Real-Time Radar Image Understanding: A Machine-Intelligence Approach

Ann Marie Aull, Robert A. Gabel, and Thomas J. Goblick

■ Machine intelligence (MI) techniques have been combined with conventional signal processing and image processing techniques to build a software package that automatically recognizes reentry vehicles from a sequence of radar images. This software package, called ROME (Radar Object Modeling Environment), takes as its input a time sequence of range-Doppler images of a target produced by the Lincoln Laboratory Discrimination System from raw radar-data samples. ROME then processes this image sequence to extract radar features and track them from image to image. From these feature tracks, ROME then constructs a three-dimensional model of the target. The object model derived from the image sequence is then compared to a catalog of models of known objects to find the best match. If no sufficiently close match is found, the observed object is declared unrecognized. The object-model catalog is constructed by adding new models derived from radar data that do not match any model already in the catalog. Thus ROME is "trained" to recognize objects by using real radar data from known objects.

Because of the strong interest in real-time recognition of reentry vehicles, the entire ROME system was recoded for parallel execution on the MX-1 multiprocessor, which was developed in the Machine Intelligence group at Lincoln Laboratory. This machine was designed for MI applications that involve intensive numeric as well as symbolic computation, and that have real-time processing requirements. The final version of the ROME object-recognition system, coded in a combination of parallel Common LISP and C, runs in real time on the 16-node MX-1 multiprocessor.

For many years Lincoln Laboratory has been involved in research pertaining to defense against intercontinental ballistic missiles, including research on the detection, tracking, and recognition of reentry vehicles. During this time several approaches to the problem of rapid and reliable recognition of reentry vehicles have been pursued. In this article we report on an approach that combines conventional signal processing and image processing techniques with machine intelligence (MI) techniques to recognize reentry vehicles at an early stage in their trajectory, prior to their interaction with the atmosphere. In this exoatmospheric region, object trajectories are independent of vehicle shape and mass; hence gross vehicle dynamics cannot be used as discriminants. The approach taken here is based on the appearance and local dynamics of the object as inferred from a sequence of radar images.

The concept of this approach to the early recognition of reentry vehicles is to have a radar detect an unknown object, track it, and take enough data samples to form a sequence of range-Doppler images.

Then, while this image sequence is being processed to recognize the object, the radar can detect and take data on another object. The radar can thus be time shared to handle multiple targets. Of course, real-time operation of the overall detection-and-recognition process is the primary objective, which in this context means that the object-recognition processing should take no longer than the amount of time needed to gather the radar data on the object.

Conventional approaches to object classification typically involve parameter estimation, multidimensional pattern classification, or template matching. These approaches have been successful in many contexts, but are difficult to apply to the recognition of a three-dimensional object from a sequence of noisy images. They also can be computationally intensive and sensitive to noise, artifacts, and minor object alterations.

Human analysts, however, have been successful at extracting estimates of radar scattering-center motion even from noisy data at low imaging rates. This observation motivated us to pursue the approach of building a three-dimensional model of an object as a collection of radar scatterers on a rigid frame that exhibits local dynamics about the gross trajectory. Hence we take an approach that is more like recognition of visual objects, rather than the conventional statistical-decision-theory or pattern-classification approach.

Knowledge-Based Signal Processing

The goal of our approach is to combine the power of conventional methods with the flexible representations and control structures from the field of machine intelligence. A first step in our object classification is to derive primitive features from which a semantic model can be built. The semantic model then can be readily manipulated and interpreted by a human observer as well as by automatic recognition, discrimination, and generalization procedures.

We have built a recognition system and interactive workstation software package that we call the Radar Object Modeling Environment (ROME). Three major conceptual modules are in our current system. The first of these modules is based on an approach known as knowledge-based signal processing [1]. This approach involves a toolbox of signal processing primitives, termed *knowledge sources*, that are designed to select data subsets, extract features, and compare extracted features with the data to produce confidence measures. These primitives include such standard signal processing procedures as clustering algorithms, correlation measures, spectral analysis, and parametric curve fitting; their choice and design are based on procedures used in the manual analysis and identification of these objects.

The second system module is the semantic model-building and matching scheme. This component takes the data-derived features and produces a semantic model that is then matched against a catalog of stored semantic models for object identification. The semantic model represents the extracted information as physical components, intercomponent relationships, and properties of the components and their relationships. The extracted model and the stored catalog model are identical in form and thus can readily be compared and contrasted. In addition, the system can generalize several examples of semantic models to form a single semantic model representation. This feature is useful when the system is presented with data of objects for which we do not have *a priori* information.

The final system module is the control mechanism based on the *blackboard structure* developed in the field of MI [2]. The blackboard structure consists of a global memory (the *blackboard*) in which information is posted as it accumulates. Knowledge sources that embody specific information-processing algorithms check the state of the blackboard to see if their expertise can be applied to move the processing toward its final result. A knowledge source extracts information currently on the blackboard, does its own processing, and posts its results back on the blackboard. The system control continues in this manner until no knowledge sources remain to be applied. A scheduler controls the triggering of knowledge sources in cases when more than one is ready to perform some action. The blackboard design is inherently modular and opportunistic, allowing both data-driven and model-driven processing.

The ROME user interface displays the recognition and blackboard control operations as they proceed.

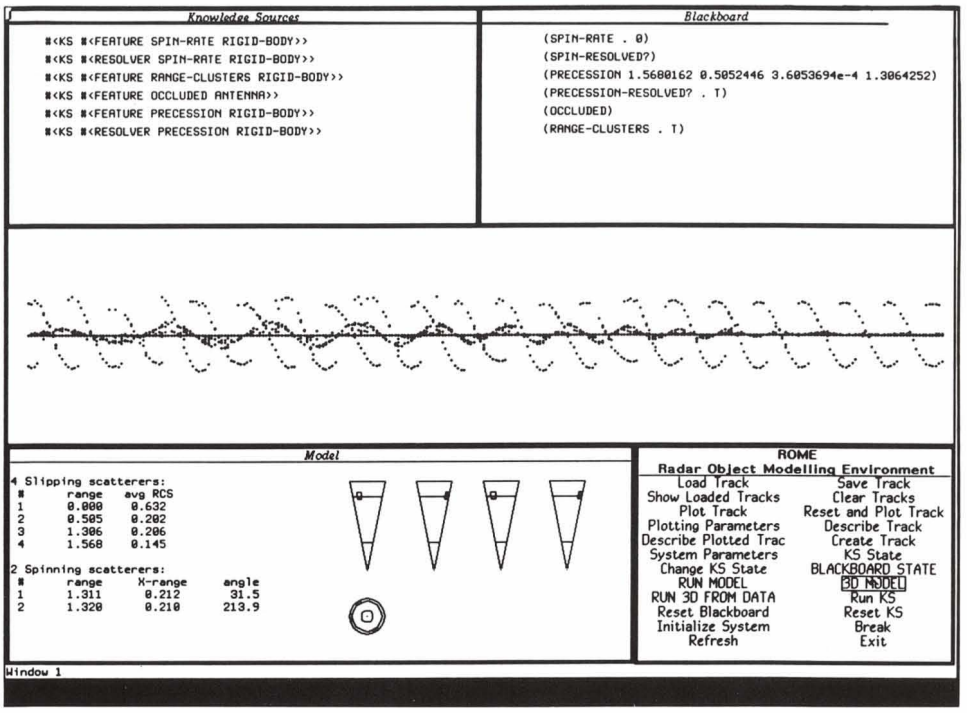


FIGURE 1. Presentation of information in a screen image taken from a Symbolics workstation running ROME. The input data, in a Doppler-versus-time format, is shown in the middle panel. Directly below it are parameters of the derived model together with a cartoon outline of the essential features of the vehicle. Monitor panels at the top left and right show, respectively, the state of active knowledge sources and of the blackboard. At the bottom right is the mouse-controlled operator's panel.

In addition to its recognition capabilities, ROME can act as a workstation for manual analysis of target characteristics. The menu-driven workstation provides a useful interactive analysis tool, and it allows users to learn from the manual operation how to build better deductive procedures. Figure 1, a typical presentation from the Symbolics workstation screen, shows the data track in the middle panel, the operator's control at lower right, the knowledge source at upper left, the blackboard status at upper right, and a representation of the derived model—obtained without operator intervention—at the bottom.

In this article we first discuss the basics of range-Doppler imaging that form the physical model for our processing. We then describe major ROME system modules in more detail, including motivations for our approach, and provide some performance evaluation results. Finally, we present a summary and directions for future work.

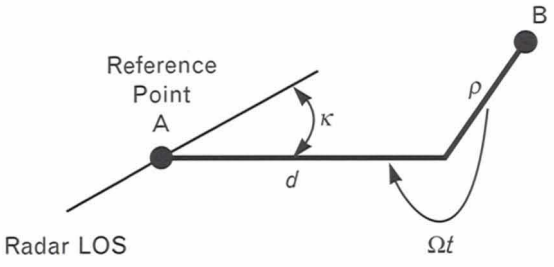


FIGURE 2. Simplified target model, showing two scattering centers. The parameters of importance are the relative axial difference d, the radial distance ρ, the aspect angle κ, and the angular rotation rate Ω.

Radar Range-Doppler Imaging

The data used in this analysis are taken from a range-Doppler imaging radar, which produces a sequence of two-dimensional portraits of the target being observed. Figure 2 shows two scattering centers (i.e., points of reflection) on a generic target model. As-

sume that the radar tracker has removed the gross target motion, and that we measure the relative range of scatterer B with respect to scatterer A. Assume further that the scatterers are part of a rigid body spinning around its axis with an angular rate of Ω radians/sec, with B at axial distance d and radial distance ρ with respect to A, and with aspect angle κ between the spin axis and the radar line of sight (LOS). The range increment r from A to B along the radar LOS is given by

$$r(t) = d\cos\kappa + \rho\sin\kappa\sin(\Omega t), \tag{1}$$

which induces a relative carrier phase shift given by

$$\psi(t) = 2\pi\frac{2r(t)}{\lambda},$$

where $2r(t)$ is the round-trip range increment and λ is the carrier wavelength (this simplified discussion assumes that the vehicle velocity is small compared with the speed of light c).

Suppose that a sequence of pulses is collected at a fixed pulse-repetition interval. By observing the time rate of change of $\psi(t)$, for example by performing a Fourier analysis of the (complex-valued) time-sampled and range-sampled data, we obtain

$$\frac{d\psi(t)}{dt} = 2\pi\frac{2}{\lambda}\frac{dr(t)}{dt},$$

which leads to

$$\begin{aligned} f_D(t) &= \frac{1}{2\pi}\frac{d\psi(t)}{dt} \\ &= \frac{2\rho\Omega}{\lambda}\sin\kappa\cos(\Omega t). \end{aligned} \tag{2}$$

By combining Equations 1 and 2 we see that the locus of scatterer B forms an elliptical path in the range-Doppler plane, as shown in Figure 3. Because the Doppler frequency f_D is directly proportional to the distance ρ, the Doppler dimension is commonly called the cross-range axis. We emphasize that, in this interpretation, the Doppler frequency is measured as a phase progression across a set of pulses within a single range bin, with a target-tracking algorithm that holds the range reference at a fixed position on the body.

For a signal bandwidth of W Hz and an imaging interval of T sec, with corresponding time and frequency resolutions $\Delta t = 1/W$ and $\Delta f_D = 1/T$, the axial resolution Δd and radial resolution $\Delta\rho$, given by

$$\Delta d = \frac{c\Delta t}{2\cos\kappa} = \frac{c}{2W\cos\kappa}$$

and

$$\Delta\rho = \frac{\lambda\Delta f_D}{2\Omega\sin\kappa} = \frac{\lambda}{2\Omega T\sin\kappa},$$

are independent of the target range. We assume that the imaging interval T is smaller than the time taken by the imaged point to move from one range-Doppler cell to another, and that the scatterer motion is approximately linear over this interval. (For more details and a relaxation of this requirement, see Reference 3.) This presentation of data differs from the optical case; rather than the conventional angle-angle scan, we illuminate the object from the radar direction and project the reflection intensity onto a plane parallel to the radar LOS. Note, too, that the assumed motion is necessary to form the image. With $\Omega = 0$, no cross-range information is obtained.

We can use this analysis to model a complex rigid body as a collection of scatterers whose projections in the range-Doppler plane traverse elliptical paths, per-

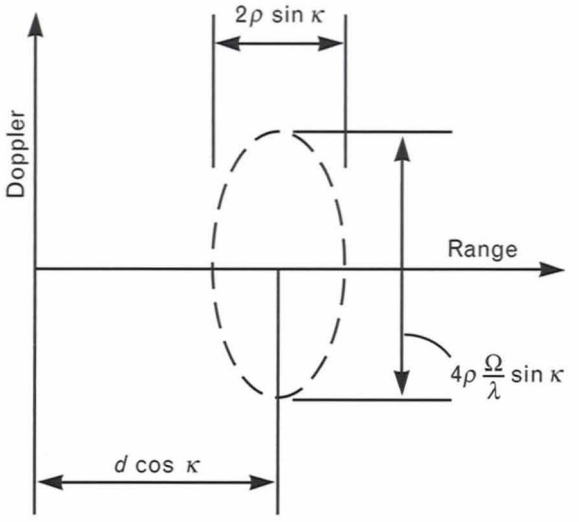


FIGURE 3. Range-Doppler locus of rotating scatterer. The axial dimension is scaled by $\cos\kappa$ and the radial dimension is scaled by $\sin\kappa$.

haps with occlusion as individual scatterers are masked from radar illumination by another part of the object. By looking at successive range-Doppler images, we can observe the motion of the rotating scattering centers. This motion can be described graphically as a helical locus of scattering-center locations in range-Doppler-time space, as seen in Figure 4. This confusing collection of points, each of which is associated with an amplitude peak from a complex-valued image, forms the input to our image understanding system.

With data such as those presented in Figure 4, we wish to produce a model in the form of Figure 2 that best explains these data. Our task is complicated by the occurrence of extra points due to noise-induced peaks in the range-Doppler plane, by missing data due to dropped detections, and by the multivalued nature of the data for each image time as scatterers move in and out of shadows in the radar-beam illumination.

Feature Extraction and Model Derivation

A radar sensor observes an object as a number of physical features that reradiate incident radar energy. Some features, such as attachment points, rotate with the body and can be used to extract spin information. Other features, such as the nose, aft end of the cone, and joints (or *joins*) between body parts, are known as *slipping scatterers* because they appear to slip around the body as it turns, while always maintaining the same apparent position with respect to the radar. These target features are evidenced by a set of reflections of measured amplitude, or radar cross section (RCS), in (time, range, Doppler) space. By properly associating a subset of these reflections with a physical feature on the object, we can parametrically model the reflections and deduce the physical position and reflection attributes of the feature on the body. The collection of these features, together with a description of the object dynamics, then becomes the object

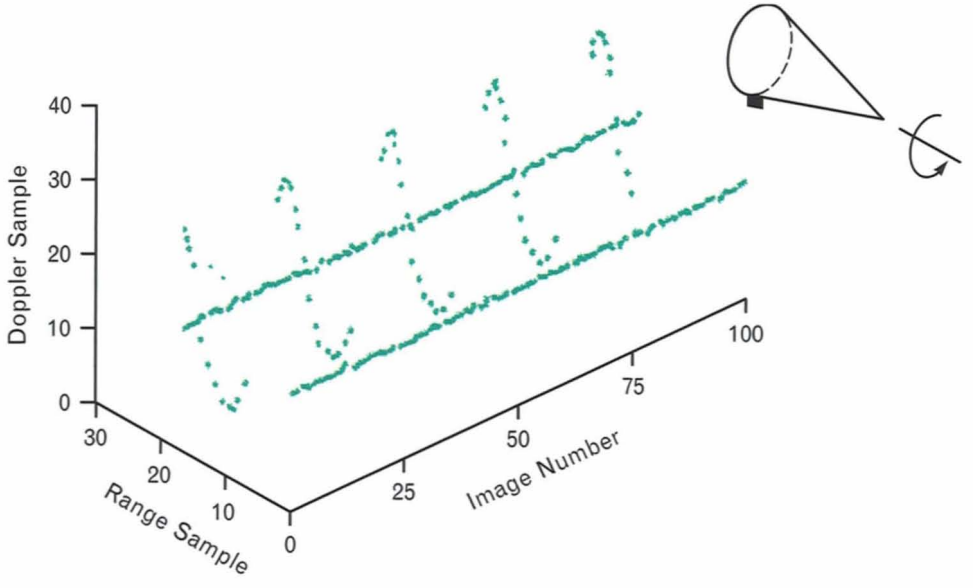


FIGURE 4. Range-Doppler-time loci of multiple scatterers. Note the occlusion of the rotating scatterer and the intertwining of the scatterer tracks.

model. Because the reflections intertwine and overlap in the observation space, however, it is difficult both manually and automatically to associate reflection subsets properly with the corresponding physical body features.

Range Clustering

A first step in the association of reflection subsets within ROME is the clustering of range returns in an attempt to distinguish those which arise from different parts of the object. Range smoothing of the returns is carried out by considering each observed image feature as a point; the set of returns is convolved with an appropriate range kernel to yield an RCS-versus-range portrait such as the one shown in Figure 5. This clustering, which is done in the range dimension, readily distinguishes returns from the nose, mid-body joins, and aft end. A simple threshold comparison then suffices to separate the returns. The nose and join returns can be isolated and treated as slipping scatterers, with the spinning scatterers on the aft body to be modeled as described below.

Spinning-Scatterer Extraction

Figure 6 shows a time-Doppler portrait of the scatterers from the aft end of a typical object. All these scatterers were found in the aft range cluster of Figure 5, and the reflections of several physical features are superimposed. The task of the spinning-scatterer knowledge source is to resolve these features and to deduce the object motion. Several techniques are employed in this effort, including the direct modeling of time-Doppler sinusoids, Hough-transform extraction of sinusoidal segments, and clustering in time-Doppler space. Once the reflections from individual physical scatterers are correctly associated, the parameterization of the scatterers in terms of their body coordinates and collective spin rate is straightforward.

Doppler-Time Sinusoidal Fit

Because our modeling of the scattering centers follows from a physical understanding of body dynamics and the radar-imaging process, we can begin with the sinusoidal model of scatterer motion in the time-Doppler plane. Treating the scatterer's locus, or *track*, as a sampled sinusoid, we choose three points near a peak to estimate frequency (i.e., body roll rate) by using a one-parameter linear predictive coding (LPC) fit. Specifically, we choose the parameter a to satisfy the homogeneous difference equation

$$y_k - ay_{k-1} + y_{k-2} = 0,$$

where y_k, y_{k-1}, and y_{k-2} are successive Doppler values, $a = 2\cos(2\pi/T)$, and T is the scatterer roll period measured in time samples. The track is then extrapolated by using a sinusoidal model to locate successive

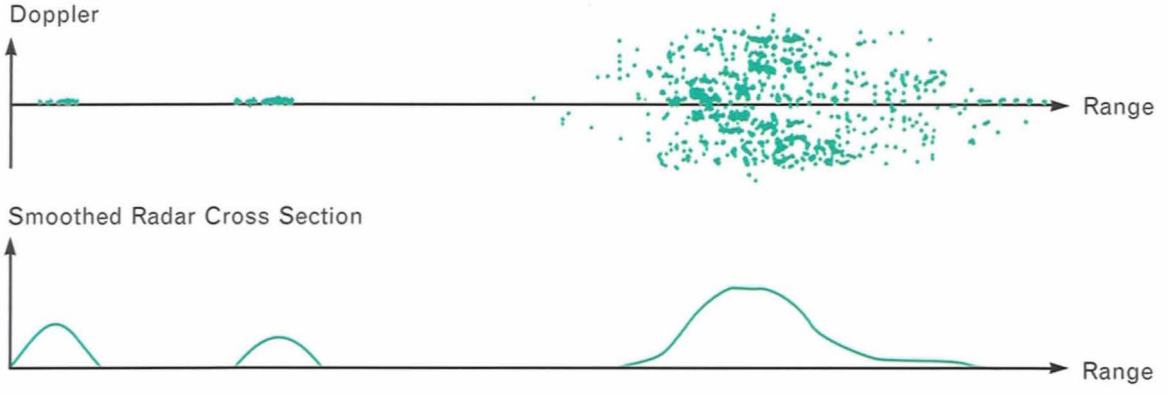


FIGURE 5. Typical range profile: (a) The top portrait shows a projection of the range-Doppler-time points onto the range-Doppler plane. (b) The bottom portrait shows the result of smoothing these data to obtain an estimate of the radar cross-section distribution versus range.

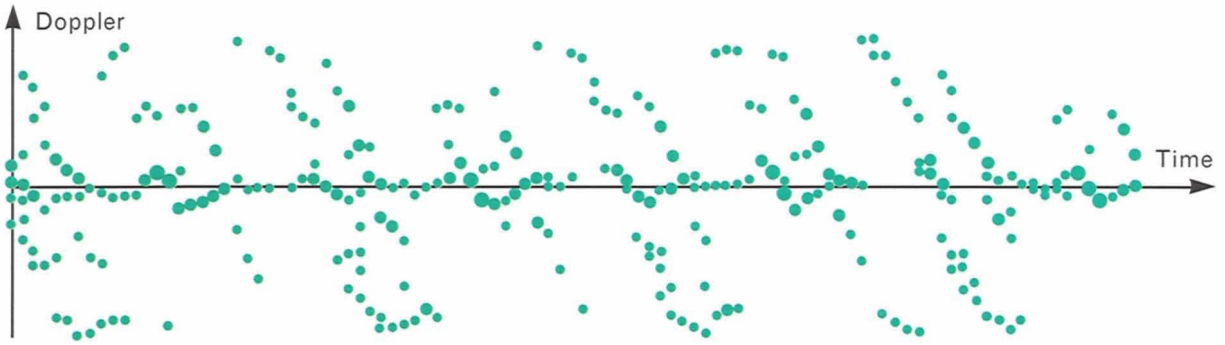


FIGURE 6. Time-Doppler view of the aft scatterer cluster. The individual loci of the five scatterers must be identified and parameterized to model this portion of the object.

points, as illustrated in Figure 7. As each new point is added to the track, the LPC fit is repeated to refine the frequency estimate iteratively by defining the equation

$$y_k - ay_{k-1} + y_{k-2} = \varepsilon_k ,$$

and then minimizing

$$\sum_k \varepsilon_k^2$$

with respect to a to obtain

$$a = \frac{\sum y_k y_{k-1} + \sum y_{k-2} y_{k-1}}{\sum y_{k-1}^2} .$$

The preliminary track extrapolation is terminated when we reach the break caused by occlusion of the spinning scatterer. With a roll-rate estimate based on all points of the track segment, we now correlate the sequence $\{y_k\}$ with a sine and cosine at that frequency to estimate the amplitude and phase of the best-fit sinusoid. (If the track does not exhibit occlusion, it is taken as evidence of a slipping scatterer exhibiting precession; the treatment of this slipping scatterer is described in a later section.)

Moving one estimated period in time along the data, we now locate points that are hypothesized to belong to the same feature track. The LPC fit is repeated independently of the first estimate, and the two sinusoid parameter estimates are compared. If the parameters are sufficiently close, the frequency estimate is refined by using the track zero crossings, and the fit of amplitude and phase is revised by using the combined data from both segments. This procedure

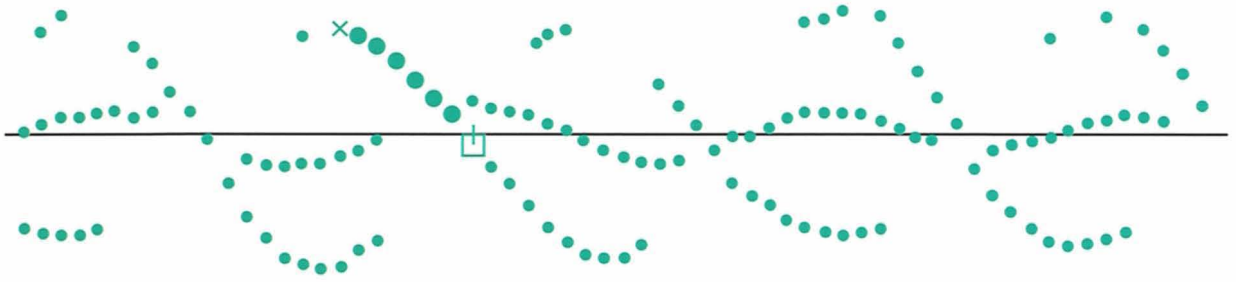


FIGURE 7. Extraction of sinusoidal track. This figure shows a sinusoidal model being fit to the Doppler-versus-time locus of one scatterer. The sinusoid is being extended bidirectionally, with the cross and the square box indicating the next points to be tested for inclusion with the other data for this scatterer.

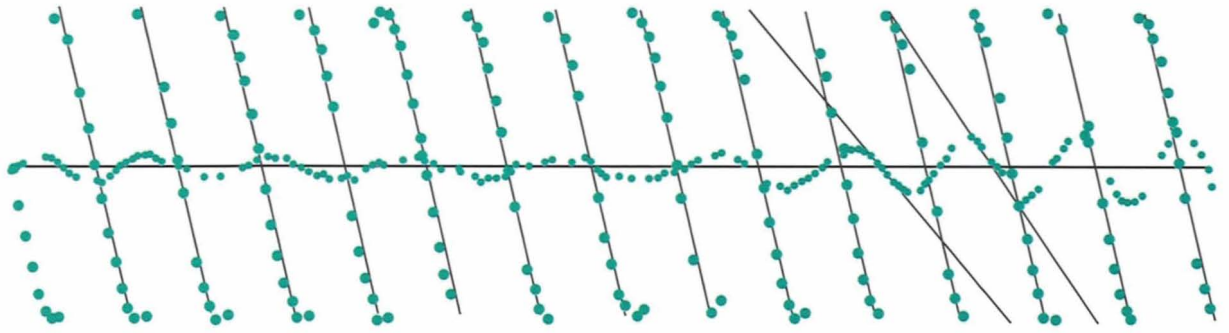


FIGURE 8. Hough transform extraction of scatterer tracks. Several fits are shown for the central portion of the scatterers' loci. Note the anomalous fits for the third scatterer.

is then repeated out to the time limits of the data segment.

This process of extracting the parameters of a single physical body feature is termed *bottom up,* or *data driven;* given the data points, we estimate parameters according to some predefined fit criterion. We now take a top-down viewpoint. Expecting a sinusoidal pattern, we compare the parametric fit with all data points, associating those points which can be explained with the given model component and removing them from the data set.

The remaining points are now left to be associated with additional object features, and the process begins anew with the reduced data set. We repeat the search procedure, successively extracting tracks and interpreting them as spinning scatterers on the model. This stage of processing ends when no further tracks can be extracted. The remaining data points are assumed to be anomalies (poor fits or noise) or the tracks of slipping scatterers, which do not exhibit periodic occlusion.

Hough-Transform Application

The difference-equation approach described above works well in many cases, but can lose track in the presence of precession, data dropouts, random displacement of time-Doppler features, and similar anomalies. This problem is especially troublesome at the beginning of the object-recognition process, when relatively few points are used to extrapolate the track.

An alternative is to take a more global search for sinusoidal fits and use a standard Hough transform to identify the roughly parallel middle portion of successive sinusoids. An interesting problem arises in the interpretation of peaks in (slope, intercept) space; although we are interested in points clustered along regularly spaced lines with negative slope, which the eye readily extracts from the data of Figure 6, a simple peak detection in the Hough (slope, intercept) plane leads to the detection of many lines present in the data but inconsistent with the physical model [4].

Figure 8 shows a typical result, with a number of correct fits as well as several anomalies from the precession of slipping scatterers. A consistency check of slope is used to filter good fits from poor ones, and the period is initially estimated as a factor of approximately three times the time span over which the mid-sinusoid fit is close. Alternatively, an LPC-period estimate can be formed from the point cluster for each line. The number of spinning scatterers is then extracted by an analysis of periodicities in the zero crossings for a given spin-period estimate, which allows for missed line detections and for anomalous detections representing false lines, as seen in Figure 8. A consistency check of zero crossings is then used to refine the period estimates, and finally a fit of model to data is used to resolve ambiguities. The primary test in this portion of the modeling is for consistency among the spin periods observed, according to the implicit rigid-body assumption.

As an alternative to the standard Hough approach, we could employ a generalized Hough transform by using sinusoids parameterized by amplitude, period, and phase to match the entire pattern for each track

of the scatterer. The search space in this case, however, is considerably larger, more processing time is required to form the estimated scatterer position, and this approach is more sensitive to body precession.

Time-Doppler Clustering

A third approach is warranted in cases when the Doppler coordinates of the data points are too noisy to link with the LPC approach, and the presence of multiple scatterers leads to ambiguous detections with the Hough-transform line extraction. The time-Doppler clustering algorithm begins with the evaluation of a track autocorrelation to detect periodic structure in the data. Let the kth image feature have coordinates (t_k, D_k) and RCS equal to σ_k. Now define the autocorrelation measure $R(\tau)$ as

$$R(\tau) = \sum_k \sum_{k'} \sigma_k \sigma_{k'} W_1\left(t_k - t_{k'} + \tau, D_k - D_{k'}\right), \tag{3}$$

where the kernel function $W_1(\delta_t, \delta_D)$ is nonzero for only a limited extent in time and Doppler. The effect is to test the association of points on the given track with those on a τ-shifted version of itself. Relative peaks in the function $R(\tau)$ indicate time shifts at which the track is self-similar, suggesting potential spin-period hypotheses. These peak spacings are readily found by examining the amplitude spectrum of $R(\tau)$.

Figure 9(a) shows a typical track for which the time-Doppler clustering algorithm is appropriate, because here the scatterers do not persist long enough to apply the LPC or Hough approaches. Figure 9(b) shows the corresponding autocorrelation $R(\tau)$ on the same time scale. The recurring peaks every $T/2$ seconds are evident in the amplitude spectrum of Figure 9(c). We make allowance for potential symmetry in the physical location of scatterers on the body, which induces a submultiple of the spin period in the autocorrelation-function peak spacing. If the spin-period estimate is ambiguous, alternative values may have to be carried through succeeding processing stages before a fit estimate can resolve the uncertainty.

The track data are then wound up on the selected spin period T, with track points at time $t + nT$ overlaid on those at time t, as shown in Figure 9(d). A spreading function W_2 is used to smooth the points in time and Doppler, yielding the function

$$A(t, D) = \sum_k \sigma_k W_2\left(t - t_k, D - D_k\right), \quad 0 \le t < T, \tag{4}$$

where σ_k and (t_k, D_k) are the RCS and (time, Doppler) coordinates of the kth track point, and the sum is taken over all points on the wound-up track. The function $A(t, D)$ is evaluated on a fine grid in the time-Doppler plane, local peaks are identified, and (t, D) points are collected along ridges emanating from each peak. A sinusoid of period T is fit to the points forming each ridge. This set of sinusoids is now compared with the original track data, and individual sinusoids are selected in the order of best fit to the data until a lower threshold is reached in fit quality. Estimates of sinusoid amplitude and phase are refined through comparison with close data points. At this stage, ambiguities in spin rate are resolved to obtain the appropriate multiple of the apparent spin period and the correct number of physical scatterers.

Resolution of Spinning-Scatterer Extraction

The previous three sections outline three independent approaches for extracting a spinning-scatterer description from the data signature. The choice of algorithm could be decided on an *a priori* basis, depending on the data quality, object class, or other criteria. Alternatively, the three algorithms could be run in parallel on the same data, and an *a posteriori* selection made on the basis of fit quality between model and data. The spin-fit resolver could then be viewed as a fourth knowledge source. The results of the various spinning-scatterer extraction algorithms are posted to subareas of the system blackboard, and the resolver combines these pieces of evidence to report a single evaluated result to the spinning-scatterer domain of the blackboard. The flexibility of this modular structure has expedited the construction of our system, with algorithmic implementations developed by different analysts and merged at well-defined interfaces.

Slipping-Scatterer Extraction

Following the modeling of spinning scatterers, we turn to the parameterization of slipping scatterers,

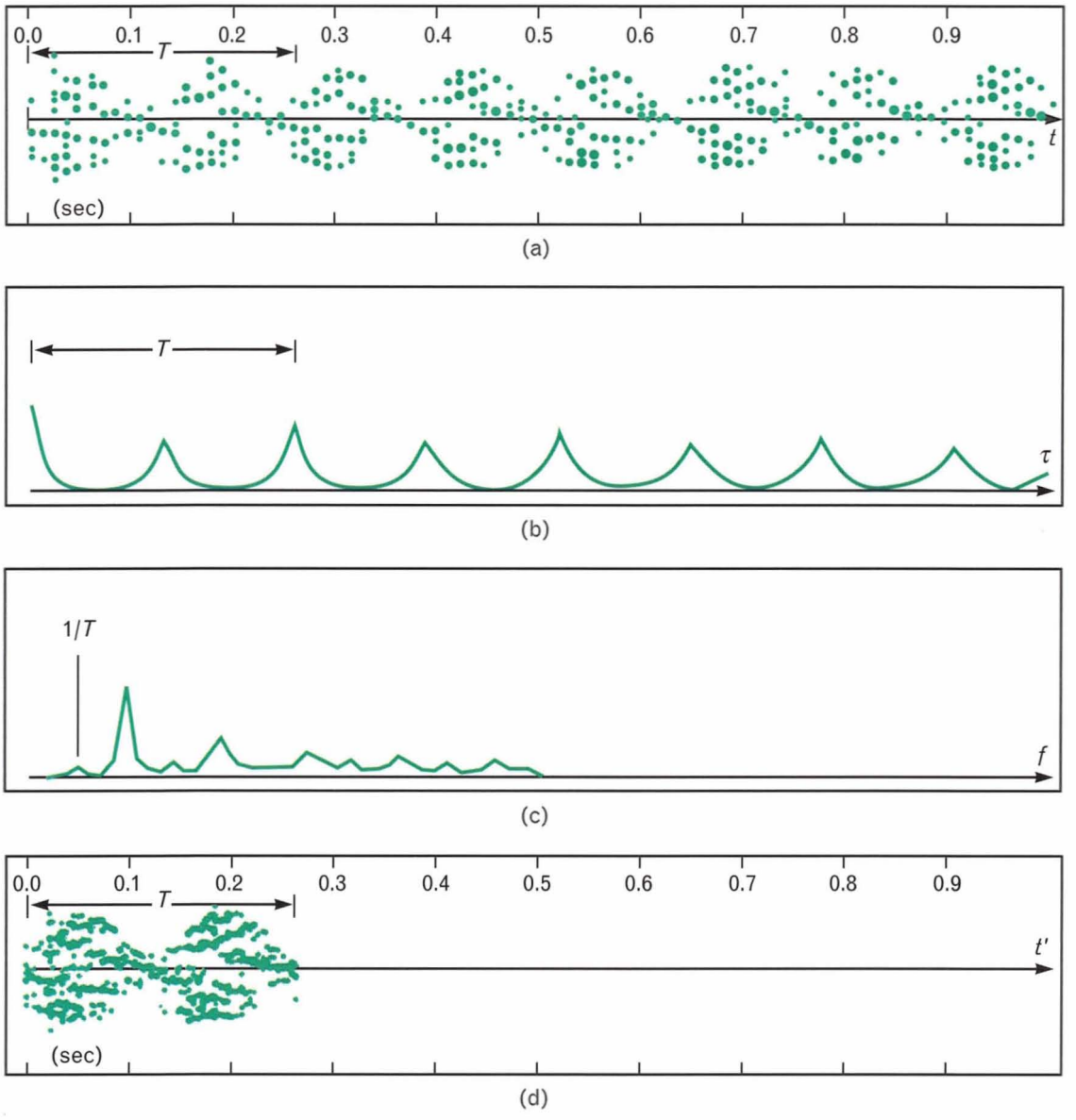


FIGURE 9. (a) Time-Doppler portrait of the aft scatterer cluster. Note the difference in appearance for this class of scatterers from those shown in Figure 6. (b) Generalized correlation for the data in part *a*. This function, evaluated according to Equation 3, shows peaks at intervals for which the data in part *a* are similar; in this case these peaks occur at submultiples of the roll period T. (c) Discrete Fourier transform of track autocorrelation. This spectral portrait of the data in part *b* shows a strong peak at twice the roll frequency. Because the body is symmetric, there is minimal energy at the true roll frequency, which must be deduced from sinusoidal fits to the original data. (d) Wound-up track, with $t' = t \bmod T$. This figure shows the superposition of several periods from the data in part *a*; the scatterer structure will be extracted from these data.

which are annular reflecting structures such as the join between body parts or reflections from the aft end of the body cone. These scattering-center tracks are generally characterized by significantly lower Doppler excursions, unoccluded visibility, relatively constant RCS versus roll angle, and motion characterized as precession. The tracks of slipping scatterers are found from the initial range clustering and from the

remnants of the slipping-scatterer extraction. Because precession is generally variable in amplitude and frequency, especially in the presence of forces induced by atmospheric interactions, a time-varying model is used to describe it. By letting ζ represent the precession amplitude, θ the precession angle, d the axial distance from the slipping scatterer to the reference point, and κ the aspect angle as before, we have a Doppler signature of approximately

$$f_D(t) = \frac{2\zeta(t)\,\dot{\theta}(t)\,d\,\sin\kappa}{\lambda}\cos\theta(t).$$

Here, ζ and θ are smooth functions of time (we chose multiple pairs of low-order polynomials on disjoint time intervals).

The digital signal processing operations employed include sign extraction from a weighted sum of Doppler values for each time, median filtering to remove dropouts, zero-crossing detection, sinusoidal fit over consecutive estimated half-periods, and a least-mean-square polynomial fit of the magnitude and net angle versus time for successive time segments. In the design of the fit algorithm for this time-varying phenomenon, we look for a balance between the closeness of the match and the length of the time segment over which it is valid. The results of the precession-estimation algorithm are reported to the precession-estimates domain of the blackboard. As with spin extraction, a precession-resolver knowledge source examines the results for ambiguities or conflicts, and posts final results to the resolved motion domain.

Model Generation, Comparison, and Catalog Generation

The choices of model representation and model-matching strategy are tightly coupled and depend on the objects being modeled, the nature of the data, the types of features extractable from the data, and the general approach to the problem (e.g., data driven versus model driven). The characteristics of range-Doppler images of simple rigid objects with a tractable motion solution provide several specifications for a model representation. First, we can image the entire object through several rotations because its spin rate is considerably lower than the imaging rate and the observation time can be on the order of several periods of rotation. Thus a three-dimensional description of the located scatterers can be derived. Second, the distinguishing factor between different imaged objects is the general body structure and motion solution, as opposed to more detailed numerical specifications. Therefore, a suitable object model should symbolically represent the structural components, their interrelationships, a motion description, and numerical quantifiers. Finally, while there is a set of known objects for which we would like to build and store models to compare with imaged objects, we also need to acquire new models from the data as new, previously uncataloged objects are presented to the system. Thus the ability to derive a workable model directly from the data is important. Furthermore, this data-derived model should be identical in form to the stored catalog models to facilitate building and generalizing new catalog entries directly from the data.

Model Representation

A representation known as a *semantic network* was chosen to satisfy the above criteria. This representation is a network of nodes denoting physical features of the object, and links relating pairs of nodes [5, 6]. In addition, descriptive property nodes can point to and quantify either nodes or links. This structure is a common model representation for structured objects or scenes from imagery [7, 8]. We can think of this structure as a way of encompassing the information that one analyst passes verbally to another in describing the data-derived features of an object of interest. These features then lead to a three-dimensional object description as a collection of parts, positions, and parameters. Note that models can be built from sparse features in cases in which not all features are detected.

As a simple example, a description of a hammer might consist of a list of the parts *handle* and *head*, with the head further represented as *claw* and *face*, and with *length* and *weight* attributes assigned to the handle and head. Figure 10 shows the corresponding semantic net. This descriptive framework allows us to distinguish among different hammer types (claw, sledge, ball peen) on the basis of their parts (claw, face, ball) and attributes (weight, handle length).

Within ROME, a typical semantic network representation of a reentry vehicle is as shown in Figure 11.

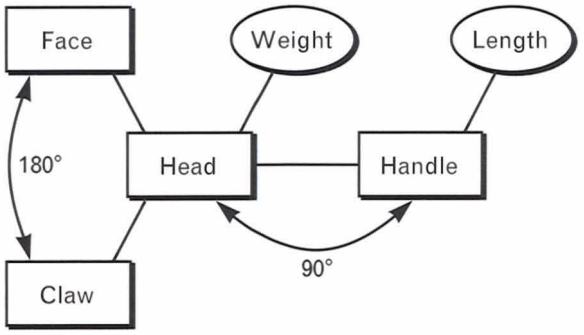


FIGURE 10. Semantic network model for a simple object, which in this example is a claw hammer. Nodes in the network correspond to parts of the hammer, and links serve to relate and characterize the parts.

This network represents the reentry vehicle in terms of its scatterers, the geometric relationships between them, associated quantifiers, and the object motion. Nodes in the network correspond to point scatterers, ring scatterers, and the entire rigid body as a unit. The relational links between the nodes represent relative differences in range, cross-range, or angle. Quantifiers pointing to the nodes contain more detailed descriptive information such as scatterer type, radar cross section, or polarization attributes. The spin rate is a property attached to the entire body node, because all scatterers extending from the rigid-body node must rotate together. The data-derived features are represented in the identical format used in the catalog of objects available for comparison.

Model Comparison

Once a data-derived semantic model has been built, it can be matched against a catalog of semantic models. The catalog models are established directly from known physical models or from previous data-derived models. The matching strategy consists of finding the best correspondence between parts on the derived model and parts on the catalog model. This correspondence is evaluated and compared to the correspondences from the other catalog models. Because of the symmetry of the objects and the occasional discrepancy in number of parts between the derived model and the catalog model, several potential correspondences between point and ring scatterers usually exist. These correspondences are based on the general structure rather than on quantifiers. The correspondences are constrained because range and angular-position information limits the possible permutations for these correspondences.

Once a set of realizable correspondences is established between the scatterers from the derived model and the scatterers from the catalog model, these correspondences must be evaluated and scored to assess the best fit. The basis for evaluation is finding the correspondence that minimizes the distances in property space. For example, assume that in a certain correspondence a ring from the derived model is associated with a ring from the catalog model. The properties attached to both rings (such as range and RCS) are plotted in their property space, and Euclidean distances are computed between derived and cataloged properties. This computation is done for all parts in the correspondence, and for all correspondences. The correspondence with the greatest number of minimum distances across its parts is marked as best. The scoring metric for each correspondence is the number of minimizations in property space less the number of missing parts between the derived and catalog models. Some cost must be associated with a derived model having either too few or too many parts compared to the catalog model. The resulting normalized scores are compared across the catalog, and the maximum score is chosen as the best match from the catalog to the derived model. Note that the spin rate and the apparent number of point scatterers are mutually dependent pieces of evidence; this fact can be used to refine an object model in a model-driven processing pass.

Catalog Generation

Because of the variety of missions of the radars from which we obtain data, images of unknown objects are often presented to the system. In these cases, the data correspond to objects for which no *a priori* models exist. An important part of our system is the ability to generate a semantic model from any number of data segments associated with a particular object.

Generalization, which is a subproblem in the more general field of learning [9], is the ability to learn a general model from several specific examples. P.H.

Winston advocates an inductive learning strategy known as learning by example [10]. An external teacher presents the system with examples and counterexamples, and the system induces the general concept description. As new inputs are presented to the system, they are matched against the current models. New inputs can also be used to refine the internal representations continually.

We approach generalization not as a learning system that continually changes its knowledge and performance based on examples, but chiefly as a way to train the system as to how a particular uncataloged object should be represented, based on several examples derived from data. This information is then entered into the current catalog. Once the training is complete, the system performs in its normal mode, treating the generalized model as a catalog entry.

The generalization process is guided by a human operator who selects acceptable data-derived models for a given object. Acceptable models as well as counterexamples are determined by the expertise of the human operator. The resulting models can thus be ambiguous because of the complexity of the data and the inconsistency among experts; these ambiguities are resolved through generalization.

All the data-derived models that were selected result from the initial processing of the ROME system (up to the matching stage) to produce semantic models, as we discussed previously. The first derived model initializes the generalized model. As other examples are presented, the generalization associates new structural components with current structural components in the generalized model. When an association is made (similar to the matching correspondence problem), the structural components increases a number-of-occurrences counter. Each structural component

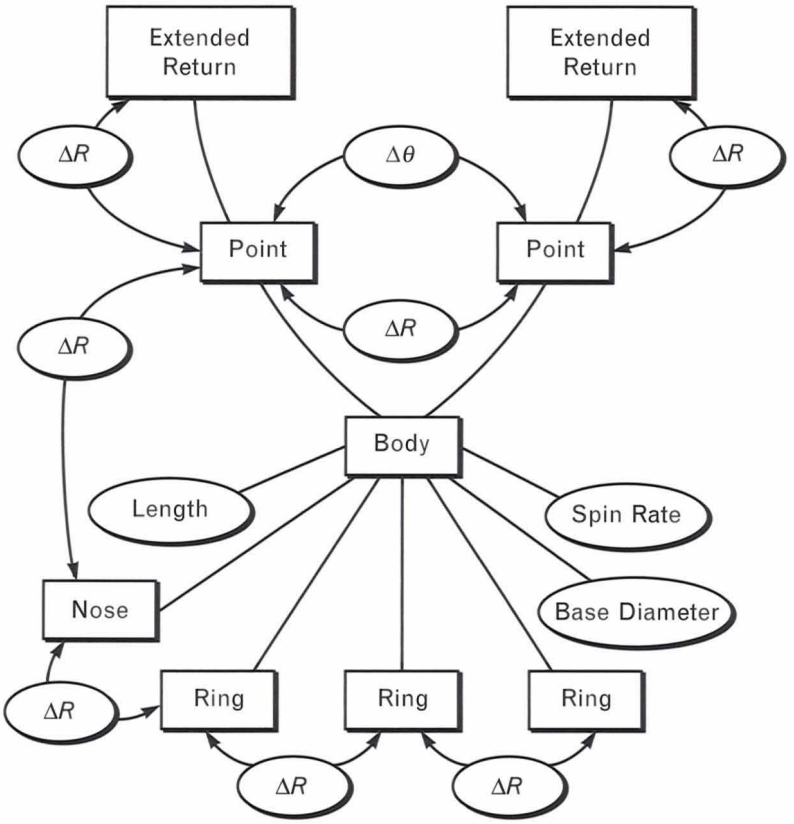


FIGURE 11. Semantic network model for a reentry vehicle. Nodes correspond to the body as a whole and to various scattering centers. Links relate the attributes and relative positions of the scatterers.

Table 1. ROME Classification Performance

	Catalog						
Test Data	RV1	RV2	RV3a	RV3b	RV4*	RV5	RV6
M00 (4)	100%	—	—	—	—	—	—
M01 (9)	—	100%	—	—	—	—	—
M10 (9)	—	—	45%	55%	—	—	—
M02(8)	—	—	—	—	100%	—	—
M03/LDS013 (8)	—	—	—	—	100%	—	—
LDS014 (5)	—	—	—	—	—	100%	—
LDS027 (5)	—	—	—	—	—	20%	80%

* Catalog models are based on a mission different from the one tested.

keeps track of the number of associated occurrences as well as the total number of training instances. Thus each component in the generalized model has a weight attached to it equal to the number of occurrences divided by the number of training sets. This weight, which is between 0 and 1, is used to estimate the importance of that structural component in the matching. Components that are not associated with existing parts in the generalized model are added as additional parts, but their weight is low compared to parts that do find appropriate associations.

Once an example model is generalized for a particular object, this model is entered as the catalog entry for the unknown object, and the system performs its normal operations. As new data come into the recognition system and a semantic model is built, the derived model is compared against the trained examples. The number of examples needed for a particular object depends on the diversity of the models in the entire catalog and the inherent variability in the data and in the data processing. As discussed in the next section, results from the generalization of models have been encouraging. The margin between match scores for correct matches versus incorrect matches is significant.

In the discussion above, we implicitly assumed that the aspect angle κ between the radar LOS and the vehicle spin axis is known, so that absolute dimensions could be compared in the match assessment. If κ is not known, then the look angle between the radar LOS and the vehicle velocity vector can be used as an estimate of κ (this estimate assumes that the spin axis is aligned with the trajectory, which is not necessarily true). Alternatively, κ can be left as a free parameter to be estimated in the search, and chosen—for each association of derived and catalog model—to maximize the range and cross-range match score. The maximum score is taken to indicate the assessed vehicle type, and the corresponding value of κ that was used to maximize the corresponding score is taken as an estimate of aspect angle.

Detection Performance Results

For evaluation of the classification performance of ROME, data from several missions were collected by operational radar systems, processed through the Lexington Discrimination System (LDS) facility [11], and passed to ROME as sets of feature vectors defining the locations of peaks in the respective sequences of range-Doppler images. Five different vehicle types were represented by one mission each; a sixth type was represented by two independent missions. The data from each of the seven missions were partitioned into data segments. ROME independently processed

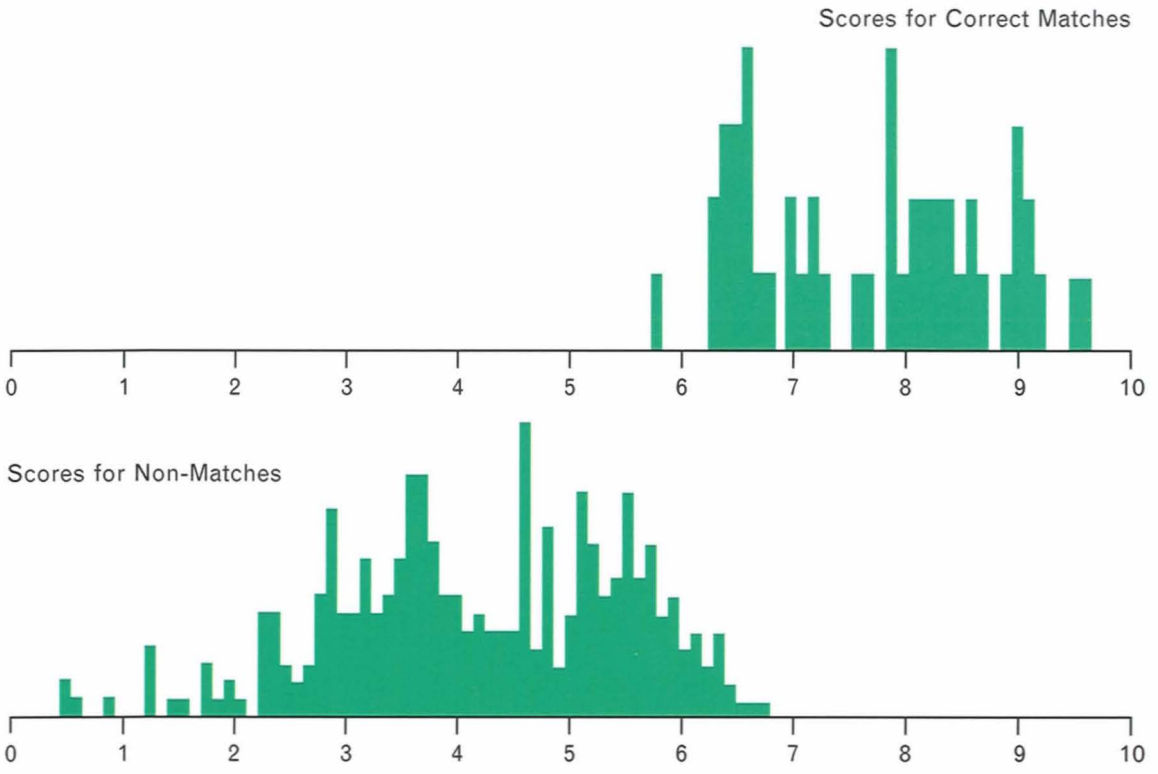


FIGURE 12. Distribution of matching scores for the results shown in Table 1. Note the bimodal nature of the matching score distribution, with the scores for nearly all correct matches lying well above the scores for all incorrect matches. This distribution suggests that the match score can be compared with a threshold for introducing a new model.

each data segment to generate derived semantic models for the corresponding vehicles. In all, there were 48 data segments and 48 corresponding individual models. These models were combined by vehicle type to form respective catalog models for the six vehicle types.

Each data segment was taken in turn and its individual model was tested as follows. The remaining data segments from that mission were used to form a modified catalog model for the vehicle; note that the tested segment was not represented at all in this modified model. The derived model for the data segment under test was then compared with the catalog, which now contained the modified model in place of the original one, and the best match was chosen. Then the next data segment was chosen, a new modified catalog entry was formed, and so on for each of the 48 track segments. This testing methodology corresponds to the standard *leave one out* experiment, where the models against which a data segment is tested do not depend on that data segment.

Table 1 shows the test results. The mission designation is shown in the first column, with the number of data segments shown in parentheses; the top row indicates the cataloged vehicle models. The nine individual derived models from the data segments of mission M10 formed two clusters in feature space, and were assigned two entries in the catalog. The model labeled RV4 was derived from missions M02 and M03/LDS013; in testing the derived models from these missions, we used only derived models from the other mission to generate the catalog entry. Of the 48 data segments, only one was misclassified—one of the LDS027 data segments was declared to be vehicle type RV5 instead of its correct identity of RV6. Figure 12 displays a histogram of the numerical matching scores from the correct matches as compared with those of the incorrect pairings. The minimal overlap

of these distributions suggests that the match score can be used as an absolute measure of classification confidence. We note that the system parameters were not tuned to the vehicles tested; as successive vehicles were added to the database during the development of this code, only the catalog entries were altered.

Of course, accurate assessment of the system's classification performance will require a greater quantity of data, with multiple missions included for all vehicle types. Also, because a primary goal of this system is the discrimination of reentry vehicles from decoys and associated objects, the system must be tested with a wider range of objects. We found these preliminary results encouraging, however, because similar reentry vehicles were correctly classified on the basis of their appearance through the use of real data taken on exoatmospheric flight objects.

Parallel Processing for Real-Time ROME Execution

Now we turn to the second requirement of this project: real-time execution of the ROME algorithms. The basic objective of this project was to assess the applicability of MI techniques to the problem of automatic recognition of reentry vehicle types. The initial emphasis was on developing algorithms that worked well with real input data and actually recognized the different reentry vehicle types. The next phase was to optimize these algorithms to achieve the best possible recognition performance by using a database of real radar data on real objects. Because any practical application of these techniques would require real-time execution, the emphasis then shifted to defining the computational requirements for real-time execution, recoding the ROME algorithms, and demonstrating actual real-time execution on a parallel processor.

ROME Processing Requirements

The execution time of ROME on a uniprocessor LISP workstation (a Symbolics 3645 LISP machine) with a typical data track was measured as 45 seconds of processing per second of radar data. The data had already been preprocessed by the LDS computer system [11] and sent to ROME in the form of feature vectors (i.e., locations of peaks in the range-Doppler images). Thus for achieving real-time execution, a speedup factor on the order of 100 would be needed. This factor allows a margin for variation in processing time due to data-dependent factors such as signal-to-noise ratio, object complexity, imaging rate, and data length. In addition, the catalog size has a strong but well understood effect on the overall processing time, but we do not have to deal with this issue with the small catalog size that ROME uses.

A number of approaches can be used to speed up the ROME execution time. One obvious method is to use a uniprocessor computer that can run Common LISP 100 times faster than a Symbolics LISP machine. Such a computer would require a processor that runs Common LISP with a cycle time considerably less than a nanosecond; unfortunately, no such machine is available. Recoding ROME in some other language, such as C or FORTRAN, would not reduce the required processor speed significantly enough to make the uniprocessor approach feasible or appealing.

The alternative approach is parallel processing, which offers several interesting options. The obvious parallel implementation to achieve a speedup of a factor of 100 is to use a machine with 100 processors, or *nodes*, each of which could run Common LISP programs as fast as a LISP workstation. Each node could run the uniprocessor ROME code to process radar data corresponding to a different interval of time, say T seconds, and produce its results in 50 to 100 times T seconds. Each node would then be able to process data fast enough to handle one out of every 100 data intervals, and the entire parallel system could then keep up with real time. This solution would be unwieldy, however, and the latency of the results would be unacceptably long (for $T = 2$ sec, each node would take 100 to 200 seconds to produce results). This approach would therefore not be useful in a system requiring a fast response.

Let us now consider more practical parallel implementations of ROME involving fewer than 100 processors. In the late 1980s, low-cost processors that could run Common LISP as fast as a LISP machine were unavailable, so achieving real-time execution with the ROME system was a considerable challenge. How can we currently realize a speedup of a factor of 100 relative to a LISP machine with processors that can-

not run Common LISP as fast as a LISP machine? The answer, in fact, is that it cannot be done.

To achieve real-time ROME execution, we must focus in more closely and consider the resources that are specifically required to do the ROME computations, rather than just running Common LISP. Compared to other types of processors such as digital signal processor (DSP) computers, LISP machines are relatively slow at performing numeric computations such as signal processing operations. If ROME involved only signal processing computations, we could easily put together a multiprocessor computer with a modest number of DSP chips to obtain a large speedup in execution relative to a uniprocessor LISP machine. ROME, however, involves a mixture of symbolic computations as well as signal processing.

One practical approach is to build a multiprocessor with a modest number of nodes, each of which can run Common LISP as fast as a LISP machine, and use a special coprocessor to perform signal processing computations much faster than a LISP machine. Thus the key issue to focus on is the amount of numeric computation versus the amount of symbolic computation done in ROME. The more numeric computation in ROME, the more a fast numeric coprocessor could contribute to the overall speedup. Figure 13 shows the major processing steps in ROME. The numbers in the figure indicate how many seconds of processing each module takes per second of radar data on a uniprocessor LISP machine. These numbers indicate that the bulk of the processing time is concentrated in the modules that involve numeric computations. Estimates of exactly how much of the total ROME uniprocessor execution time involves numeric computation range from 75% to as much as 90%. This percentage would be much greater if the processing done in LDS were included.

In the box entitled "Uniprocessor Speedup with a Numeric Accelerator" we use a simple model to calculate the potential speedup factor for a program in which the numeric computation in a uniprocessor takes R times as long as the symbolic computation time. The model assumes the use of a special processor that performs numeric computations σ times as fast as the original processor and symbolic computations at the same rate as the original processor.

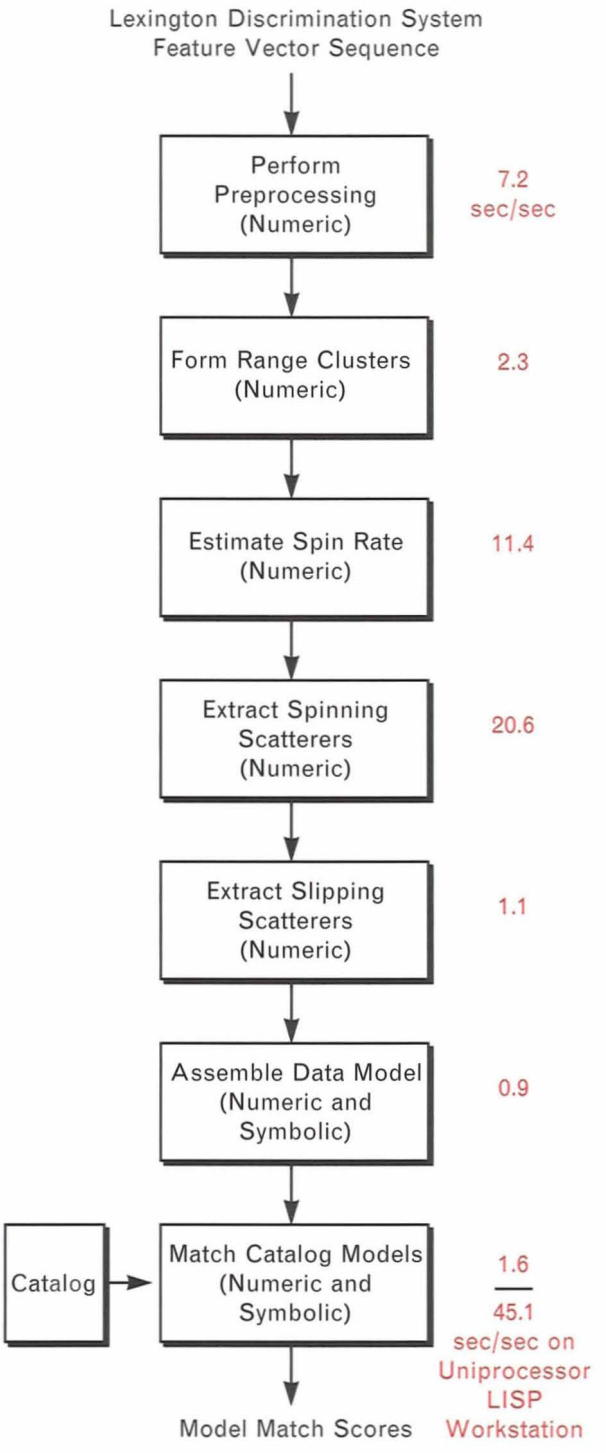


FIGURE 13. Major processing steps in ROME. The numbers reflect execution times in seconds of processing per data second for a uniprocessor LISP workstation.

Figure A indicates that for a mix of times R from 3 to 10, a numeric speedup of greater than 20 achieves

UNIPROCESSOR SPEEDUP WITH A NUMERIC ACCELERATOR

CONSIDER A UNIPROCESSOR computer executing an application program that consists of two different types of operations, for example, symbolic and numeric computations. Suppose the symbolic computations take T_s sec to execute while the numeric computations take T_n sec to execute in each complete execution of the application. Suppose next that another computer is considered that can execute symbolic computations σ_s times faster and numeric computations σ_n times faster than the original computer. The execution time of this application on the second machine would then be

$$T = \frac{T_s}{\sigma_s} + \frac{T_n}{\sigma_n}$$

compared to $T_s + T_n$ on the original machine. We could then define the *speedup* in execution time as

$$S = \frac{T_s + T_n}{T_s/\sigma_s + T_n/\sigma_n} = \frac{1 + R}{1/\sigma_s + R/\sigma_n},$$

where we have defined R as the ratio of numeric to symbolic computation time on the original machine for this application; i.e.,

$$R = \frac{T_n}{T_s}.$$

Figure A shows a plot of the overall speedup S obtained by using the second machine versus the ratio R of numeric-to-symbolic computation in the application. We assumed that the second machine performed numeric computations σ_n times faster than the original machine, and it performed symbolic computations at the same speed as the original machine (i.e., σ_s was equal to one). Note that the speedup realized by doing numeric computations faster depends on the relative amount of numeric computation in the application; that is, larger values of S are obtainable for larger values of R. But note also that accelerating numeric computations has diminishing returns. For R = 10 (i.e., 10 times as much numeric computation as symbolic computation), S is barely improved when σ_n is increased beyond 20. In general, there is little benefit in increasing σ_n much beyond twice the value of R in the application because the speedup factor is already close to its asymptotic value for $\sigma_n \to \infty$. For an application with R ranging somewhere between 4 and 8, the maximum speedup is seen to be limited to a factor of between 5 and 9, and these speedups can be nearly achieved by numeric acceleration values of σ_n between 10 and 20.

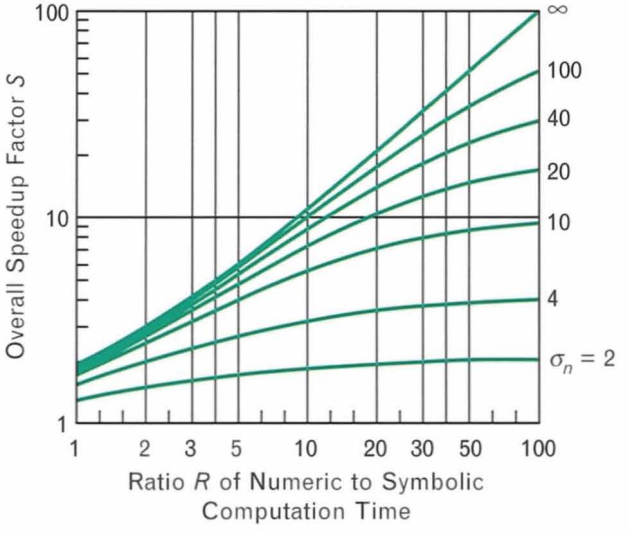


FIGURE A. Overall processing speedup versus ratio of numeric to symbolic processing time. Curves are parameterized by the speedup of numeric computations, with symbolic computation speed held constant.

essentially all the speedup possible with this mix of computation (i.e., effectively an infinite speedup of numeric computation), and the overall program speedup ranges from a factor of 4 to 10.

The goal of speeding up ROME execution time by a factor of 100 was ultimately achieved by a combination of measures. We first conservatively estimated that streamlining the ROME code would speed up execution by a factor of two, because the code was originally written for accurate recognition rather than fast execution. Another factor of five could be obtained from a processor that performs numeric computations 20 times as fast as a LISP machine. That would leave a factor of 10 to be realized by using multiple nodes working in parallel on ROME computational tasks. Cooperative parallel computations require communication between processors; hence, a multiprocessor with more than 10 nodes would be needed to achieve real-time execution of ROME.

The MX-1 Multiprocessor

The ROME system is one example of a large class of MI applications that we term *sensor data understanding*, which involve a mixture of intensive numeric computation with symbolic computation. Moreover, these applications must frequently execute in real time. Other example applications are robotic vision, automatic-target-recognition systems, and machine recognition of spoken language. Real-time execution of sensor-data-understanding applications often presents difficulties for the same reasons already discussed; namely, artificial-intelligence workstations perform numeric computations inefficiently, and parallel computers that run LISP are not common. Thus the Machine Intelligence Group set out to develop a multiprocessor computer specifically to facilitate development of parallel algorithms for high-speed execution of this class of MI applications. The result of this effort is the MX-1 multiprocessor, which is a shared-memory, tightly coupled system with a parallel LISP programming environment that also provides powerful numeric computation capability [12].

The MX-1 multiprocessor consists of 16 processing nodes interconnected with a crossbar network. Each node consists of a Motorola 68020 microprocessor and 68882 math coprocessor chip (running at a clock rate of 16 MHz), 8 Mbytes of random access memory (RAM), and an independent DSP computer designed around a Weitek DSP chip set. The DSP of each node has a peak math computation rate of 20 Mflops (single precision) and 256 Kbytes of data memory. Figure 14 shows a block diagram of the MX-1 system, including its host, which is a Symbolics 3675 LISP machine. The host provides the user interface, the network interface, and the hard-disk mass storage.

The crossbar has one-byte-wide bidirectional data paths that operate at 16 MHz. The crossbar control allows broadcasts to be made from any node to any subset of other nodes, including all other nodes. The crossbar was motivated by the unpredictability of memory-access patterns in LISP systems doing symbolic processing. Each RAM memory module is connected to a processing node so that accesses to local memory are faster than going over the crossbar to another memory module. In the MX-1, a single-word (4 bytes) memory access over the crossbar is approximately seven times longer than a local memory access. The shared-memory software environment can correctly access data anywhere in memory, but using the crossbar for every memory access would significantly increase access time. The user has the option of setting up data structures in specific memory modules to try to maximize locality of memory references for faster execution.

The interface between the LISP machine host and the MX-1 is complicated by the differing word lengths of the Symbolics and 68020 processors. This interface thus employs a 68020 processor similar to one of the MX-1 processing nodes. This processor, which is called the Interface Processing Element, handles all communication between the MX-1 and the host machine (including network functions, the user interface, and the mass storage), and it also can participate fully in the MX-1 computations.

Each node also has a high-speed data-input port with a peak data rate of 16 Mbyte/sec to accommodate rapid entry of sensor data for real-time processing experiments. The concept of operation of the MX-1 in real-time applications is as follows. The sensor data are entered into the data memories of the DSP processors via the high-speed data ports on each

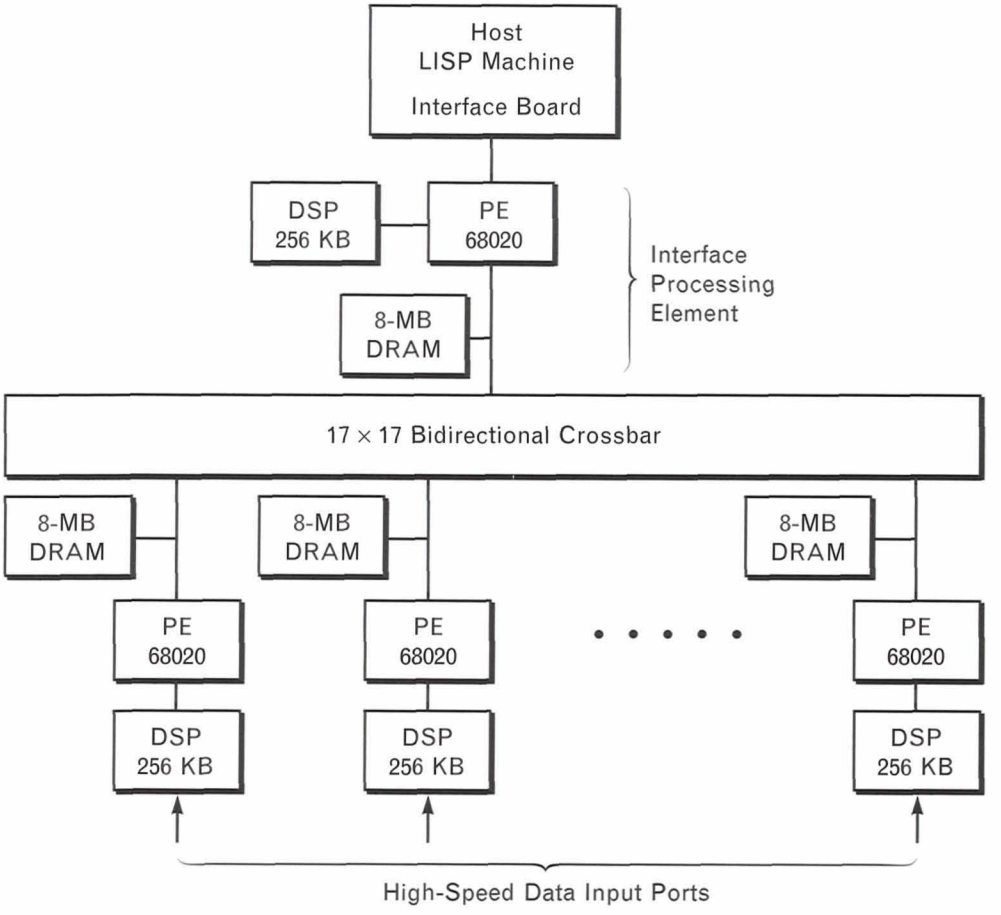


FIGURE 14. MX-1 multiprocessor architecture. Each processing node contains 8 Mbytes of dynamic random access memory (DRAM), a 68020 processing element (PE), and a DSP coprocessor. Communication is handled by a 17 × 17 crossbar. The host is a Symbolics 3675 LISP machine.

node. At this point, front-end signal processing (data-independent numeric operations) is performed by the DSP processors executing C functions. Computation in sensor data understanding systems starts out almost exclusively as signal processing, then becomes more data dependent (with more branching), and then becomes largely symbolic computation. Thus when the computation becomes more symbolic and less efficient to perform on the DSP processors, the computation is performed on the 68020 processors, which can execute both C programs and compiled Common LISP programs. In this way, the symbolic parts of the application code are speeded up by parallel execution on the 68020 processors, while the numeric parts are speeded up by parallel execution and by the faster DSP processors.

Although readers might conclude from this article that the ROME application motivated the MX-1 design, this was actually not the case. The MX-1 design was well under way by the time the focus in the ROME project turned to real-time execution. In fact, because the data input to the ROME processing system consists of feature vectors already computed in the LDS computer, much of the front-end numeric processing, on which MX-1 could achieve a spectacular speedup, has already been done. ROME is thus not an ideal first application with which to demonstrate MX-1 in its best light, because the LDS processing has already greatly reduced the ratio of numeric processing to symbolic processing.

The MX-1 Programming Environment

The programming language of the MX-1 is Common LISP. More specifically, we use the Kyoto Common LISP (KCL) software developed at Kyoto University, Japan, which is in the public domain in the U.S. (including sources). This software was benchmarked by using a number of the Gabriel LISP benchmarks, and it compares favorably with the performance of Symbolics LISP workstations [13]. Of course, Common LISP programs must include numerous declarations to compile efficient run-time code, which partly offsets the advantages of the LISP language. But new code can be prototyped rapidly in the LISP style without declarations by using all of the prototyping and debugging features of LISP while paying a penalty in slower execution time during this phase of program development. When the code has been debugged, declarations can then be added to produce more efficient compiled code for faster execution.

KCL compiles to C code, which, in turn, is compiled to 68020 machine code for execution. Using C as an intermediate language allows easy porting of LISP applications to other processors when necessary. The Machine Intelligence Group has added a number of extensions to KCL for parallel programming. The primary extension is called the *promise*, which is a construct that causes a LISP procedure to be executed on a specified node of the MX-1. The node can be precisely specified (e.g., node seven) or loosely specified as the next one available to run a procedure.

The single address space of the MX-1 is arranged to provide a portion of each local memory for use as *private* memory for the processing element, a portion for use as *shared* memory, and a portion for use as *broadcast* memory. Private memory is used for operating-system code and application code; shared memory can be accessed by all other processing elements. The contents of the broadcast portion of each processing-element memory are identical, which allows us to simulate the access of a single data structure by all processing elements, while still allowing locality of reference for each processor. Broadcast memory exploits the broadcast capability of the crossbar by handling all writes to broadcast space with a single crossbar write operation.

The standard parallel-programming paradigm on the MX-1 is for a user to write a LISP procedure to process data in a data structure such as an array, and to then debug the procedure. The array is then partitioned (divided into parts) and distributed to a set of processing elements (put in their shared memory). The processor can then perform a *promise-funcall* with the procedure, specifying the processing elements that contain the data structures to execute the promise. The box entitled "Parallel Language Constructs" gives a simple example of a parallel program that performs a vector dot product. The actual parallel program is not greatly extended in length relative to the uniprocessor version of the code for the procedure. The *promise-funcall* is a basic construct that allows us to build many different types of parallel programs. The *wait-for-promise* construct provides a basic primitive for synchronization of subprogram execution on different nodes. The *pyramid-pe* construct combines these other two into a mechanism for distributing tasks and combining results.

The underlying C software runs on a distributed operating system called MXOS, which was derived from UNIX 4.3. C programs can be executed on the MX-1, but the parallel-programming tools are all oriented toward parallel KCL. KCL, however, does provide a convenient interface with the C environment that allows us to load a C program into the MX-1, give it a LISP name, and then call it from the KCL environment. In this way, certain numeric-intensive computations can be coded in C and called from a KCL program to execute in the 68882 math coprocessor.

The selection and use of the DSP coprocessors also relies on the convenient C interface of KCL. The Weitek DSP chip set used in the MX-1 comes with a C compiler. Thus C programs can be written and compiled for the DSP coprocessors, and debugged by using the special tools of the DSP programming environment. We developed a linkage to KCL to allow DSP routines to be loaded in the same way that C routines are loaded and linked to the KCL environment so they can be called from a KCL program. The DSP linkage automatically handles the movement of the C routine arguments to the DSP coprocessors, loads the C code, executes the routine, and commu-

PARALLEL LANGUAGE CONSTRUCTS

A NUMBER OF parallel programming primitives are available to the applications programmer who wishes to use Parallel Kyoto Common LISP; these primitives are shown in Figure A. The *promise-funcall* is a processing request that identifies a function to be called, the function arguments, the target processor identification, and a strictness parameter. A high strictness indicates that the function must be run on the indicated processor; a low strictness directs that the function be executed by the next available processor. The promise-funcall is a pending operation that may be assigned to a named variable (e.g., to the variable *promise-object* in this example). Synchronism is established with a *wait-for-promise*, which returns the processing result when it is available, meanwhile suspending further operation.

These two primitives are built into a basic parallel mechanism, the *pyramid-pe*, which defines a set of active processing elements, an operation to be performed in parallel on all processors, another operation that pairwise combines their partial results, and separate strictness measures for the base operation and the combination operation. For execution on N processors, the pyramid-pe orchestrates the issuing of N promise-funcalls and the corresponding N wait-for-promises for the base function, $N/2$ promise-funcalls and wait-for-promises for the first-level combination, $N/4$ promises and wait-for-promises for the second-level combination, and so on to the last promise-funcall and wait-for-promise that yield the final result.

Thus, as shown in Figure B, a vector inner product can be defined as the result of a pyramid-pe that calls the basic vector-multiply operation to evaluate a partial inner product on each of the active processing elements and a vector-sum operation to combine the results of any two vector-multiplies. Each processing element knows the total number of processors to be used, the length of the vectors, and its own identity; from this information, the local initial and final vector index limits are computed and a local sum is evaluated by each of the processing elements. The N local sums are combined in parallel, by pairs, through $\log_2(N)$ stages, to yield the final result.

```
(promise-funcall pe-number strictness #'function arg1 ... argn)

(setq promise-object (promise-funcall...))

(setq result (wait-for-promise promise-object))

(pyramid-pe
      number-of-pes
      #'base-function
      #'combine-function
      base-strictness
      combine-strictness)
```

FIGURE A. Parallel language constructs. These parallel programming primitives are used by the applications programmer to code an algorithm for execution on the MX-1.

```
(defun inner-product (arg1 arg2)
    (pyramid-pe number-of-pes
          #'vector-multiply arg1 arg2
          #'vector-sum
          3
          3))

(defun vector-multiply (vector1 vector2)
   (let ((start-index (compute-start-index vector1 this-pe-num))
         (end-index (compute-end-index vector1 this-pe-num)))
      (do ((index start-index (1 + index))
           (partial-sum 0))
          ((= index end-index) partial-sum)
        (setq partial-sum (+ partial-sum
                             (* vector1[index] vector2[index])))))))

(defun vector-sum (promise1 promise2)
    (+ (wait-for-promise promise1)
       (wait-for-promise promise2)))

(setq S (inner-product v1 v2))
```

$$S = \sum_{n=0}^{nmax} v1(n)\ v2(n)$$

FIGURE B. Vector inner-product computation. This example shows the coding of a vector inner product in the form of a parallel program for execution on the MX-1. The vector-multiply function forms a partial product and the vector-sum function combines partial sums in a pairwise fashion.

nicates the results back to the 68020 processor of the node. In fact, the DSP environment can also handle a sequence of calls to the DSP that might not need to have all results at intermediate stages of processing moved to the processing element. The DSP coprocessors, which have been benchmarked on a variety of numeric-intensive routines and compared to the 68882 math coprocessor, consistently execute these routines more than 20 times faster than the 68882. The overall speedup in execution time will be somewhat less than this factor because of the time needed to handle the communication between the 68020 and the DSP.

The parallel-programming extensions to KCL, the MXOS distributed operating system, the use of broadcast memory space, and the software to communicate with and run C programs on the DSP coprocessors were all developed in the Machine Intelligence Group at Lincoln Laboratory.

Parallel Processing Issues and Results

The ROME code was modified in a version for parallel KCL implementation called MX-ROME. Modifications amounted to identifying the code segments that would benefit from parallel execution, such as range clustering, autocorrelation evaluation (Equation 3), and time-Doppler clustering (Equation 4), and casting them in a form that allows a run-time selection of the number of processing elements. MX-ROME was then tested on the MX-1 to evaluate its execution speed. All processing elements ran identical code; the only difference in execution was a local selection of processing subset based on the identity of the individual processor and the total number of pro-

cessors used. In a typical implementation, upper and lower loop index values were computed locally so that the processors shared the computation load fairly.

The system was then run with a fixed data set for a variable number of active processing elements. Figure 15 shows a typical profile of execution speed versus number of processors. This figure shows what at first might be considered a surprising result: as more processors are employed, the execution time decreases initially, achieves a minimum, and then increases thereafter. In addition, the relationship of the processing time to the number of processors exhibits local extrema.

To understand this phenomenon, consider the components of processing time inherent in the parallel execution of an algorithm with N processors. Initially, a fixed time is required for initialization of variables and data access, independent of N. Then the processors are passed data and requested to do their part of the processing; in our implementation, this task involves sending individual promises to the active processors, which requires time proportional to N. The processors then perform their part of the computation, which takes time proportional to $1/N$. Results are gathered in pairs (simultaneously to the extent possible), which requires time proportional to the depth of the binary collection tree, or $\uparrow \log_2 N \uparrow$, where $\uparrow \cdot \uparrow$ means "least integer greater than or equal to." Thus our processing model predicts a processing time T_p of the form

$$T_p = \alpha_0 + \alpha_1 N + \alpha_2 / N + \alpha_3 \uparrow \log_2 N \uparrow .$$

Thus T_p behaves as $1/N$ for small N, as N for large N, and it jumps as N is increased from (for example) 4 to 5 or 8 to 9. The optimum number of processors varies with the application, and greater computation requirements favor larger numbers of processors. For the ROME application, including C code running on the DSP processors, the optimum number is found to be six processing elements. If the DSP processors are not used, the optimum number is seven processing elements, and if all processing is done in LISP, the processing time continues to decrease out to the full complement of 16 processing elements.

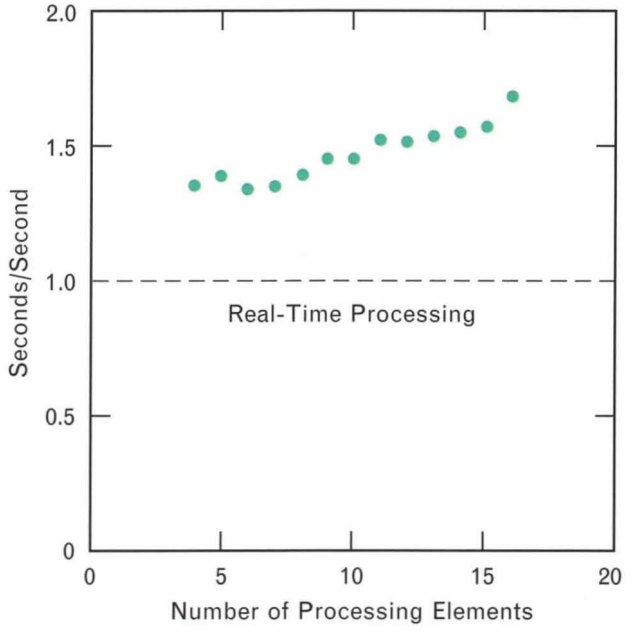


FIGURE 15. MX-ROME processing time versus number of processing elements, in seconds of processing per second of data. The value 1.0 corresponds to real-time operation on this data set. The increase in processing time as processors are added is due to interprocessor communication.

Figure 16 displays the execution time of ROME for various computation platforms. In its initial LISP form running on a Symbolics 3645 workstation, the program required 45 sec of processing for each second of a typical data segment. When ported to a single processing element of the MX-1, the corresponding processing requirement increased to 54 sec per data second, which reflects both the difference in CPU (MX 68020 versus Symbolics 68030) and the alteration of certain code constructs from their Symbolics ZetaLISP form (which is tuned for the Symbolics CPU) to the corresponding generic KCL equivalents. With all 16 processing elements running Parallel KCL, this processing time dropped to 4.3 sec per data second (a speedup of approximately 80% of the maximum possible 16-fold improvement). Rewriting four of the numeric-intensive routines in C and running these routines on the DSP processors gave a further improvement in processing time to 1.3 sec per data second.

The parallel-programming result, however, was still slower than what was needed for real-time operation. This realization led to a structuring of the processing

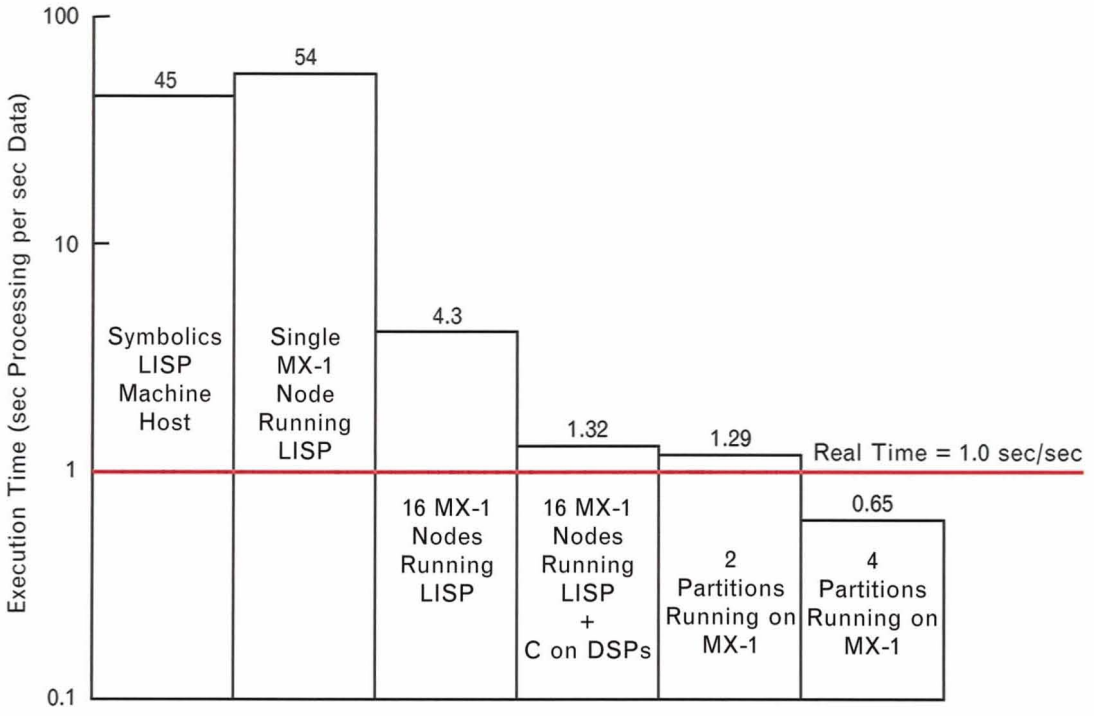


FIGURE 16. MX-ROME execution times for various platforms and software configurations, in seconds of processing per second of data.

elements in clusters, as shown in Figure 17. In this scheme the operating system is designed so that all clusters can simultaneously run code on different data sets with no confusion of identically named variables among the clusters (including system-level global variables such as the number of processing elements). In this implementation of the control mechanism, the processing elements can be assigned arbitrarily to clus-

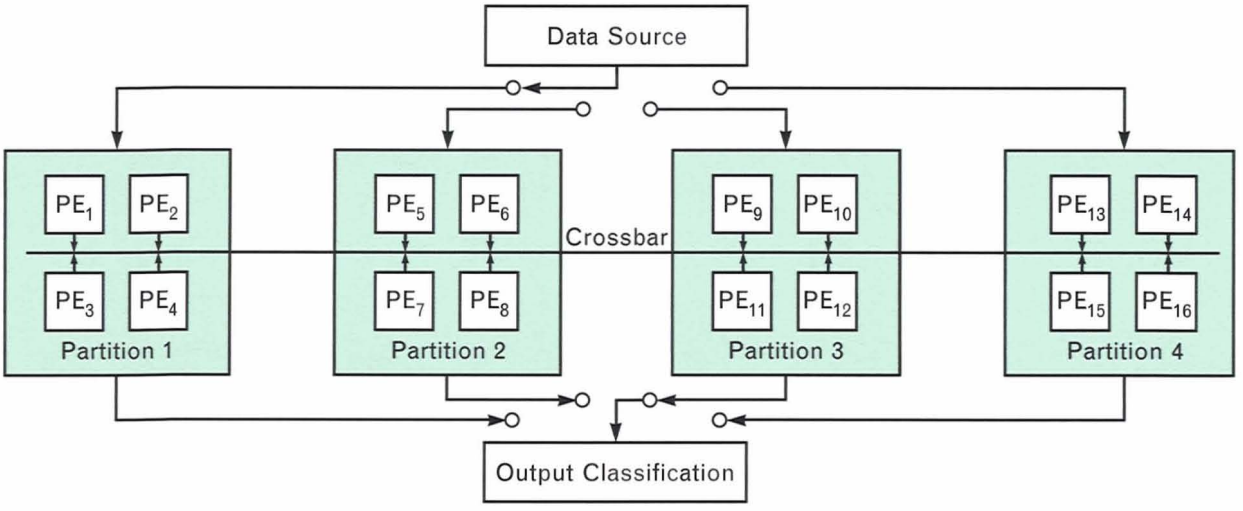


FIGURE 17. Partitioning the MX-1 into clusters of processing elements. For MX-ROME all clusters ran identical code on independent data sets, although more general operation is possible.

ters, clusters can be of unequal size, and each cluster can run an arbitrarily assigned code segment. Of course, fewer processing elements are available to run each cluster's code, and a penalty must be paid in overhead to coordinate the communication of data to and between processing clusters. We found, for example, that a two-cluster partitioning of the MX-1 with eight processing elements per cluster ran the code at approximately the same rate as an eight-node version of the MX-1 without the partitioned code.

With the optimal partitioning of the MX-1 into four clusters of four processing elements each, we achieved a run time of 0.6 processing seconds per data second, well within the required real-time performance limit. The latency is approximately 2.4 sec per data second; i.e., for a two-second data segment, the classification result will be available within five seconds after the data are input to the MX-1.

Summary

The accomplishments of this project are numerous. The ROME software was the first program that used MI techniques to analyze radar imagery and recognize reentry vehicles automatically. This software uses MI techniques to implement the knowledge sources, the blackboard control structure, and the semantic-net object representations. It also builds its own catalog of object models from real data, and in this sense it is trainable. The recognition performance of ROME, when measured on real data, has been excellent. The system was coded for parallel execution on a multiple-instruction multiple-data multiprocessor, the MX-1. It was the first serious MI application to run on the MX-1, and as such, it motivated valuable enhancements to the MX-1 parallel programming environment.

The parallel version, called MX-ROME, makes significant use of the MX-1 performance-monitoring system to time the execution of various code segments and processing steps, which is necessary to optimize parallel programs. With the numeric-intensive processing coded in C and executing on the DSP processors, MX-ROME ran nearly at real time. Real-time execution was achieved on the MX-1 by concurrently running four versions of MX-ROME, each on different subsets of four nodes.

Future Directions

ROME can be enhanced in many ways to make it more effective in the general discrimination problem for ballistic missile defense systems. The ROME code could be extended to incorporate radar polarization data when available and to characterize the individual scatterers by their RCS-versus-azimuth portraits. Efficient use of radar resources suggests developing a look-process-look strategy, whereby data from disjoint observation intervals are combined to make a refined classification decision. Early selection of reentry vehicles from a cloud of objects that includes booster fragments, decoys, and countermeasures will involve building an enhanced object catalog with provision for on-line catalog updating. The general discrimination problem will require the capability of performing all of these functions at real-time rates.

The MX-2 Multiprocessor

A need exists for even more powerful parallel computers to execute general discrimination code handling multiple objects in real time. Since the MX-1 multiprocessor design was finalized and construction began in 1987, microprocessor technology has continued to improve at a steady pace. When reduced instruction-set computer (RISC) chips became commercially available, their potential use in upgrading the MX-1 to an improved multiprocessor—the MX-2—was investigated. Even though RISC microprocessors offer faster computation rates for both numeric and symbolic processing, they cannot simply replace the Motorola 68020 microprocessors used in the MX-1 because the crossbar interconnection network is not fast enough to keep up with the RISC processors. This enhanced computation rate would create a communication bottleneck that would result in an unbalanced multiprocessor architecture and inefficient use of the RISC processors.

To obtain more processing power for real-time MI applications, the MX project has turned its focus to a new multiprocessor design based on the use of these RISC processors. Optical communication techniques will be used for building a very high-speed interconnnection network to avoid the communication bottleneck. The applications of primary concern for

the MX-2 are the same as those addressed in the MX-1 design, namely, MI applications that involve a mix of both numeric and symbolic computations and that have a real-time processing requirement. Therefore, the programming environment envisioned for the MX-2 is essentially that of the MX-1, which allows a user to write parallel programs using a combination of Common LISP and C. The operating system, however, will be MACH instead of a UNIX derivative, to provide better support for parallel processing.

Acknowledgments

Naming all those who have contributed to the success of this project is not practical, but a few of the outstanding contributors should be mentioned. We wish to thank Al Hearn, whose early support for our ideas led to the connection with on-going discrimination and classification research programs at Lincoln Laboratory. Don Johnson developed algorithms for spin estimation; John Delaney developed the initial version of the parallel ROME code for the MX-1; Gary Choncholas first achieved real-time execution of the ROME code on the MX-1 by running several copies of the code concurrently on subsets of processing nodes; Rich Jaenicke converted the parallel ROME code to a uniprocessor benchmark program; and Ellen Glover provided extensive technical support throughout the project. The MX-1 hardware and software teams, led by Paul McHugh and Paul Harmon, respectively, contributed their time generously in helping port the first version of parallel ROME code to the MX-1.

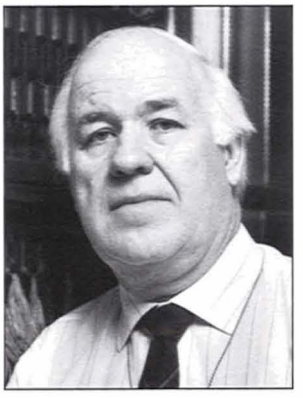
This work was supported by the Department of the Army and the Air Force.

REFERENCES

1. G.E. Kopec, A.V. Oppenheim, and R. Davis, "Knowledge-Based Signal Processing," *Trends Perspectives in Signal Processing* **2**, 1 (July 1982).
2. H. Penny Nii, "Blackboard Systems: The Blackboard Method of Problem Solving," *AI Mag.* 7, 38 (Summer 1986).
3. D.A. Ausherman, A. Kozma, J.L. Walker, H.M. Jones, and E.C. Poggio, "Developments in Radar Imaging," *IEEE Trans. Aerosp. Electron. Syst.* **20**, 363 (1984).
4. D.H. Johnson, "Application of the Hough Transform to Doppler-Time Image Processing," *Proc. ICASSP, New York, 11–14 Apr. 1988,* p. 1212.
5. A. Barr and E.A. Feigenbaum, eds, *The Handbook of Artificial Intelligence, Vol. 1* (William Kaufmann, Los Altos, CA, 1981).
6. P.H. Winston, *Artificial Intelligence* (Addison-Wesley, Reading, MA, 1984).
7. J.H. Connell, "Learning Shape Descriptions: Generating and Generalizing Models of Visual Objects," *Technical Report 853,* MIT Artificial Intelligence Laboratory (Sept. 1985).
8. T.E. Weymouth, "Using Object Descriptions in a Schema Network for Machine Vision," *Technical Report 86-24,* University of Massachusetts, Computer and Information Science (1986).
9. R.S. Michalski, J.C. Carbonell, and T.M. Mitchell, *Machine Learning* (Tioga Pub., Palo Alto, CA, 1983).
10. P.H. Winston, "Learning Structural Descriptions from Examples," *Technical Report 231,* MIT Artificial Intelligence Laboratory (1970).
11. S.B. Bowling, R.A. Ford, and F.W. Vote, "Design of a Real-Time Imaging and Discrimination System," *Linc. Lab. J.* **2**, 95 (1989).
12. T.J. Goblick, P.G. Harmon, J.I. Leivent, J.J. Olsen, D.S. Cogen, J.J. DiTuri, P.G. McHugh, and R.L. Walton, "A Multiprocessor Computer System for Integrated Numeric and Symbolic Computation," *Proc. 5th IEEE Int. Symp. on Intelligent Control, Philadelphia, 5–7 Sept. 1990,* p. 336.
13. R.P. Gabriel, *Performance and Evaluation of LISP Systems* (MIT Press Series in Computer Systems, Cambridge, MA, 1985).

ANN MARIE AULL
is a technical staff member in the Machine Intelligence Technology group. Her current research interests are in the field of radar image understanding, including the application of machine-intelligence techniques, neural networks, and implementations for parallel-programming environments. She received a B.S. degree from Purdue University and an S.M. degree from MIT, both in electrical engineering. While at MIT she worked in the Speech Recognition group under Dr. Victor Zue. Prior to joining Lincoln Laboratory in 1986, she worked at Voice Processing Corporation on developing speech-recognition algorithms.

ROBERT A. GABEL
is a senior staff member in the Machine Intelligence Technology group. His research interests are in innovative applications of signal processing techniques. He received a B.S.E. degree and a Ph.D. degree in electrical engineering from Princeton University, where his doctoral thesis was on computer-generated binary holograms. For ten years he was a faculty member in the Department of Electrical and Computer Engineering at the University of Colorado, where he developed courses and laboratories in signal processing, and where he received awards for his teaching. Bob has also worked at the Bell Telephone Laboratories on PCM communication systems while earning an M.S.E.E. degree at New York University, and at the Sperry Research Center on sonar signal processing and array beamforming techniques. He is coauthor of the textbook *Signals and Linear Systems*, and he has been at Lincoln Laboratory since 1983.



THOMAS J. GOBLICK
is an associate leader of the Machine Intelligence Technology group. He received a B.S. degree in electrical engineering from Bucknell University in 1956 and a Ph.D. degree from MIT in 1963, where his doctoral research was in information theory. He was a Fulbright Scholar in 1958 at the University of London's Imperial College of Science and Technology. He has worked at Lincoln Laboratory since the completion of his graduate work, except for sabbaticals at Washington University in St. Louis in 1967 and at MIT in 1983. During his career he has worked on many aspects of information processing, including spread-spectrum communication, satellite navigation, and speech compression. He was involved in the initial studies of satellite navigation that led to the now operational Global Positioning System. He also worked on the first Federal Aviation Administraton program at Lincoln Laboratory, which developed the Mode-S air-traffic-surveillance and data-link system, scheduled to be operational this year. His sabbatical at MIT was spent with the Department of Electrical Engineering and Computer Science and the Laboratory for Computer Science, where he worked on computer technology and artificial intelligence. His current interests are image understanding, parallel computing architectures, and optical interconnection networks for parallel computers.