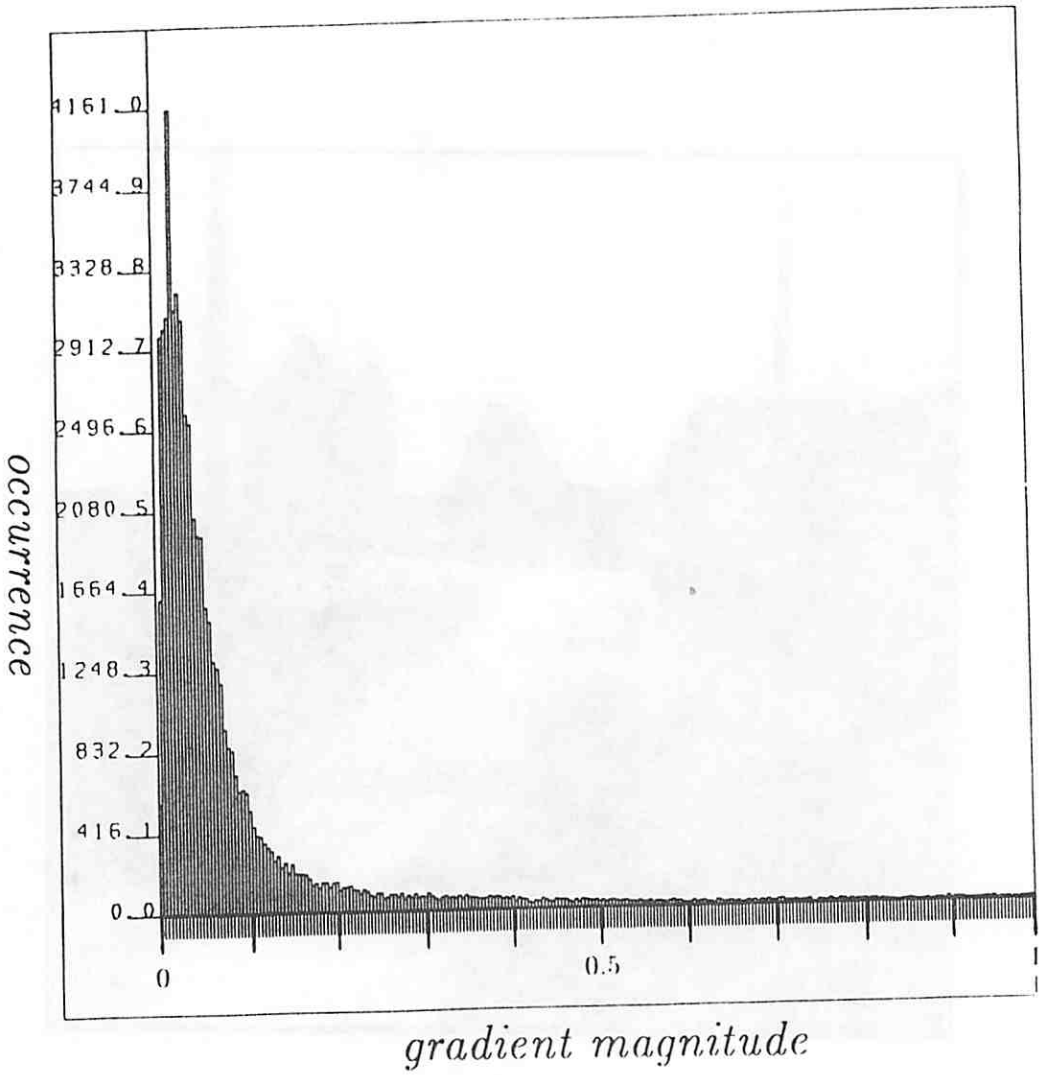

Figure 3: Histogram of gradient magnitudes computed from the original image shown in figure 2.

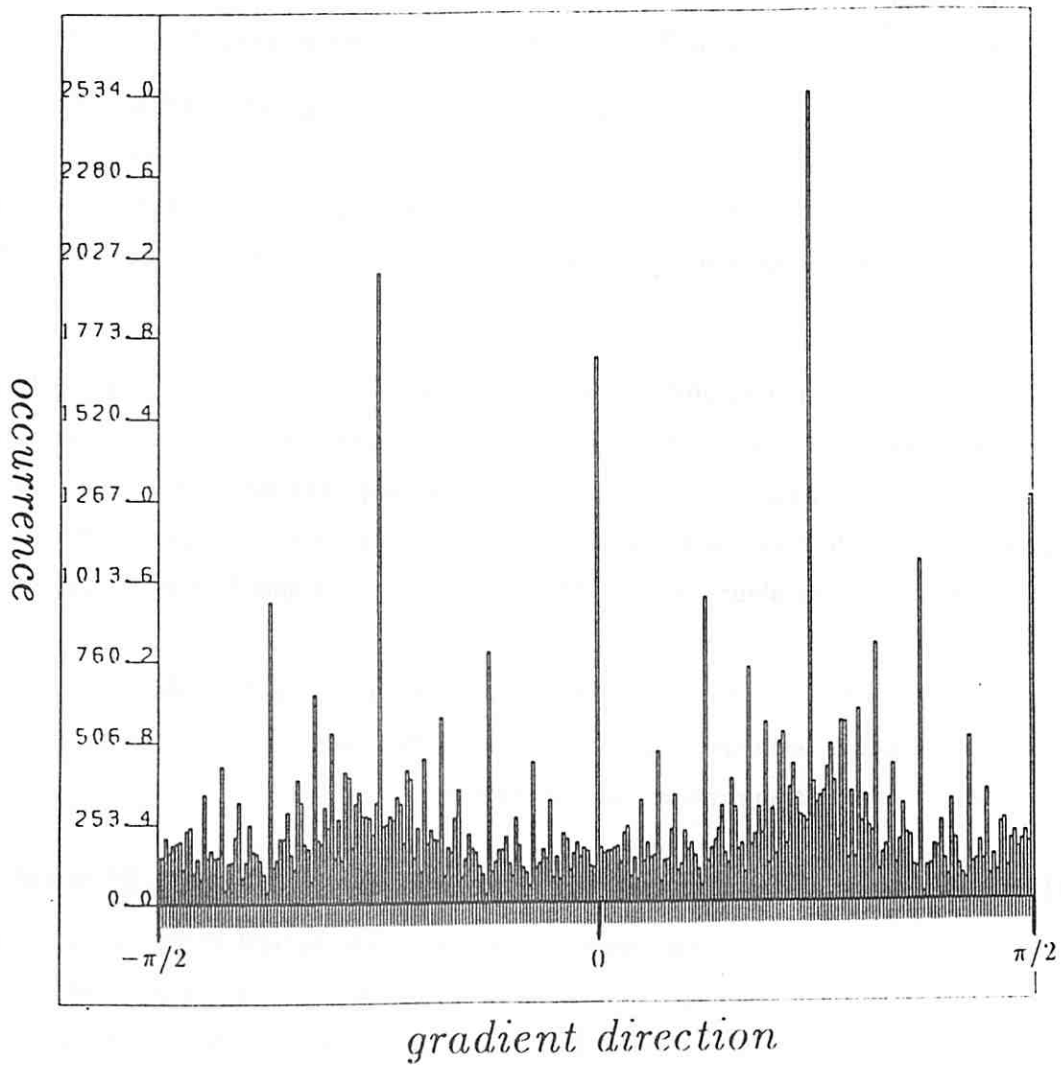

Figure 4: Histogram of gradient directions computed from the original image shown in figure 2. Gradient direction was computed as $\arctan(I_y/I_x)$; the histogram does not distinguish between gradients with exactly opposite directions (i.e., $\pi/2$ and $3\pi/2$).

For 4-connected neighborhoods, the length of a line (measured in pixel units) can be no greater than the number of pixels in its line support region; the area of support regions are generally much larger than the length of the lines fit to them. Hence, a minimum line length can be established by excluding pixels from further processing when they reside in a line support region which contains less than a minimum number of pixels. Most line support regions contain a small number of pixels, so even a small threshold on region size can exclude a large number of pixels from further processing. Excluding pixels at this stage saves the effort of fitting regions with unwanted small lines.

Fitting lines to line support regions is costly (as will be shown in section IIID). As discussed, previous stages in the computation can greatly reduce the number of pixels and regions which need to be fitted with a line. Since line fitting is the last computation in the line extraction process, overall computation at this stage can be further reduced only by the the use of an efficient line fitting algorithm.

This section has shown that careful ordering of subcomputations can significantly reduce the number of pixels processed. Expectation-driven line finding allows more discriminate computation which wastes no effort on unwanted results; more expensive steps can be deferred until necessary to take advantage of the decreased number of pixels in later steps. Reducing the number of pixels processed by each step in the algorithm (i.e., filtering) can result in speedups of several orders of magnitude.

Efficient implementation of steps in a pixel-based algorithm is critical for real-time applications. The next section examines techniques used for the Fast Line Finder (FLF) program. It assumes the order and use of information discussed in this section.

III. Efficient Algorithms for Processing Relevant Pixels

As discussed in the last section, there are four main steps in a gradient-based and region-based line extraction algorithm: compute gradients, classify pixels into buckets by gradient direction, apply a connected components algorithm to form line support regions, and fit lines to the line support regions. We now discuss some methods for computing these steps.

A. Computing Gradient Direction and Magnitude

Gradients can be simply computed using convolution masks to determine the derivatives of the image intensity in the x and y directions, as is done in the Prewitt, Sobel, or similar edge operators. Other techniques have certain advantages, but computational cost makes them inappropriate for most real-time applications. (Discussion of edge operators can be found in [2,5,10].) The size of the convolution mask affects the noise-immunity and accuracy of measured gradients. Kitchen and Malin [6] found that 2×2 masks produce relatively large errors in computed gradient direction. The 3×3 masks were found to be more accurate (e.g., worst case error of about 7° [6]). We did not consider larger masks because they require additional computation without yielding justifiably better results.

The output of the Sobel operator is not significantly better than the Prewitt operator [6]. For the work described here we used the Prewitt operator, which has an advantage in that it can be computed by two additions and three subtractions; the Sobel mask requires additional multiplications, shifts, or additions (depending upon how the mask is computed). On a conventional computer, this small difference saves several operations for every pixel in the image; usually this tangibly decreases computation. This choice may not be justified for some specialized image processing hardware in which there may be no difference in computation time between different convolution operators; the Sobel might then be the better choice due to its slightly greater accuracy.

Gradient magnitude is computed from the derivatives in the x and y directions. That is, $m = \sqrt{I_x^2 + I_y^2}$ where m is gradient magnitude, I_x and I_y are the derivatives of image intensity in the x and y directions, respectively. Exactly computing gradient magnitude in this way is expensive since it requires one addition, two multiplications, and a square root. Moreover, I_x and I_y are themselves only approximations of the true gradient components [6], and so extreme accuracy when computing gradient magnitude is a waste of resources. "City block" computation of gradient magnitude provides a useful approximation: that is, $m \approx |I_x| + |I_y|$. It never underestimates gradient magnitude, and thus, city block is a conservative measure in that thresholding on the approximation will not incorrectly exclude pixels from further processing. As will be discussed, the line fitting stage of our algorithm (as does that of Burns *et al.* [3]) uses gradient magnitude to weight the placement of lines in support regions. Inaccuracies caused by city block approximation are directionally dependent. Because pixels in line support regions are defined by similar gradient directions, the errors caused by city-block tend to similarly affect pixels in a region; hence these inaccuracies do not significantly affect the fitting of lines to support regions.

B. Coarse Quantization of Gradient Directions Into Buckets

As shown in figure 1, the gradient direction at a pixel is used to classify pixels into one of a fixed number of directional buckets. These buckets will later be used to form line support regions.

The directional buckets are most conveniently specified by angular measures (e.g., radians); yet, only the derivatives in the x and y directions are available from the image using the operators discussed. A straightforward way to classify pixels into buckets is to convert the derivatives into an angular measure, and then use the angular measure to determine the appropriate bucket. The conversion to radians is $\alpha = \arctan(I_y/I_x)$ (corrected for quadrant). Unfortunately, the division and trigonometric function make this a computationally expensive solution. Because many pixels must be classified into buckets, this approach requires substantial computation. Further, this approach is excessively expensive because it computes angular direction more precisely than is necessary to classify pixels into buckets. The gradient directions computed from images are somewhat inaccurate [6], noisy, and buckets are coarsely quantized; thus, precise methods are not justified.

A simple two-dimensional lookup table may be used to directly map derivatives into buckets. I_x and I_y at each relevant pixel can index into the table position which contains the associated bucket label. The table represents the mapping from the directional derivatives to buckets, and it only requires a single table access. The coarseness of this mapping is determined by the lookup table size. A smaller table incurs larger discretization errors, and errors get larger as the origin is approached. A larger table better approximates the actual computed values, but memory limits place a practical upper bound on table size. Figure 5 shows the discretization error (one degree per isocontour) in the positive quadrant for a 65×65 bucket lookup table (i.e., a table that covers the range -32 to $+32$ in both I_x and I_y). Directional derivatives need not directly index into this table since they can be "folded" until they are small enough to index into the table. For example, if $I_x = 255$, $I_y = 102$, and the table is 65×65, then these directional derivatives can be progressively halved until both are inclusively within the range -32 to $+32$ (i.e., they should be halved three times until $I_x = 31$ and $I_y = 12$). This "folding" preserves the direction of the vector (with some small rounding error) while allowing significantly smaller tables. Though larger tables reduce the average error, a 65×65 lookup table was found to be sufficiently accurate for the FLF program; the error caused by the table is less than that caused by the edge operators themselves [6].

The most straightforward way to create the lookup table computes the angular gradient direction for each index into the lookup table, coarsely quantizes that direction using the bucket specifications, and places the resulting bucket label into that table position. The angular direction

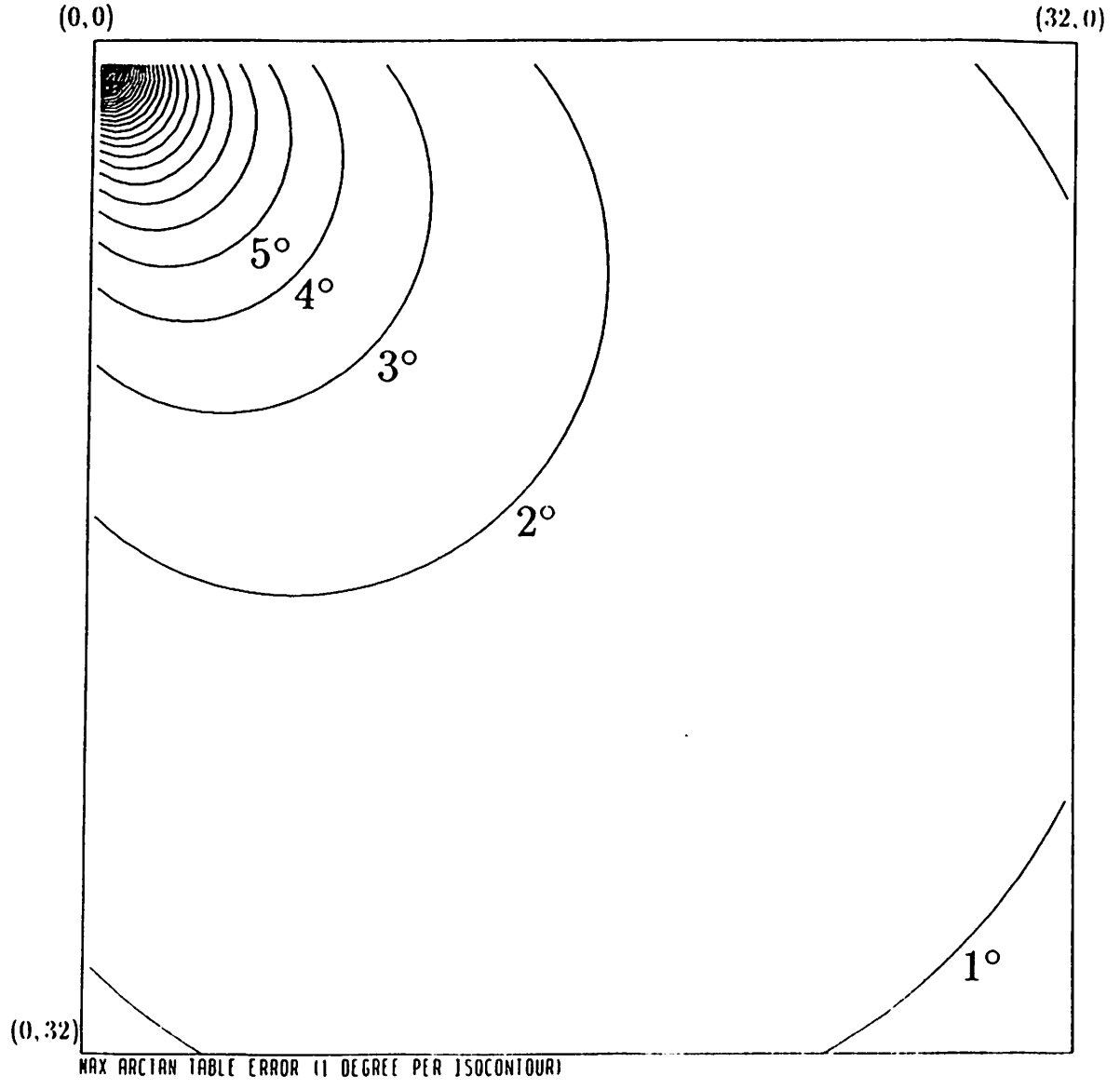

Figure 5: Discretization error (one degree per isocontour) in the positive quadrant for a 65 · 65 directional derivative to gradient direction bucket lookup table.

for each table position can be computed once and used to initialize other tables to avoid recomputing them. This approach is simple to implement and fairly efficient. More sophisticated approaches are possible (e.g., by observing that buckets occupy contiguous positions in the table), but the time spent on table loading does not often justify the added complexity.

Quantization into buckets introduces problems when line orientations fall on bucket boundaries, which can sometimes fragment line support regions. Burns *et al.* [3] overcame this problem by running the line extraction process twice; lines are extracted for the original buckets and then lines are extracted for buckets rotated by half the bucket width. Thus, two complete sets of line support regions are produced; final line selection is based upon the number of pixels "voting" for each line support region. Though this technique works well, it requires more than double the computation. Further, since we permit arbitrary bucket specification (not just the uniform quantization shown in figure 1b and used in [3]), it is sometimes not possible to rotate the buckets in a well defined way. Empirically, the lines extracted by the FLF program do not suffer greatly from boundary fragmentation. This is due, in part, to some of our particular applications in the domain of mobile robotics, which allows expectations to constrain desired lines, so that buckets can be centered at the expected line orientations to reduce line fragmentation. For applications not allowing this approach, the voting scheme of Burns *et al.* [3] can be adopted.

C. Connected Components Algorithm (CCA) to Form Line Support Regions

Connected components algorithms (CCA) are commonly used in computer vision, image processing, and a wide range of other fields. In gradient-based and region-based line extraction algorithms, the CCA groups adjacent pixels with identical bucket labels into line support regions. When the CCA has completed, each contiguous group of pixels sharing identical bucket labels will have been assigned a unique region number. Later line fitting will determine what line best fits each of these regions.

This section provides only the bare details of the connected components algorithm (CCA) used in the fast line finder (FLF). A more detailed understanding can be obtained from [3,9].

The FLF uses a four-connected neighborhood to define adjacency among pixels. That is, only the pixels above, below, left, and right of a given pixel are considered that pixel's neighbors. Other neighborhood definitions can be used [9] (e.g., eight-connected), but a four-connected neighborhood provides good results without incurring the additional computation required to evaluate larger neighborhoods.

The CCA algorithm operates in several stages. The first stage scans over all "relevant" pixels

(as previously determined by their gradient), and it groups adjacent pixels that share identical bucket names into region *fragments*. Region fragments occur because a raster scan is used. For example, scanning the upper half of a "U" shaped region would result in grouping the left and right portions of the region into different fragments. Only when the bottom part of the region was scanned would it be found that the two fragments are adjacent and share identical buckets, and thus, they are part of a single region. These *fragment equivalence relations* are stored in an adjacency list as they are encountered for later use in forming complete regions. The first stage of the CCA produces an image containing fragment labels, a fragment equivalence list, and a count of the number of pixels contained in each fragment.

The second stage of CCA processing combines region fragments by assigning each fragment a unique region label in which the fragment is contained. This is easily done using the fragment equivalence list. The index into the adjacency list is a fragment label; each position contains a linked list of all other fragments in the same region. Regions are formed by merging fragments which share an equivalence relation. Each fragment is assigned a region label so a single level of redirection is required to identify a region for each position in the fragment labelled image. This level of redirection requires the same amount of computation as methods which explicitly create a region labelled image, yet the redirection significantly reduces space requirements. It is good practice to have conditionally compiled code that produces a region labelled image for debugging and other purposes.

As previously discussed, final line lengths can be restricted by fitting lines only to line support regions containing more than a certain number of pixels. Only pixels contained within these minimum-sized regions are considered "relevant" to the line fitting process. This can easily be accomplished in the second stage of the CCA. We know the number of pixels contained within each fragment (these values were accumulated in the first step). When stage two in the CCA merges these fragments into regions, the number of pixels contained within the line support region as a whole can be computed. Any region which contains less than a certain number of pixels can be eliminated by simply assigning an "irrelevant" region label to its constituent fragments. The line fitting step will bypass fitting any lines to pixels contained within "irrelevant" regions.

D. Fitting Lines to Line Support Regions

Very generally, a line fitting algorithm should "best" fit a line to the line support regions and their underlying intensity surfaces. There are many algorithms for fitting lines to a collection of points, and the required amount of computation can vary widely. Burns *et al.* [3] fit a plane to

the intensity surface underlying each line support region. Lines are determined by intersecting this plane with a horizontal plane at the mean region intensity value. Though the results obtained with this technique are good, other methods can fit equally good lines while requiring far less computation.

Another way to fit lines to support regions views region pixels as a cluster of points to which a line must be fit. Computing the principal axis does just this, and it requires significantly less computation than required for plane fitting and intersection. The partial statistics needed from the region pixels can be computed in a single pass over the image; fitting a line can then be completed by performing a final computation on these partial statistics. A fuller discussion of the principal axis computation may be found in [2,5,10].

Partial sums in the form of a scatter matrix yield eigenvalues which provide the principal axis of a cluster of points. The scatter matrix for a line support region is

$$\begin{pmatrix} a & b \\ b & c \end{pmatrix} = \begin{pmatrix} \sum wX^2 & \sum wXY \\ \sum wXY & \sum wY^2 \end{pmatrix}$$

where w is a weighting factor for each pixel in the region, and (X, Y) are the region pixel coordinates whose origin has been relocated to the centroid of the support region. The gradient magnitude is used as the weighting factor w (as is done by Burns *et al.* [3]). Because this formulation requires two passes over the image (one to find the centroids, and one to compute the scatter matrix), we use the standard alternative form which allows partial sums to be accumulated in a single pass:

$$a = \sum wx^2 - \frac{(\sum wx)^2}{\sum w}$$

$$b = \sum wxy - \frac{\sum wx \sum wy}{\sum w}$$

$$c = \sum wy^2 - \frac{(\sum wy)^2}{\sum w}$$

where a, b, and c are the elements of the scatter matrix, and (x, y) are the region pixel coordinates.

The line which best describes a group of pixels in a support region is determined by finding the eigenvalues of the scatter matrix. The characteristic equation of the scatter matrix is

$$\begin{pmatrix} a & b \\ b & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \lambda \begin{pmatrix} x \\ y \end{pmatrix} \equiv \begin{vmatrix} a - \lambda & b \\ b & c - \lambda \end{vmatrix} = 0$$

which can be solved using the quadratic formula

$$\frac{a+c \pm \sqrt{(-a-c)^2 - 4(ac-b^2)}}{2}$$

A less expensive form to compute is

$$\frac{a+c}{2} \pm \sqrt{\frac{(a-c)^2}{4} + b^2}$$

The solutions to the characteristic equation yield two eigenvalues. The small eigenvalue V_S is obtained by subtracting the second term, and the large eigenvalue V_L is obtained by adding the second term. The ratio of these large and small eigenvalues provides an extremely useful measure of the straightness and width of the line support region about the fitted line. For V_S/V_L, small ratios correspond to small region widths (which are more "line-like") and large ratios correspond to large region widths (which are more circular).

The orientation of the best fitting line is given by the eigenvector derived from the small eigenvalue and the scatter matrix

$$\vec{V}_S = \arctan\left(\frac{V_S - a}{b}\right)$$

where $\vec{V}_S$ is the orientation of the fitted line (corrected for quadrant). This computation does not encode the *sense* of the line (i.e., the direction of brightness). If the coordinate system sweeps counterclockwise from the x axis and the protocol is adopted that the direction of brightness is $\vec{V}_S + \frac{\pi}{2}$ (i.e., "right is bright"), the sense of $\vec{V}_S$ is correct when

$$\cos(\vec{V}_S)\sum I_x + \sin(\vec{V}_S)\sum I_y \geq 0$$

where $\sum I_x$ and $\sum I_y$ are accumulated partial statistics for the line support region. If the sense of $\vec{V}_S$ is incorrect, it is simply corrected by adding π to it.

Numeric capacity is important when accumulating the partial statistics. Some of these statistics sum the square of image coordinates, and for large regions, these numbers can sometimes cause numeric overflow. Though not a serious concern, care should be taken to ensure that partial sums never exceed the representation. This problem can often be eliminated by moving the origin to the center of the image, scaling down all coordinates and gradient magnitudes, restricting the number of pixels sampled within any line support region, or using large numeric representations (though this can incur excessive computation for numeric operations).

It has been assumed that every pixel within a relevant line support region is used in accumulating partial statistics. Though this makes sense for small regions, this fine granularity may not be justified for larger regions. That is, the relative contribution of a single pixel in a region decreases as the total number of pixels in the region increases. Sampling only a subset of pixels in large support regions can dramatically reduce the amount of computation devoted to accumulating partial statistics. The sampling rate within each region should be determined by the number of pixels contained within that region; large regions are more sparsely sampled than smaller regions. Care should be taken that aliasing effects do not result from the particular sampling technique; random sampling using precomputed sampling tables offers an efficient solution. Obviously, for sampling to be worthwhile, the effort required to determine which pixels to sample must be less than the time required to sample the pixel. Further, unless a large number of image pixels are contained in large regions, sampling may not be justified. Though sampling was not incorporated into the FLF program, it would have likely resulted in a small performance improvement.

The principal axis for each line support region determines the line orientation $\vec{V}_S$ and the region centroid anchors the position of that line. The endpoints of the line segment must still be determined.

Conceptually, it is difficult to define exactly where the endpoints of a line fitted to a region *should* be placed. A reasonable definition would place the endpoints where the two region pixels furthest from the centroid project perpendicularly onto the line. Though intuitive, this solution has problems with irregularly shaped regions, and it requires an extra pass through the image (since the projection cannot be done until after the line has been fitted).

A simpler way to determine line endpoints was used by the FLF. The endpoints for each line were determined by intersecting the principal axis line with an upright box bounding its support region. The length of lines may be increased by enlarging all bounding boxes by a fixed amount. The box bounding the line support region can be easily obtained when the partial statistics are gathered to build the scatter matrix; the bounding box is defined by the smallest and largest (x, y) pixel coordinates in the region. This method is efficient because defining the bounding box and intersecting it with the principal axis line is computationally simple. This technique causes some anomalies (e.g., rotating a noncircular region in the plane changes the endpoints somewhat), but it works extremely well for elongated support regions. It is not meaningful to fit a line to an irregular region (as when the ratio of the eigenvalues is large), and so unusual fitting of lines to such regions is not a serious concern.

E. The Two-Pass Line Extraction Algorithm

The components of the FLF algorithm that have been described fit together to form the full algorithm:

1.0 PASS 1 over the image

- 1.1 **for** each image pixel **do**
  - 1.1.1 Compute the gradient direction and magnitude
  - 1.1.2 Threshold on gradient magnitude
  - 1.1.3 Use gradient direction to classify suprathreshold pixels into buckets. (Pixels below threshold and pixels whose gradient is not in a direction of interest are specially labelled and ignored in all subsequent processing.)
  - 1.1.4 Perform connected components analysis to group adjacent pixels that share identical bucket labels into region fragments, and build a *fragment equivalence list*
- 1.2 **for** every unprocessed region fragment **do**
  - 1.2.1 Use the *fragment equivalence list* to merge fragments into line support regions and set the region pixel count to the sum of the fragment pixel counts
  - 1.2.2 Eliminate regions whose pixel count is below a certain threshold by tagging their constituent fragments "insignificant" so they will be ignored in subsequent processing
  - 1.2.3 Assign region labels to fragments which are part of these "significant" regions

2.0 PASS 2 over the fragment labelled image and gradient magnitude image

- 2.1 **for** each image pixel in a "significant" region **do**
  - 2.1.1 For the region within which the pixel is contained, accumulate statistics needed to compute the scatter matrix and endpoints (for line fitting)
  - 2.1.2 If desired, build a region labelled image
- 2.2 **for** each "significant" region **do**
  - 2.2.1 Compute the scatter matrix from the accumulated statistics
  - 2.2.2 Compute the best-fit line orientation and centroid anchor position
  - 2.2.3 Compute the line endpoints
  - 2.2.4 Put the fitted line and associated statistics into a line list to be output by the FLF program

A one-pass algorithm is possible, but it cannot eliminate pixels from complete processing; hence, the two-pass version is faster. The two-pass FLF algorithm was chosen because the first pass can exclude the bulk of the image from being processing by the second pass (the line fitting stage).

IV. Effect of Parameters on Line Extraction and Program Performance

A key aspect of the fast line finder (FLF) program is that it has been designed for vision-guided navigation in the UMASS AuRA mobile robot project [1]. The robot is supposed to move through its environment using visual information to localize relative object positions. An internal spatial description of structures in the environment provide *expectations* which limit the position, length, and orientation of significant lines in the visual field. Visual localization is accomplished by comparing the expected placement of environmental structures against the lines extracted from the image. This approach is highly effective in road following, and it appears to be capable of achieving an advanced level of autonomous mobile robot navigation through the environment.

Extracting unwanted lines from the image wastes computation. This can be avoided by manipulating one or more parameters in the fast line finder (FLF) program which control extracted line characteristics. Though the results obtained by [3] can be approximated by appropriate parameter settings (except for the additional set of lines from the rotated buckets), structural image expectations can provide better performance. This section describes the general effect of these parameters upon the line extraction process.

A. Effect of Gradient Magnitude Threshold

As shown in figure 3, thresholding small gradient magnitudes can significantly decrease the number of pixels fully processed by the algorithm. Eliminating pixels at this stage is inexpensive and it avoids the substantially more expensive computation at later stages in the algorithm. The gradient magnitude threshold is one of the most important parameters which controls the amount of overall computation. Without question, at least a small gradient magnitude threshold is essential; small gradients are noisy, they constitute a large portion of the image, and they rarely yield useful structural image information.

Increasing a small gradient magnitude threshold usually decreases the number of extracted line support regions; that is, all regions in which all pixels have gradient magnitudes below threshold are eliminated. At small thresholds, a large number of these regions usually occur in an image. As the threshold increases, the number of regions which contain *only* subthreshold pixels decreases. As a result, higher thresholds tend to eliminate only a portion of pixels contained within regions that would have otherwise been formed when using a lower threshold. When the pixels eliminated by a higher threshold bisect a region, the region fragments into two smaller regions. So in general, larger gradient magnitude thresholds tend to fragment regions into smaller regions and they tend to

decrease the average region size. Smaller line support regions translate into smaller line segments being extracted from the image. Thus, larger gradient magnitude thresholds tend to produce shorter line segments.

Sensor noise levels define the smallest meaningful gradient magnitude threshold, though as discussed above, larger thresholds can substantially reduce the number of pixels processed without seriously affecting the quality of extracted lines. There is no strict method for determining an appropriate threshold. When used for the UMASS AuRA project, thresholds were determined by empirical analysis of outdoor image sequences. A gradient magnitude threshold of about 5% (threshold/range) excluded a large number of pixels from full processing while avoiding significant region fragmentation. Another approach initially uses a low threshold to extract lines. The average gradient magnitude for all extracted lines found to be *structurally significant* [1] could then be used to set a larger, more appropriate threshold for processing future frames.

B. Effect of Bucket Width and Direction

The buckets determine how pixels are classified (e.g., figure 1b). Restricting the orientation of extracted lines can result in substantial savings in computation since eliminating pixels which contribute to unwanted lines requires only a single table lookup.

As bucket width increases, the size of line support regions tends to increase since the grouping of pixels is on the basis of gradient direction less restricted. At the extreme, if all gradient directions were contained within a single bucket, then all contiguous and "relevant" pixels would be grouped into a single region; then, only the threshold on gradient magnitude would control region formation. Conversely, decreasing bucket width requires region pixels be more uniform in gradient direction. Thus, smaller buckets tend to fragment regions and decrease region size.

For any given image, there is a meaningful upper and lower bound on bucket width. A very small bucket would be susceptible to minor variations in the intensity surface and bucket boundary fragmentation; typically, resulting lines would be small and less structurally related to the environment. Similarly, very large buckets fail to discriminate between unrelated pixels; resulting line support regions would be large, less straight, and less structurally related to the environment. The lower bound for bucket sizes is determined by the inherent inaccuracies in determining gradient direction [6]. Appropriate bucket widths can be determined by empirical investigation or recursive line extraction/parameter setting. Burns *et al.* [3] discuss bucket size issues.

Line orientation can be restricted by excluding a bucket definition for undesired orientations. Since gradient direction is strongly related to line orientation, excluding pixels whose gradient directions are not within desired line orientations has little effect upon the quality of extracted

lines.

C. Effect of Region Pixel Count Threshold

As previously mentioned, a line can be no longer than its line support region (for 4-connected neighborhoods). Final line lengths may thus be restricted by eliminating regions which contain less than a certain number of pixels. Eliminating subthreshold regions avoids fitting unwanted lines; this can substantially increase performance.

The exact relationship between line length and region size depends upon the linearity of the line support regions about the fitted lines. We assume that line lengths are expressed in pixel units. At the extreme, when the line exactly fits the line support region, the region contains the same number of pixels as the length of the fitted line. Such a precise fit is extremely rare. Most often, even the best of line support regions are a few pixels wide. Therefore, the region pixel count threshold should be some multiple of the minimum desired line length.

The multiple of line length used to set the region pixel count threshold can be determined in several ways. A conservative multiple can be chosen by empirical investigation (e.g., region width of five pixels). This method was successfully used by the UMASS AuRA project.

The region width is related to the ratio of the eigenvalues; small ratios correspond to small region widths and large ratios correspond to large region widths. This observation facilitates another approach which initially sets a low region pixel count threshold and it then determines an appropriate region width threshold from extracted lines whose eigenvalue ratios are small. Similar approaches which do not use the ratio of eigenvalues can be used (e.g., the ratio of region pixel counts to fitted lines can be used), although eigenvalue ratios seem particularly appropriate.

VI. Preprocessing on Specialized Image Processing Hardware

Substantial speedups in the FLF algorithm can be realized by preprocessing on specialized image processing (IP) hardware. Specifically, gradients and perhaps coarse quantization of gradients into buckets could be computed on these devices. Subsequent processing steps (e.g., the connected components algorithm, line fitting, etc.) are not pixel oriented and so they are not feasible for most IP devices. This section discusses initial FLF algorithm steps which may be put on specialized IP hardware.

Computing gradients is an elementary operation on most image processing hardware. Generally, this is accomplished by convolving the image with appropriate masks (e.g., Prewitt, Sobel, etc); convolution hardware is particularly well-suited. The computational concerns on a sequential

machine that argued for the Prewitt operator do not apply to IP devices. Instead, a better quality mask may often be used with no resulting increase in computation. Selection of convolution masks depends upon the amount of computation required by the specific IP device and the desired level of accuracy. Computing gradient magnitude from the directional derivatives was discussed in section IIIA, and this computation is particularly simple on most IP hardware.

For specialized IP hardware to be useful, it must take less time to obtain the preprocessed data than it would have taken on a conventional sequential machine. The total preprocessing time is the sum of the IP computing and data transfer times. Most often, the IP hardware is connected to a host computer. Since we assume IP devices can preprocess faster than a sequential machine, the feasibility of using these devices depends upon the data transfer rate. Preprocessing on IP hardware attached to a host computer doubles or triples the number of images to transfer, so care must be taken to ensure that increased data transfer does not undermine its usefulness. IP devices with direct memory access (DMA) to the host are usually required to obtain real-time performance; IP devices are then feasible when the databus has capacity to transfer three or four images without significant delay.

The UMASS AuRA project is currently downloading the computation of gradients and directional buckets onto a *Gould DeAnza IP8500* image processing device using a *DEC VAX-11/750* as the host computer with DMA. The video camera on the mobile robot uses a VHF transmitter to transfer the real-time images to a digitizer on the *IP8500*. Preprocessing on the *IP8500* is desirable since the images are already on the device. Highly efficient image transfer required significant effort, but was found to be possible. Timing results are not available at this time, but it appears that a $10 - 20\%$ speedup will be possible over the current version.

VII. Results

This section demonstrates the line extraction algorithm on an outdoor scene which was one of a sequence of images acquired from a mobile robot moving through the environment; Arkin *et al.* [1] describes how these lines can be used to guide a robot through the environment.

Figure 2 shows the original image of an outdoor scene in which the robot is currently positioned in the center of the path. Executing steps $1.0 - 2.1$ of the line extraction algorithm (as shown in section IIIE) results in the line support regions shown in figure 6; the default eight buckets shown in figure 1b were used, and a gradient magnitude threshold of about 2% (threshold/range) and a region pixel count threshold of 10 was used. Figure 7 shows the lines fitted to the line support regions shown in figure 6 (as described in sections IIID and IIIE).

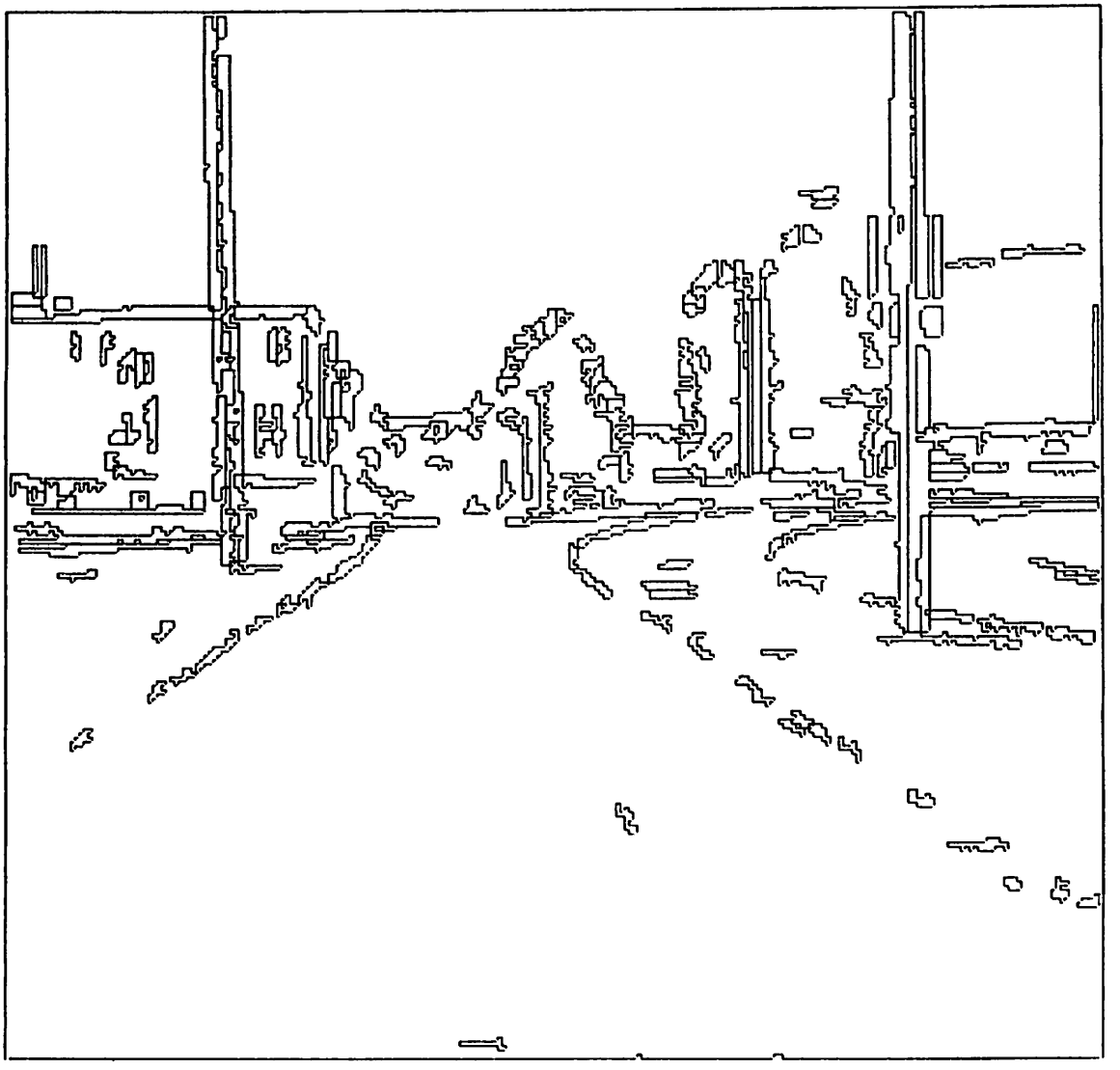

Figure 6: Line support regions extracted from the image shown in figure 2. The default eight buckets shown in figure 1b were used; a gradient magnitude threshold of about 2% (threshold/range) and a region pixel count threshold of 10 was used.

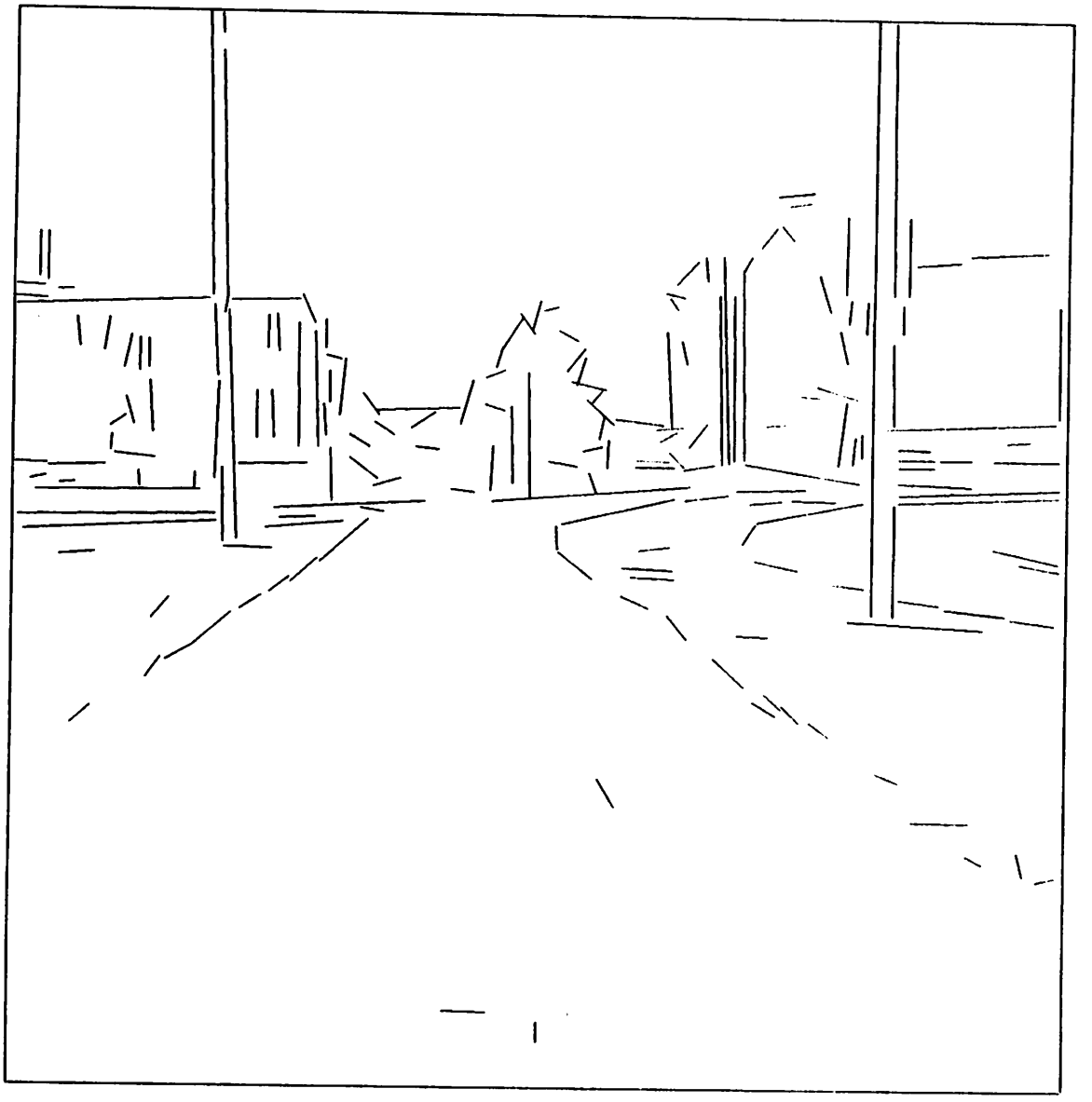

Figure 7: Lines fitted to the line support regions shown in figure 6 using the algorithms described in sections IIID and IIIE.

As discussed in [1], path-following was achieved by directing the robot along the center of the path; the centerline was determined from the current robot position and the vanishing point of the path which was found by intersecting the lines fit to the left and right sides of the path. For this simple method of navigation, only lines which demarcate the path sides provide useful information; effort spent extracting lines which are not part of the path wastes computational resources. Figure 8 shows the lines extracted when the directional buckets are tuned to the expected orientation of the path sides. That is, two buckets centered on the expected path edge orientations were used. In addition to reducing overall computation, this tuning largely avoids bucket boundary fragmentation as discussed in section IIIB. Thus, as shown in figure 8, longer line segments are obtained than when bucket tuning is not performed (as shown in figure 7).

VIII. Discussion

This paper has presented a methodology for developing fast pixel-based algorithms for feature extraction. This methodology was developed in the context of a fast line finder (FLF) which was effectively used for real-time vision-guided robot navigation (as described in [1]). As a rough comparison, the FLF program ran in about two seconds for a 256×256 image on a *DEC VAX-11/750* versus several minutes required for the original Burns *et al.* algorithm.

In addition, the line extraction algorithm summarized in section IIIE was modified in a simple manner to provide a table-based region segmenter that has also been used by the UMass AuRA project. Instead of computing gradients from an intensity image and then mapping them into buckets, the modified program allows a user-specified lookup-table to map pixel intensity values from a general image into corresponding labels. Analogous to thresholding on gradients, this modified version indicates (via the lookup-table) what pixel values are not relevant to the desired end result. The resulting labelled image is then processed by a connected components algorithm to produce the segmented regions. Rather than fit lines to these regions, the modified program outputs statistics describing each segmented region (e.g., centroid, mean value, dispersion as measured by the ratio of eigenvalues, etc.). This modified program was thus called a Fast Region Finder (FRF). Arkin *et al.* [1] describe how the FRF can be used to support autonomous mobile robot navigation.

The extent to which vision can support robot applications is directly affected by the speed of visual processing. The availability of a fast line extraction algorithm allowed the UMass AuRA project to visually guide a mobile robot through the environment in real-time. Similarly, the methodology presented in this paper can be extended to other feature extraction techniques which can support robot navigation, industrial robotics, and a host of other applications. Hopefully, this

melding of vision and robotics will lead to a far more advanced level at which machines can interact with their environment.

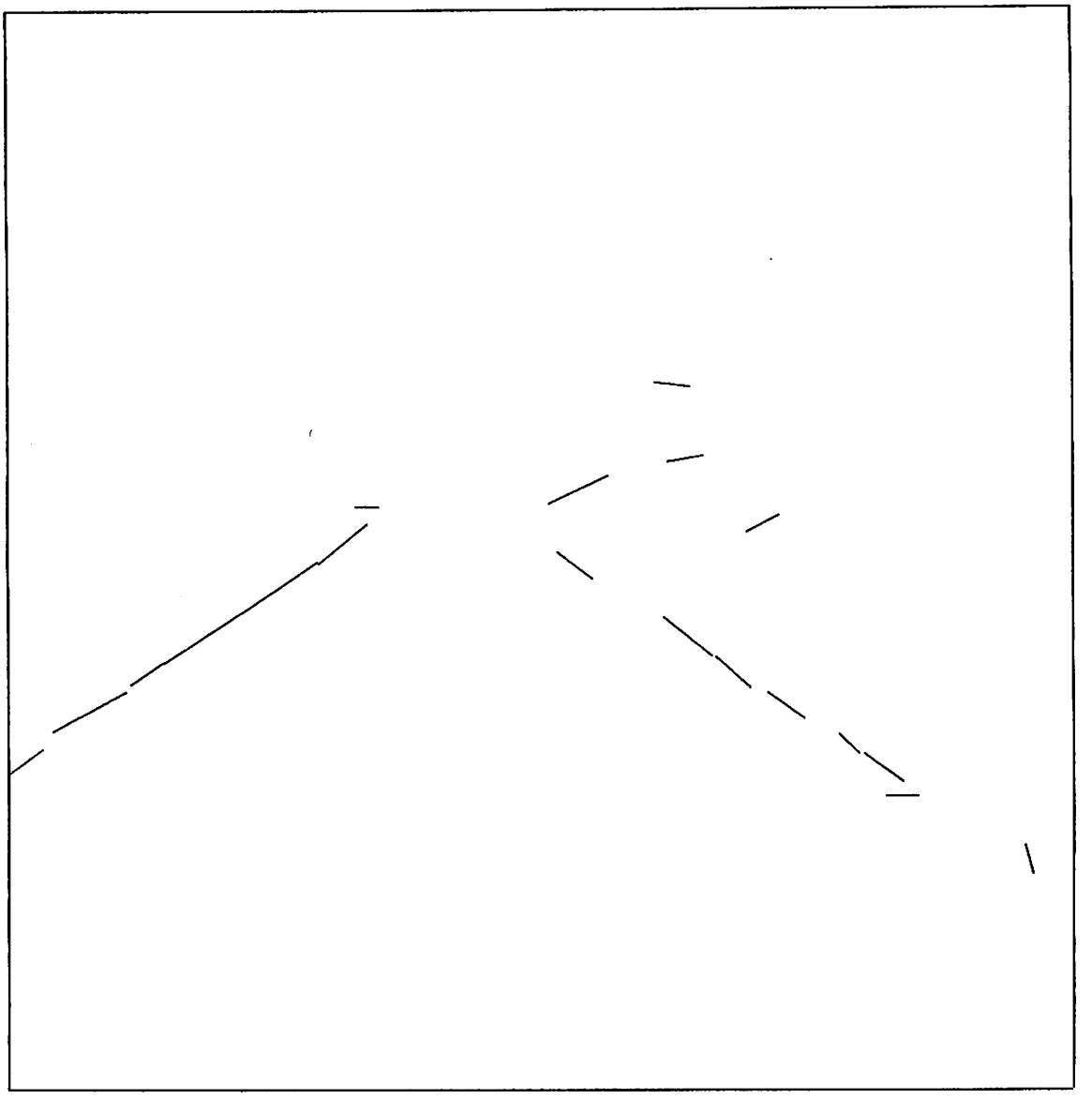

Figure 8: Lines extracted from the image shown in figure 2 when the directional buckets are tuned to the expected orientation of the path sides.

Acknowledgements

We are endebted to J.B. Burns for his assessments of performance issues and solution techniques. Special thanks are due to R.C. Arkin, M. Boldt, and P. Anandan.

References

[1] R.C. Arkin, "Working Towards Cosmopolitan Robots: Intelligent Navigation In Extended Man-Made Environments", Ph.D. dissertation, Computer & Information Science, University of Massachusetts at Amherst, September, 1987.

[2] D. Ballard and C.M. Brown, *Computer Vision*, Prentice-Hall: NJ, 1982.

[3] J.B. Burns, A.R. Hanson, and E.M. Riseman, "Extracting Straight Lines," *IEEE Transactions on Pattern Analysis & Machine Intelligence,* vol. PAMI-8, no. 4, pp. 425–455, July 1986.

[4] J.H. Burrill, "Low Level Vision System," Technical Report 87-14, Computer and Information Science Department, University of Massachusetts, Amherst, MA, January 1987.

[5] Duda and Hart, *Pattern Classification and Scene Analysis,* Wiley: NY, 1973.

[6] L.J. Kitchen and J. Malin, "The Effect of Spatial Discretization on the Magnitude and Direction Response of Simple Differential Edge Operators on a Step Edge, Part 1: Square Pixel Receptive Fields," Technical Report 87-34, Computer and Information Science Department, University of Massachusetts, Amherst, MA, April 1987.

[7] S. Peleg, "Straight Edge Enhancement and Mapping," Computer Science Technical Report 694, University of Maryland, College Park, MD, September 1978.

[8] F.M. Vilnrotter, R. Nevatia, and K.E. Price, "Structural Analysis of Natural Textures," *IEEE Transactions on Pattern Analysis & Machine Intelligence,* vol. PAMI-8, no. 1, pp. 76-89, January 1986.

[9] C. Ronse, *Connected Components Algorithms for Binary Images,* Research Studies Press: NY, 1984.

[10] A. Rosenfeld and A.C. Kak, *Digital Picture Processing,* Academic Press: NY, 1976.