REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	20 Oct 95	

4. TITLE AND SUBTITLE A comparison of the Performance of Non-Parametric Classifiers with Gaussian Maximum Likelihood for the Classification of Multispectral Remotely Sensed Data.

5. FUNDING NUMBERS

6. AUTHOR(S)

Steven W. Nessmiller

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

AFIT Students Attending:

USAF Academy

8. PERFORMING ORGANIZATION REPORT NUMBER

95-125

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DEPARTMENT OF THE AIR FORCE
AFIT/CI
2950 P STREET, BLDG 125
WRIGHT-PATTERSON AFB OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

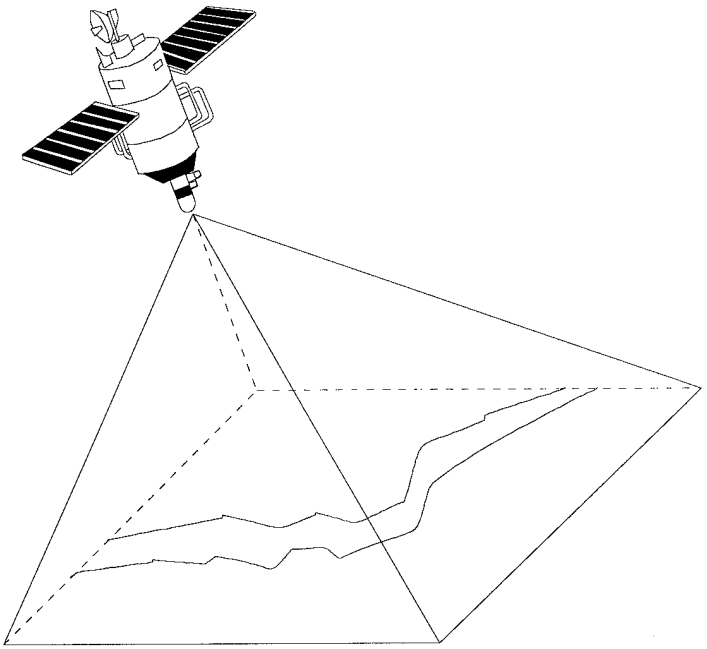
Approved for Public Release IAW AFR 190-1
Distribution Unlimited
BRIAN D. Gauthier, MSgt, USAF
Chief Administration

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS			15. NUMBER OF PAGES 133
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

A Comparison of the Performance of Non-Parametric Classifiers with Gaussian Maximum Likelihood for the Classification of Multispectral Remotely Sensed Data



by:
Steven W. Nessmiller
Captain, USAF
Bachelor of Science - Engineering Sciences, Control Theory
United States Air Force Academy

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in the Center for Imaging in the College of Imaging Arts and Sciences of the Rochester Institute of Technology

Dr. John Schott, Thesis Advisor
Dr. Peter Anderson, Committee Member
Dr. Roger Easton, Committee Member
Dr. Harvey Rhody, Committee Member

A COMPARISON OF THE PERFORMANCE OF NON-PARAMETRIC CLASSIFIERS WITH GAUSSIAN MAXIMUM LIKELIHOOD FOR THE CLASSIFICATION OF MULTISPECTRAL REMOTELY SENSED DATA

by
Steven W. Nessmiller
B.S. United States Air Force Academy
(1988)

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the Center for Imaging Science
Rochester Institute of Technology

September 1995

Signature of the Author 

Accepted by  Sept. 6, 1995
Coordinator, M.S. Degree Program Date

Center for Imaging Science
Rochester Institute of Technology
Rochester, New York

Certificate of Approval

Master of Science Degree Thesis

The Master of Science degree thesis of Steven W. Nessmiller
has been examined and approved by the thesis
committee as satisfactory for the thesis requirement
for the Master of Science Degree

Dr. John Schott, Thesis Advisor

Dr. Peter Anderson, Committee Member

Dr. Roger Easton, Committee Member

Dr. Harvey Rhody, Committee Member

Sept 6 1995
Date

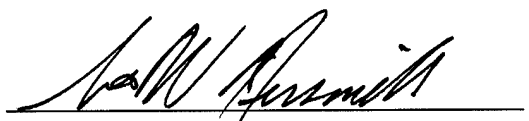
Center for Imaging Science
Rochester Institute of Technology
Rochester, New York

Thesis Release Permission Form

Thesis Title:

A Comparison of the Performance of Non-Parametric Classifiers
with Gaussian Maximum Likelihood for the Classification
of Remotely Sensed Multispectral Data

I, Steven W. Nessmiller, grant permission to the Wallace Memorial Library of the Rochester Institute of Technology to reproduce this thesis in whole or in part provided any reproduction will not be of commercial use or for profit.



Steven W. Nessmiller, Captain USAF

13 Sep 95

Date

A Comparison of the Performance of Non-Parametric Classifiers with Gaussian Maximum Likelihood for the Classification of Remotely Sensed Multispectral Data

by:

Steven W. Nessmiller

1.0 Abstract

This study compares the performance of two non-parametric classifiers and Gaussian Maximum Likelihood (GML) for the classification of LANDSAT TM 30-meter resolution six-band data. The mathematical assumptions made in developing GML are valid if the pixels that constitute the training classes are normally distributed. Since it requires a model of the data, GML is termed a "parametric" classifier. Of current interest are new classification methodologies that make no assumptions about the statistical distribution of the pixels in the training class; these approaches are termed "non-parametric" classifiers. This study will compare the n-Dimensional Probability Density Function (nPDF) essentially a projection technique that reduces data dimensionality, and an advanced neural network that utilizes fuzzy-set mathematics, the Fuzzy ARTMAP, to the traditional GML approach to image classification. The different approaches will be compared for statistical classification accuracy and computational efficiency.

Acknowledgments

I would like the efforts of the following people to be noted. This study would not have been possible without their combined efforts.

Dr. John Schott, my thesis advisor, for his guidance, patience, and ability to keep me busy for a week after a ten-minute meeting.

Dr. Roger Easton, for correcting my, grammar and speeling, and constantly challenging me in the classroom.

Dr. Carl Salvaggio, for helping me develop the basic idea for this study, and then promptly moving to a different state.

Dr. Harvey Rhody for sparking and kindling an interest in the fascinating field of pattern recognition.

The programming wizardry of Mr. Stephen L. Schultz.

Mr. Frank Tantalo for providing someone to study with, bounce ideas off of, and ensure that each other was still sane.

Major Harry Gross, Amy Gross, and all the little Grosses (Grossi?) for their dedicated support of the Feed a Single Captain Program (FSCP).

Our Senior Ranking Officer Major Joe Chapa, who constantly proved to be much more than just an excellent officer. I think I'm finally starting to understand some of the things you've told me.

Sue Chan and Val Heminck for helping ensure I ended up in the right classes, had enough research hours, and on and on....

The following people from the Eastman Kodak corporation who made my stay with them very enjoyable as well as educational: Joe Westbay, Jennifer LeBaron, Don Oinen, Ken Mason, Mark Janosky, Bob Breary, Patty Hook, Sue Roncone, Sue Cardot, Donna Rankin-Parobek, Bob Fiete, Jim Salacain, Jim Mooney, Leslie Marangola, Don Agnew, Laura Mellberg, and Vince Piarulli.

Dedication

First, this thesis is dedicated to the people who have had the greatest impact on my life, my parents. I think that all of the Lego, the Erector set, and the collection of "How and Why" books may have finally paid for themselves. I promise to keep doing my best.

Next, it is dedicated to the vast number of instructors who have taken the time, effort, and interest necessary to expertly instruct a wide array of subjects. I am proud to say that my sister is among your ranks continuing this critical profession.

Finally, this study is dedicated to the reader.

Table of Contents

List of Tables

List of Figures

2.0 Introduction

The objective of this thesis is to compare the performance of two non-parametric classifiers (a fuzzy ARTMAP neural network and the nPDF algorithm) with the classical Gaussian maximum likelihood (GML) approach. All parametric classification schemes, including GML, make some assumption about the statistical distribution of the training-class pixel intensity vectors. This assumption is utilized to determine a statistical decision rule (i.e. the Mahalanobis distance) for classification purposes. The GML has been shown to be a robust classifier, but its effectiveness suffers when the training-class pixel distribution varies markedly from normality or when class means are only slightly separated (Frey, 1994). Non-parametric classifiers make no assumption of the distribution of the pixels in the training classes. As such, they exhibit increased accuracy, but are extremely sensitive to biased training sets.

Four multispectral images of varying composition (i.e. agricultural, rural, urban, and forest) will be classified with the same training data by each algorithm. Four images will be classified by each algorithm to minimize the potential of anomalous effects arising from a particular image and algorithm combination.

The LANDSAT satellite collects multispectral information as it orbits above the earth's surface. The images collected by this system are composed of pixels that nominally represent the irradiance gathered from a 30-meter-square patch of the earth's surface. Each pixel in an image can be described by a six-dimensional intensity vector whose elements are the 8-bit digital count (DC) value (an integer ranging from 0 to 255). The spectral sensitivity of each band is summarized in table 2.0.1 (Richards, 1993). Note that band 6, the thermal band, is not included. This is due to the fact that this information is not correlated with that in the other bands. As such, it typically is not utilized in image classification operations.

Table 2.0.1 - Spectral sensitivity of LANDSAT TM bands

TM Band	Bandpass (μm)	"Color"
1	0.45 - 0.52	blue
2	0.52 - 0.60	green
3	0.63 - 0.69	red
4	0.76 - 0.90	near IR
5	1.55 - 1.75	mid IR
7	2.08 - 2.35	mid IR

Similarly, the M-7 airborne system is another multispectral sensor. It utilizes a line scanning system and can measure light at wavelengths in the range of 0.33 to 14.0 μm. Up to 19 different bands in this range can be collected simultaneously, with a typical ground sample distance of approximately 5 meters. Table 2.0.2 details the selected bands utilized in this study to mimic the LANDSAT TM sensor.

Table 2.0.2 - Spectral sensitivity of selected M-7 Bands

M-7 Band	Bandpass (μm)	"Color"
3	0.44 - 0.46	blue
6	0.52 - 0.55	green
8	0.60 - 0.67	red
10	0.83 - 1.00	near IR
12	1.50 - 1.90	mid IR
13	2.10 - 2.60	mid IR

The importance of high-quality training data cannot be overstated, especially for the non-parametric classifiers. By using the same training data for each classification algorithm, the potentially negative effects of any variation will be eliminated. The

distance separating the cluster centers of the target classes will be decreased to evaluate the impact of decreased separation on the effectiveness of each algorithm. Additionally, a hybrid classification approach which combines the strengths of the different algorithms will also be studied.

The different approaches to classification were compared statistically in terms of their classification accuracy via a confusion matrix and the kappa coefficient. Computational efficiency was compared in terms of elapsed run time, training time, and system resource requirements. This comparison highlights the relative strengths and weaknesses of the different classifiers and determine the set of conditions where a specific classifier is best employed.

3.0 Objectives and Deliverables

Statement of Work

- Implement the various classifiers in the Advanced Visualization System (AVS), an interactive data visualization environment.
- Select at least four multispectral images of varying composition.
- Utilize the fuzzy K-means algorithm to collect trusted and spectrally pure training data for the classification algorithms.
- Utilize an AVS module to collect user-defined training data for the classification algorithms.
- Evaluate the classification algorithms statistically in terms of their classification accuracy on both dependent and independent training sets.
- Evaluate the classification algorithms computationally in terms of their efficiency.
- Experiment with hybrid classification methodologies.

List of Deliverables:

- A Gaussian maximum likelihood classification module for AVS environment.
- An AVS module to perform fuzzy K-means clustering to create truth images and collect trusted training data.
- An nPDF classification and module for LANDSAT TM images for use in the AVS environment that will permit user definable classification boundaries.
- An AVS module that implements the fuzzy ARTMAP neural network algorithm to classify LANDSAT TM images.
- An AVS module to compute confusion matrices and classification accuracy statistics.
- A written document covering the theory, background, approach, and results of the study.

4.0 Background and Approach

4.1 Acquisition of Training Data

The importance of high-quality training data in the parametric and non-parametric classification algorithms has been introduced. The various classification algorithms require labeled training data to calculate representative statistics, to train a neural network, or to define classification boundaries. The method in which these data will be acquired from each image bears some explanation. Consider the simplified representation of a LANDSAT TM scene (Figure 4.1.1) in the following discussion. We will assume that there are four classes of interest: coniferous trees, deciduous trees, water, and grass. The Gaussian maximum likelihood (GML) parametric classifier assumes that the training-class pixels are distributed in a multivariate normal manner. This assumption can be validated through application of the central limit theorem, but this theorem also imposes a restriction on the minimum number of pixels that must be present in each training class. Swain and Davis (1978) state that the minimum number of pixels per training class is 10D, where D is the number of bands or dimensions being utilized in the classification algorithm, while 100D would be "highly desirable". Other references state that 30 examples per class produce results that are "quite good" for a one-dimensional or single-band case (Dougherty, 1990). The increase in the number of training-class pixels required to describe a class in a feature space of higher dimensionality can be intuitively explained quite readily. As dimensionality increases, the probability of a particular spectral band being inadequately represented also increases. To offset this, training class size must be positively correlated with dimensionality.

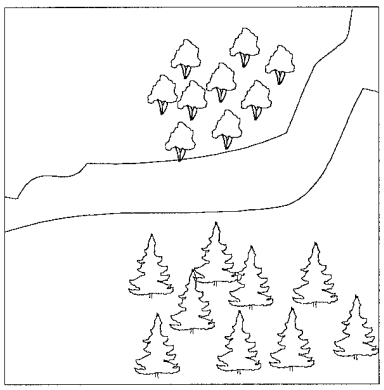


Figure 4.1.1 - Depiction of a LANDSAT TM scene

Because non-parametric classifiers make no assumption of the underlying statistical distribution of the pixels, no mathematical inference of class membership may be made. Because of this limitation, non-parametric algorithms require even more robust training data than the parametric classifiers. With these considerations in mind, this study will limit training class membership to not less than 30D and will strive for 100D whenever possible. In addition, all algorithms will use the same training data to classify each image, thereby eliminating any potential effects due to variations in training data.

Typically, training sets are defined by drawing polygons on the image that delineate the extent of a target class. Figure 4.1.2, is identical to Figure 4.1.1 except for of the superimposed training class polygons. Class 1 represents deciduous trees, class 2 is composed of coniferous trees, class 3 is comprised of water, and class 4 is made up of grass pixels. Note that in this example there were not enough contiguous grass pixels in one region to adequately describe the grass class. Because of this, two regions had to be defined to meet the previously discussed minimum membership criteria. The algorithm then extracts the pixels within the polygon and "labels" them as belonging to the indicated target class. The classification algorithms can then be presented with a set of

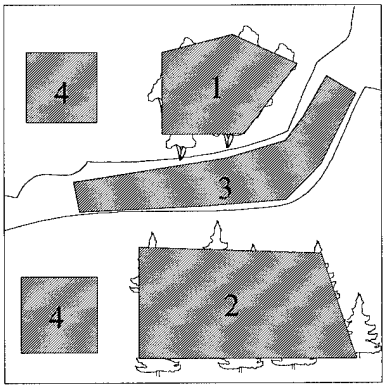


Figure 4.1.2 - Depiction of LANDSAT TM scene with overlayed training class polygons

labeled training data for each target class so that statistical or other calculations can be made.

The preceding discussion describes the most common method to collect training data for *supervised* classification algorithms. The term "supervised" implies that a human operator specifies the class membership of the test pixels. One of the main shortcomings of this process is that it is extremely difficult, if not impossible, to collect sets of "pure" training data. This difficulty springs from the inability to collect and label pixels that belong to only one spectral class. To envision this problem, consider the areas of the LANDSAT TM image depicted in figure 4.1.1 that are primarily composed of deciduous and coniferous trees. It would be extremely difficult to draw polygons that encompassed just tree pixels without accidentally including some of the background grass pixels. Training sets will likely always be "polluted" with such impurities.

Unsupervised image classification techniques often rely on information gathered from determining centers of *clusters* belonging to naturally occurring classes of spectral vectors in the image. To continue with the simplistic LANDSAT image example, it would be reasonable to assume that there are 4 major clusters of spectral vectors, one for each of the broad classes. A clustering algorithm typically determines the location of the cluster centers in *pixel space* in an iterative manner. Once the location of the clusters have been determined, some type of *minimum-distance-to-the-means* classification can be readily accomplished. This simple classification algorithm determines the Euclidean distance of the intensity vector of a pixel to the various cluster centers, and then assigns the pixel to the closest class. This set of spectrally pure and labeled pixels could provide an excellent data set to train the supervised classification algorithms.

The ISODATA, or K-means algorithm is a well known method of determining cluster centers. The parameter "K" represents the number of cluster centers to locate. The algorithm was developed by G. H. Ball and D. J. Hall in 1967 and is implemented in

many multispectral image processing packages. In 1973, J. C. Dunn developed a version of the algorithm which employed *fuzzy set theory* to determine the degree of membership of a spectral vector to a given cluster. This version generally converges faster and is less likely to divide naturally occurring clusters, than the crisp set theory implementation. The simplest explanation for the desirable qualities of this algorithm is that individual pixels need not be modeled as belonging to a cluster, but the degree of their *membership* can be determined. Pixels with high membership values greatly influence the cluster center, while outlying pixels with lower membership values influence it to a lesser degree. The mathematical development presented here essentially follows that in Dunn (1973).

Fuzzy set theory, to be described in greater detail in the section 4.4 of this report detailing the fuzzy ARTMAP neural network, is often "injected" into existing algorithms by implementing a *membership function*. This function returns a value which represents the degree to which a given data point belongs to a set. Large membership values, typically thresholded to unity, indicate that the pixel intensity vector displays many of the qualities of the set to a great degree. Lower values, typically closer to zero, indicate that the particular example does not fully represent all qualities of the set. Membership values are analogous to probabilities, but have different underlying mathematical properties and cannot be treated in the same manner. The great strength that this type of fuzzy measurement brings to an algorithm is that data elements can contribute to multiple sets rather than to the membership of only one set. Consider the representation of the boundary between neighboring sets. In one case the sets have crisp boundaries; in the other, the boundaries are fuzzy (Figure 4.1.3).

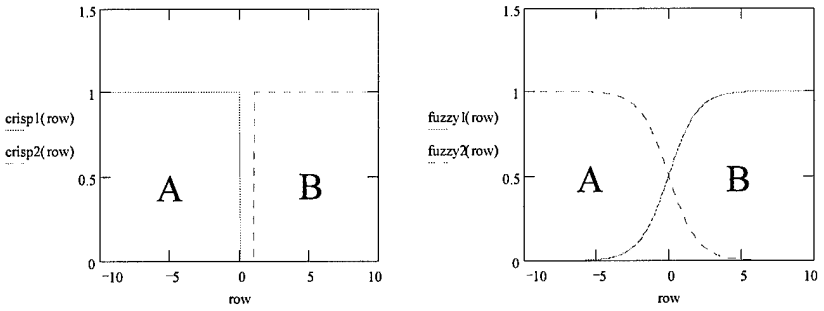


Figure 4.1.3 - Depiction of the boundary between two crisp sets (A and B at left) and the boundary defined by the membership function between two fuzzy sets (right).

The vertical axis of each plot represents the membership that a given point has in set A or set B. For the crisp set case, this value can be either 0 or 1, while the points belonging to the fuzzy set can have membership values anywhere in the unit interval. To see the power of a fuzzy set representation, consider the pixel located at x = +2. In the crisp set representation, the only information we have concerning this data point is that it has membership in set B. No knowledge about the strength of its membership, or the fact that it is located near a boundary, is conveyed. In the fuzzy-set view of the same data point (Figure 4.1.4), we see that this data point has strong membership (0.9) in set B, but that it also has some of the qualities (0.1) of the data points that constitute set A. The information conveyed through the membership function allows us to treat this data point more appropriately. Whenever a decision is made to assign a data point to a particular set, some information is lost. Fuzzy-set theory combats this

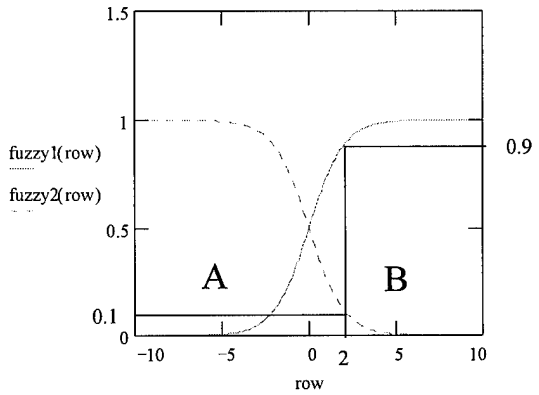


Figure 4.1.4 - Depiction of the membership of a point in two neighboring fuzzy sets

information loss by relaying a confidence measure about the quality or strength of the decision.

At the start of the fuzzy clustering algorithm, the number K of clusters to be found must be estimated. This can be approximated by visually inspecting the image, or set based on a priori knowledge of how many classes are necessary. Many non-trivial papers are devoted to intelligent determination of the number of clusters present in a data set, but further discussion of this subject is beyond the scope of this report. After a value for K is selected, that K pixels are chosen at random from the input image. This ensures that the algorithm is starting with valid solution positions and is not outside the solution space. Another way to begin the algorithm would be to determine the minimum and maximum spectral intensity vectors across all bands, and then construct a line in the N-dimensional space (where N represents the number of data bands) with K evenly spaced points as initial solutions. The membership with respect to each of the K clusters is then determined. Consider the following membership equation:

$$m_i(\bar{\mathbf{x}}) = \frac{\dfrac{1}{\left(\bar{\mathbf{x}}-\bar{\mathbf{u}}_i\right)^2}}{\sum_{j=1}^{k}\left(\dfrac{1}{\left(\bar{\mathbf{x}}-\bar{\mathbf{u}}_j\right)^2}\right)} \qquad \text{for } 1 \le i \le k \tag{4.1.1}$$

where $\bar{\mathbf{x}}$ is the N-dimensional spectral vector for a given pixel, $\bar{\mathbf{u}}_i$ is the spectral vector representing the center of the "*i*th" cluster, and $m_i(\bar{\mathbf{x}})$ is the membership of the pixel with respect to the *i*th cluster. The distance measure employed in this implementation should be recognized as the square of the simple Euclidean distance. Note that the membership equation approaches unity for points near to one class and far from all others. Numerical problems associated with being very close (or directly on) a cluster center are handled by checking for very small distance measures and then setting the membership of the pixel to that cluster to one. Once the membership of the pixels in the image have been

determined with respect to each cluster, the new cluster centers can be calculated. Consider the following equation:

$$\bar{\mathbf{u}}_i = \frac{\sum_{x \in \chi} [m_i(\bar{\mathbf{x}})]^2 \bar{\mathbf{x}}}{\sum_{x \in \chi} [m_i(\bar{\mathbf{x}})]^2} \qquad \text{for } 1 \leq i \leq k \tag{4.1.2}$$

where $x \in \chi$ implies that the sum is taken over all the pixels in the image. Note that this equation must be calculated K times to find the elements of $\bar{\mathbf{u}}_i$. This forms a weighted average of the influence of each pixel on each cluster center through the use of the membership function. Once the new cluster centers have been calculated, the distance that the center vectors have moved since the last iteration is determined. If the maximum movement is below some user-defined threshold, we can conclude that the cluster centers have converged to their final locations. After the cluster centers have converged, the "*cluster map*" image is constructed by checking the membership value of each pixel in the image with respect to all of the clusters. If the largest membership value is equal to or greater than a user-defined membership threshold, the pixel is "labeled" as part of that class. A set of training data can then be built by scanning the cluster map image for pixels assigned to clusters, retrieving the corresponding multispectral pixel from the original image, sorting them by cluster number, and writing the resulting ordered set to disk. In pseudocode, the entire fuzzy K-means algorithm can be expressed as:

```
pick K clusters at random from the image
        while clusters have not converged
                calculate membership
                calculate new cluster centers
        check for convergence
build cluster map
build training set
```

It is important to note that while we have constructed a set of labeled spectrally pure pixels for a data set, that we have potentially "colored" them by the collection

methodology. We have employed a Euclidean distance measure that we will later see is unable to account for a data set's inherent distribution. Also note that the algorithm may not find clusters in the image of interest to the analyst, and, furthermore, it may then be difficult for the analyst to assign meaningful labels to each of the derived clusters.

Image Classification Algorithms

4.2 Gaussian Maximum Likelihood

Gaussian Maximum Likelihood (GML) is perhaps the most popular classification algorithm due to its employment of classical mathematics and the fact that it generates a measurement of membership certainty of a pixel to a class. GML is widely taught in introductory courses in both remote sensing and pattern recognition, and it is implemented in many image processing packages. As such, it is a standard to which the performance of other algorithms can be compared.

GML is a parametric classifier, based on the assumption of normally distributed pixel intensity vectors. It has been shown that the assumption of normally distributed pixel intensity vectors is valid in a typical remote sensing application (Frey, 1994). This can be attributed to the averaging effect of the LANDSAT TM sensor as it detects the irradiance emitted or reflected from a nominally 30-meter square patch on the earth. Recall that the density of the sum of two independent random variables is the convolution of their individual densities. The averaging of the TM sensor essentially convolves the probability density functions of the materials from which the detected photons were emitted or reflected. Repeated convolution of the individual probability density functions rapidly approaches a normal distribution. This is the basis for the central limit theorem. Nevertheless, the GML algorithm has also been shown to be extremely robust, even when dealing with data sets that deviate markedly from normality (Frey, 1994). The development presented here essentially follows that in Richards (1993), with the link to the χ^2 (chi squared) distribution developed by Johnson and Wichern (1992).

4.2 Mathematical Development of GML

In the development of the algorithm we will assume that the components of the n-dimensional vector $\bar{\mathbf{x}}$ represent the intensity vector of a pixel, and the m-dimensional vector $\bar{\mathbf{w}}$ represents the target classification classes.

$$\bar{\mathbf{x}} = \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} \qquad \bar{\mathbf{w}} = \begin{pmatrix} w_1 \\ w_2 \\ \cdot \\ \cdot \\ \cdot \\ w_m \end{pmatrix}$$

From the training data, we can calculate the following conditional probability,

$$p(\bar{\mathbf{x}}|w_i) \tag{4.2.1}$$

which should be interpreted as the probability that a pixel with spectral vector $\bar{\mathbf{x}}$, is a member of class w_i. A much more useful measure to develop would be:

$$p(w_i|\bar{\mathbf{x}}) \tag{4.2.2}$$

which is the probability of membership in class w_i for a specific pixel. This measure could be utilized for classification by determining the most probable class for a pixel. This measure can be derived through the application of *Bayes' Rule* which can be stated as:

$$p(a|b) = \frac{p(a \cap b)}{p(b)} \tag{4.2.3}$$

$$\text{similarly, } p(b|a) = \frac{p(b \cap a)}{p(a)} = \frac{p(a \cap b)}{p(a)} \tag{4.2.4}$$

where the $\cap$ operator indicates simple intersection of two sets. By setting the equivalent terms equal and regrouping, we can state:

$$p(b|a) = \frac{p(b)p(a|b)}{p(a)} \tag{4.2.5}$$

In our previous notation, a useful result is obtained:

$$p(w_i|\bar{\mathbf{x}}) = \frac{p(w_i)p(\bar{\mathbf{x}}|w_i)}{p(\bar{\mathbf{x}})} \tag{4.2.6}$$

Two terms in equation 4.2.6 require some clarification. The term $p(w_i)$ often is titled an *a prior probability* as it acts as a weighting function which reflects how much of a given image is composed of a particular class. This term normally is set equal to 1/M , where M is the number of classes. When this condition is obviously not true, the term will be reasonably estimated. The $p(\bar{\mathbf{x}})$ term represents the probability of a pixel with spectral vector $\bar{\mathbf{x}}$ being present in the image. Note that pixels are classified by comparing the various conditional probabilities of each spectral intensity vector for each target classification class. When utilizing this methodology, the term $p(\bar{\mathbf{x}})$ will appear in the denominator of both conditional probabilities on both sides of the inequality. Since it is a common term, it can be canceled. This results in the following classification logic:

$$\bar{\mathbf{x}} \in w_i \text{ if } p(w_i|\bar{\mathbf{x}}) > p(w_j|\bar{\mathbf{x}}) \text{ for all } i \neq j \tag{4.2.7}$$

It is important to realize that equation 4.2.7 states that all spectral vectors present in the image will be assigned to one of the target classes in the image. This may not be desirable, and the requirement can be eliminated by setting a threshold for class membership assigning a pixel whose probability is less than the threshold to a "background" class.

If we now assume that the spectral intensity vectors are distributed in a multivariate normal manner the conditional probability can be stated in the following manner:

$$p(\bar{\mathbf{x}}|w_i) = \frac{1}{(2\pi)^{\frac{D}{2}}\sqrt{|\Sigma_i|}}\, e^{-\frac{1}{2}(\bar{\mathbf{x}}-\bar{\mu}_i)^T \Sigma_i^{-1}(\bar{\mathbf{x}}-\bar{\mu}_i)} \tag{4.2.8}$$

Where Σ_i^{-1} is the inverse of the *variance-covariance* matrix for class i, $\bar{\mu}_i$ is the mean vector for the "ith" class, $\bar{\mathbf{x}}$ is the spectral intensity vector whose class membership is being evaluated, and D is the number of dimensions or spectral bands being used. The operator "| |" designates matrix determinant and the transpose operator "T" converts the column vectors to row vectors and vice versa. The variance-covariance matrix contains a wealth of information concerning the dispersion of data in a target class. As such, it

warrants some explanation. The general form of a variance-covariance matrix is:

$$\Sigma_i \quad \begin{vmatrix} \sigma^2_{11} & \sigma^2_{12} & \sigma^2_{13} & \cdots & \sigma^2_{1n} \\ \sigma^2_{21} & \sigma^2_{22} & & & \\ \sigma^2_{31} & & \sigma^2_{33} & & \\ \vdots & & & & \\ \sigma^2_{n1} & & & & \sigma^2_{nn} \end{vmatrix}$$

Figure 4.2.1 - general form of Σ matrix

where σ^2_{ij} is the covariance of band i with respect to band j. The matrix can be calculated by:

$$\Sigma_i = \varepsilon\{(\bar{\mathbf{x}} - \bar{\mu}_i)(\bar{\mathbf{x}} - \bar{\mu}_i)^T\} \tag{4.2.9}$$

Where ε is the expectation operator. Typically an unbiased estimator is utilized to approximate the variance-covariance matrix numerically:

$$\Sigma_i = \frac{1}{K-1} \Sigma_{i=1}^{K} (\bar{\mathbf{x}} - \bar{\mu}_i)(\bar{\mathbf{x}} - \bar{\mu}_i)^T \tag{4.2.10}$$

where *K* represents the number of data elements in the training class. It is important to realize that this calculation must be performed once for each of *m* classes, so that all the target classification classes are described mathematically. Also realize that σ^2_{ij} must equal σ^2_{ji}, and that σ^2_{ii} is the simple empirical variance. In general, if any of the off-diagonal terms (σ^2_{ij}) have large amplitudes, bands i and j are highly correlated. This implies that information about one band can be used to predict the value of another, or that the information contained within the bands is not independent, but correlated. More simply put, if bands i and j are positively correlated, an increase in band *i* will be accompanied by a corresponding increase in the observed values in band *j*. If the off-diagonal values are small, then the data are independent and cannot be described by some linear combination of the spectral bands. As previously mentioned, the information contained within the variance-covariance matrix permits more accurate modeling of the dispersion of the data that constitutes a training class. Specifically, the information

contained within a variance-covariance matrix defines an n-dimensional hyperellipsoid in feature space. The eigenvalues of the matrix define the lengths of the axes of the hyperellipsoid, while the associated eigenvectors determine their orientation (Johnson and Wichern, 1992).

Returning to the development of GML, we will at this point introduce the *discriminant* function, which will result in some mathematical convenience:

$$g_i(\bar{\mathbf{x}}) \equiv \ln\{p(\bar{\mathbf{x}}|w_i)\} \tag{4.2.11}$$

and applying this operation to the multivariate normal distribution yields:

$$g_i(\bar{\mathbf{x}}) = -\frac{D}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma_i| - \frac{1}{2}(\bar{\mathbf{x}} - \bar{\mu}_i)^T\Sigma_i^{-1}(\bar{\mathbf{x}} - \bar{\mu}_i) \tag{4.2.12}$$

Note that a number of common terms can be eliminated in classification, resulting in a discriminant function that can be stated as:

$$g_i(\bar{\mathbf{x}}) = -\ln|\Sigma_i| - (\bar{\mathbf{x}} - \bar{\mu}_i)^T\Sigma_i^{-1}(\bar{\mathbf{x}} - \bar{\mu}_i) \tag{4.2.13}$$

Classification may be accomplished by employing the following rule:

$$\bar{\mathbf{x}} \in w_i \text{ if } g_i(\bar{\mathbf{x}}) < g_j(\bar{\mathbf{x}}) \text{ for all } i \neq j \tag{4.2.14}$$

Note that the last term of equation 4.2.13 resemble a distance measure. By eliminating the negative signs and considering the special case where Σ_i is the identity matrix, the classification rule reduces to the simple Euclidean distance measure. This distance measure can be described mathematically as:

$$d_E(\bar{\mathbf{x}}, \bar{\mu}_i)^2 = (\bar{\mathbf{x}} - \bar{\mu}_i)^T(\bar{\mathbf{x}} - \bar{\mu}_i) \tag{4.2.15}$$

and the classification rule becomes:

$$\bar{\mathbf{x}} \in w_i \text{ if } d_E(\bar{\mathbf{x}}, \bar{\mu}_i)^2 < d_E(\bar{\mathbf{x}}, \bar{\mu}_j)^2 \text{ for all } i \neq j \tag{4.2.16}$$

In feature space, equation 4.2.16 defines a set of hyperspheres located concentrically about the mean vector of the class. This Euclidean distance measure is the heart of the minimum-distance-to-the-means classification methodology.

Incorporation of the data distribution information in the variance-covariance matrix into a distance measure can be accomplished quite readily. Consider the following

measure, often termed the *Mahalanobis* or *statistical* distance:

$$d_M(\bar{\mathbf{x}}, \bar{\mu}_i)^2 = \ln|\Sigma_i| + (\bar{\mathbf{x}} - \bar{\mu}_i)^T \Sigma_i^{-1} (\bar{\mathbf{x}} - \bar{\mu}_i) \tag{4.2.17}$$

A development identical to that leading to equation 4.2.14 results in the following classification rule:

$$\bar{\mathbf{x}} \in w_i \text{ if } d_M(\bar{\mathbf{x}}, \bar{\mu}_i)^2 < d_M(\bar{\mathbf{x}}, \bar{\mu}_j)^2 \text{ for all } i \neq j \tag{4.2.18}$$

which represents the desired Gaussian maximum likelihood classifier. As compared to the previous Euclidean distance measure classifier (equation 4.2.16) note that this algorithm defines an n-dimensional hyperellipsoid in feature space. This is accomplished by the "space scaling" effect of the variance-covariance matrix.

It is generally desirable, though computationally intensive, to account for the dispersion of the training data. The validity of this statement can be visualized readily. Consider the identical two-dimensional feature spaces in figures 4.2.3a and 4.2.3b. The information about the covariance of the cluster is not utilized in the first example (Euclidean distance measure and minimum-distance-to-the-means), but is employed in the second case (Gaussian maximum likelihood employing the Mahalonobis distance). The pixel to be classified is represented by the cross near the middle of the feature space. In Figure 4.2.3a, the pixel of interest would be assigned to class one as its vector is closer to μ_1 in a Euclidean sense. By taking into account the inherent dispersion of the data, as

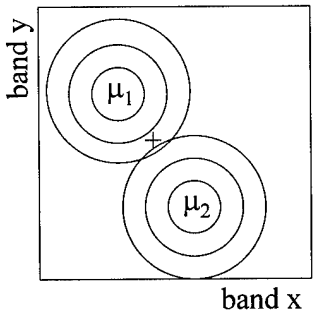


Figure 4.2.3a
hyperspheres resulting from
minimum distance to the means

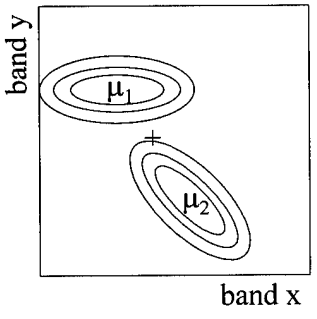


Figure 4.2.3b
hyperellipsoids resulting from GML

depicted in Figure 4.2.3b, the GML classifier would assign the pixel to class two. Note that this effect is most easily explained by realizing that the information contained within the variance-covariance matrix essentially scales feature space differently in different directions thereby accounting for the data's dispersion.

As previously mentioned, Gaussian maximum likelihood always assigns a pixel to one of the target classification classes. This can be undesirable especially in the case where a particular pixel is below some threshold for membership in any target class, and it would be most advantageous to assign this pixel to a "background" or "other" class. Recall the Mahalanobis distance measure as defined in equation 4.2.17:

$$d_M(\bar{\mathbf{x}}, \bar{\mu}_i)^2 = \ln|\Sigma_i| + (\bar{\mathbf{x}} - \bar{\mu}_i)^T \Sigma_i^{-1} (\bar{\mathbf{x}} - \bar{\mu}_i) \tag{4.2.19}$$

and its associated classification rule equation 4.2.18:

$$\bar{\mathbf{x}} \in w_i \text{ if } d_M(\bar{\mathbf{x}}, \bar{\mu}_i)^2 < d_M(\bar{\mathbf{x}}, \bar{\mu}_j)^2 \text{ for all } i \neq j \tag{4.2.20}$$

We desire to incorporate a threshold value into this measure to describe the level of certainty to be attained prior to assigning a pixel to a given class. Mathematically, this can be expressed as:

$$\bar{\mathbf{x}} \in w_i \text{ if } d_M(\bar{\mathbf{x}}, \bar{\mu}_i)^2 < d_M(\bar{\mathbf{x}}, \bar{\mu}_j)^2 \text{ for all } i \neq j$$
$$\text{and } d_M(\bar{\mathbf{x}}, \bar{\mu}_i)^2 \leq T_i \tag{4.2.21}$$

where T_i is the threshold value to attain membership in class i. It can be shown (Johnson and Wichern, 1992) that for a D-dimensional multivariate normal distribution, the vectors have a χ^2 (chi squared) distribution if the hyperellipsoids of vectors satisfy the following relationship:

$$(\bar{\mathbf{x}} - \bar{\mu}_i)^T \Sigma_i^{-1} (\bar{\mathbf{x}} - \bar{\mu}_i) \leq T_i \tag{4.2.22}$$

This distribution has D degrees of freedom and a probability of 1-α. In a more compact notation this is typically represented as $\chi^2_D(\alpha)$ Utilizing this result to our distance measure results in the following classification algorithm:

$$\bar{\mathbf{x}} \in w_i \text{ if } d_M(\bar{\mathbf{x}}, \bar{\mu}_i)^2 < d_M(\bar{\mathbf{x}}, \bar{\mu}_j)^2 \text{ for all } i \neq j$$
$$\text{and } d_M(\bar{\mathbf{x}}, \bar{\mu}_i)^2 \leq \chi^2_D(\alpha) \tag{4.2.23}$$

Intuitively, equation 4.2.23 is comforting. As the value of α is decreased, thereby increasing the probability that a particular spectral intensity vector is within the hyperellipsoid of a given class, the $\chi^2_D(\alpha)$ distribution returns larger and larger values. This can be visualized as the hyperellipsoid swelling or inflating in feature space to encompass a greater and greater volume.

4.3 n-Dimensional Probability Density Functions (nPDF)

The nPDF algorithm was codeveloped at the Department of Earth and Atmospheric Sciences at Purdue University by Haluk Cetin and Donald W. Levandowski. This method utilizes what are known as *frequency perspective* plots to allow the display of multi-dimensional data on two-dimensional display devices (such as a CRT), to reduce data dimensionality in a manner similar to the Karhunen-Loe've transformation, to classify the multidimensional data in either a supervised or unsupervised manner, and to perform cluster analysis prior to classification. In essence, the algorithm projects n-dimensional data onto a two-dimensional plane through the use of two distance measurements. The straightforward mathematical development that follows will center on the derivation of the projection technique and how the resulting projection can be utilized for classification. The development presented here essentially follows that presented by Cetin and Levandowski (1991).

4.3.1 Mathematical Development of nPDF

As previously mentioned, the nPDF algorithm essentially projects n-dimensional data onto the two-dimensional plane where considerably simplified classification techniques can be applied. In a two-dimensional feature space, a feature vector is defined by:

$$\bar{\mathbf{x}} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

A simple way to uniquely describe the position of the point (x_1, x_2) in this feature space is as the intersection of two arcs originating from two reference points or "corners". Consider the diagram in Figure 4.3.2 where band 1 and band 2 are two bands of interest in the multispectral data. The range of the axes is assumed to be 256 (2^8 as per 8-bit TM data). The magnitudes of the radii of the arcs are:

$$d_1 = \sqrt{x_1^2 + x_2^2} \tag{4.3.1}$$

$$d_2 = \sqrt{(R - x_1)^2 + x_2^2} \tag{4.3.2}$$

where R represents the maximum value of the data (255). By extending this concept to a three-dimensional feature space, Figure 4.3.2 is easily obtained.

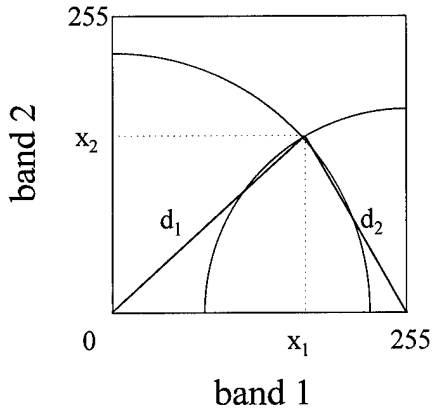


Figure 4.3.1
a two-dimensional feature space

$$\bar{\mathbf{X}} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

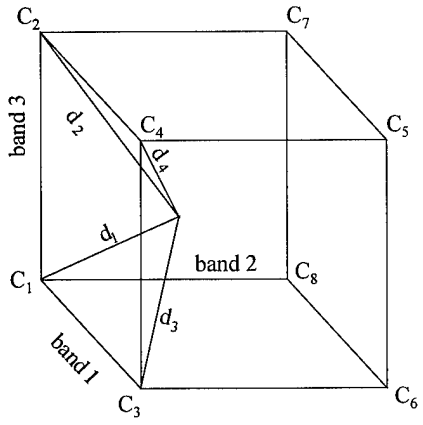


Figure 4.3.2
three-dimensional vector and
its feature space representation

As previously defined, the distances d_1 and d_2, from corners C_1 and C_2 to the point of interest are:

$$d_1 = \sqrt{x_1^2 + x_2^2 + x_3^2} \tag{4.3.3}$$

$$d_2 = \sqrt{x_1^2 + x_2^2 + (R - x_3^2)} \tag{4.3.4}$$

Complimentary equations for the other distances from the other reference corners of the

feature space can be easily derived following this same process. This process may be extended to higher dimensions where the feature vector is:

$$\bar{\mathbf{X}} = \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix}$$

and the distance measures can be generalized to:

$$d_i = \sqrt{\Sigma_{j=1}^{n} \left(x_j^2 * a_j + (R - x_j)^2 * b_j \right)} \tag{4.3.5}$$

where i is the reference "corner number" of the hypercube, j is the band number, n is the number of bands, and a_j and b_j are values necessary to calculate the distances to the corners of the n-dimensional hypercube. The values of a_j and b_j necessary to calculate the d_1 through d_4 measurements on the LANDSAT TM six-dimensional data are summarized in table 4.3.1.

Table 4.3.1 - values of a_j and b_j

	d_1		d_2		d_3		d_4	
TM Band	a_j	b_j	a_j	b_j	a_j	b_j	a_j	b_j
1	1	0	1	0	1	0	1	0
2	1	0	1	0	0	1	0	1
3	1	0	0	1	1	0	0	1
4	1	0	1	0	1	0	1	0
5	1	0	1	0	0	1	0	1
7	1	0	0	1	1	0	0	1

With these distance measures developed, we are now ready to derive the rest of the nPDF components. Consider the following equation:

$$nPDF_i = S * \frac{d_i}{2^{BIT}\sqrt{NB}} \tag{4.3.6}$$

where d_i is the distance from the "ith" corner to the point of interest as previously defined, *BIT* is the number of bits in the input data (8 for LANDSAT TM), *NB* is the number of bands being utilized (6 for LANDSAT TM), and *S* is a scale factor. Note that the ratio term in equation 4.3.6 returns a value between zero and slightly less than one. Multiplication by the scale factor allows the frequency perspective plot to be stretched so that finer details in the plot may be observed. The nPDF components are utilized in pairs to produce the nPDF, or frequency perspective, plots. By our previous definitions, 6 plots are possible utilizing the different distance measures arising from the following unique combinations: C_1-C_2, C_1-C_3, C_1-C_4, C_2-C_3, C_2-C_4, and C_3-C_4. Utilization of different principal corner pairs produces differing nPDF plots. By choosing different corner pairs, the separation between desired target classes may be increased to yield increased classification accuracy.

To illustrate the procedure of creating the nPDF plot and the ensuing classification process, a synthetic 3-dimensional training set consisting of 4 normally distributed classes was created. The data set ranges in value from 0 to 31. As previously defined, *R* is set equal to 31, *BIT* is set equal to 5 (because $2^5 = 32$), *NB*=3, and *S*=64. The scale factor dictates the size of the resulting nPDF plot, and in this case, a 64-by-64 plot will be created. Because of this a 64-by-64 array must be allocated and initialized to contain all zeros prior to any further calculations. The nPDF plot is most simply considered to be a height field and the corresponding array will be referenced by height_field[row][col]. The following algorithm in pseudocode will produce an nPDF plot for the synthetic data set utilizing the d_1 and d_4 measurements:

for row = 1 to num_rows_in_image
for col = 1 to num_cols_in_image

$$d_1 = \sqrt{\Sigma_{j=1}^{3}\left(x_j^2 * a_{1_j} + (R - x_j)^2 * b_{1_j}\right)}$$

$$d_4 = \sqrt{\Sigma_{j=1}^{3}\left(x_j^2 * a_{4_j} + (R - x_j)^2 * b_{4_j}\right)}$$

$$npdf_1 = S * \frac{d_1}{2^{BIT}\sqrt{NB}}$$

$$npdf_4 = S * \frac{d_4}{2^{BIT}\sqrt{NB}}$$

height_field[$npdf_1$][$npdf_4$] = height_field[$npdf_1$][$npdf_4$] + 1

next col
next row

where a_j and b_j reference the components of the following vectors as defined in table 4.3.1:

$$\bar{a}_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \qquad \bar{b}_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \qquad \bar{a}_4 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad \bar{b}_4 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Note that the algorithm essentially "counts up" the number of data elements (or pixels) in the training set that share the same $npdf_1$ and $npdf_4$ values and "bins" them together. This is accomplished by rounding off the calculated nPDF value to the nearest integer and using this value as the index for the height_field array which is then sequentially incremented. This process produces the height-field effect as previously discussed. The synthetic training data set was created with the four normally distributed classes, each with unit variance, and the following mean vectors:

$$\bar{m}_1 = \begin{pmatrix} 28 \\ 28 \\ 28 \end{pmatrix} \qquad \bar{m}_2 = \begin{pmatrix} 4 \\ 4 \\ 4 \end{pmatrix} \qquad \bar{m}_3 = \begin{pmatrix} 16 \\ 4 \\ 4 \end{pmatrix} \qquad \bar{m}_4 = \begin{pmatrix} 20 \\ 20 \\ 10 \end{pmatrix}$$

Table 4.3.2 lists the nPDF values for the mean vectors:

Table 4.3.2 - values of $npdf_1$ and $npdf_4$ for the synthetic data

class	$npdf_1$	$npdf_4$
1	56	33
2	8	44
3	20	48
4	35	36

We expect to see concentrations of pixels scattered around these coordinates in the nPDF plot. The height field and contour diagrams below were created by applying the nPDF algorithm to the synthetic data and utilizing the d_1 and d_4 measurements. Examining the plots confirms the previous conjecture as spikes in the height field and "blobs" in the contour plot are located at the previously calculated nPDF coordinates for the centers of the training class distributions.

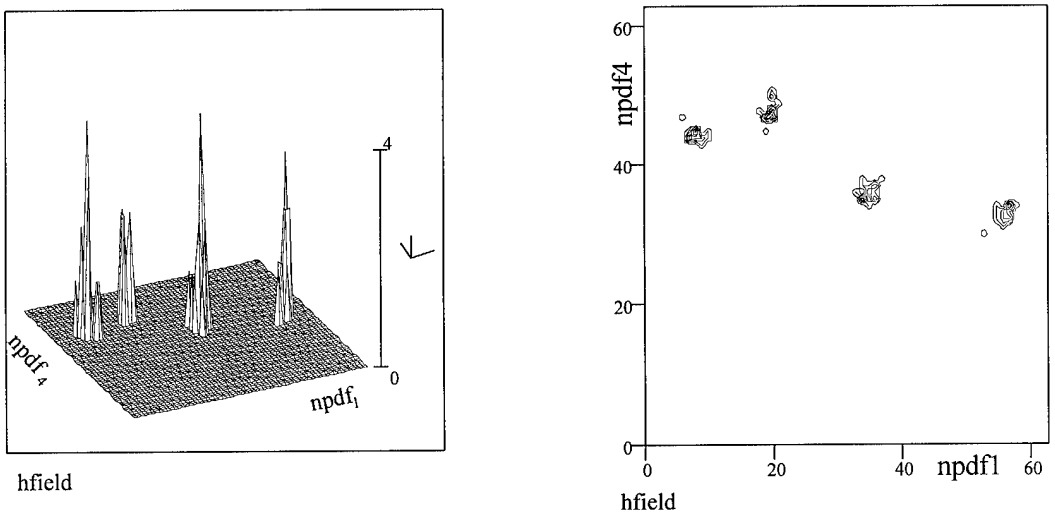


Figure 4.3.3 - nPDF height-field and contour plot

Once the location of the class mean vectors are calculated in the nPDF space, an image classification algorithm very similar to the minimum-distance-to-the-means can be easily developed. Consider the following image classifier logic:

$$\bar{\mathbf{x}} \in w_i \text{ if } d_{nPDF}(\bar{\mathbf{x}}, \bar{\mu}_i)^2 < d_{nPDF}(\bar{\mathbf{x}}, \bar{\mu}_j)^2 \text{ for all } i \neq j \qquad (4.3.7)$$

where $d_{nPDF}(\bar{\mathbf{x}}, \bar{\mu}_i)^2 = (\bar{\mathbf{x}} - \bar{\mu}_i)^T(\bar{\mathbf{x}} - \bar{\mu}_i)$. This is identical to the Euclidean distance measure, with the important exception that this calculation need be performed only in the two-dimensional nPDF feature space. While there is considerable computational savings in performing this measure in the space with reduced dimensionality, recall that the intensive projection calculations must be completed for every pixel in the image prior to its classification. As intuitively expected, the performance of this image classifier is very similar to the multidimensional minimum-distance-to-the-means algorithm, and is fraught with the same inability to account for data dispersion in the training classes.

After calculating the nPDF plot for the training classes, classification of the entire image can be accomplished in a unique manner. Consider the previously calculated contour plot that has been arbitrarily segmented as shown in Figure 4.3.4. The regions marked R_1 through R_4 correspond to the previously defined target classes. This mapping can be utilized as a lookup table (LUT) if we fill the array elements corresponding to a particular polygon with the numerical value of that class (a 1, 2, 3, or 4 in this simple example). The $npdf_1$ and $npdf_4$ values are calculated for each pixel in the original image. With these values, we simply look up the target classification class value and assign the pixel to it.

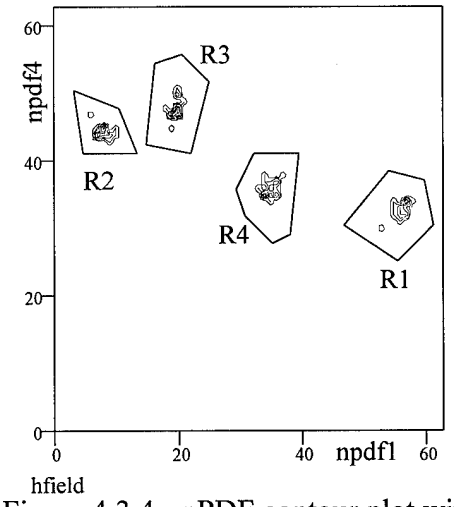
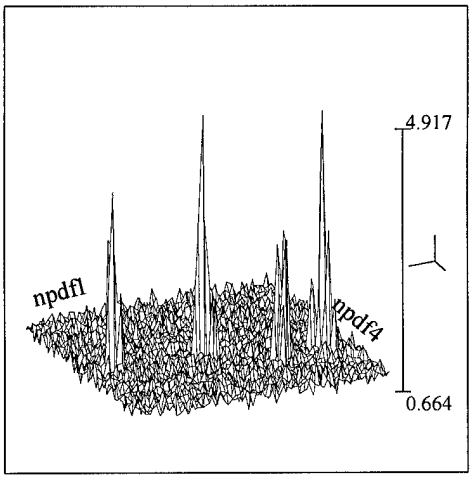


Figure 4.3.4 - nPDF contour plot with overlayed classification boundaries

This method of supervised classification, where a human specifies the classification boundaries in a projection of feature space, has some notable advantages. Foremost of these is that the classification performance depends entirely on the users selection of bounding polygons, which can be readily modified to account for subtle shape fluctuations in the data distribution that could be extremely difficult to model mathematically. Complicated and time consuming calculations of the multi-dimensional bounding volume or distance measures need not be completed as the simplified two-dimensional boundaries can be implemented as LUTs in the nPDF space. Also no difficult statistical calculations are required to determine if a pixel should be assigned to a "background" or "other" class as is necessary in GML classification. The most notable disadvantage of this method is the subjective placement of the boundaries which are difficult to reproduce accurately.

The nPDF algorithm also has considerable utility for estimating the number of target classes present in a data set. This information is very useful in cluster analysis, and is at the heart of unsupervised classification methodologies. Illustrating this facet of the algorithm is also quite simple. If the nPDF plot is computed for the entire image, we would expect that masses of pixels with similar spectral intensity vectors would generate high peaks in the plot. By simply counting the number of these spikes, a reasonable estimate of the number of target classification classes K can be made. This process is illustrated in Figure 4.3.5, which was created by calculating an nPDF plot for an



image_hfield

Figure 4.3.5 - nPDF height-field plot of 4 clusters in the presence of noise

"entire image" of the synthetic data set, to which some random "background" noise was added. The random noise appears as small data fluctuations at the base of the diagram while the four peaks represent the major data clusters in the data set.

4.4 Fuzzy ARTMAP Neural Network

A review of the current literature in the field of remote sensing will indicate that many neural-network architectures have been utilized for classifying image data. The chief advantage of utilizing neural networks for classification arises from the fact that a single flexible learning algorithm is able to derive an optimal decision rule for a given situation from the training data. The chief problems of this family of algorithms are long learning times and the inability to deal with "fuzzy" data. The term "fuzzy" relates the concept of the extent to which a given feature is present in an exemplar. Terms such as "close to home", "much greater than five", and "quite young", are linguistic examples that portray this concept. Human beings are naturally fuzzy in their decision-making processes. While it is quite simple for a graduate student to label a given instructor as a "good teacher", it has traditionally been impossible to provide a computer with the tools to make this same type of classification. Neural networks are typically implemented on general-purpose computer hardware and are hindered by their inherent binary nature. In the mid 1960's, L. A. Zadeh derived the mathematical groundwork for a field of study that would become known as fuzzy logic. Armed with these new tools, engineers and researchers have utilized these concepts to permit the machine computation of fuzzy data.

An advanced neural network, the fuzzy ARTMAP, attacks both traditional shortcomings of neural networks in a very direct manner. Conventional neural network methodologies typically require training data to be applied repeatedly, often thousands of times, to determine a set of decision rules that produce a desired level of performance. By comparison, the Adaptive Resonance Theory and MAPping (ARTMAP) approach can perform the same classification to the same level of accuracy after only a handful of training cycles, or epochs. The introduction of fuzzy logic to this family of algorithms greatly increases their flexibility because features that previously could only be said to be present or absent can now be more completely described. When dealing exclusively with

binary data, the algorithm reduces to its crisp set theory predecessors. The overview in section 4.4 essentially summarizes that of Carpenter *et al.*, (1992).

4.4 Overview of fuzzy ARTMAP Neural Network Architecture

Consider the overall system diagram for the neural network of interest:

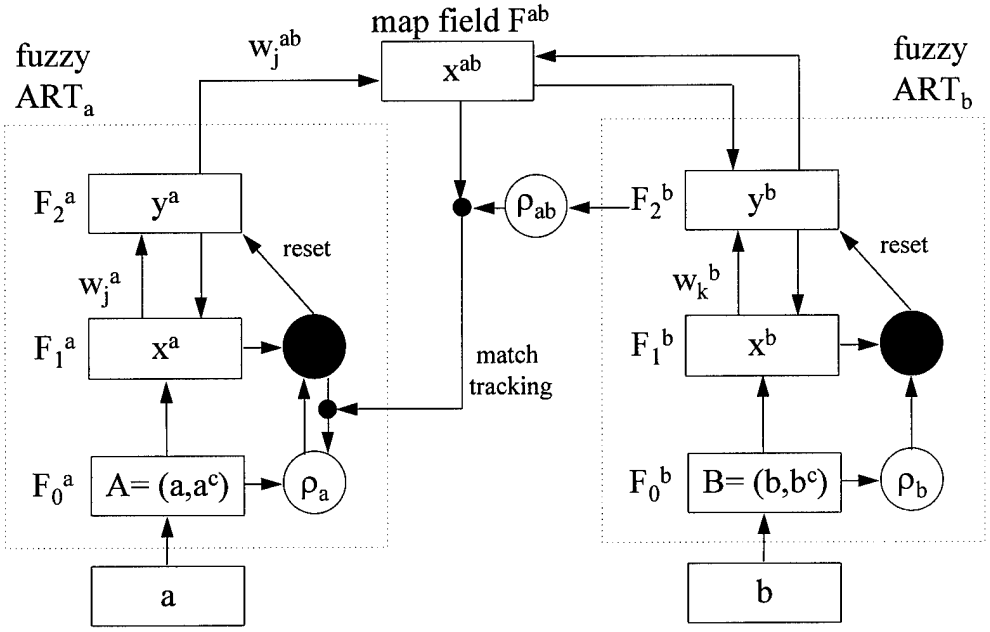


Figure 4.4.1 - Fuzzy ARTMAP Architecture

This neural network is not as complex as it may appear. The network must be trained before the LANDSAT TM images can be classified. This is accomplished by applying the spectral intensity vectors of the training-class pixels at point **a**, while the corresponding label or target class will be applied simultaneously at point **b**. An important calculation, known as *complement coding*, takes place on the input data in the *preprocessing fields* F_0^a and F_0^b. Initially the input data, (represented here as the vector $\bar{\mathbf{I}}$), must have its elements scaled between 0 and 1. Mathematically we can describe the rescaling operation as:

$$\bar{\mathbf{I}} = \frac{\bar{\mathbf{a}}_i - \min}{\max - \min} \tag{4.4.1}$$

Where $\bar{\mathbf{a}}_i$ represents an individual element of the intensity vector, and *max* and *min* denote the maximal and minimal data values. The rescaling operation can be accomplished with either global or local maxima and minima. In this study, to account for all possible occurrences, the minimum digital count value will be set to 0 and the maximum value to 255. Note that the rescaling operation preserves amplitude information and provides a very useful mathematical view of the world. Consider the following pixel intensity vector and its normalized variation, $\bar{\mathbf{I}}_{norm}$.

$$\bar{\mathbf{I}} = \begin{bmatrix} 80 \\ 20 \\ 144 \\ 36 \\ 71 \\ 54 \end{bmatrix} \qquad \bar{\mathbf{I}}_{norm} = \begin{bmatrix} 0.314 \\ 0.078 \\ 0.565 \\ 0.141 \\ 0.278 \\ 0.212 \end{bmatrix}$$

The *complement* of the input vector must also be calculated. This can be mathematically defined as:

$$a_i^c = 1 - a_i \tag{4.4.2}$$

$$\text{and we redefine } \bar{\mathbf{I}} \equiv (a, a^c) \tag{4.4.3}$$

Accomplishing the complementation operation on the input pixel results in:

$$\bar{\mathbf{I}} = \begin{bmatrix} 0.314 & 0.686 \\ 0.078 & 0.922 \\ 0.565 & 0.435 \\ 0.141 & 0.859 \\ 0.278 & 0.722 \\ 0.212 & 0.788 \end{bmatrix}$$

The data at the *input field* of ART_a, F_1^a, is now a normalized and complemented vector

with 2N elements, where N is the number of bands. A value representing the target class with 2M elements, where M is the number of target classes, is similarly represented as a vector in input field F_1^b. The target class data label for this field must be encoded in a binary manner. If the previously defined training class examples are used, we will encode grass as 0001, water as 0010, pine trees as 0100, and deciduous trees as 1000. Their complements obviously are 1110, 1101, 1011 and 0111, respectively. As previously mentioned, the fuzzy ARTMAP reduces to a crisp set theory implementation in the presence of binary data. It is interesting to note that in this study that the ARTa module will be utilizing fuzzy logic, while its linked sister module will perform similar calculations using traditional crisp set theory mathematics.

The adjustment of the weights that provide the system's *long-term memory* will now be discussed. As in a regular ART network, all nodes in the input field are fully interconnected, and the F_1 activity vector will be represented by: $\bar{\mathbf{x}} = (x_1, ...x_{2N})$. Similarly, the *classification vector*, or field activity vector F_2, will be represented by: $\bar{\mathbf{y}} = (y_1 ....y_{2L})$. Note that the number of nodes in either field can be arbitrary, but there must be more nodes in the classification field than classification categories. We will also define the weight vector (or long term memory trace) between the "*j*th" node of F_2 and all nodes in the input field as: $\bar{\mathbf{w}}_j = (w_{j1}, ...w_{j2N})$. Note that, while the crisp-set theory

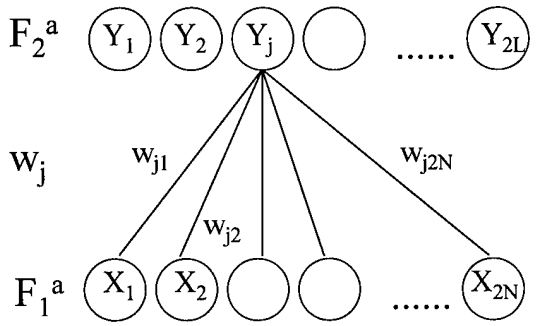


Figure 4.4.2 - depiction of the weight vector $\bar{\mathbf{w}}_j$
between the input field (F_1^a) and classification field (F_2^a)

implementation of the ARTMAP algorithm has both a bottom-up and a top-down weight vector, the fuzzy ARTMAP has only this single weight vector for each node. Initially this weight vector has all elements set to unity. Figure 4.4.2 illustrates the dynamics between the F_1 and F_2 fields.

As in a regular adaptive resonance theory network, the role of the classification field is to determine a "winning" node and pass this information to the rest of the system. In the fuzzy ARTMAP, this category choice is accomplished through the use of fuzzy mathematics. Consider the following *category choice* function:

$$T_j(\bar{\mathbf{I}}) = \frac{|\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_j|}{\alpha + |\bar{\mathbf{w}}_j|} \tag{4.4.4}$$

where "|.|" denotes the norm operator, which is defined as:

$$|\bar{\mathbf{a}}| = \sum_{i=1}^{N} \bar{\mathbf{a}}_i \tag{4.4.5}$$

The α term is a *choice* parameter which is set to be greater than zero. The fuzzy AND operator " $\wedge$" is a relative of the logical AND operator and simply returns a vector whose elements are the minimum value of the corresponding elements in the set of vectors being processed. This "min" operation is easily defined as:

$$(\bar{\mathbf{p}} \wedge \bar{\mathbf{q}})_i = \min(\mathbf{p}_i, \mathbf{q}_i) \tag{4.4.6}$$

Mathematically speaking, equation 4.4.4 determines the degree to which the weight vector is a *fuzzy subset* of the input vector. Note that when the weight and input vectors are very similar, $(\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_j) \cong \bar{\mathbf{w}}_j$ and T_j will approach unity, implying that node J is the best choice. A category choice is made when only one F_2 node becomes *activated.* A node is said to have become activated when it has a value of one and all others have a value of zero. This operation is often referred to as "winner take all". The chosen node index J is governed by the following simple relation:

$$J = \max\{T_j(\bar{\mathbf{I}}) : j = 1, 2...2L\} \tag{4.4.7}$$

In the event of a tie, the node with the lowest numerical value is chosen. When the signal from the input and classification fields reinforce one another, resonance occurs. This process occurs when:

$$\frac{|\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_j|}{|\bar{\mathbf{I}}|} \geq \rho \tag{4.4.8}$$

where ρ is the *vigilance parameter*. As ρ is increased, the algorithm has to be more and more certain of the of assignment of a particular exemplar to a specific class before classification is executed. This is exactly as mathematically stated in equation 4.4.8. Technically, the equation returns a measure of the degree to which the input vector is a fuzzy subset of the weight vector. More intuitively put, note that the proportion only becomes large (near unity) when the input and weight vectors have similar values. This is exactly why the weight vector is also termed a "long-term memory trace". The weight vector of each node essentially "learns" (defines) a multidimensional region that is populated solely by vectors from one target class. After the network is trained, new input vectors located within the bounds of this region are assigned to that class. Whenever an input pixel is determined to reside within the bounds of or close to a target region as determined by the vigilance parameter, resonance will occur between the input and classification fields and the same F_2 node will be activated. Learning will occur at this point, and this process is governed by the relation:

$$\bar{\mathbf{w}}_j^{new} = \beta(\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_j^{old}) + (1-\beta)\bar{\mathbf{w}}_j^{old} \tag{4.4.9}$$

Note that when the weight vector is essentially a fuzzy subset of the input vector, then $\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_j \cong \bar{\mathbf{w}}_j$ and little "learning" will occur. This is desirable and leads to enhanced stability. The parameter β is termed a *learning-rate parameter* and is bounded between zero and one. For fast learning, $\beta \approx 1$. This makes the weights closely track the input vectors, and is useful when initially training the network. For fast encoding of noisy data sets (the typical real-world situation) the learning parameter is set to 1 until a node becomes initially activated. When a node is initially assigned to represent a target

classification class, it is said to have been *committed.* Once a node has been committed, the value of β is reduced. Though not inherently obvious at this point, it is important to note that the elements of the weight vectors can only decrease monotonically. This is directly related to classification boundaries expanding in feature space. This important facet of the algorithm will be highlighted in a simple two-dimensional and two-class example to follow.

If a pixel is determined not to fall within or close to a particular classification region, resonance will not occur. The degree of "closeness" to the classification region is user definable through the vigilance parameter. Resonance does not occur when the *vigilance criterion* (equation 4.4.8) is not met. This occurs when:

$$\frac{|\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_j|}{|\bar{\mathbf{I}}|} < \rho \tag{4.4.10}$$

When this happens, *mismatch reset* is said to have occurred, and a search operation begins. The algorithm determines if the pixel is close to or within any of the regions defined by the weight vectors of the F_2 nodes. If no suitable region can be found, another F_2 node can be committed or a new node can be created. When a new node is created, another classification region is defined internally. Once resonance is attained, learning can be accomplished by altering the node's weight vector as with the learning rule as previously described.

For insight into how this algorithm divides up feature space, consider the following two-dimensional example. Given a general feature vector, the preprocessing, or fields F_0, will produce the normalized four-dimensional vector:

$$\bar{\mathbf{I}} = (\bar{\mathbf{a}}, \bar{\mathbf{a}}^c) = (a_1, a_2, 1 - a_1, 1 - a_2) \tag{4.4.11}$$

As expected, the weight vector will attempt to map these input values to a region of feature space through application of the learning rule. The weight vector, in complement coded form, can then be expressed as:

$$\bar{\mathbf{w}}_j = (u_j, v_j^c) \tag{4.4.12}$$

Geometrically speaking, the weight vector can be thought of as describing a rectangle in feature space. Consider the diagram in Figure 4.4.3. We will assign the name R_j to the region near the center of feature space. A measure of its size can be computed by adding its height and width as follows:

$$|R_j| = |v_j - u_j| \tag{4.4.13}$$

During learning, the region R_j grows as the elements of its associated weight vector decrease monotonically. Consider the case where a new pixel (denoted as $\bar{\mathbf{a}}$) meets category choice criteria for this same region. This will cause the very same classification field node to become active, and the elements of the weight vector will begin to monotonically decrease during learning. The algorithm seeks to determine the minimum-sized rectangle in feature space that will encompass all of the input vectors that cause this node to become active. At this point, we will introduce the fuzzy OR operator (denoted by the "$\vee$"). As might be expected, it is a relative of the crisp OR operator. Unlike the fuzzy AND operator, this process returns the maximum value of corresponding elements of the vectors. In feature space, the previously defined rectangle has enlarged (Figure 4.4.4) to reflect the decrease in the weight vector's elements and the assignment of the new data point to this class. The weight vector associated with this new region is then:

$$\bar{\mathbf{w}}_j = (\bar{\mathbf{a}} \wedge u_j, (\bar{\mathbf{a}} \vee v_j)^c) \tag{4.4.14}$$

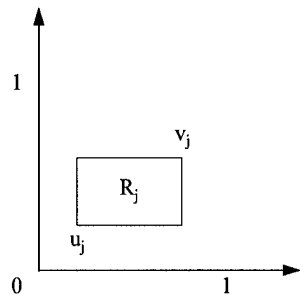


Figure 4.4.3 - weight vector represented as rectangle R_j in feature spac

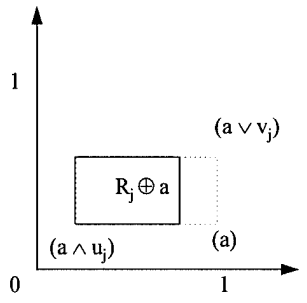


Figure 4.4.4 - region R_j has "grown" to include vector $\bar{\mathbf{a}}$

It is important to note that the weight vector essentially has "learned" the lower left and upper right corners of the new region. As already defined, the size of this new region is:

$$|R_j \oplus \bar{\mathbf{a}}| = |(\bar{\mathbf{a}} \vee v_j) - (\bar{\mathbf{a}} \wedge u_j)| \tag{4.4.15}$$

and the $\oplus$ operator implies a bounded sum which cannot exceed some value. This process must be bounded by the vigilance parameter to prevent the entire feature space from being enclosed by a single rectangle:

$$|R_j| \leq (1-\rho)N \tag{4.4.16}$$

where N is the number of bands or dimensions being used (2 in this case). Note that as the vigilance parameter ρ is increased, the feature space is segmented by smaller and smaller rectangles and thus becomes more and more finely granularized.

In general, for an M-dimensional vector and given $\bar{\mathbf{A}}$, the set of all vectors that activate the node of interest, the following fuzzy-set statements can be made concerning the class' hyper-rectangle in feature space. The vertices of the hyper-rectangle are represented by the minimum and maximum values of the individual elements that encompass the vectors that define class $\bar{\mathbf{A}}$. Mathematically, this is simply:

$$u_j = (\wedge_j \bar{\mathbf{A}})_i = \min\{\bar{\mathbf{A}}_i\} \text{ and similalry, } v_j = (\vee_j \bar{\mathbf{A}})_i = \max\{\bar{\mathbf{A}}_i\} \tag{4.4.17}$$

Recall that in the preceding two-dimensional case, u_j and v_j were the lower left and upper right corners respectively. It follows that the size as defined in equation 4.4.13 of the corresponding hyper-rectangle is:

$$|R_j| = |\vee_j \bar{\mathbf{A}} - \wedge_j \bar{\mathbf{A}}| \tag{4.4.18}$$

The corresponding weight vector can therefore be expressed as:

$$\bar{\mathbf{w}}_j = (\wedge_j \bar{\mathbf{A}}, (\vee_j \bar{\mathbf{A}})^c) \tag{4.4.19}$$

Applying the norm operator to this results in:

$$|\bar{\mathbf{w}}_j| = \Sigma_i(\wedge_j \bar{\mathbf{A}}) + \Sigma_i(1 - (\wedge_j \bar{\mathbf{A}}) = M - |\vee_j \bar{\mathbf{a}} - \wedge_j \bar{\mathbf{a}}| \tag{4.4.20}$$

Substituting into the size equation (4.4.13) for the hyper-rectangle yields:

$$|R_j| = M - |\bar{\mathbf{w}}_j| \tag{4.4.21}$$

Realizing that $|\bar{\mathbf{w}}_j| \geq \rho N$, we obtain the very useful realization:

$$|R_j| \leq (1-\rho)N \tag{4.4.22}$$

which was utilized in the preceding example (eq. 4.4.16) without the benefit of this proof. It is worth mentioning again that this important result states that increasing the vigilance parameter ρ results in a decrease in the size of the corresponding target classification hyper-rectangles decrease in size. This produces a finely partitioned feature space with many classification hyper-rectangles. In feature space, this entire process basically learns an exciting variation of the classical stacked parallelepiped classification algorithm (Richards, 1993). The most important difference between the two implementations is that the fuzzy ARTMAP implementation permits what is known as *exception handling,* where separately recognizable subclasses are defined within larger classes. This aspect of the algorithm is depicted in Figure 4.4.5.

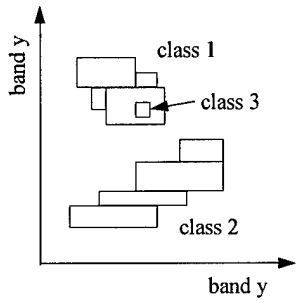


Figure 4.4.5 - stacked classification rectangles
with exception handling in feature space

Illustration of a two-dimensional two-class example would demonstrate the utility of the previously defined category choice, vigilance criteria, and learning/weight adjustment rules. The two classes in this example will be water and pine trees and only the "blue" and "green" bands will be used. Consider pixels with the following values:

class	blue digital counts	green digital counts
water	230	26
pine	61	191
pine	71	204

After normalization and complement coding, the pixels are represented by the following vectors:

class	vector	normalized blue digital counts	normalized green digital counts	complement of normalized blue	complement of normalized green
water	I_1	0.9	0.1	0.1	0.9
pine	I_2	0.24	0.75	0.76	0.25
pine	I_3	0.3	0.8	0.7	0.2

We will assume that the network is being initially trained. Category choice, or which classification node will become activated, is given by:

$$T_j(\bar{\mathbf{I}}) = \frac{|\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_j|}{\alpha + |\bar{\mathbf{w}}_j|} \tag{4.4.23}$$

Recall that all of the initial elements of the weight vector have a value of unity, and that the choice parameter α, is some small positive number. This results in:

$$T_j(\bar{\mathbf{I}}_1) = \frac{|\bar{\mathbf{I}}_1 \wedge \bar{\mathbf{w}}_j|}{\alpha + |\bar{\mathbf{w}}_j|} \cong 1 \text{ for j=1 to 2L} \tag{4.4.24}$$

which means that node one of the classification field will become active as the lowest numerical value of j is always chosen. Resonance occurs when the input and classification fields reinforce one another. This occurs when:

$$\frac{|\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_1|}{|\bar{\mathbf{I}}|} \geq \rho \tag{4.4.25}$$

We will choose to set the vigilance parameter $\rho = 0.9$. Since the node was previously uncommitted, resonance will occur because:

$$\frac{|\bar{\mathbf{I}} \wedge \bar{\mathbf{w}}_1|}{|\bar{\mathbf{I}}|} = 1 \geq \rho \tag{4.4.26}$$

Learning will now ensue. As previously stated, it is governed by the learning rule (equation 4.4.9). Given that we are in fast learning mode ($\beta = 1$) this results in the following calculation and weight vector:

$$\bar{\mathbf{w}}_1^{new} = \beta(\bar{\mathbf{I}}_1 \wedge \bar{\mathbf{w}}_1^{old}) + (1-\beta)\bar{\mathbf{w}}_1^{old} = [0.9, 0.1, 0.1, 0.9] \tag{4.4.27}$$

Note that this is exactly the input vector! If we perform similar calculations for the first pine tree pixel, we will find that resonance will not occur on the first node due to the value of the vigilance parameter. Due to this and the previously stated conditions, the second classification node will become active and its weight vector will be set equal to:

$$\bar{\mathbf{w}}_2^{new} = \beta(\bar{\mathbf{I}}_2 \wedge \bar{\mathbf{w}}_2^{old}) + (1-\beta)\bar{\mathbf{w}}_2^{old} = [0.24, 0.75, 0.76, 0.25] \tag{4.4.28}$$

Once again, the weight vector becomes the input vector. Recall that in feature space these weight vectors each define a two-dimensional rectangle. The maximum and minimum (lower left and upper right) coordinate pairs of the vertices of these rectangles can be calculated by simply taking the first two components of the weight vector and complement coding the last two elements. This results in two points in the two-dimensional feature space:

$$(u_1, v_1^c) = [0.9, 0.1, 0.9, 0.1]$$

$$\text{and similarly, } (u_2, v_2^c) = [0.24, 0.75, 0.24, 0.75] \tag{4.4.29}$$

Recall that the weight vectors can only decrease monotonically, and as they decrease the corresponding classification rectangle in feature space grows. When the next tree pixel is evaluated by the network, resonance does not occur with the first classification node due to the vigilance parameter, however resonance does occur with the second node as $\rho = 0.945 > 0.9$. As discussed earlier, learning will now occur resulting in the following new weight vector:

$$\bar{\mathbf{w}}_2^{new} = \beta(\bar{\mathbf{I}}_3 \wedge \bar{\mathbf{w}}_2^{old}) + (1-\beta)\bar{\mathbf{w}}_2^{old} = [0.24, 0.75, 0.70, 0.20] \tag{4.4.30}$$

and this weight vector defines a classification rectangle with the following coordinates:

$$(u_2, v_2^c) = [0.24, 0.75, 0.3, 0.8] \tag{4.4.31}$$

But before the previous assertion can be made, we must check that the rectangle does not exceed the maximum allowed size given by:

$$|R_2| \leq (1-\rho)M = [0.3 - 0.24 + 0.8 - 0.75] < (1-0.9) * 2 \tag{4.4.32}$$

and we find that $0.11 < 0.2$, so the rectangle grows within allowable size limits. Note that in feature space (Figure 4.4.6) the region describing the pine class has grown the minimum amount necessary to surround both pixels that belong to this class.

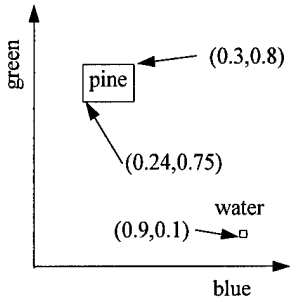


Figure 4.4.6 - two-dimensional feature space representation of the pine and water weight vectors

The inter-ART field, represented as map field F^{ab} in Figure 4.4.7, has two missions. First, it maps the classification from ART_a to the classification output of ART_b, and secondly it realizes the match tracking rule. The dynamics of the map field are illustrated in Figure 4.4.7 below.

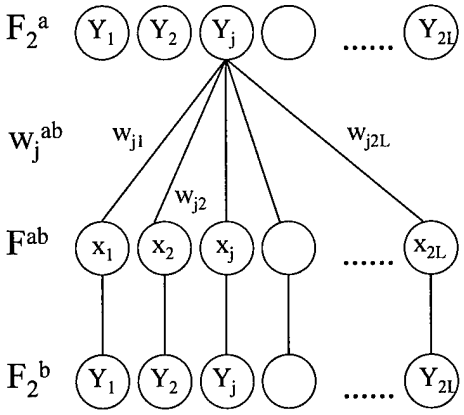


Figure 4.4.7 - Representation of the dynamics of the inter-ART field

We are initially interested in what occurs when the ART_a and ART_b modules are active and in agreement. When this occurs x^{ab} (the output vector from field F^{ab}) becomes:

$$\bar{\mathbf{x}}^{ab} = \bar{\mathbf{y}}^{b} \wedge \bar{\mathbf{w}}_J^{ab} \tag{4.4.33}$$

which should be interpreted as the fuzzy AND of the classification output from the ART_b module and the weights between ART_a's *J*th classification field node and the map field. Note that all components of this weight vector, like the others previously discussed, are initially set equal to one (the one-to-one mapping between F^{ab} and $F_2^{\,b}$ is always accomplished with unity gain). Fuzzy ANDing the classification field weight vector with the classification result from ART_b permits the network to derive a mapping from the input vectors that activate the same node in ART_a to the correct classification node in ART_b. Once node J in $F_2^{\,a}$ learns to predict node K in F^{ab}, one element of the weight vector between them is set to one for all time. In our notation, this rule can be represented as:

$$w_{JK}^{ab} = 1 \tag{4.4.34}$$

It is crucial to note that the activation of different nodes in $F_2^{\,a}$ may be mapped to the same output classification class. This permits the mapping of "many to one". Understanding this result is vital to understanding one of the greatest strengths of this network. Even though many different grass pixels may activate different classification nodes in ART_a (and therefore must be located in different hyper-rectangles in feature space) they are nevertheless still part of the same classification class and will be correctly mapped to it. The "many-to-one mapping" is depicted in Figure 4.4.8 on the following page.

When there is a mismatch during training between the output of ART_a and the correct classification of ART_b, *match tracking* occurs. This is mathematically triggered when:

$$\left|\bar{\mathbf{x}}^{ab}\right| < \rho_{ab}\left|\bar{\mathbf{y}}^{b}\right| \tag{4.4.35}$$

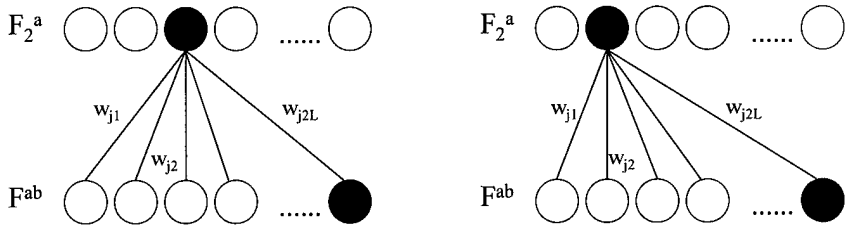


Figure 4.4.8 - Graphical representation of the mapping of many classification nodes to the same output classification class

The match tracking rule then increases ART_a's vigilance parameter ρ_a until the correct node in ART_a is activated, and this occurs when:

$$|\bar{\mathbf{x}}^a| = \left|\bar{\mathbf{I}} \wedge w_J^a\right| \geq \rho_a \left|\bar{\mathbf{I}}\right| \tag{4.4.36}$$

and this will drive the output of the field map to be:

$$\left|\bar{\mathbf{x}}^{ab}\right| = \left|\bar{\mathbf{y}}^b \wedge w_J^{ab}\right| \geq \rho_{ab} \left|\bar{\mathbf{y}}^b\right| \tag{4.4.37}$$

In the event that no node can be found that satisfies these equations, all of the nodes in F_2^a are set to zero. In essence, by shutting down all classification nodes due to a pixel that does not map to any of the hyper-rectangles in feature space, the neural network is responding, "I don't know". This powerful result will cause pixels that are not in any of the classes to be mapped to the background class.

Once the network is suitably trained, the ART_b module is disconnected. Pixels from the image are sequentially presented to the preprocessing fields of ART_a and their resulting classification is read at the output of the map field. At classification time, the output of the map field is:

$$\bar{\mathbf{x}}^{ab} = w_J^{ab} \tag{4.4.38}$$

which is simply the weight vector between F_2^a and F^{ab}. Recall that this vector will have all of its elements equal to zero except for one whose value will be one. This vector simply contains the encoded classification class. When all of the nodes in F_2^a are equal to zero, the output from the map field is:

$$\bar{\mathbf{x}}^{ab} = 0 \tag{4.4.39}$$

which implies a result of "I don't know" or assignment to the background class.

The fuzzy ARTMAP architecture is uniquely suited to classifying remotely sensed images. No other neural networks combine the great strengths of ARTMAP, such as ART dynamics, exception handling capability, and the ability to effectively deal with analog data in such a stable and rapid learning environment.

4.5 Reporting the Results

As previously discussed, the simple LANDSAT TM image in Figure 4.5.1 is composed of four major classes, water, grass, deciduous trees, and pine trees. Assume that the image is 100 pixels by 100 pixels. We will also assume that 15% of the image (1,500 pixels) is composed of deciduous trees, 25% is made up of pine trees (2,500), 20% is water (2,000), and the remaining 40% (4,000) is made up of grass.

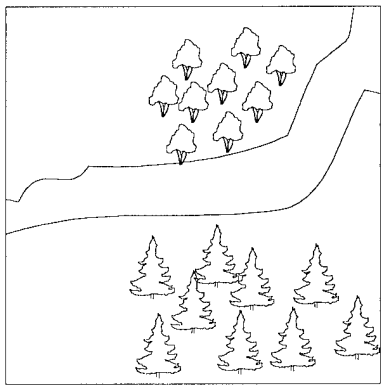


Figure 4.5.1 - depiction of LANDSAT TM scene

The accuracy of a particular classification algorithm often is illustrated through the use of a simple mathematical construct termed a *confusion* or *error* matrix. Consider the confusion matrix in Figure 4.5.2, which represents the results attained by segmenting the artificial LANDSAT TM scene with a particular classification algorithm.

	grass	water	pine	leaf	classification total
grass	3,885	0	20	5	3,910
water	0	2,000	0	0	2,000
pine	90	0	1,985	392	3,178
leaf	25	0	495	1,103	912
ground truth total	4,000	2,000	2,500	1,500	10,000

Figure 4.5.2 - a simple confusion matrix

The entries along the bottom of the matrix represent the ground truth classes and the entries along the right edge represent the classification as performed by the algorithm. The classification total represents the sum across a particular row, while the ground truth total is calculated by summing down each column. The results in this matrix can lead to some important conclusions. The "grass" column indicates that of the 4,000 ground truth grass pixels, 3,885 were correctly classified, 90 were incorrectly classified as pine trees, and 25 were also incorrectly classified as deciduous (leaf) trees. Note that none of the grass pixels were incorrectly classified as water. This should not be surprising as their respective mean vectors are probably widely separated, leading to low classification error and little confusion. This is not true for the pine and leaf columns, where considerable confusion (between 20 and 26 percent) is evident between the two tree types. This confusion is not unexpected as the two mean vectors are likely to be separated only slightly, leading to some misclassification.

The overall accuracy of a classification algorithm often is reported as a *simple accuracy.* This metric is calculated by summing the correctly classified pixels along the diagonal of the confusion matrix and dividing by the total number of pixels. For the preceding confusion matrix, this results in a simple accuracy measurement of:

$$\frac{3885+2000+1985+1103}{10000} = 89.7\% \qquad (4.5.1)$$

Note that this measure does not account for any off-diagonal terms in the confusion matrix. Also note that this figure of merit is artificially inflated due to the highly accurate, although easily accomplished, segmentation of the water pixels.

In an attempt to account for varying class content in an image, the *weighted accuracy* measurement has been proposed. It is calculated by first dividing the number of correctly classified pixels by the number pixels present in the class. This term is then divided by the number of classes present in the image. In the preceding case, this classification accuracy would be calculated and reported as:

$$\frac{\frac{3885}{4000}+\frac{2000}{2000}+\frac{1985}{2500}+\frac{1103}{1500}}{4} = 87.5\% \qquad (4.5.2)$$

Also note that this figure does not account for the off-diagonal terms in the confusion matrix.

To combat the problems associated with the simplistic computation of classification accuracy just described, a measure will be introduced that accounts for the off-diagonal terms, but compensates for chance agreement. This measurement is termed the *kappa coefficient* and is denoted $\hat{\kappa}$. The development that follows essentially summarizes the development in Rosenfield and Fitzpatrick-Lins (1986) with the considerations for other coefficients coming from Foody (1992).

The confusion matrix in Figure 4.5.3 is identical to that in Figure 4.5.2 with the addition of the simple row and column marginals:

	grass	water	pine	leaf	classification total	row marginal $P_r(i)$
grass	3,885	0	20	5	3,910	0.39
water	0	2,000	0	0	2,000	0.2
pine	90	0	1,985	392	2,467	0.25
leaf	25	0	495	1,103	1,623	0.16
ground truth total	4,000	2,000	2,500	1,500	10,000	
column marginal $P_c(i)$	0.4	0.2	0.25	0.15		

Figure 4.5.3 - a simple confusion matrix
with row and column marginals added

The marginals, P_r and P_c, are simply calculated by dividing the sum across a row or down a column by the total number of data elements. With these simple calculations, the kappa coefficient can be calculated as follows:

$$\hat{\kappa} = \frac{P_o - P_E}{1 - P_E} \tag{4.5.3}$$

where the observed proportion of agreement P_O is defined as the proportion of the correctly classified pixels. Completing these calculations for the previously defined example results in:

$$P_O = \frac{3885+2000+1985+1103}{10000} = .8973 \tag{4.5.4}$$

This is simply the sum of the diagonal terms of the confusion matrix divided by the total number of data elements. The proportion of agreement due to chance, P_E, is defined as:

$$P_E = \Sigma_{i=1}^{M} P_r(i) P_c(i) \tag{4.5.5}$$

where M is the total number of target classification classes (4 in this case). In this case:

$$P_E = \Sigma_{i=1}^{M} P_r(i) P_c(i) = 0.39 * 0.4 + 0.2 * 0.2 + 0.25 * 0.25 + 0.16 * 0.15 = 0.2825 \tag{4.5.6}$$

Substitution yields a value for the kappa coefficient:

$$\hat{\kappa} = \frac{P_o - P_E}{1 - P_E} = \frac{.8973 - 0.2825}{1 - 0.2825} = 0.8569 \tag{4.5.7}$$

This value typically is multiplied by 100 and presented as a percentage. This results in a kappa coefficient of 85.69%, somewhat lower than the 87.5% weighted classification accuracy as previously calculated for the same confusion matrix.

Foody (1992) notes that P_E is calculated from the diagonal terms of the confusion matrix. These terms denote actual agreement, and therefore improperly inflate the proportion of agreement due to chance. He suggests utilizing the following measurement; named Brennan and Prediger's Kappa after its creators:

$$k_{B\&P} = \frac{P_O - \frac{1}{M}}{1 - \frac{1}{M}} \tag{4.5.8}$$

where M once again represents the total number of target classification classes. The 1/M terms can be justified by the argument that the marginal terms in a typical image classification algorithm are free parameters and not fixed *a priori*. Therefore the

probability of chance agreement reduces simply to 1/M. In this example, the metric has a value of:

$$k_{B\&P} = \frac{P_O - \frac{1}{M}}{1 - \frac{1}{M}} = \frac{.8973 - \frac{1}{4}}{1 - \frac{1}{4}} = .8631 \tag{4.5.9}$$

which is slightly larger than the kappa coefficient. Intuitively, the larger value makes sense, as we have removed some of the actual agreement which previously contributed to the probability of agreement due only to chance. This study will calculate both the classical and Brennan and Prediger's kappa coefficient for each image and each classification methodology and compare their results in actual use.

Classification accuracy obviously is one of the chief concerns in this study, but other metrics were applied to the various segmentation algorithms. They also were compared in terms of execution time required for image segmentation. The time function of the computer was queried prior to entering and after completing the segmentation phase of each algorithm. The difference between these two values is published with the results of each classification run. Note that the training time for the neural network will be included in this measure, while the creation of bounding polygons for the nPDF space will not. The creation time of the nPDF classification polygons requires direct user intervention and considerable care to produce accurate results. If possible, the various modules will also be "profiled" to determine where most of the computing time is spent. The end result of compiling and presenting these various measurements will produce a "snapshot" of the best conditions and related difficulties of the differing approaches.

5.0 Using the Classification Modules in the AVS Environment

The Advanced Visualization System from Advanced Visual Systems was chosen to support this study for a number of reasons. Most importantly, this system provides an extremely powerful and user extensible data visualization and image computing environment for the programmer. A competent "C" programmer can expect to be able to write AVS modules in no more than a couple of weeks. The very rapid ramp-up time can be attributed to relieving the programmer of the considerable intricacies of the X-Windows graphical environment and simplifying the problems of data transfer between the modules. As such, virtually all modules for the environment are composed of two distinct code sections. The first defines the interfaces with other modules and how control parameters are passed to the second code section. This second section, the compute function, is almost entirely composed of standard C calls, the only exceptions being the routines to access shared data between the modules. The complete source code for each module developed for this study is included in appendix A of this report.

Possibly the most attractive feature of the AVS environment is that computational chains can be readily represented as "networks" or "procedures" of simple modules. Imaging operations that traditionally would take considerable time to uniquely code by hand, either in a low-level programming language or a mathematics package, can be created by merely connecting a few modules into a network. In addition, many modules for common imaging operations, such as reading, writing, and displaying standard image formats, have already been developed and tested. This study extended the use of AVS at RIT into image classification, but it required only a handful of specialized modules. All of the inherently compatible base functionality already had been developed.

In the following sections of this report, a representative processing network for each of the classification operations will be described along with detailed operating

instructions for each of the classification modules. As such, this chapter can serve as a user's guide to image processing with the supplied modules.

5.1 Collecting Training Data in the AVS Environment

Two separate modules and associated networks were constructed for this study to support the collection of training data for the classification algorithms. The method by which training data is interactively defined by the user by superimposing polygons on the image proved to be a difficult project for even an experienced software engineer at RIT. For this reason, its development was delayed and a method to test the evolving classification modules in a timely manner became critical. To this end, the *Fuzzy K-Means* module was developed and implemented. This module serves a dual role. Besides being able to collect spectrally pure training data, it can also be used for unsupervised image classification. Consider the AVS network in Figure 5.1.1.

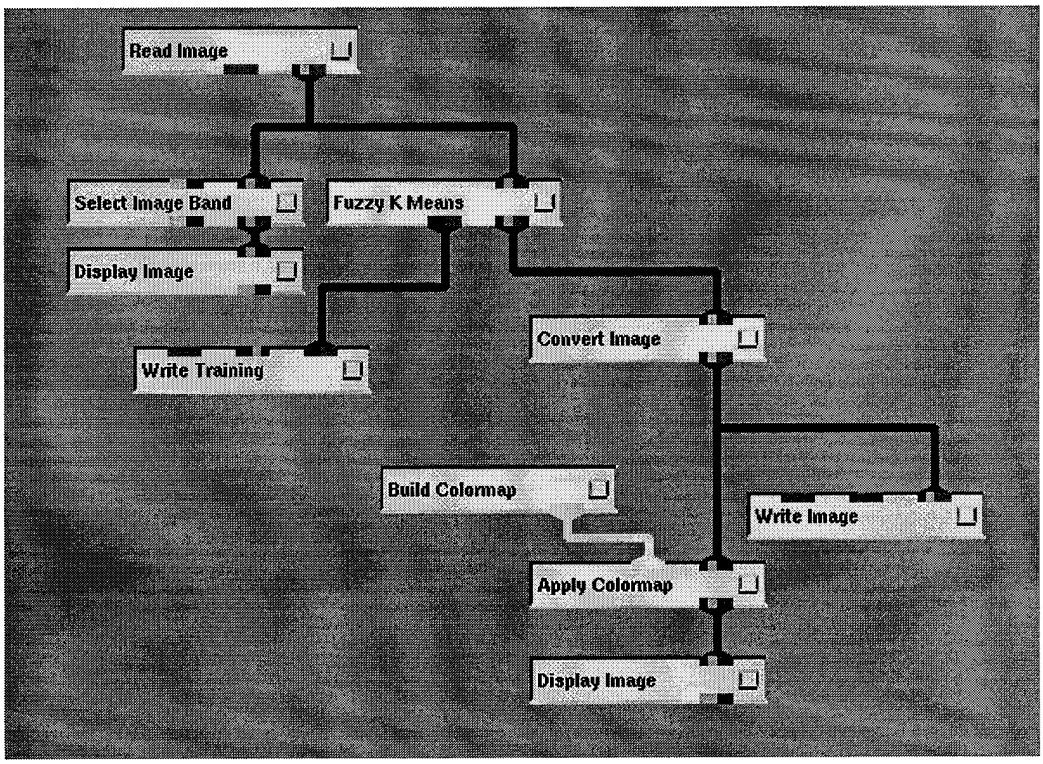


Figure 5.1.1 - Depiction of Fuzzy K-Means network

In operation, the network reads in a user-selected image, selects a single band, and displays the image. Pointers to the multispectral image data in shared memory are then passed to the to *Fuzzy K-Means* module. The control panel for the module is depicted in Figure 5.1.2. The user provides the algorithm with a number of parameters before clustering can occur. First, the user must enter the number of clusters to locate and select a membership value. By setting the membership to a high value (near 1) , a few pixels in very tight spectral groups will be collected for each cluster. When operated in this manner, the module supports the collection of training data. If the membership is instead set to a low value, the image can be classified in a unsupervised manner by calculating the cluster centers and performing a form of minimum-distance-to-the-means classification with respect to the value of the membership function. The "Sample offset" parameter permits the input image to be subsampled. By entering a value of two, every other pixel in every other row of the image will be used for cluster computations. As expected, increasing this value dramatically reduces processing time as only a fraction of the pixels in the input image need be processed. The "Maximum iteration" parameter permits the user to define the maximum number of clustering iterations. Similarly, the "Cluster shift limit" parameter is used by the algorithm to determine when the clustering algorithm has converged. The value of four indicates that the greatest shift of any cluster from one iteration to the next must be less than or equal to four to end processing. Once the user has supplied these parameters, only the "Generate clusters" or "Auto-cluster"

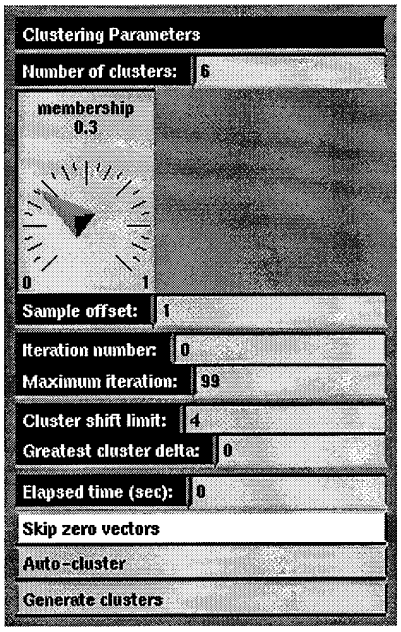


Figure 5.1.2 - Depiction of the control panel for the Fuzzy K-Means module

toggle switch need be selected. The auto-clustering feature will eventually allow a number of images to be processed successively. The module then executes the compute function code. It first selects the appropriate number of pixels pseudorandomly from the input image to serve as initial approximations of the cluster centers. If the "Skip zero vectors" toggle has been set, the module checks for and skips any pixels with zero magnitude. This permits segmented images to be rapidly processed. It then allocates a "membership" array for each cluster that has the same dimensions as the input image. The membership value for each pixel from the image or subsampled image is then calculated with respect to each cluster center and then the center approximations are recomputed. This process is repeated until the algorithm converges and the display is updated to show the iteration number and greatest cluster shift. The clustering algorithm obviously is computationally intensive and may take a considerable amount of processing time before convergence occurs. Once the conditions for convergence are met, the elapsed time is reported and the required number of iterations is displayed. At this point, the training data set is collected and the unsupervised classification map is constructed. The membership values with respect to each cluster center for each pixel in the input or subsampled image are compared to find the maximum value. If this value is equal to or greater than the value membership function, the pixel is assigned to the corresponding cluster and its multispectral pixel is added to the training data linked list. If the greatest membership value is not greater than or equal to the value of the membership parameter, the pixel is assigned to the background class. The resulting training data in the form of a linked list is then transferred to the write training module so that it can be optionally saved to disk. Similarly, the classification map can be saved to disk in a standard image format. A colormap is constructed and applied to the unsupervised classification map so that the clustering and classification results can be readily visualized.

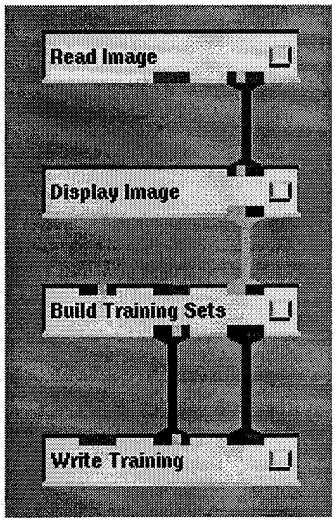


Figure 5.1.3 - Depiction of network to gather user defined training data

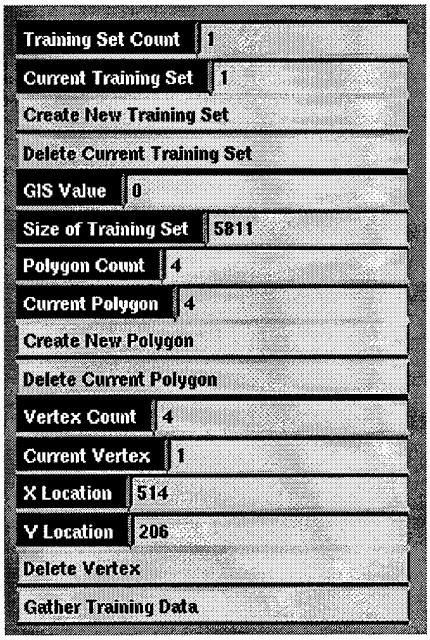


Figure 5.1.4 - Depiction of the control panel for the Build Training sets module

The network that supports gathering training data from the input multispectral image in an interactive manner will now be discussed. This module, and the supporting modules that read and write the training data structures, were written by Stephen Schultz, resident software engineer at RIT. The programming of these modules required an in-depth understanding of the X windows environment that is currently beyond the capability of the author. Consider the network in Figure 5.1.3. This network reads and displays the user-selected image. References to the input data are passed to the *Build Training Sets* module. The control panel for this module is depicted in Figure 5.1.4. To define training-class polygons, the user simply designates the polygon vertices with the cursor and a mouse click. The module allows multiple polygons to define a single class. Additional polygons are added to the current training set by selecting the "Create New Polygon" button. After the user has defined a given class, the process is repeated for a new class after selecting the "Create New Training Set" button. To correct any errors, the module permits the user to remove the current vertex, polygon, or entire training class. A subsection of an image with

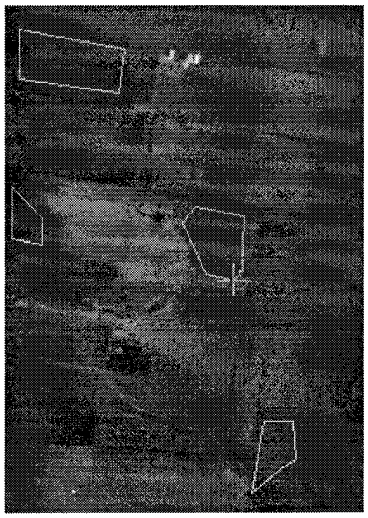


Figure 5.1.5 - Depiction of image with overlaid training class polygons

overlaid training class polygons is depicted in Figure 5.1.5. After all polygons for all classes have been defined, the user selects the "Gather Training Data" button. This causes the module to retrieve the designated multispectral pixels from the input image, label them as belonging to a given class, and chains them together in the form of a linked list. The linked list is then passed on to the *Write Training* module where it is written to a user-specified disk file. The training data file can then be readily processed by the modules that support the classification operations.

5.2 Gaussian Maximum Likelihood in the AVS Environment

Accomplishing image classification with the *GML Classify* and *Class Statistics* modules is a straight forward process in the AVS environment. Before an image can be classified, the representative statistics for the training classes must be calculated. Figure 5.2.1 depicts the AVS network that is used to create the required statistics files. Training data collected by either the fuzzy K-means or the user-interactive module is transferred in the form of a linked list from the *Read Training* module to the *Class Statistics* module. The data is then moved from the linked list to an internal data structure to support further operations. The control panel that supports the user interface to the statistics module is depicted in Figure 5.2.2. Once the training data has been received and an output statistics filename has been selected, the interface is updated. It displays the number of training sets or classes, the total number of training class pixels, and the dimensionality of the training set. When the user selects either the "Generate statistics file" or "Auto-generate statistics file" button, the module begins the compute function's statistical operations. In more somewhat greater detail, the mean vector, the variance-covariance matrix, and its inverse are calculated for each training class. Matrix inversion is accomplished through Lower-Upper (LU) decomposition and backsubstitution. When these operations are complete, the interface is updated to

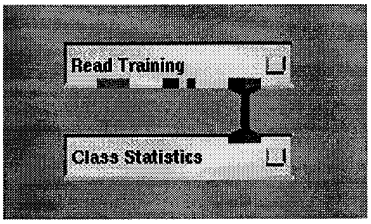


Figure 5.2.1 - Depiction of the class statistics network

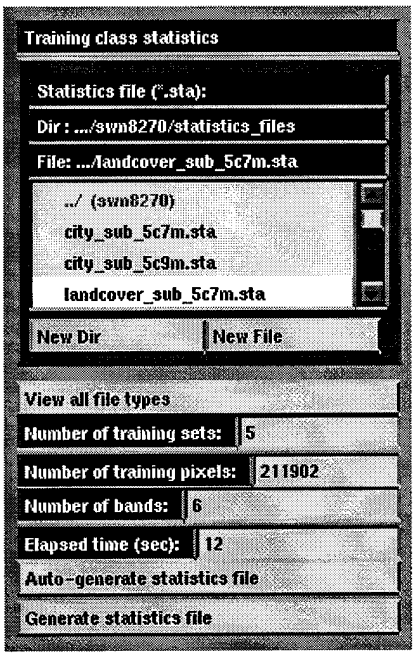


Figure 5.2.2 - The class statistics module control panel

reflect the elapsed processing time and the resulting class statistics are written to the user specified disk file. A sample statistics file is depicted in Figure 5.2.3.

```
training class statistics file for AVS GML
6   bands
2   classes

These are the mean vectors:
101.558304 87.162682 79.549095 108.158653 117.790672 91.485168 3473.000000
105.620270 99.025017 91.263634 127.766518 130.973694 98.951248 1559.000000

These are the inverted matrices:
0.148350 -0.031817 0.079483 -0.010912 -0.002541 -0.030949
-0.031817 0.339271 -0.272071 -0.005770 0.050755 0.034068
0.079483 -0.272071 0.444432 0.002016 -0.055556 -0.025385
-0.010912 -0.005770 0.002016 0.160764 -0.005757 0.015016
-0.002541 0.050755 -0.055556 -0.005757 0.211132 -0.045791
-0.030949 0.034068 -0.025385 0.015016 -0.045791 0.203757

0.402535 -0.241097 0.073095 0.037794 0.001006 -0.067547
-0.241097 0.601827 -0.238123 -0.037646 0.051879 0.028189
0.073095 -0.238123 0.422447 0.031728 -0.047003 0.000208
0.037794 -0.037647 0.031728 0.184273 -0.029169 0.011340
0.001006 0.051879 -0.047003 -0.029169 0.233353 -0.056189
-0.067547 0.028189 0.000208 0.011340 -0.056189 0.237680

These are the natural logs of the determinants:
9.728889 7.587222

These are the normalized vc matrices:
7.810011 -0.872250 -1.863393 0.423204 0.059587 1.082154
-0.872250 6.033162 3.718243 0.165424 -0.659772 -0.838456
-1.863393 3.718243 4.887069 -0.022048 0.320907 -0.222125
0.423204 0.165424 -0.022048 6.296008 0.039846 -0.421174
0.059587 -0.659772 0.320907 0.039846 5.272109 1.341222
1.082154 -0.838456 -0.222125 -0.421174 1.341222 5.517158

3.484727 1.426702 0.223407 -0.536731 -0.159543 0.808821
1.426702 2.768975 1.273246 -0.003784 -0.368406 -0.010975
0.223407 1.273246 3.100528 -0.270882 0.305396 -0.005112
-0.536731 -0.003784 -0.270882 5.693850 0.592060 -0.283535
-0.159543 -0.368406 0.305396 0.592060 4.767593 1.096926
0.808821 -0.010975 -0.005112 -0.283535 1.096926 4.711347

These are the determinants:
16795.867188 1972.824951
```

Figure 5.2.3 - Sample statistics file

Once a statistics file has been created, image classification can be accomplished readily. Consider the network in Figure 5.2.4 on the following page that is utilized to perform GML classification.

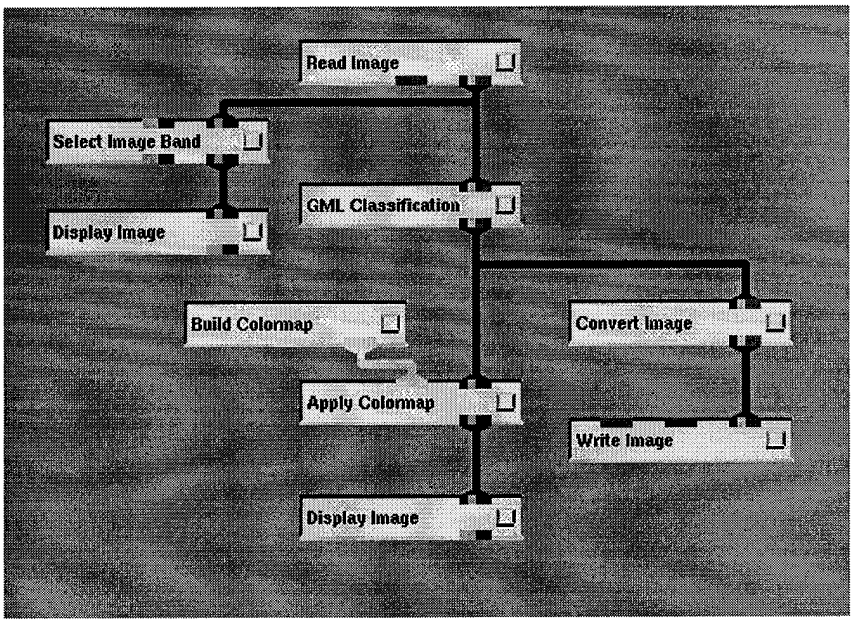


Figure 5.2.4 - GML classification network

The network reads the multispectral image and permits the user to select a single band to display the input image. A reference to the multispectral image data (in the form of a pointer to shared memory) is transferred to the *GML Classification* module. The control panel for module is depicted in Figure 5.2.5. This parametric classifier requires a model of the training data created by the *Class Statistics* module as previously discussed. Once the user selects a valid statistics file, the data from the selected file is printed to the standard output device and the interface is updated to reflect the classification parameters, the number of target classes, and the dimensionality of the data. At this point, the user need enter only an appropriate value for the Chi squared (χ^2) distance and select either the "Generate GML map" or "Auto-generate GML" button. The auto-generation feature allows the same image to be easily reclassified by just entering a different value of χ^2. The module then enters its compute function. The "Skip zero vectors" toggle allows segmented images to be rapidly processed. Before being processed, the magnitude of

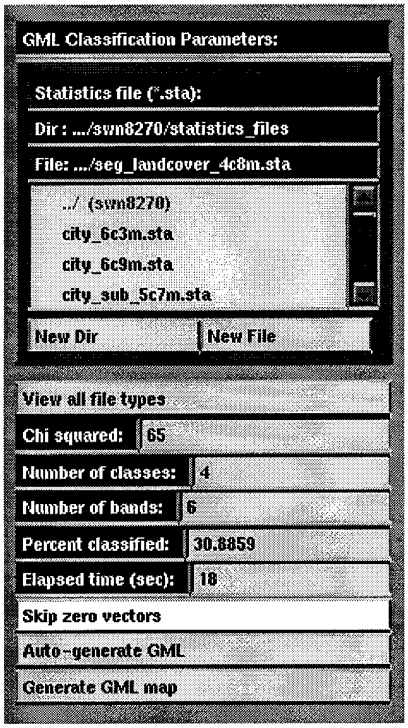


Figure 5.2.5 - The GML classification control panel

each pixel is calculated; if zero, the pixel is skipped. It is important to note that the Mahalanobis distance for each non-zero pixel in the input image is then calculated with respect to the mean vector of each target class utilizing its unique variance-covariance matrix. As expected, if the image size or the number of target classes is increased, classification time proportionally increases. If the minimum distance value to one class is less than that of the user input value for χ^2, the pixel is assigned to that class. The value in the classification map at the position of the input pixel then is updated to reflect this assignment. If the minimum distance exceeds that of the limiting value, the pixel is assigned to the background class. Once the entire classification map is constructed in the manner just described, the interface is updated to reflect the elapsed computation time in seconds. The network then constructs a colormap and applies it to the classification map so that the classification results can be readily visualized. The classification map is then converted to a standard image format and optionally is written to a user-defined file.

5.3 Performing nPDF Classification in the AVS Environment

The inherent graphical nature of the AVS environment is well suited to performing classification with the n-Dimensional Probability Density Function (nPDF) algorithm. Recall that this approach requires that the multispectral training data must first be projected into "nPDF" space to construct a LookUp Table (LUT) for image classification. The layout in Figure 5.3.1 depicts the AVS network that is used to project the multispectral training data into the two-dimensional nPDF space.

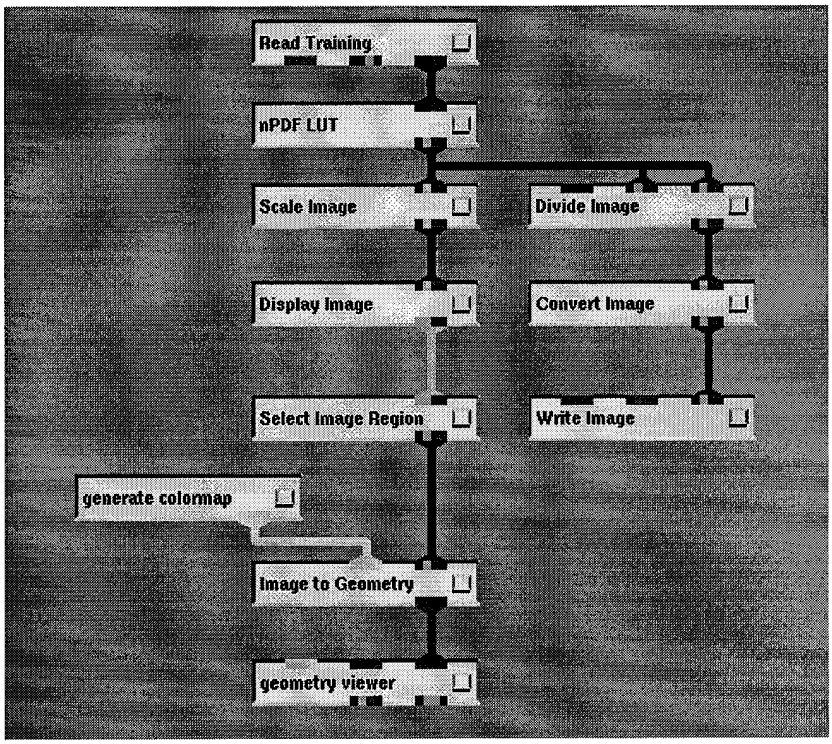


Figure 5.3.1 - Depiction of the network to project training data into nPDF space

This network reads in a user-specified training data file and passes it as a linked list to the *nPDF LUT* module. At that module, the training data is transferred from the linked list into a more easily manipulated internal data structure. The control panel for the *nPDF*

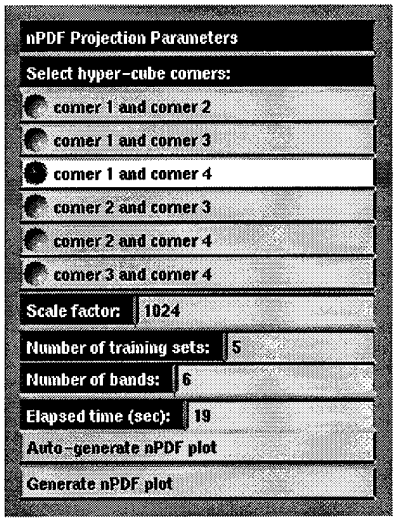


Figure 5.3.2 - Control panel for the nPDF LUT module

LUT module is depicted in Figure 5.3.2. The user then selects a pair of hypercube pairs by depressing one of the radio button options, and enters a value for the scale factor for the resulting nPDF projection. After these two parameters have been supplied, the user need only select the "Generate nPDF plot" or "Auto-generate nPDF plot" to begin the projection process. The auto-generation feature allow the same input training set to be repeatedly projected with a variety of corner choices and scale factor values. Once the module enters the compute function, the interface is updated to reflect the dimensionality and the number of classes in the training data set. The algorithm then projects each pixel from the training set into nPDF space as defined by the corner choice and scale factor parameters. When all pixels have been processed, the interface is updated to display the elapsed processing time and the projection is passed to the rest of the network. First, the nPDF projection is divided by itself to form a binary image, converted to a standard image file format, and then optionally written to a disk file. The binary projection image must be processed by another network before it can be used as a LUT for classification purposes. The projected image is scaled and displayed to permit visual inspection. If no bundles of the training data overlap, good classification results can be expected. If the bundles do overlap, the corner pairs and scale factor are varied until the bundles no longer intersect, or such intersection is minimized. Continuing on with the network, a subsection of the displayed image can be selected. A colormap is then generated and applied to this subsection. The resulting nPDF projection can be converted to a height field. The resulting colored three-dimensional field can be viewed

with the AVS geometry viewer. In Figure 5.3.3, the nPDF projection for a five-class training set is depicted as a flat projection and as a height field.

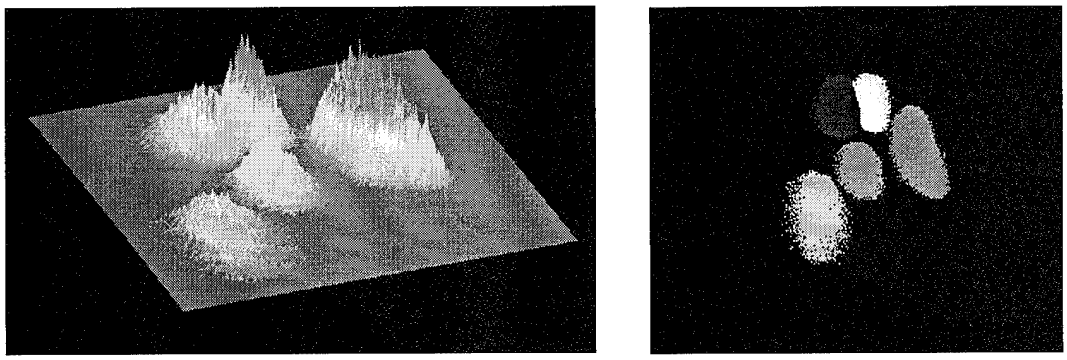


Figure 5.3.3- nPDF projection of training data depicted as a height field and a flat projection.

As mentioned in section 4.3 of this report, the nPDF projection must be processed before it can be used as a LUT for classification purposes. In a manner similar to that for constructing polygons around training data of interest, a boundary is drawn around each individual cluster. Care must be taken to completely encompass a class and to provide precise boundaries between clusters, as the accuracy of the resulting classification depends entirely on the accuracy of the LUT. It is also important to realize that the complete spectral extent of a class must be encircled by its class polygon in nPDF space. One method for accomplishing this is to attach a piece of acetate to the computer's display and sketch the position of the training data clusters. Once this has been accomplished, the nPDF projection of the entire image to be classified is then displayed. By utilizing a straightedge and a pen, the projection of the entire image can be segmented readily into classification regions. Once a classification region has been completely surrounded, the region is then filled with the numerical value for that class. Each individual region is saved to a disk file as an image. The network in Figure 5.3.4 page is utilized to create the individual classification regions and fill them with the

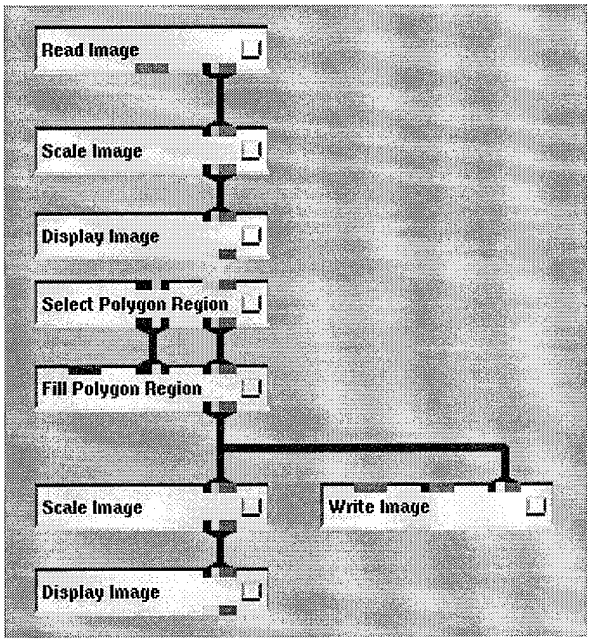


Figure 5.3.4- AVS network to create individual classification regions in nPDF space.

appropriate class value. The network reads in an image (typically the nPDF projection of the image to be classified) scales its values to promote interpretability, and displays it. The classification regions are individually designated with the *Select Polygon Region* module. These regions are passed onto the *Fill Polygon Region* module where the designated classification regions are filled with the appropriate class value. The separate classification region images are then written to a disk file in a standard format.

Once the individual classification regions have been designated, they must be combined into a single LUT. The network depicted in Figure 5.3.5 is utilized for this purpose. This AVS procedure reads in two separate classification region images at a time, scales each image, and displays them. They are then combined with logical OR operation as provided by the *Or Image* module. The resulting image is written to a disk file. In operation, the user must read the class-one and class-two image and combine them to form the intermediate LUT for classes one and two. This process is repeated until all of the individual class images have been combined into a single LUT.

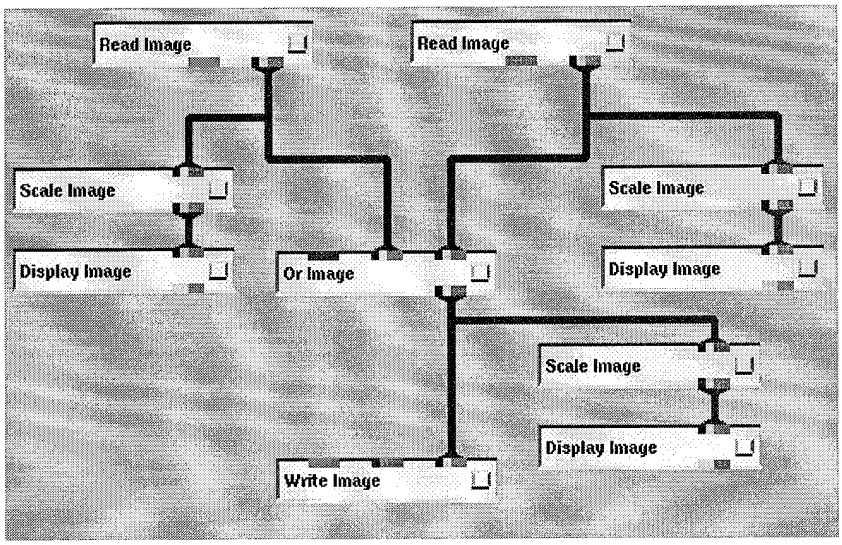


Figure 5.3.5- AVS network to combine the individual classification region images into a LUT.

The images in Figure 5.3.6 highlight the process from nPDF projection to classification LUT.

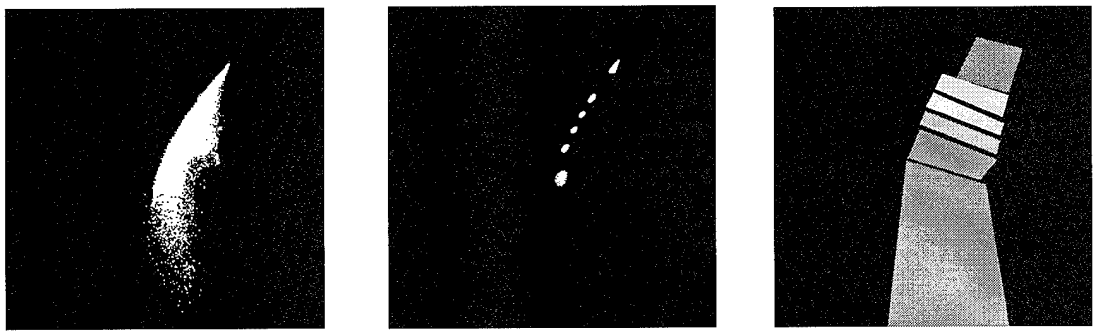


Figure 5.3.6- nPDF projection of an image, nPDF projection of training data from image, and resulting classification LUT

Once the LUT has been constructed, the image classified easily. The AVS algorithm for nPDF classification is depicted in Figure 5.3.7. This network reads in the LUT, displays it, and converts it to the format expected by the classifier. Similarly, the input image is read, a single band is selected, and the image is displayed. The LUT data

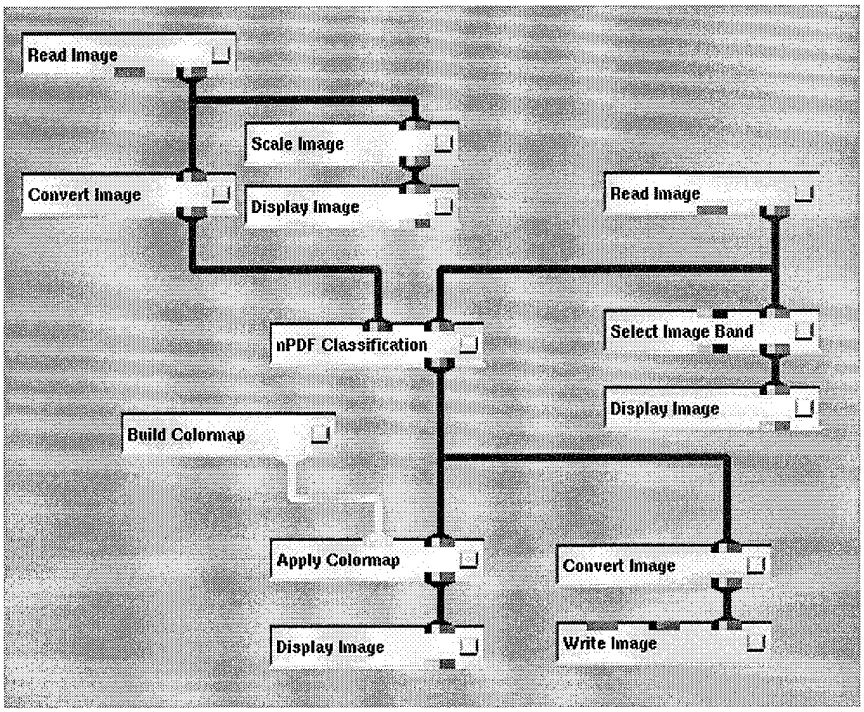


Figure 5.3.7- Depiction of network to accomplish nPDF classification

and the multispectral image data are passed to the *nPDF Classification* module. The control panel constituting the user interface to this module is depicted in Figure 5.3.8. To begin classification, the user needs to select the same corners of the hypercube as were used to project the data. At present, AVS does not permit this information to be transferred in any other manner. The simplest workaround is to include the corner choice in the name of the LUT. Once the user has supplied the corner-choice selection, the "Classify Image" or "Auto-classify Image" must be selected before classification can begin. The auto-classification feature eventually will allow multiple images to be successively processed by the module. Once classification has begun, the interface is updated to reflect the scale of the LUT, which is either its height or width dimension. The "Skip zero vectors" toggle allows for the rapid processing of images with many zero

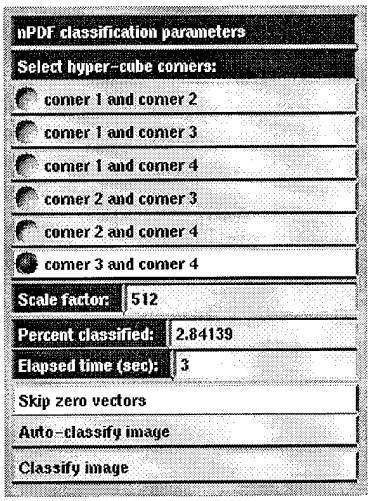


Figure 5.3.8 - Control panel for the nPDF Classification module

magnitude vectors. Each non-zero pixel from the input image is projected into nPDF space as defined by the combination of the corner choice and the scale factor. To achieve classification, the coordinates of each input pixel in nPDF space are used to reference a location in the LUT. If the pixel projects into one of the classification regions, the numerical value of that region is retrieved. If instead the pixel projects onto an undesignated region, the pixel is assigned to the background class. Note that increasing the number of classes has no impact on classification time, but the LUT will be more difficult and time consuming to generate. Once all of the input image's pixels have been classified, the elapsed computational time is displayed. The network completes operation by building a colormap, applying it to the classification map, and displaying the results to aid visual inspection. The classification map is then converted to a format to support standard image formats and is may be written to a disk file.

Any classification algorithm can also be utilized for image *segmentation*. This operation screens out all pixels that do not belong to a desired class and forwards all multispectral pixels that do compromise a selected class. Instead of creating a classification map, a new multispectral image is created comprised solely of the desired class or classes.

Image segmentation via the nPDF algorithm will be used as the first step in *hybrid* image classification. In this case, the term hybrid implies that a nPDF segmented image will be passed to either the GML or fuzzy ARTMAP algorithm for further processing. In this manner, the strengths of either classification algorithm can be tested on tightly

grouped spectral classes. The network in Figure 5.3.9 supports the first phase in hybrid image classification by nPDF segmentation.

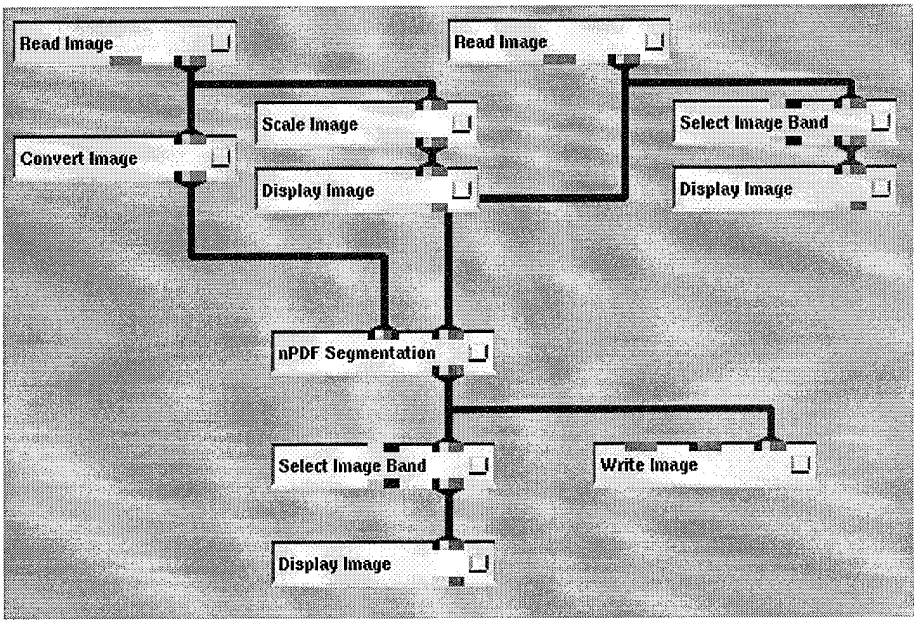


Figure 5.3.9 - AVS network to accomplish nPDF image segmentation

This AVS procedure functions identically to that for nPDF classification with the exception that it produces a multispectral image of the class or classes defined in the LUT. The resulting image is then written to disk in a standard format.

5.4 Accomplishing Fuzzy ARTMAP Classification in the AVS Environment

Accomplishing image classification with the *Make ARTMAP* and *Fuzzy ARTMAP* modules is a very similar to the process outlined for the GML module. Before an input image can be classified, a fuzzy ARTMAP neural network and associated parameter file must be created from the training data. The parameter file conveys information about the constructed network such as the dimensionality of the input space, the number of ART_a classification nodes, and the number of target classes (or ART_b classification nodes). A complete commented parameter file is depicted in Figure 5.4.1.

```
1       #define randomInit - seed for random number generator "1" = read time clock
1       #define orderedSet "1" = read in the patterns in the same order as original file
0       #define employ_weighting "1" = input features are differentially weighted
0       #define on_line - "1" = system goes through training set once
1       #define num_voters
1       #define num_runs - number of complete training runs
10      #define max_iterations - maximum training iterations
1       #define complement "1" = complement is presented
2       #define a_length -  the dimensionality of the input space
2       #define b_length -  the dimensionality of the output space
2       #define num_data_category - number of classes
700     #define train_pats - the number of training patterns
0       #define predict_pats - the number of predict patterns
300     #define test_pats - the number of test patterns
0       #define on_line_recast - method of recasting to handles inconsistent cases
1.0     #define dnInit - the initial top down weights of both ART modules
0.001   #define Alpha - parameter used in choiceOrderfunction of ART1
0.001   #define epsilon   IA-match -> rho goes to current match + epsilon.
1.0     #define rate - under learning, wij -> (1 - rate) * wij + rate * fast_learn_limit
0.0     #define z_bar - in art1.c, if top down weight drops below zbar, both weights -> 0
0.0     #define min_arho - minimum ART-A rho value
1.0     #define brho - minimum ART-B rho value
0.0     #define noise_rate at which recent success or failure change confidence of nodes
0.0     #define noise_tolerance below which an ART-A node will be destroyed
0       #define trace - if > 0, then progresses of program will be reported
0       #define trace_weight - allows features to be weighted separately
1       #define display_confusion_matrix - if 1, confusion matrix is displayed
0       #define incorporate_rule
0       #define extract_rules
0       #define quantize_rules
0       #define quantization_step
0       #define individual match_tracking
1       #define num_F2_winners - number of winning classification nodes
0.0     #define threshold
```

Figure 5.4.1 - Example ARTMAP parameter file

In general, a casual user need not be concerned with many individual parameters in the file as they are either defaults, have been optimized to process multispectral images, or are automatically updated by this module. The neural network file that is created by this module contains the weight vectors for each node in both ART modules, and the values stored in the inter-ART field which maps an ART_a classification node to an ART_b classification node representing the output class. The AVS procedure pictured in Figure 5.4.2 depicts the AVS network that is used to create the required neural network files.

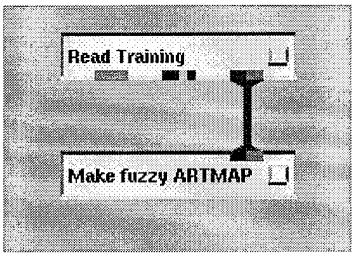


Figure 5.4.2 - Depiction of the network to construct the parameter and network files

As with the other classifiers, training data collected either by the fuzzy K-means or the user interactive module is transferred in the form of a linked list from the *Read Training* module to the *Make fuzzy ARTMAP* module. The data then is moved from the linked list to an internal data structure to support further processing. The control panel that supports the user interface to the *Make fuzzy ARTMAP* module is depicted in Figure 5.4.3.

Once the training data has been received and a filenames for both the parameter and network files have been selected, the interface is updated. It displays the number of training sets or classes, the total number of training class pixels, and the dimensionality of the training set. At this point, the user must select a value of the vigilance parameter (ρ) for the ART_a module. Recall that this determines the size of the region in feature space associated with a given node. Increasing ρ will decrease the maximum allowable size of the hyper-rectangle in feature space resulting in finely granulized classification and the creation of a large number of nodes. As expected, if the user selects a smaller value for the vigilance parameter, the classification regions in feature space grow and fewer nodes need to be created. The "Recast inconsistent cases" button permits the algorithm to check for inconsistencies in the training data. For example, if there were

some grass pixels in the tree class and a separate grass class, this preprocessing algorithm would attempt to relabel the incorrect pixels. At this point, the user need only select a value for ρ and either the "Generate network" or "Auto-generate network" button to create the fuzzy ARTMAP neural network. The auto-generation feature permits the network to be easily recreated for different values of the vigilance parameter. The module then enters its compute function. As each of the training class pixels is operated upon, the region into which it fits best with respect to the vigilance parameter must be determined. If found to fall within a region corresponding to a classification node, no further action need occur. If near to a classification region, and the degree of "nearness" is controlled by ρ, the weight vector of the node is updated and learning has occurred. If the pixel does not fall in or near any classification region, a new node is created. Once all training class pixels have been processed to 100% recognition, the interface is updated to reflect the number of classification nodes, the number of learning iterations, and the elapsed computation time. The network and parameter files are then written to disk.

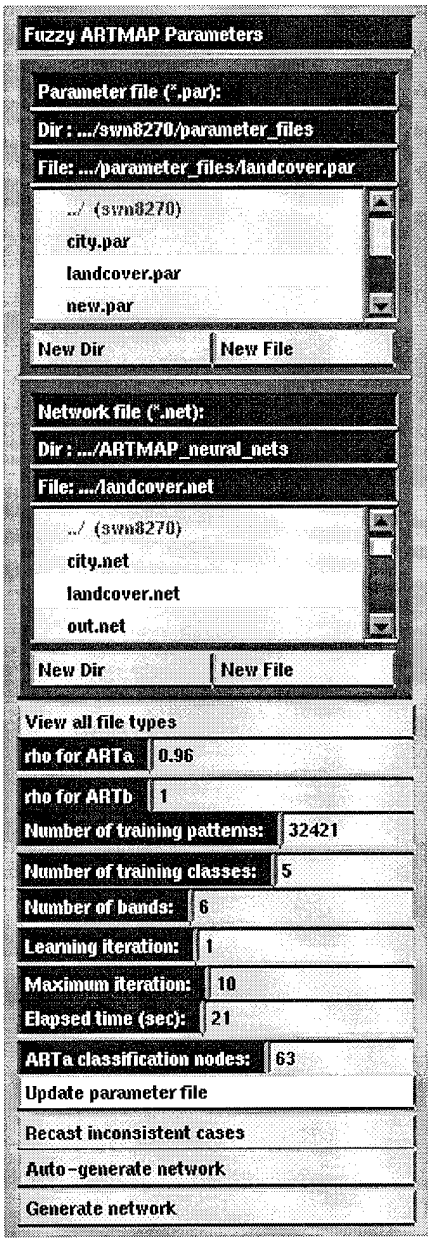


Figure 5.4.3 - Depiction of the control panel for the Make fuzzy ARTMAP module

At this point, image classification with the Fuzzy ARTMAP module can be readily accomplished. Consider the network depicted in Figure 5.4.4.

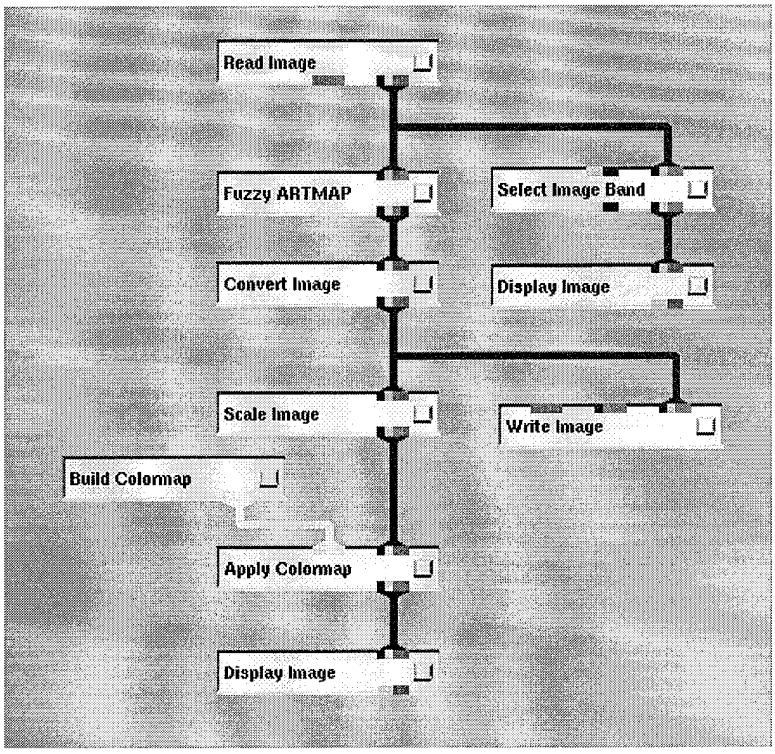


Figure 5.4.4 - Depiction of the network to perform fuzzy ARTMAP image classification in the AVS environment

The AVS procedure reads in a multispectral image and permits the user to select a single band to display the input image. A reference to the multispectral image data (in the form of a pointer to shared memory) is transferred to the *Fuzzy ARTMAP* module. The control panel supporting the user interface to this module is depicted in Figure 5.4.5. To perform image classification, this non-parametric classifier requires both the parameter and network file created by the *Make ARTMAP* module as previously discussed. Once the user selects a valid parameter and network file, the data from both the selected files is read from disk files and displayed on the standard output device. The module's interface

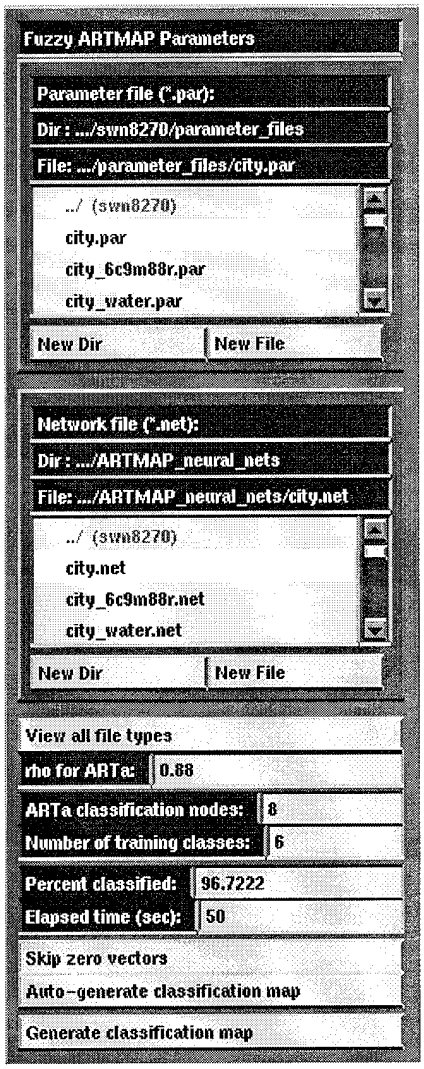


Figure 5.4.5 - Depiction of the control panel for the fuzzy ARTMAP module

also is updated to reflect important classification parameters to include the value of the vigilance parameter, the number of target classes, and the number of classification nodes. At this point, the user need only select either the "Generate classification map" or "Auto-generate classification map" button to continue. The auto-generation feature allows the same image to be re-classified by just selecting different network and related parameter files. The module then enters its compute function. The "Skip zero vectors" toggle allows rapid processing of segmented images. It is important to realize that each non-zero pixel in the input image must be compared to each classification node's weight vector. If the pixel falls within the classification region of a node, it is assigned to that class. If it does not fall within any classification region, it is assigned to the background class. The value in the classification map at the same position as each input pixel is then updated to reflect this assignment. Once the entire classification map is constructed in the manner just described, the interface is updated to reflect the elapsed computation time in seconds. The network then constructs a colormap and applies it to the classification map so that the classification results can be readily visualized. The data comprising the classification map is then converted to support standard image formats and is optionally written to a

user defined file. As expected, if the image size or the value of the vigilance parameter is sufficiently large to force the creation of many classification nodes, classification time correspondingly increases.

5.5 Using the Confusion Matrix module in AVS

Once the images have been classified by the various algorithms, the classification accuracy needs to be assessed. This is a simple process in the AVS environment after a *truth* image has been constructed. A truth image is a classification map constructed by hand or by a trusted algorithm that reflects ground truth. Consider the AVS network in Figure 5.5.1.

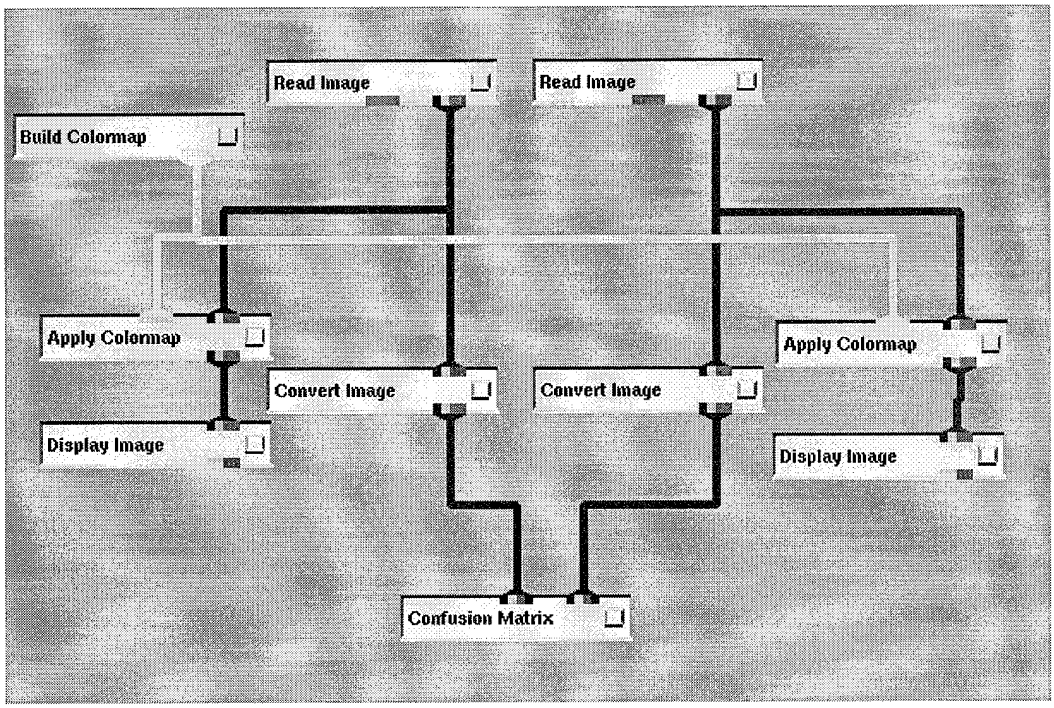


Figure 5.5.1 - Depiction of the network to assess classification accuracy

This network compares a classification map constructed by one of the classification modules and a corresponding truth image. A colormap is constructed and applied to both incoming images, and the resulting colored classification maps are displayed to aid visual inspection. The classification map and truth data are then converted to a format compatible with the *Confusion Matrix* module. This module checks to ensure that the two data sets are comparable. In specific, it checks that the image sizes are equal and that

the same number of classes are present in each image. Note that it is critical that the truth image be connected to the leftmost port and that the classification map being evaluated be connected to the rightmost port of the module. To aid in correct usage, the control panel for the *Confusion Matrix* module is labeled to minimize errors. The user interface for the module is depicted in Figure 5.5.2. Once the module has received the reference to the input data and a filename for the confusion matrix, the user need only select the "Calculate confusion matrix" or "Auto-generate confusion matrix" to begin operation. The auto-generation feature permits differing classification maps from the various modules to be rapidly compared to a given truth image. When the module enters the compute function the confusion matrix is initially formed. Each pixel in both the classification map and truth image are compared in pairs. The value of the pixel in the truth image is used to index the column of the confusion matrix, while the value of the pixel in the classification map is used to index the row. If the values are the same, implying agreement between classification and truth, an on-diagonal term is incremented. If the two values differ, the corresponding off-diagonal term is incremented. Once the confusion matrix has been completely determined, the simple accuracy, the weighted accuracy, the kappa coefficient, and Brennan and Prediger's kappa are calculated and the interface is updated. The confusion matrix along with the four figures of classification accuracy are written to the disk file. A sample output file follows in Figure 5.5.3.

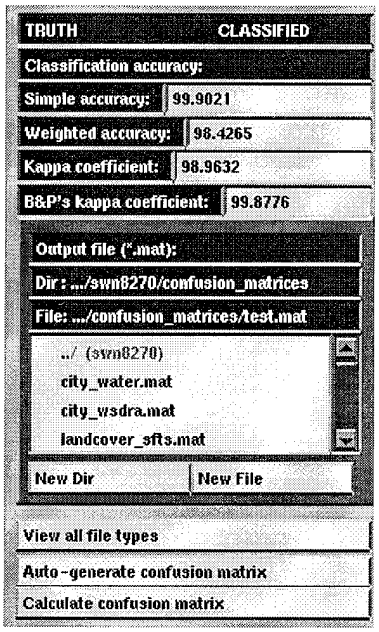


Figure 5.5.2 - Depiction of the control panel for the Confusion Matrix module

This is the confusion matrix:

262050	642	332	1	64
76	91922	4	81	0
173	861	32463	35	121
180	1430	252	16564	1292
170	0	0	0	9363

The simple accuracy = 0.986097
The weighted accuracy = 0.961072
The kappa coefficient = 0.974530
Brennan and Prediger's kappa coefficient = 0.982621

Figure 5.5.3- Sample output from the Confusion Matrix module

6.0 Image Classification Comparison

The color plates in Figures 6.0.1 through 6.0.4 are the test images for the various classification algorithms. The images include a range of class types from rural agricultural settings to densely populated urban scenes. All 6-band multispectral images. Three are from the M7 airborne sensor and the remaining image (Figure 6.0.3) is from the LANDSAT satellite. Table 6.0.1 below reports the important statistics of each scene.

Table 6.0.1 - Test image statistics

image name	size (H x V)	sensor	bands
city.lan	704 x 594	M7	3,6,8,10,12 & 13
landcover.lan	676 x 701	M7	3,6,8,10,12 & 13
roch84.lan	512 x 512	LANDSAT	1 - 5 & 7
seashore.lan	500 x 500	M7	3,6,8,10,12 & 13

Three major classification tasks were undertaken to compare the classification results derived from the varying approaches. In the following sections, the restrictions imposed upon and the desired goals of each task will be described. The procedure for how the images were classified, along with the elapsed time and accuracy measurements, will be presented for each algorithm accompanied by classification maps for visual inspection.

Figure 6.0.1
landcover.lan

Figure 6.0.2
city.lan

Figure 6.0.3
roch84.lan

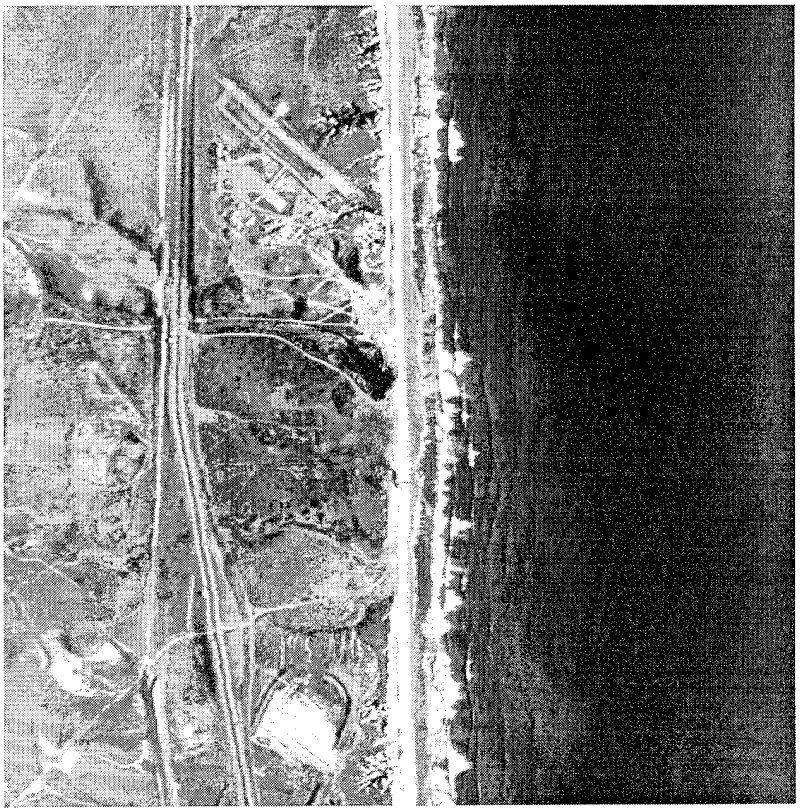


Figure 6.0.4
seashore.lan

6.1 Task 1: Discussion and Results

The first task undertaken in this study involved classifying approximately 95% of each image using training data collected from the fuzzy K-means algorithm. The goal of this task is to highlight the ability of each algorithm to classify broad target categories given trusted data for each class.

The cluster shift limit parameter of the fuzzy K-means module was set to four for all operations. This implies that all cluster centers must have moved less than a total of four pixels in the six-dimensional feature space before convergence can occur. The unsupervised classifier was set first to a large membership value, typically 0.9 or 0.85, to collect the trusted training data for each of the classifiers. The high membership value used to construct the training data was manipulated to ensure that there are at least $60D$ pixels per class, where D is the number of bands, to ensure valid statistical calculations. The clustering operation was repeated with the classifier reset to a lower membership value, typically 0.3 to 0.4, to form a truth image. The lower membership value was manipulated to ensure that approximately 95% of the image was classified. The value for the number of cluster centers was estimated from visual inspection. Recall that the cluster centers are initially selected in a *pseudorandom* manner. This ensures that the same cluster centers will be found regardless of membership value. Utilizing the fuzzy K-means clustering algorithm with a low membership value provides an easy method to construct truth images for comparison purposes. The use of the algorithm to produce truth images can be justified by realizing that this approach produces cluster centers, and the ensuing classification is based entirely upon naturally occurring "clumps" of pixels in feature space. The resulting truth images also support visual intuition of class membership and extent. The results of the clustering operation are summarized in Table 6.1.1.

Table 6.1.1 - Fuzzy K-means clustering statistics

image name	number of cluster centers	membership values	number of iterations	elapsed time (sec)
city.lan	6	0.9 & 0.3	17	2,442
landcover.lan	6	0.9 & 0.3	7	1,362
roch84.lan	5	0.85 & 0.3	6	635
seashore.lan	5	0.85 & 0.4	5	490

As expected, locating clusters of pixels within a multispectral image is computationally intensive. The wide distribution of both elapsed time, and the related value for the number of iterations to reach convergence, is a function of image size, complexity or content, and desired number of cluster centers. The reported elapsed time is the result of averaging numerous clustering operations, but very low variation was observed.

After constructing both training data sets and truth images, image classification was readily accomplished. For GML classification, the training data from the clustering algorithm must first be operated upon to construct the parametric model of the data which is stored in a statistics file. The detailed statistics files for each training set can be found in appendix B. Table 6.1.2 summarizes the number of classes in each training set, the number of points in each training set, and the elapsed time to create the statistics file for each training set. Note that required time to create the statistics file depends upon both the number of classes and the number of points in each training set.

Table 6.1.2 - Summary of GML statistics file parameters

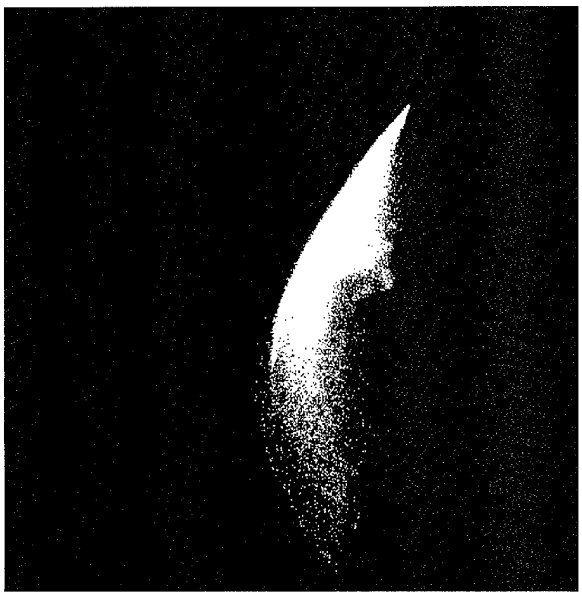
image name	number of classes	number of points in training set (sec)	elapsed time to create statistics file (sec)
city.lan	6	150,874	11
landcover.la79n	6	44,791	3
roch84.lan	5	27,177	2
seashore.lan	5	124,851	7

With a mathematical model that describes the distribution of the training data constructed, image classification with GML is a straightforward process. The value of chi squared (χ^2) was manipulated until the target classification percentage of 95% was achieved. Note that the large values of the χ^2 distance can be explained by realizing that the training data was collected at a high membership value. As such, it is tightly clustered in feature space. The large value of this distance parameter permits the resulting hyperellipsoids to expand to encompass the size of all classes. The reported classification accuracy is the simple accuracy, and the truth image for comparison purposes was created with the fuzzy K-means algorithm. Complete confusion matrices for each of the images are available for detailed inspection in appendix B. The GML classification results are summarized in Table 6.1.3.

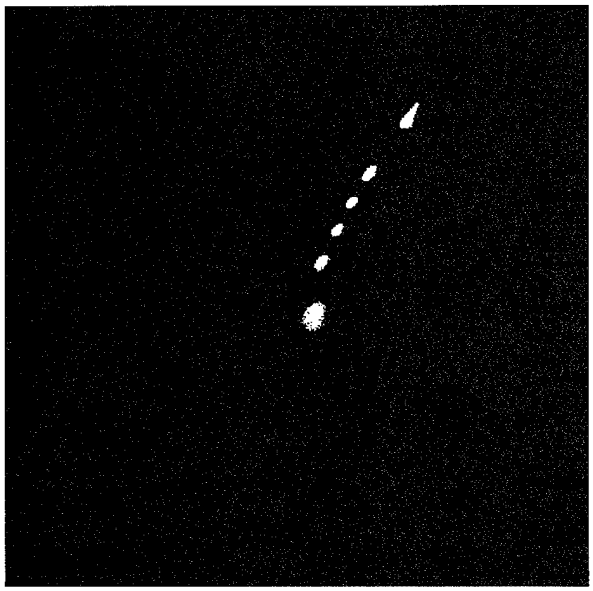
Table 6.1.3 - Summary of GML classification statistics

image name	value of χ^2	elapsed classification time (sec)	percentage of image classified	classification accuracy
city.lan	256	60	94.86	88
landcover.lan	350	61	95.1	85
roch84.lan	160	29	95.53	86
seashore.lan	145	27	95.9	90

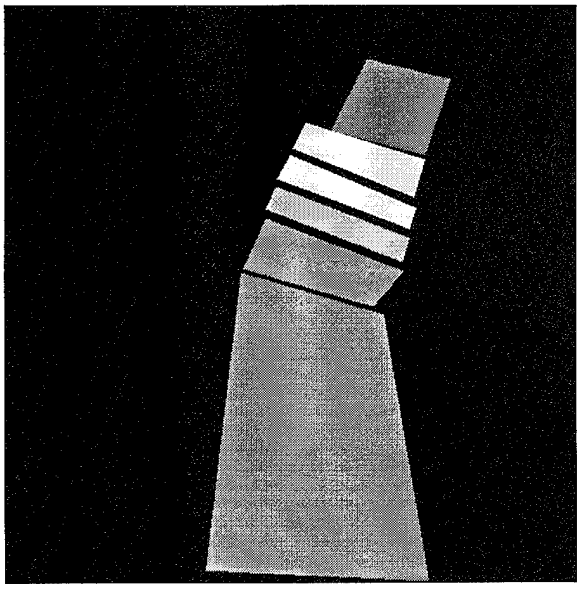
The images were classified next with the nPDF algorithm utilizing the same training data. The training data was first projected in varying nPDF spaces until no clusters overlapped, or until any such overlap was minimized. Recall that nPDF space is determined by both a hypercube corner pair and scale factor selection. The original image was then projected utilizing the same parameters so that the extent of each class in the projected space could be determined. This information was then used to build the classification LUT. This process is depicted in Figure 6.1.1 through 6.1.4.



nPDF projection
of city image



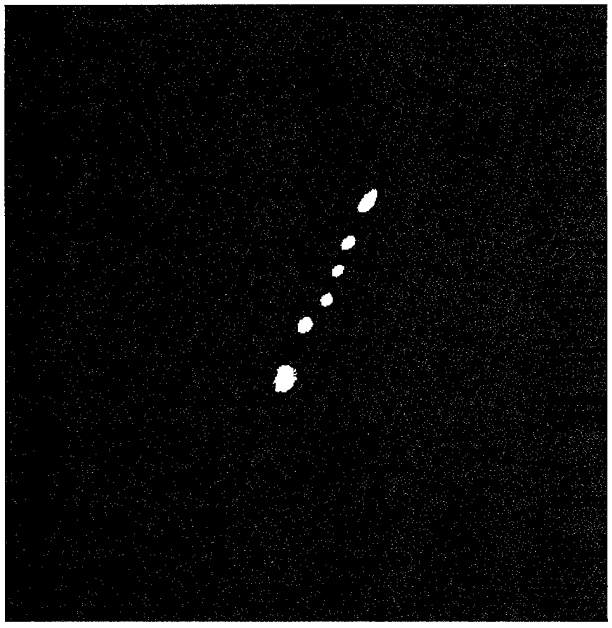
nPDF projection
of city training data



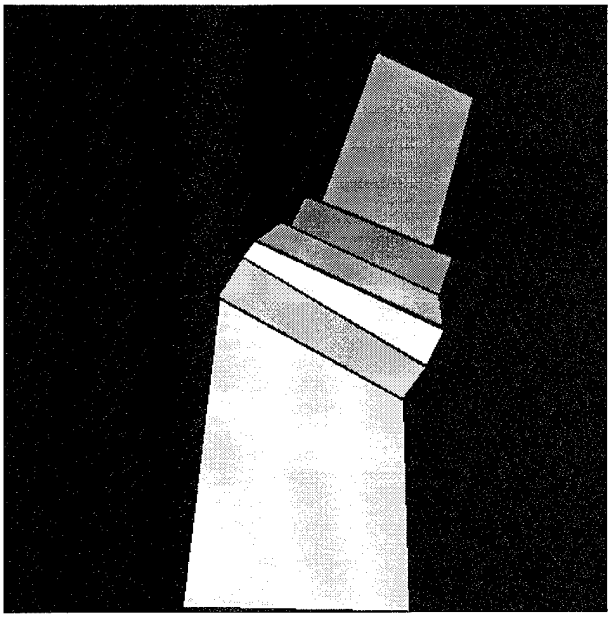
LUT created
from training data

Figure 6.1.1 - nPDF development for the city.lan image

nPDF projection of
landcover image

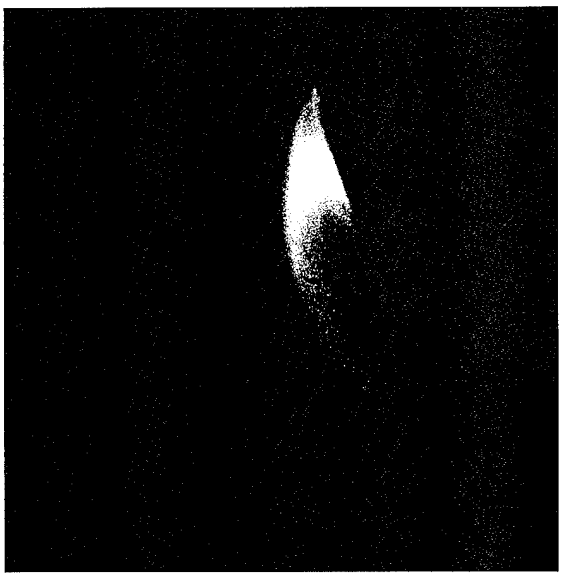


nPDF projection of
landcover training data

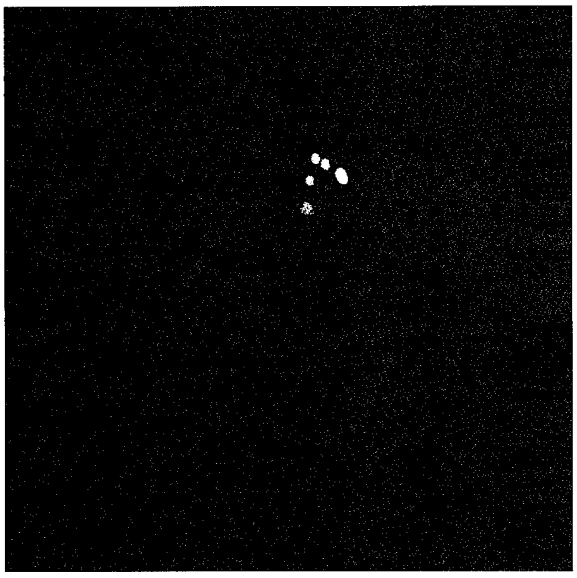


LUT created
from training data

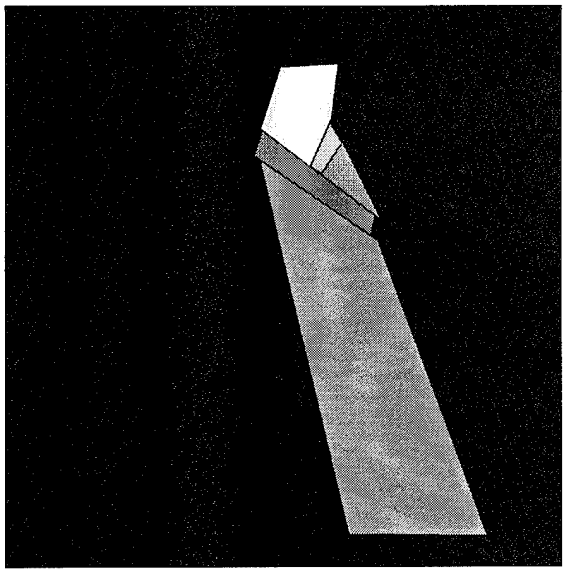
Figure 6.1.2 - nPDF development for the landcover.lan image



nPDF projection
of rochester image



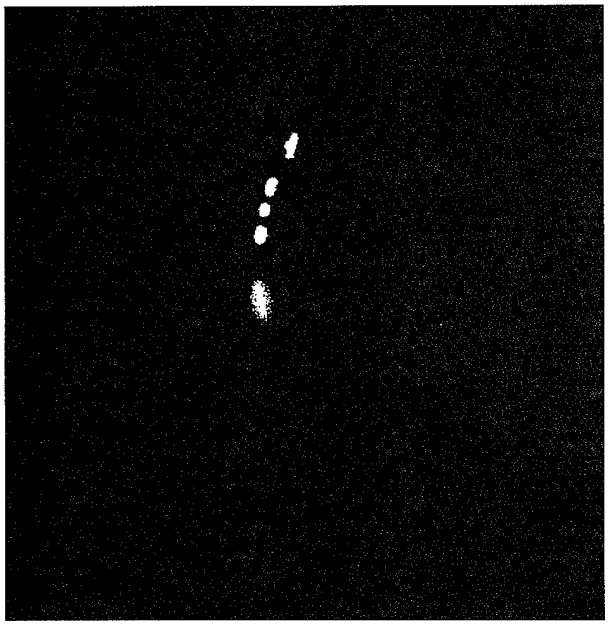
nPDF projection of
rochester training data



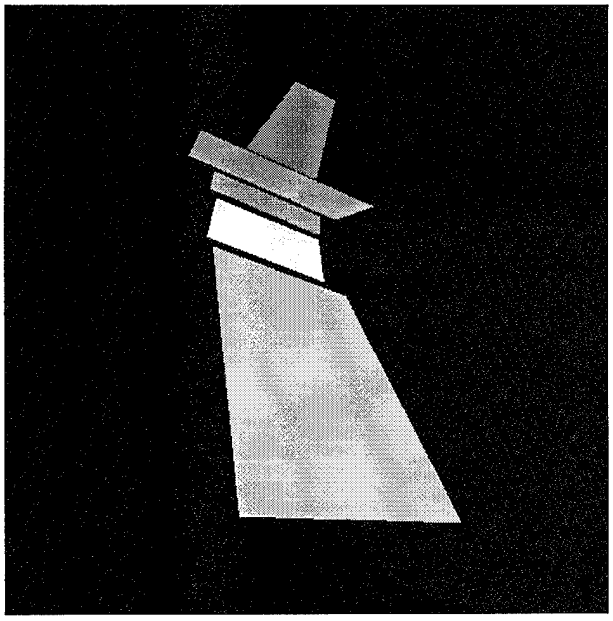
LUT created
from training data

Figure 6.1.3 - nPDF development for the roch84.lan image

nPDF projection of
seashore image



nPDF projection of
seashore training data



LUT created
from training data

Figure 6.1.3 - nPDF development for the seashore.lan image

The important nPDF LUT creation statistics are summarized in Table 6.1.4.

Table 6.1.4 - Summary of nPDF LUT statistics

image name	hypercube corners	scale factor	projection time for training data (sec)	projection time for image (sec)
city.lan	1 & 4	512	15	51
landcover.lan	1 & 4	512	4	59
roch84.lan	1 & 2	1,024	6	35
seashore.lan	1 & 3	512	14	32

Note that the elapsed time is determined by the number of pixels in the image or training set. The number of data classes has no effect on the either the projection or classification time, but it does make constructing the LUT more difficult and time consuming. In addition, it is important to note that the only LANDSAT image in the test group required the largest scale factor, and (as we shall see) has the lowest classification accuracy. The reason is the fixed gain of the LANDSAT TM sensor, which does not effectively utilize the available dynamic range of digital count values, while the exposure control of the M7 sensor does so. LANDSAT TM images will therefore always have lower dynamic ranges than comparable M7 images. In nPDF projection space, this will manifest itself as more densely packed class clusters. In turn, this forces larger scale factor values to drive the class clusters apart. Since the class clusters are closer together, any errors in the LUT that designates boundaries between the classes will lead to larger classification error and lower classification accuracy. The images were then classified with the LUTs and associated parameters just described. Reported classification accuracies reflect the simple accuracy metric with respect to the fuzzy K-means derived truth image. Table 6.1.5 summarizes the important classification parameters.

Table 6.1.5 - Summary of nPDF classification statistics

image name	elapsed classification time (sec)	percentage of image classified	classification accuracy
city.lan	53	95	88
landcover.lan	56	93	78
roch84.lan	34	91	70
seashore.lan	31	93	86

Note that it was not possible to achieve the target image classification percentage in all cases with this classification approach. This is due to the fact that it is very difficult to construct LUTs with boundaries that perfectly adjoin without overlap. Any spacing between class polygon regions results in pixels that should have been assigned to a class being improperly relegated to the background. Relatively low classification accuracies can be attributed to the inherent data dimensionality reduction of the nPDF projection operation. In moving from six- to two-dimensional space, information is irretrievably lost which leads to lower classification accuracies.

The images were then classified with the fuzzy ARTMAP neural network utilizing the same training data as the previous classification methods. As with the GML approach, the neural network must construct a mathematical model of the data. Note that the elapsed time to create the neural network depends upon the value of the vigilance parameter ρ, the number of training points and their variance, the number of training classes, and the resulting number of classification nodes created. This statement may not be readily evident, but it is easily explained. As the algorithm is creating the artificial neural network, it must evaluate the membership of each training exemplar in the training set with respect to each of the classification nodes. As the number of classification nodes created increases (recall that this number is driven by the variance of the training set and the value of ρ), more and more sets of calculations are needed for each training exemplar.

The value of ρ was adjusted to achieve the target classification percentage of 95%. Increasing the vigilance parameter would have produced a smaller hyper-rectangle in feature space leading to more accurate classification, but this would have occurred at the expense of classifying less of the input image. The fuzzy ARTMAP network creation statistics are summarized in Table 6.1.6.

Table 6.1.6 - Summary of fuzzy ARTMAP network statistics

image name	ρ for ART_a	number of ART_a nodes	number of learning iterations	elapsed time to create network (sec)
city.lan	0.88	6	2	51
landcover.lan	0.91	6	2	16
roch84.lan	0.91	5	2	8
seashore.lan	0.9	12	3	72

Once each neural network has been constructed, the images were classified. All values in the associated parameter files were identical with the exception of the size of the training set, the number of classification nodes, the number of target classes, and the value of the minimum vigilance parameter. Resulting classification time is affected by the image size and the number of classification nodes in the neural network. This is most evident in the seashore.lan image test case. Though the image is the smallest test image, the combination of the variance of the training classes and value of the vigilance parameter lead to the creation of multiple classification nodes for each class. This leads to the largest elapsed time for network creation and image classification. As the image is classified, each pixel must have its membership evaluated for each classification node. As expected, this can greatly increase classification time. Table 6.1.7 summarizes the performance of fuzzy ARTMAP image classification. The accuracy value reported is the simple accuracy with respect to the truth image created by the fuzzy K-means algorithm.

Table 6.1.7 - Summary of fuzzy ARTMAP classification statistics

image name	elapsed time to classify image (sec)	percentage of image classified	classification accuracy
city.lan	41	97	91
landcover.lan	46	96	89
roch84.lan	28	96	90
seashore.lan	46	96	93

For comparison purposes, Table 6.1.8 summarizes the classification times and accuracies for the different classification algorithms. The classification time for the GML approach reflects the sum of the time required to make the statistics file and classify the image. Similarly the time reported for the fuzzy ARTMAP approach reflects the time required to create the neural network and classify the image. The reported time for the nPDF classification approach reflects the sum of the time to project the training data and the original image. The time required to create the LUT is not reported. This procedure should not be rushed as classification accuracy is wholly dependent upon it. It can be realistically estimated to require approximately 20 minutes to construct a five- or six-class nPDF LUT.

Table 6.1.8 - Summary of classification times and accuracies

image name	GML time (sec)	GML accuracy	nPDF time (sec)	nPDF accuracy	ARTMAP time (sec)	ARTMAP accuracy
city.lan	71	88	104	88	92	91
landcover.lan	67	85	115	78	62	89
roch84.lan	31	86	71	70	36	90
seashore.lan	40	90	63	86	118	93

To aid in comparison, Figure 6.1.5 graphically displays the classification accuracy and Figure 6.1.6 depicts the required image classification time.

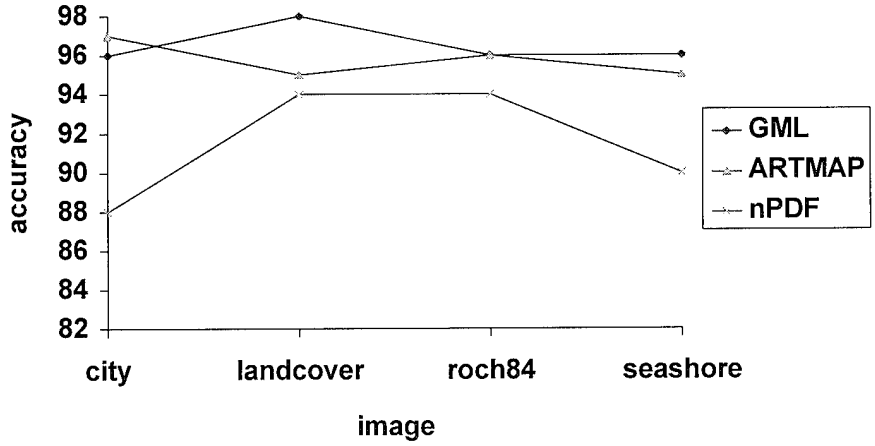


Figure 6.1.5 - Plot of task 1 classification accuracies

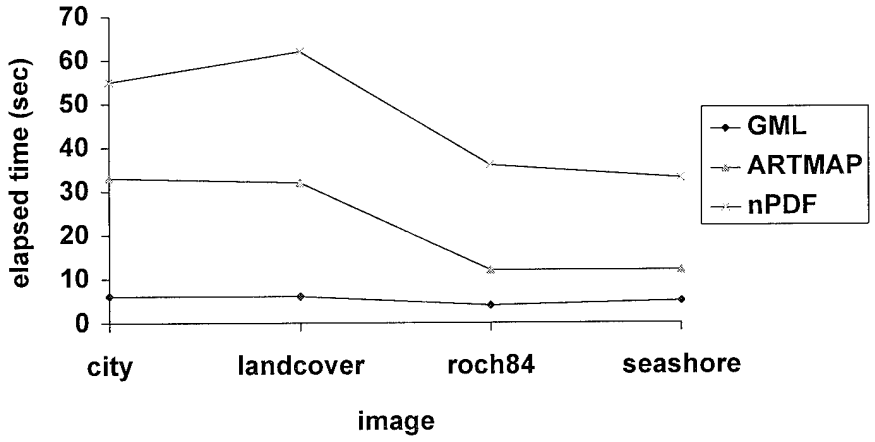


Figure 6.1.6 - Plot of task 1 elapsed classification times

Figure 6.1.7 was formed by computing the ratio of the classification accuracies to the required training/classification time. The objective of this plot is to visualize the performance of each algorithm with respect to accuracy and required classification time.

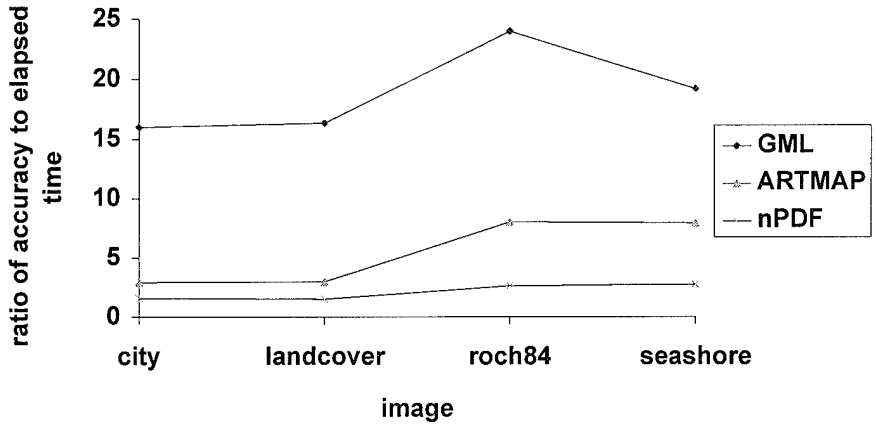


Figure 6.1.7 - Plot of task 1 ratio of accuracy to classification time

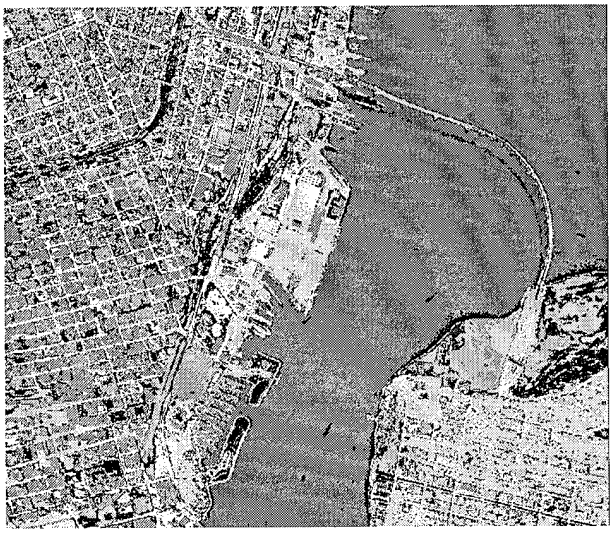
Note that the fuzzy ARTMAP classifier consistently produced the greatest classification accuracy. When the number of ART_a classification nodes equaled the number of classes of data, the algorithm's classification time is favorably comparable to that of the GML. When the vigilance parameter is increased, or there is great variation in the data, its performance suffers due to the number of calculations that must be performed on each individual pixel. The nPDF algorithm consistently produced the lowest classification accuracies coupled with the greatest classification/training time. This can be attributed to the inherent smearing and loss of data present in any projection technique. This approach is not without merit. Its greatest strength lies in its data visualization properties.

A variety of classification metrics were developed for this study. The various metrics were graphed for each image. This graph is present in appendix C. For this task,

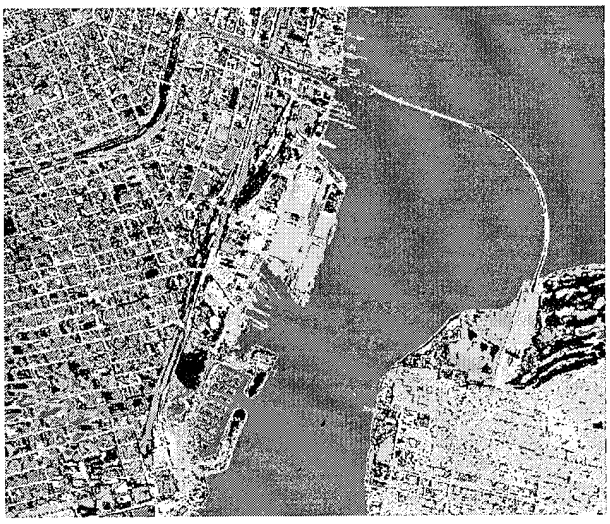
the classification accuracy metrics were consistently found to be ordered with simple accuracy first, Brennan and Prediger's Kappa, followed by the standard kappa coefficient. For this reason, only the simple accuracy metric was reported.

The color plates in Figure 6.1.8 through 6.1.11 depict the classification maps for each image and each classification methodology. All colormaps are encoded in the same manner. Class 1 is red, class 2 is blue, class 3 is green, class 4 is purple, class 5 is yellow, and class 6 is cyan. In this way the output from the class statistics module can be visually coupled with class statistics information.

fuzzy K-means



GML



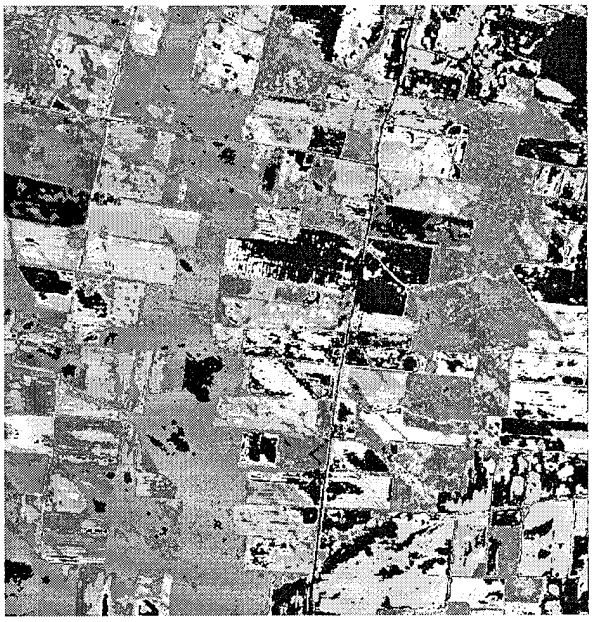
fuzzy ARTMAP

nPDF

Figure 6.1.8 - task 1 classification maps for the city.lan image

nPDF

GML



fuzzy ARTMAP

nPDF

Figure 6.1.9 - task 1 classification maps for the landcover.lan image

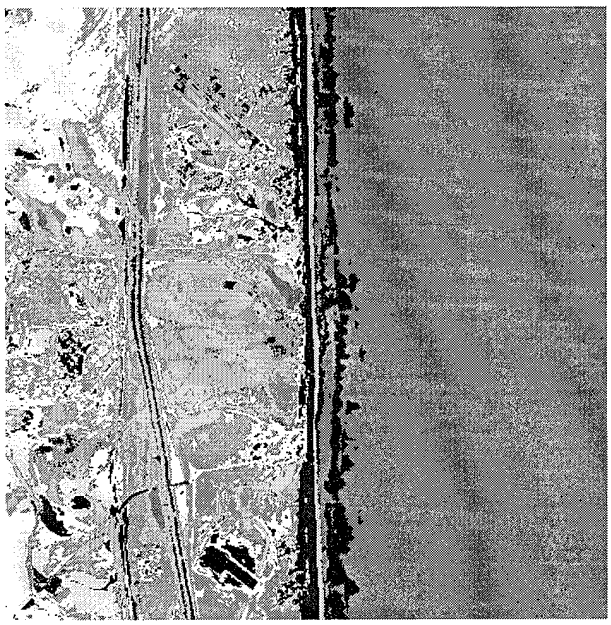
fuzzy K-means

GML

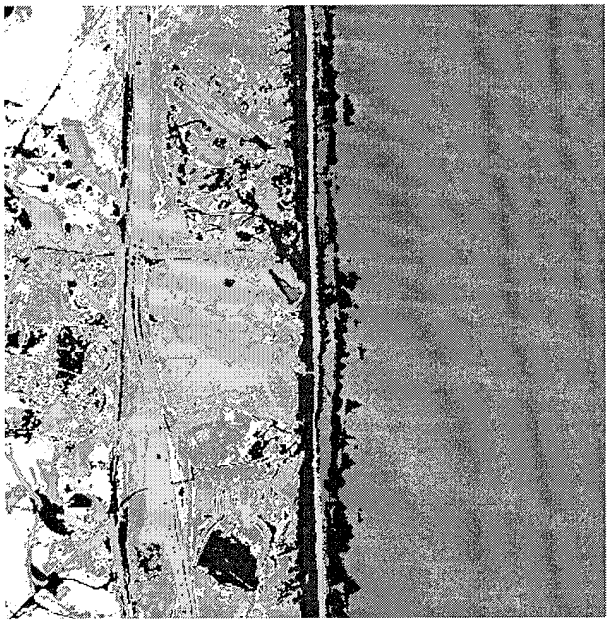
fuzzy ARTMAP

nPDF

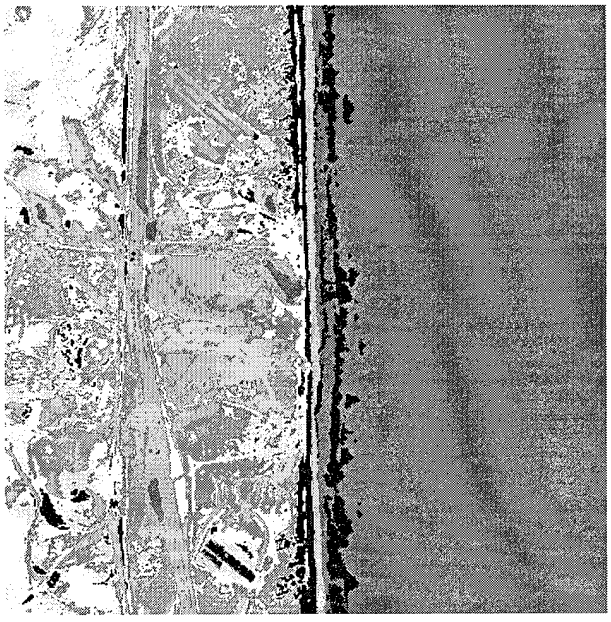
Figure 6.1.10 - task 1 classification maps for the roch84.lan image



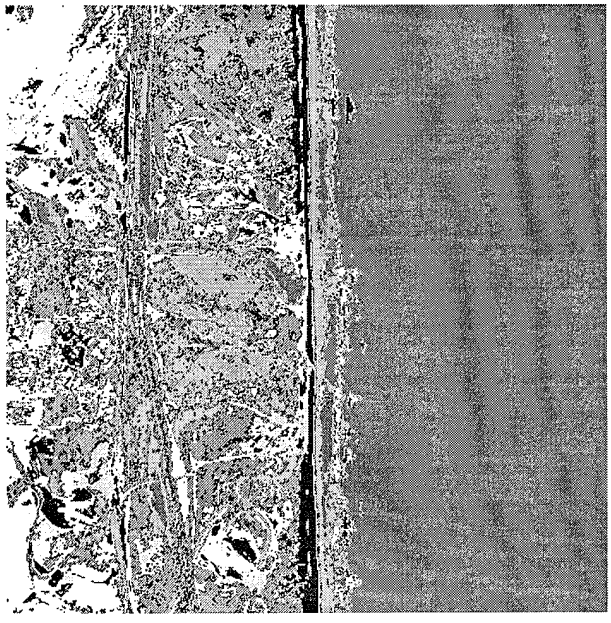
fuzzy K-means



GML



fuzzy ARTMAP



nPDF

Figure 6.1.11 - task 1 classification maps for the seashore.lan image

6.2 Task 2: Hybrid Classification Discussion and Results

Task two of this study utilized a hybrid classification methodology. In this scheme, the input image is first segmented to produce an image composed of tightly clustered data, and this image is then classified. The nPDF algorithm was used to segment the city.lan and landcover.lan images. The water class from the city image and two tightly intermingled vegetation classes from the landcover image were used to create a segmented image comprised of just the classes of interest. These segmented images were then processed by the fuzzy K-means algorithm to form trusted training classes and truth images with the method described in section 6.1 of this report. The training data for the images were passed to the GML and fuzzy ARTMAP classifiers where the segmented image was classified by each algorithm. Note that since numerous classes were made from tightly clustered data, the resulting training class sets will have mean vectors in close proximity to each other. The goals for this task were to experiment with hybrid classification, classify more than 95% of the segmented image, and to test the performance of the algorithms when operating upon data that is not well separated in feature space.

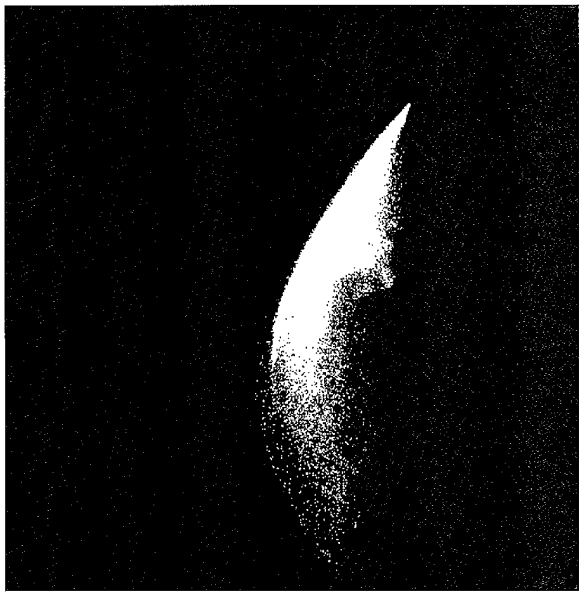
The hybrid classification methodology should produce higher classification accuracies and lower elapsed times. This observation is partly due to the fact that only a portion of the image needs to be processed. Because of this, classification times will be reduced. The "skip zero vectors" features of the classifier modules will be used to support this operation. In addition, little extraneous data is passed to the classification stage from the segmentation phase. Decreased misclassification will be observed in both algorithms due to the great reduction of extraneous data. As such, the subsequent classification algorithms can "concentrate" on the detailed classification task at hand.

As previously stated in the introduction, the nPDF algorithm was used first to segment the city and landcover images to form the city_water and seg_landcover multispectral images. This algorithm is most appropriate for the segmentation task as it lends itself to situations where class separation is large and classification accuracy is not paramount. This statement can be readily reinforced by reviewing the performance of the algorithm in task one. Select portions of the LUTs used in this task were reused to create the segmentation LUTs. In an image segmentation mode, the nPDF algorithm functions similarly to that used in a classification role with one important difference. Instead of producing a classification map, this operation produces a new multispectral image composed of pixels that fall within the classification boundaries of the LUT in the projected feature space. Any pixels that did not fall within the boundaries of the LUT classification regions were assigned zero vectors in the resulting output images. Figure 6.2.1 on the following page depicts the nPDF approach to image segmentation and Table 6.2.1 highlights the important statistics for the segmentation operation.

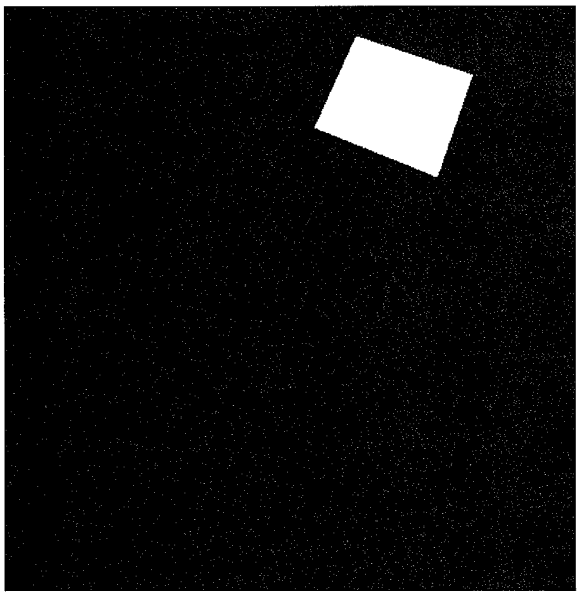
Table 6.2.1 - nPDF segmentation statistics

image name	hypercube corners	scale factor	segmentation time for image (sec)	percentage of image segmented
city_water.lan	1 & 4	512	54	37.1
seg_landcover.lan	1 & 4	512	57	39.5

The resulting segmented images were then clustered with the fuzzy K-means algorithm to create truth images and training data. Once again, the high membership value was manipulated to ensure statistically sound training sets. Similarly, the low membership value was varied to classify the vast majority of the image to form a truth image. The important parameter from the clustering operation are summarized in Table 6.2.2.

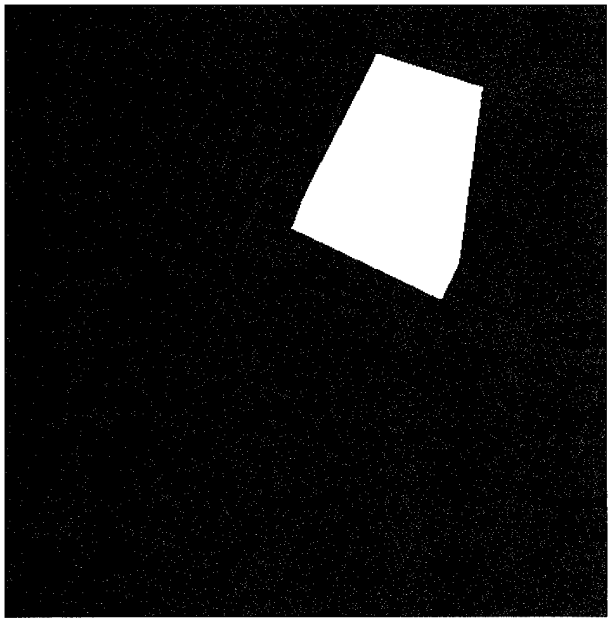


nPDF projection
of city image



Segmentation LUT

nPDF projection of
landcover image



Segmentation LUT

Figure 6.2.12 - nPDF segmentation LUT development

Table 6.2.2 - Clustering statistics for hybrid classification

image name	number of cluster centers	membership values	number of iterations	elapsed time (sec)	percentage of image classified
city_water.lan	4	0.85 & 0.3	12	711	37.1
seg_landcover.lan	5	0.85 & 0.3	12	1,041	39.5

Once the training data was obtained, a parametric model of the data was constructed for the GML classifier. Table 6.2.3 summarizes the number of pixels in each set, the number of classes in each training set and the time required to calculate the mean vectors, the inverses of the variance-covariance matrices, and their determinants.

Table 6.2.3 - Training class statistics for hybrid GML classification

image name	number of classes	number of points in training set	elapsed time to create statistics file (sec)
city_water.lan	4	52,458	3
seg_landcover.lan	5	22,788	2

With a parametric model describing the distribution of the data, GML classification was readily accomplished. The value of χ^2 was manipulated to achieve the target image classification percentage value of 95%. A percentage less than 100% was utilized to provide a fair comparison between the performance of the two classifiers. This was accomplished by forcing both the neural network and the GML classifier to discriminate the outlying data elements. If 100% classification had been used instead, the GML approach would have suffered from errors due to improper inclusion of pixels in a class due to large χ^2 distances. Similarly, the run-time performance of the ARTMAP classifier would have been artificially enhanced due to the ARTMAP's attempt to maximize its generalization of feature space division. Fewer recognition regions require fewer weight

vectors to describe them and fewer calculations on each image pixel. The fuzzy ARTMAP was forced to minimize its generalization of feature space by utilizing high values for the vigilance parameter. The reported classification accuracy metric is the simple accuracy, and it is created with respect to the truth image created with the fuzzy K-means approach. Important classification results are summarized in Table 6.2.4.

Table 6.2.4 - Hybrid GML classification results

image name	value of χ^2	elapsed classification time (sec)	percentage of image classified	classification accuracy
city_water.lan	65	15	95.6	93
seg_landcover.lan	90	26	95.2	92

Note that the values of the χ^2 distance are considerably smaller than the values that were utilized in task one. This can be readily explained by realizing that the data is tightly clustered in feature space and has considerably lower spectral extent. As such, GML's hyperelliptical classification regions need only increase a moderate amount to accomplish the desired percentage of image classification. Note that the reported value is with respect to the portion of the segmented image composed of non-zero pixel intensity vectors.

Classification with the fuzzy ARTMAP algorithm was then accomplished utilizing the same training data as for the GML approach. Since the training data is tightly clustered in feature space and having small spectral extent, high values of the vigilance parameter ρ were necessary to achieve the percentage of image classification desired. As expected, a large value for the vigilance parameter produces a neural network with finely granulized feature space recognition regions. Recall that each classification region is defined by the weight vector of a node. In this case, multiple nodes were needed to encompass the spectral extent of the target classes. Table 6.2.5 summarizes the important training statistics.

Table 6.2.5 - Hybrid fuzzy ARTMAP network statistics

image name	ρ for ART_a	number of ART_a nodes	number of learning iterations	elapsed time to create network (sec)
city_water.lan	0.97	9	2	15
seg_landcover.lan	0.94	10	2	8

Once the neural network was created, image classification was easily accomplished. Rapid image classification was realized because only the non-zero pixel intensity vectors must be evaluated. The reported accuracy metric is the simple accuracy with respect to the fuzzy K-means truth image. Important classification statistics are presented in Table 6.2.6 below.

Table 6.2.6 - Hybrid fuzzy ARTMAP classification results

image name	elapsed time to classify image (sec)	percentage of image classified	classification accuracy
city_water.lan	22	95.1	97
seg_landcover.lan	29	94.1	95

Table 6.2.7 summarizes the hybrid classification results. The times for the GML approach reflect the image segmentation operation, class statistics determination, and image classification elapsed times. Similarly, the reported elapsed times for the hybrid ARTMAP approach is the sum of the image segmentation, network creation, and image classification operations.

Table 6.2.7 - Summary of hybrid classification results

image name	hybrid GML time (sec)	hybrid GML accuracy	hybrid ARTMAP time (sec)	hybrid ARTMAP accuracy
city_water.lan	72	93	91	97
seg_landcover.lan	85	92	94	95

Figure 6.2.2 graphically depicts the simple accuracy measurements for the city_water and the seg_landcover images classification results.

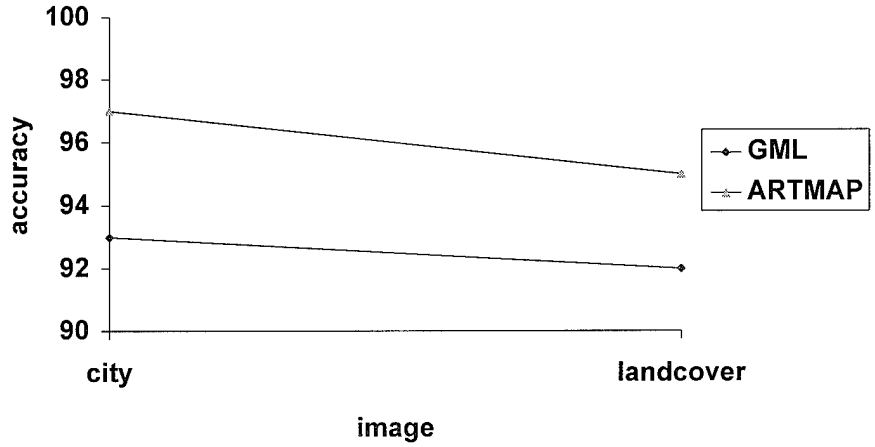


Figure 6.2.2 - task 2 image classification accuracy results

Figure 6.2.3 depicts the required classification times. Note that the reported time figure is the sum of the image segmentation, data modeling or network creation, and classification operations.

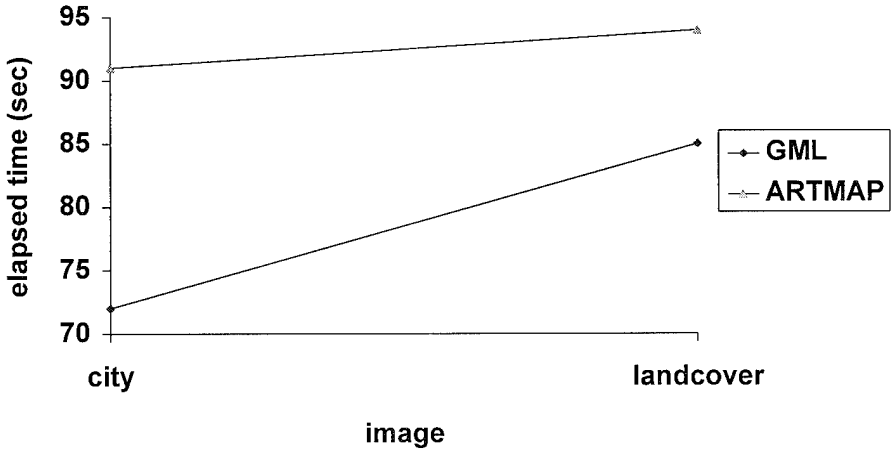


Figure 6.2.3 - elapsed time for hybrid image classification

Figure 6.2.4 was formed by computing the ratio of the classification accuracy to the required classification time.

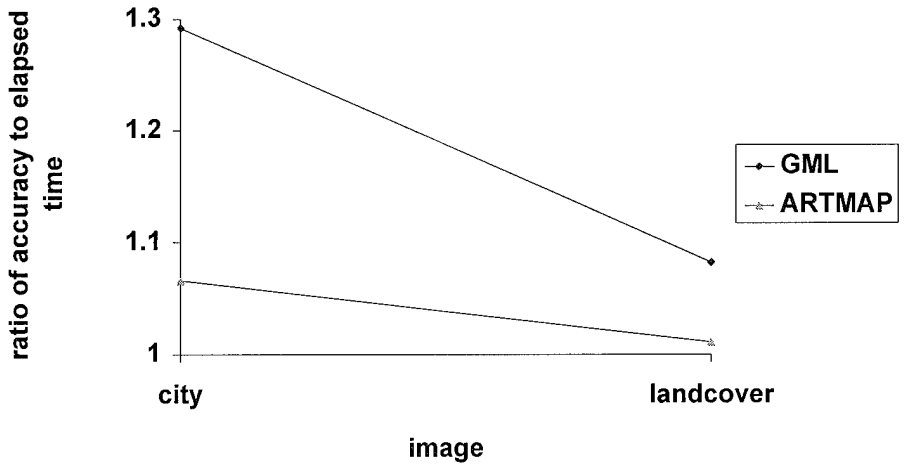


Figure 6.2.4 - ratio of classification accuray to elapsed time for hybrid image classification

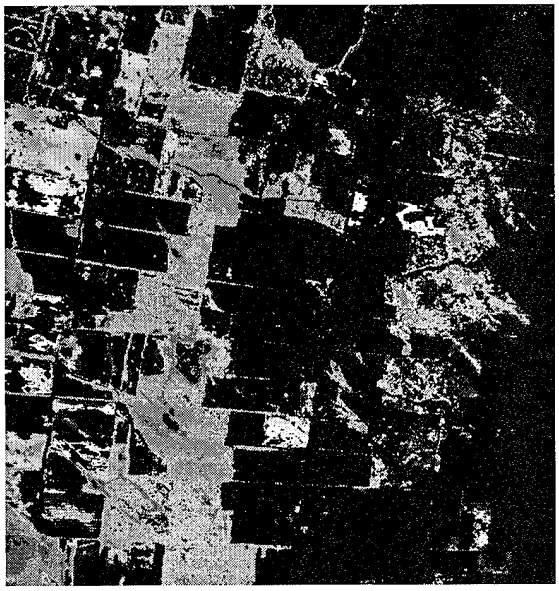
As was the case in task 1, the fuzzy ARTMAP approach produced the best accuracy, but at the cost of greater processing time than the GML approach. The effect of minimized class separation is partly the cause of this observation. It is interesting to note that the training data for the classifiers were projected into nPDF space and classification in all cases would have been difficult if not impossible with this methodology. The training data also is not necessarily distributed in a multivariate normal manner. For these reasons, the non-parametric approach may have some inherent advantage. Note that in this case, as in task 1, the training data was collected by the fuzzy K-means algorithm. The ARTMAP benefits from this as a spectrally pure closely clustered data set is presented for training. As such, it is able to create a relatively small number of recognition regions in feature space that provide for the conflicting needs of within-class generalization while providing between-class distinction. While the GML

approach inherently minimizes the effect of spurious training exemplars through the creation of the variance covariance matrix, the neural network employed in this study enjoys no such luxury. The GML approach does not greatly benefit from the spectrally pure training data because outlying data is automatically averaged out. Also note that, if the training data was collected at a very high membership value, then the training data may be spectrally colored by the mathematical processes employed by the training data collection process. This could distort the class orientation information and lead to poor classification. This effect was minimized by ensuring that the membership value produced a training set composed of a statistically significant number of values. Task 3 will explore the performance of these algorithms on user-defined data sets. These data will better describe the spectral extent of a class in feature space at the cost of necessarily including some "impure" training data.

All classification accuracy measures were reported with the simple accuracy measure. As was the case in task one, the values were always found to be in the same order with the exception of the weighted accuracy which attempts to account for size of each class. A graph comparing the various metrics is presented in appendix B.

Figures 6.2.5 and 6.2.6 represent the classification maps produced in this task. They are color coded to the information in the confusion matrices or class statistics files to support visual inspection. Class one is red, class 2 is green, class 3 is blue, class 4 is purple, class 5 is yellow, and class six is cyan.

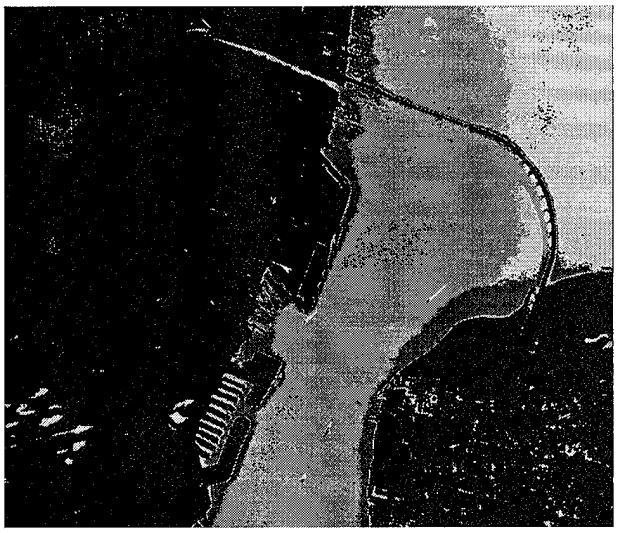
fuzzy K-means



fuzzy ARTMAP

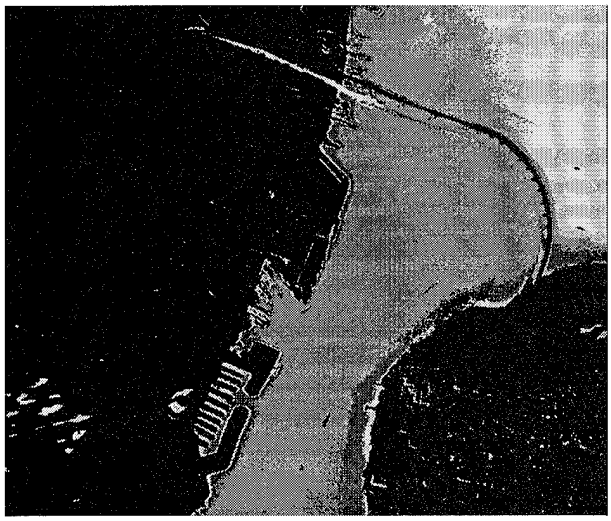
GML

Figure 6.2.5 Hybrid image classification results
for a vegetation class in the landcover.lan image



fuzzy K-means

fuzzy ARTMAP



GML

Figure 6.2.6 Hybrid image classification results
for the water class in the city.lan image

6.3 Task 3 Results and Discussion

The goal of task 3 was to evaluate the performance of the various classifiers when trained and evaluated with user-defined data. The training data were collected for the algorithms by utilizing the user-interactive module. Truth images were constructed by designating polygons representative of the training classes within the images from which the training data were collected. The individual polygons were then segmented from the image to form an evaluation and truth image. The AVS procedure in Figure 6.3.1 was utilized to create the segmented image and truth image.

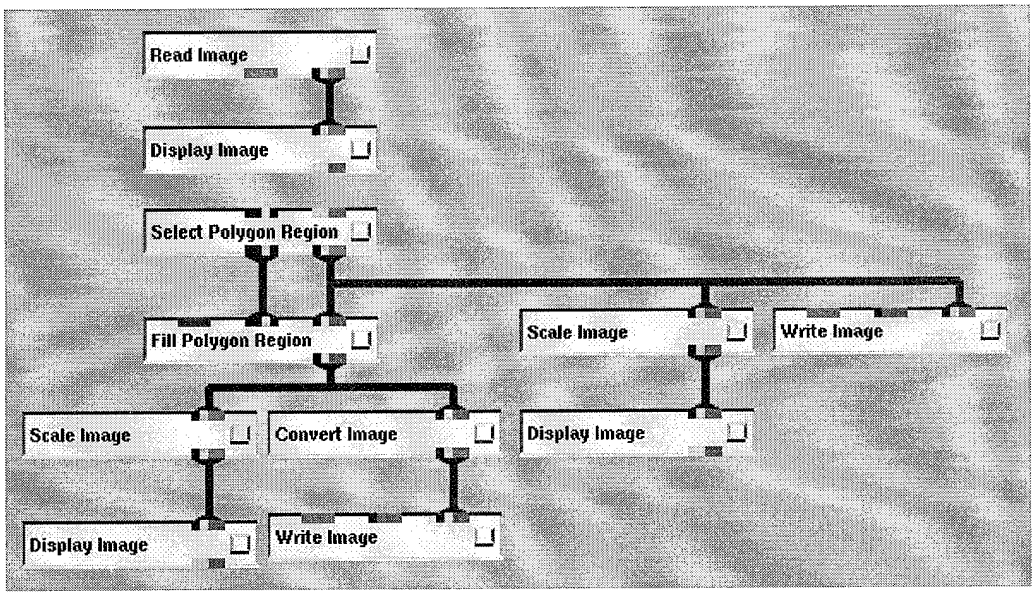


Figure 6.3.1 - AVS network to create evaluation polygons and truth images

This network reads in an multispectral image, displays it, and then permits polygon regions to be overlaid to define the evaluation polygons. This operation is supported by the *Select Polygon Region* AVS module. The multispectral data within the polygons is then passed to the rest of the network. Two important operations then occur in parallel. In the first operation, the multispectral data from the polygons are written to disk files as

individual images. These individual images are later combined into an evaluation image with the network used to combine nPDF LUTs. In a similar fashion, the polygons are passed onto the *Fill Polygon Region* module where they are filled with the class number. These polygons are written to disk and combined to form a truth image to support measurements of classification accuracy. Figures 6.3.2 through 6.3.5 graphically depict the training and evaluation polygons used for each class in each image. To aid in interpretation, training polygons are colored blue while evaluation polygons are filled with red.

Care was taken during the collection of training data to ensure that statistically significant numbers of points were included in each training class. Once the training data was collected, image classification proceeded as previously described. The first step in the GML process was the creation of the class statistics files. Table 6.3.1 summarizes the important statistics from this process and detailed statistics for each of image are presented in appendix B.

Table 6.3.1 - Summary of GML statistics file parameters

image name	number of classes	number of points in training set (sec)	elapsed time to create statistics file (sec)
city.lan	5	17,502	2
landcover.lan	4	15,422	2
roch84.lan	3	4,407	1
seashore.lan	5	9,588	1

After the statistics files have been created, GML classification was readily accomplished. Since the image being classified was segmented to support evaluation, the skip zero vectors option was employed when using the AVS modules. Table 6.3.2 summarizes the classification parameters and statistics.

training polygon

evaluation polygon

Figure 6.3.2
landcover.lan

1 = scrub
2 = vegetation
3 = trees
4 = bare soil

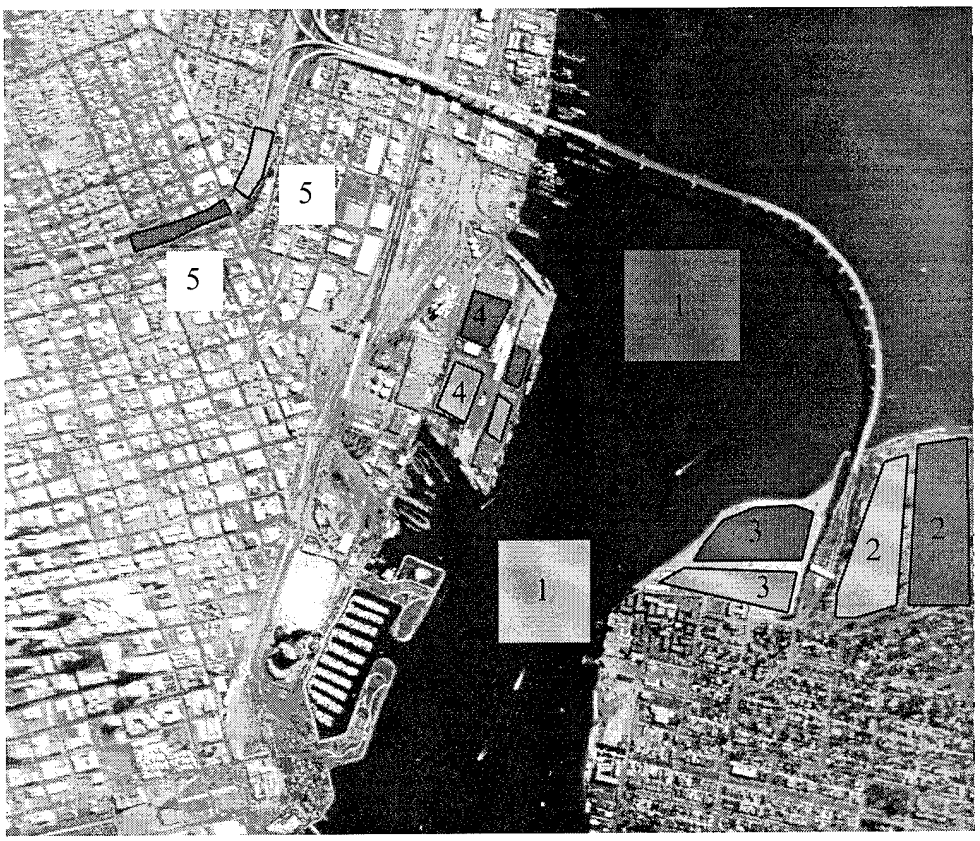
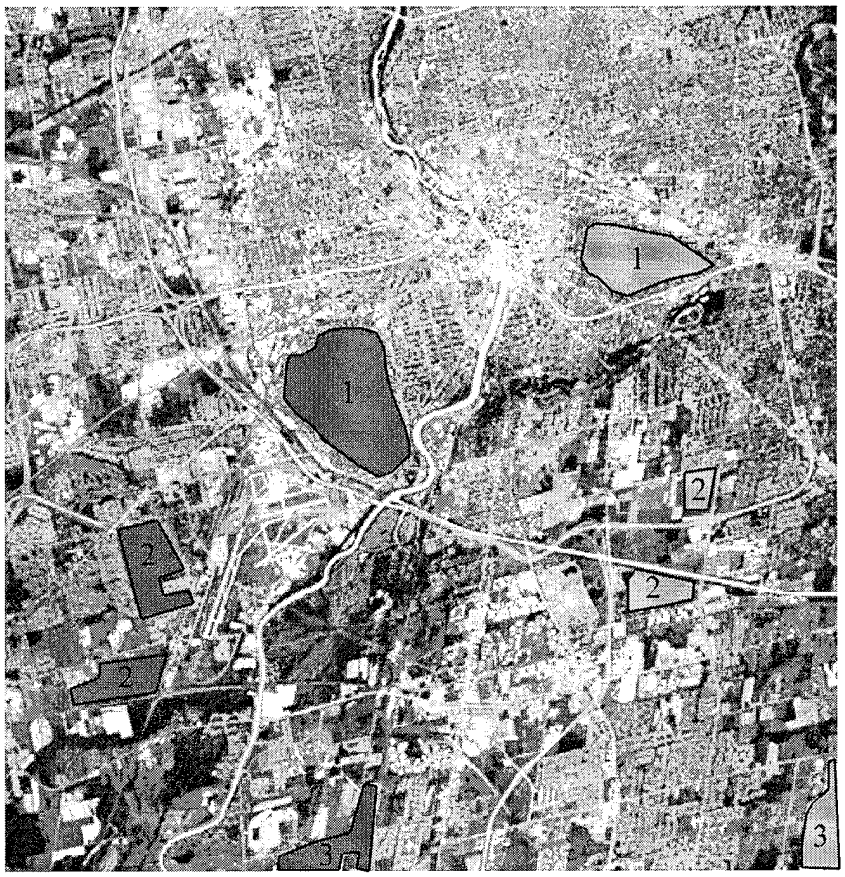


Figure 6.3.3
city.lan

1 = water
2 = scrub
3 = bare soil
4 = roof
5 = asphalt



training polygon

evaluation polygon

Figure 6.3.4
roch84.lan

1 = urban
2 = vegetation
3 = soil

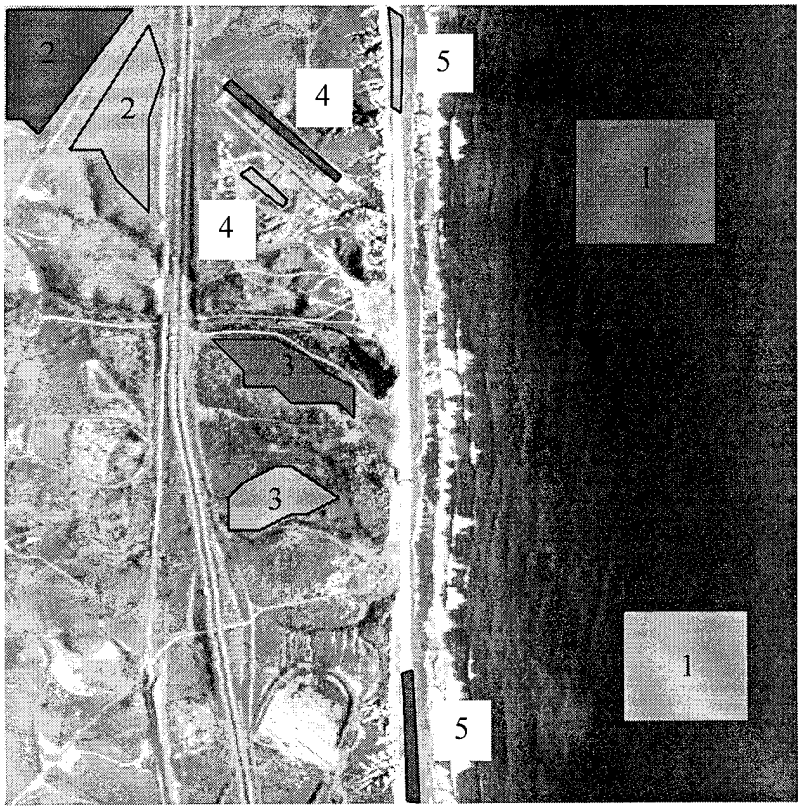


Figure 6.3.5
seashore.lan

1 = water
2 = grass
3 = scrub
4 = concrete
5 = sand

Table 6.3.2 - Summary of GML classification statistics

image name	value of χ^2	elapsed classification time (sec)	percentage of image classified	training accuracy	classification accuracy
city.lan	90	4	99	100	96
landcover.lan	100	5	99	96	98
roch84.lan	50	3	99	98	96
seashore.lan	50	4	99	98	96

The variance of the values of the χ^2 distance should be expected. The different classes in the user-defined training data vary in spectral extent in feature space. Its value was manipulated until 99% of the image would be classified with few pixels being assigned to the background class. The training accuracy metrics in the table represents the performance of the classifier on the training data. The classification accuracy metric is the simple accuracy and all detailed confusion matrices are in appendix B.

Image classification with the nPDF algorithm was accomplished next. The training data collected by the user interactive module was projected into different nPDF spaces until minimal class overlap was observed. Table 6.3.3 summarizes the parameters used and the elapsed time required to project the training data.

Table 6.3.3 - Summary of nPDF LUT statistics

image name	hypercube corners	scale factor	projection time for training data (sec)	projection time for image (sec)
city.lan	3 & 4	512	4	51
landcover.lan	1 & 2	512	3	59
roch84.lan	1 & 4	800	1	35
seashore.lan	1 & 3	512	1	32

LUTs were then constructed from the information in the projected training data. Figure 6.3.6 depicts the projected training data and the resulting LUT for visual inspection.

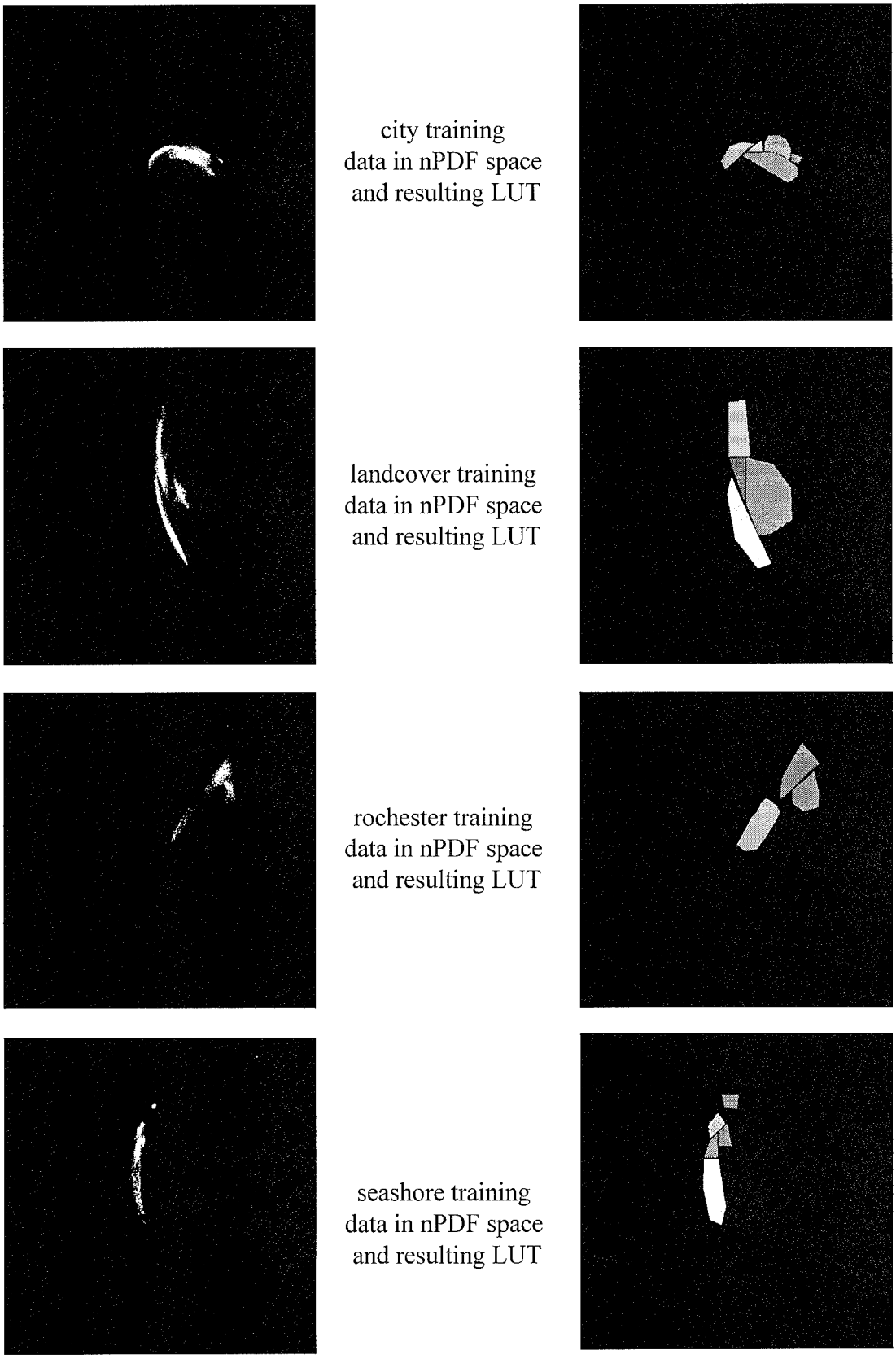


Figure 6.3.6 - nPDF development for the user defined training classes

The LUTs created from the user-defined data proved to be the most difficult to generate. This problem stems from the fact that some training classes had a relatively small number of pixels and there was considerable variance in some of the training data. This made determining classification boundaries a difficult and iterative process. Once the LUTs were constructed, nPDF classification was performed. The important statistics from the nPDF classification operation are summarized in Table 6.3.4 below.

Table 6.3.4 - Summary of nPDF classification statistics

image name	elapsed classification time (sec)	percentage of image classified	training accuracy	classification accuracy
city.lan	3	96	90	88
landcover.lan	7	97	95	94
roch84.lan	2	97	93	94
seashore.lan	3	98	91	90

Note that it was not possible to achieve the desired percentage of image classification. Once again, this is due to the fact that it is very difficult to draw classification boundaries that do not overlap. This same problem is the cause of the relatively low classification accuracy on both the dependent training used in the LUT creation and the independent data that the classification accuracy was evaluated upon.

The same data used with the GML and the nPDF approach were then utilized to train the fuzzy ARTMAP neural network. The value of the vigilance parameter was adjusted to achieve the desired level of image classification. Table 6.3.5 summarizes the network parameters, number of learning iterations, and the elapsed time required to create the networks.

Table 6.3.5 - Summary of fuzzy ARTMAP network statistics

image name	ρ for ART_a	number of ART_a nodes	number of learning iterations	elapsed time to create network (sec)
city.lan	0.91	45	2	15
landcover.lan	0.9	40	3	19
roch84.lan	0.92	58	2	5
seashore.lan	0.92	28	2	6

With the networks created, image classification was then achieved. Table 6.3.6 summarizes the resulting accuracy measurements and the elapsed time required. The training accuracy metric relates the performance of the neural networks on the training data. In all cases, training was stopped once full recognition of the training data was achieved. Note that the accuracy measurement is the simple accuracy and is measured with respect to the truth image.

Table 6.3.6 - Summary of fuzzy ARTMAP classification statistics

image name	elapsed time to classify image (sec)	percentage of image classified	training accuracy	classification accuracy
city.lan	18	99	100	97
landcover.lan	13	99	100	95
roch84.lan	7	99	100	96
seashore.lan	6	99	100	95

Table 6.3.7 summarizes the classification accuracies and elapsed times.

Table 6.3.7 - Summary of classification times and accuracies

image name	GML time (sec)	GML accuracy	nPDF time (sec)	nPDF accuracy	ARTMAP time (sec)	ARTMAP accuracy
city.lan	6	96	55	88	33	97
landcover.lan	6	98	62	94	32	95
roch84.lan	4	96	36	94	12	96
seashore.lan	5	96	33	90	12	95

Figures 6.3.7 and 6.3.8 below graphically depict the classification and elapsed computing time.

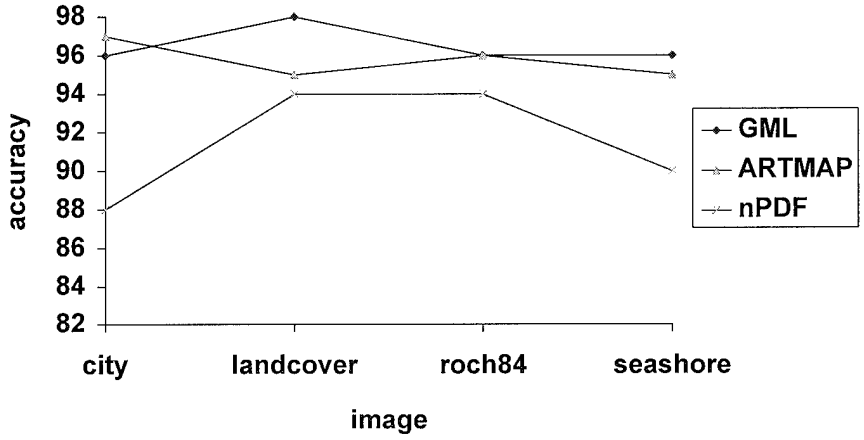


Figure 6.3.7 - Task 3 classification accuracies

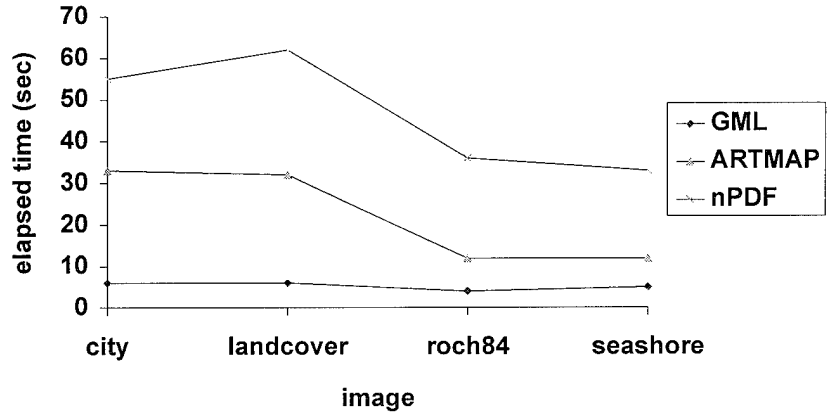


Figure 6.3.8 - Task 3 classification elapsed times

Figure 6.3.9 was created by computing the ratio of the classification accuracy to the required classification time.

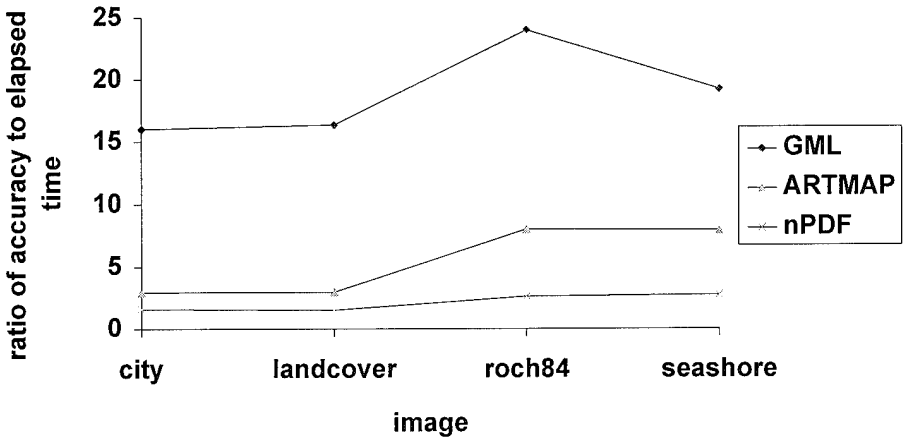
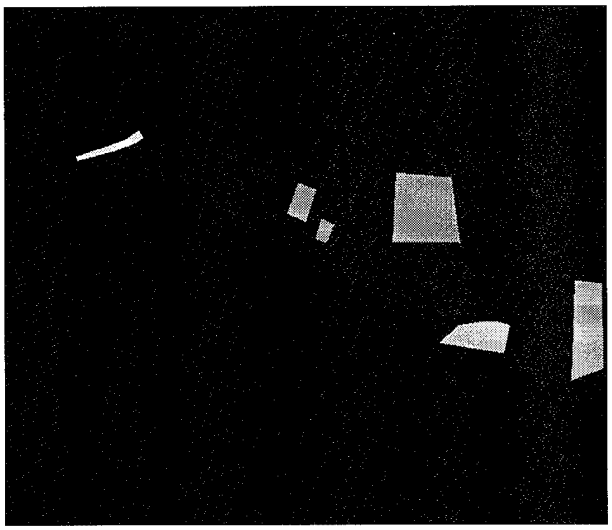


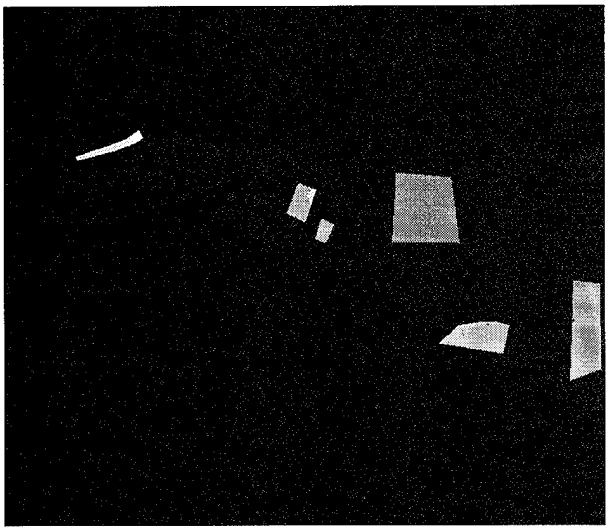
Figure 6.3.9 - Task 3 ratio of classification accuracy to classification time

The results of this task highlight the primary concern present when utilizing nonparametric classifiers. Note that in contrast to its performance in the other tasks, the GML classifier twice displayed better performance than the fuzzy ARTMAP classifier. The high classification accuracy performance of the GML classifier can be most easily attributed to the mathematical properties of the variance-covariance matrix. The matrix contains information about the shape of the training data distribution, its orientation in feature space, and its extent. As such, the parametric classifier has the ability to logically "fill in" missing data points, and the impact of noisy or spurious training data are automatically averaged out. In contrast, the classification performance of the nonparametric classifiers entirely depends on the quality of the training data, and they are inherently unable to account for missing information. Incomplete training data sets will always be encountered when the training data is interactively determined by the image analyst. Due to this reality, GML may represent the optimal image classification strategy when dealing with user-defined training data.

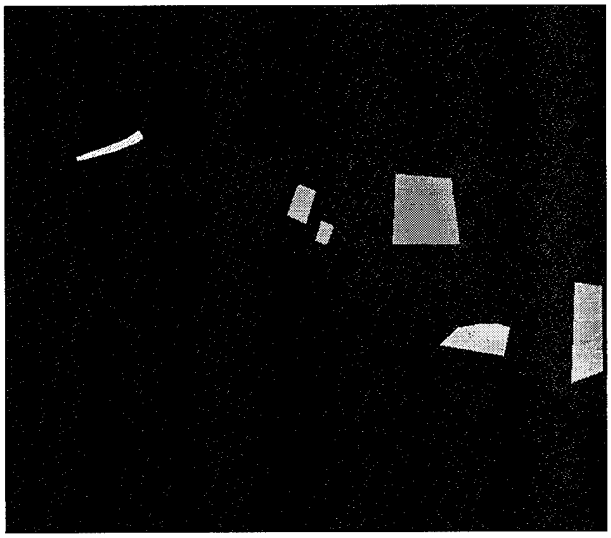
Classification maps from this task are depicted in Figures 6.3.8 through 6.3.11.



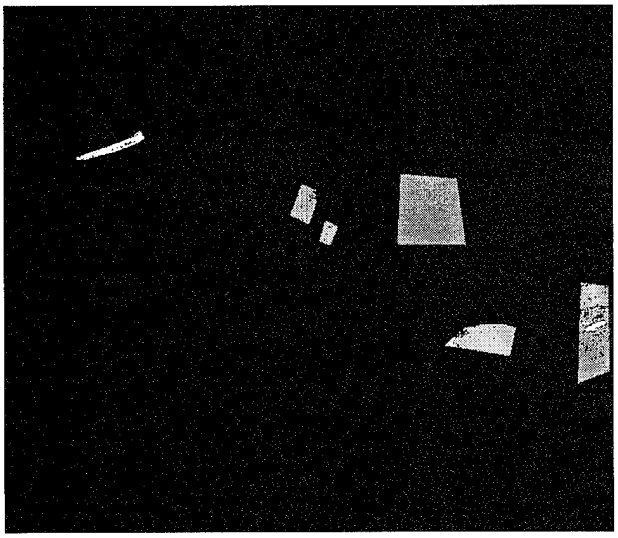
truth image



GML

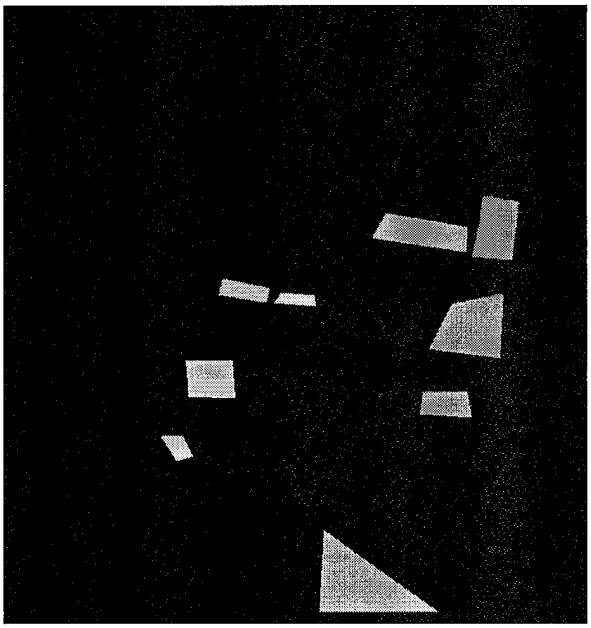


fuzzy ARTMAP

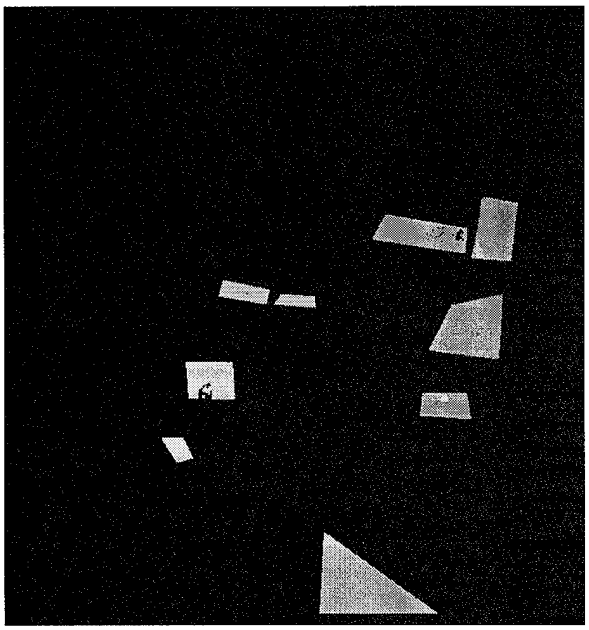


nPDF

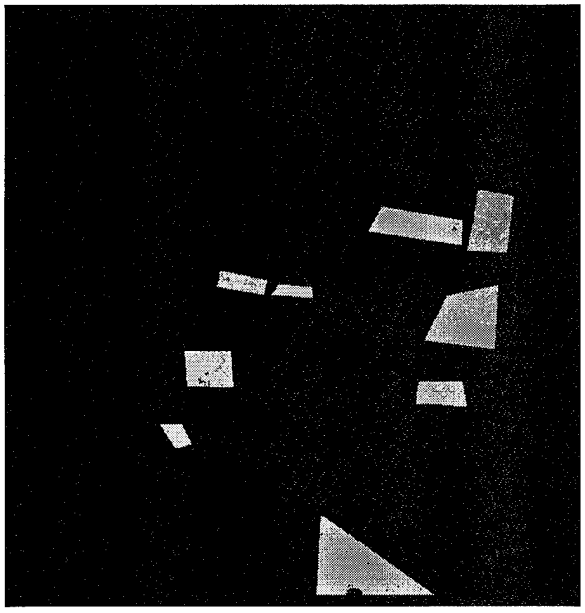
Figure 6.3.8 - task 3 classification maps for the city.lan image



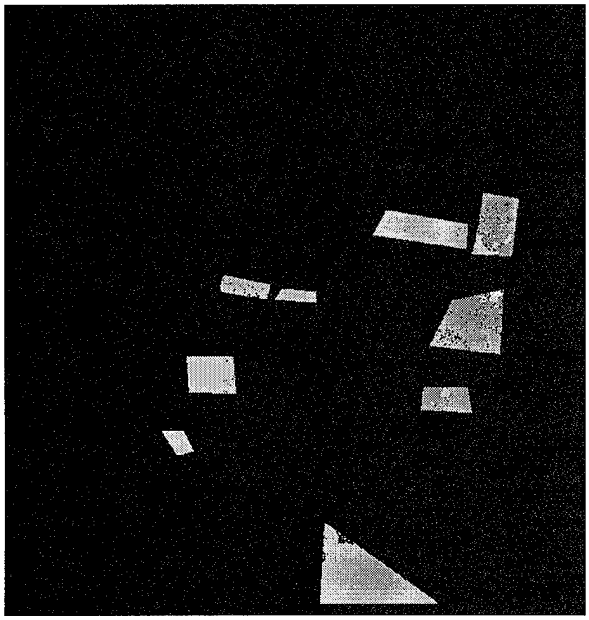
truth image



GML

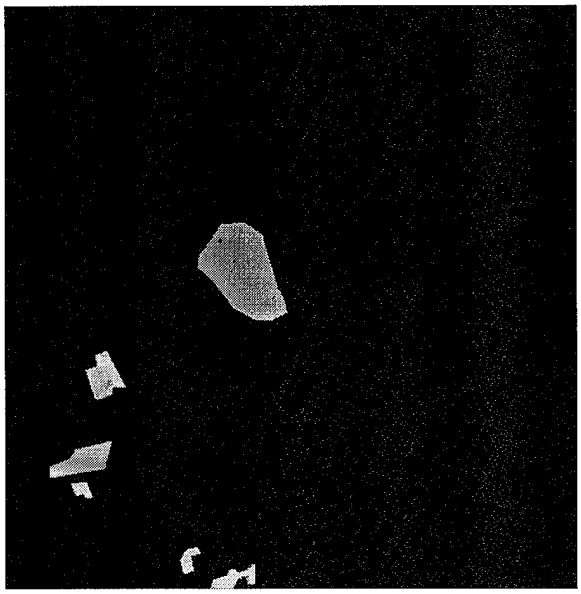


fuzzy ARTMAP

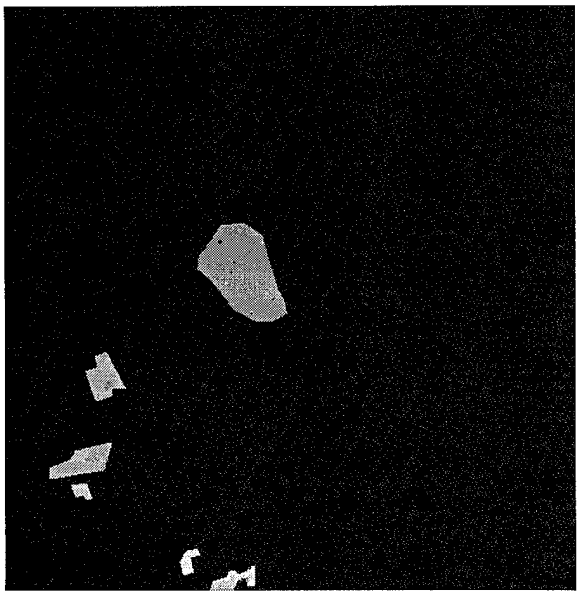


nPDF

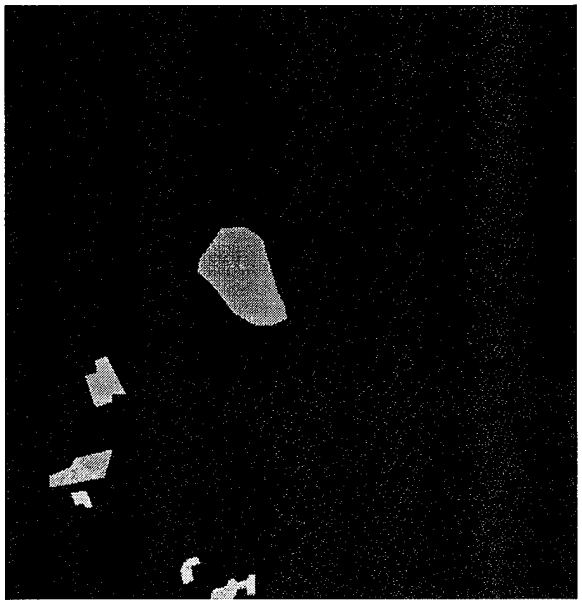
Figure 6.3.9 - task 3 classification maps for the landcover.lan image



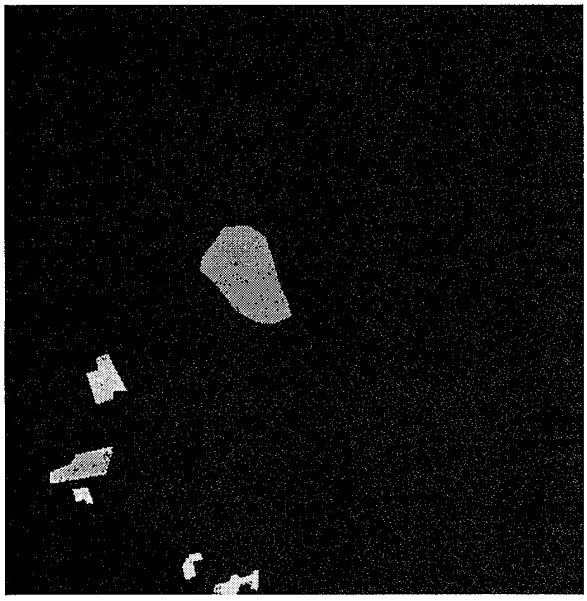
truth image



GML

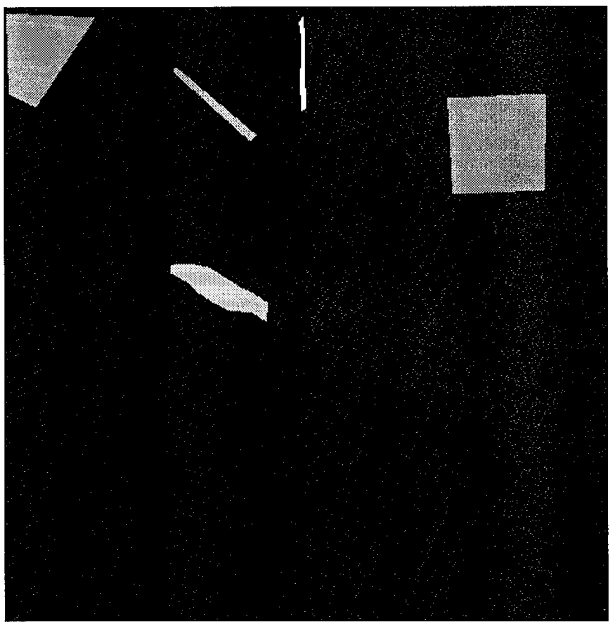


fuzzy ARTMAP

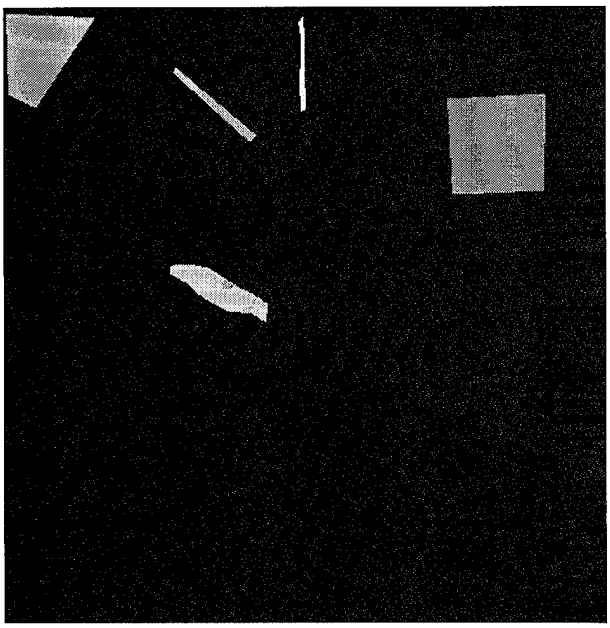


nPDF

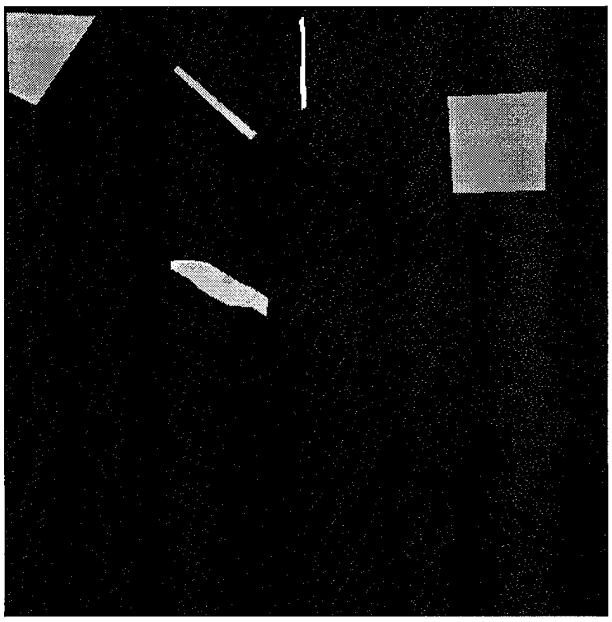
Figure 6.3.10 - task 3 classification maps for the roch84.lan image



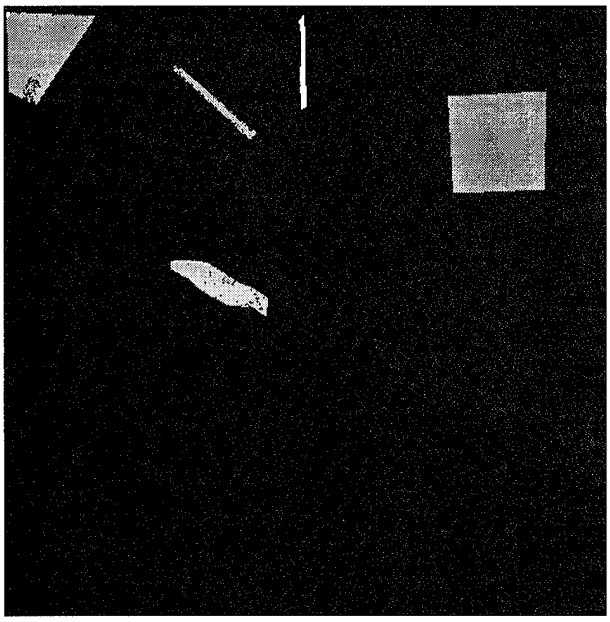
truth image



GML



fuzzy ARTMAP



nPDF

Figure 6.3.11 - task 3 classification maps for the seashore.lan image

7.0 Summary

The fuzzy K-means clustering algorithm was shown to be effective for both creating spectrally pure training data and truth images for measuring classification accuracy. This flexibility is gained by its employment of fuzzy logic through the membership function. Features within an image for which it is very difficult to collect training data, either due to size or sparse positioning, can be effectively sampled with this method. There are three main problems with this approach to image classification. First, the algorithm is extremely computationally intensive. This observation is easily explained by realizing that the membership of each pixel in the image with respect to each desired cluster center must be determined iteratively. Secondly, the clusters formed are computed in an entirely unsupervised manner and may be difficult to visually label. Finally, the data collection methodology inherently colors the training class data which destroys some class distribution information.

The heart of this study concerns itself with the manner in which each classification algorithm divides feature space into recognition regions. Gaussian maximum likelihood utilizes hyperellipsoids, the nPDF algorithm allows the analyst to define arbitrary boundaries in a projection of feature space, and the fuzzy ARTMAP neural network utilizes stacked hyper-rectangles with exception handling.

GML is the classical approach to multispectral image classification. This study has demonstrated that its classification accuracy and computational requirements on user-defined data are difficult to achieve by other methods, even an advanced neural network. The variance-covariance matrix at the core of the algorithm provides not only the location of the classes in feature space, but also a measure of their extent and orientation. In addition, the method used to calculate the variance-covariance matrix from the training data automatically weights the effects of both frequently occurring and outlying data points. Neither of the non-parametric classifiers addressed in this study are

able to accomplish this. The determination of class extent and orientation is achieved through the assumption of normally distributed pixels made when calculating the variance-covariance matrix which forms the core of its data dispersion model. The validity of the normality assumption was shown to come from the averaging effect of the sensor, and be a reasonable assumption in most remote sensing applications.

The nPDF approach to image classification was shown to uniquely involve the analyst in image classification. By interactively drawing class boundaries in a projection of feature space, subtle variations in class boundaries can be accounted for in a manner that is not possible algorithmically. It is important to note that outlying or mislabeled training data are handled in an extremely effective manner. Incorrect training data are automatically grouped into the correct class through the projection operation. This facet of the algorithm was exploited in the hybrid image classification task. The greatest strength of this algorithm is its data visualization properties as separability between classes can be readily interpreted. While class separability can be readily visually interpreted, defining accurate boundaries between the classes proved to be difficult. In addition, this method enjoys no real computational advantage in terms of elapsed time required to classify an image when compared to GML, once the time to project the original image to determine class extent is included. The introduction to this study mentioned that the nPDF algorithm could be potentially useful for determining the number of classes present in an image. This facet of the algorithm proved impossible to demonstrate with the LANDSAT or M-7 imagery used in this study. Had it been possible to achieve, distinct peaks in the nPDF projections of the images would have been noted.

Image classification with the fuzzy ARTMAP neural network produced intriguing results. When it is presented with the spectrally pure training data collected by the fuzzy K-means algorithm, its classification accuracy performance was shown to be unparalleled with only a slight increase in time required to train and classify the image as compared to

GML. This strength springs from the employment of fuzzy set theory and ART dynamics. No other neural network architecture so effectively combines these traits. When user-defined data is utilized with this approach, its greatest weaknesses are highlighted. Large variations in data coupled with cluster centers that are close to one another in feature space result in a neural network with many small hyper-rectangles dividing feature space into recognition regions. This case, which occurs often in the remote sensing application, springs from the attempt to achieve the conflicting goals of maximizing generalization while maintaining separability. This results in numerous computations being completed for each pixel in the image or training set. This manifests itself as increased learning and image classification times. If the network has not been presented with the examples of the complete spectral extent of a class, it is not capable of determining membership in the manner that GML is. Since the data distribution is not modeled, no mathematical inference other than the fuzzy "nearness" can be determined. While GML can determine that a pixel not explicitly encountered during training should "fit" in the distribution of one of its classes, the neural network cannot. It requires training sets composed of pixels that both are spectrally pure and that completely define the spectral extent of the classes in feature space.

The varying measurements of classification accuracy employed for this study were found to always follow the same pattern. The simple accuracy consistently provided the greatest measure of classification accuracy, followed by Brennan and Prediger's kappa, while the standard kappa coefficient always provided the worst measure of accuracy. The weighted accuracy, which attempted to account for class size, produced sporadic results. To compare the classification accuracy performance of the various algorithms, any measurement could be reported, and the simple accuracy was utilized in this study.

In general, it appears that the GML approach to multispectral image classification has not been "dethroned" by either of the non-parametric classifiers utilized in this study. The assumption of normal distributed training and target class data is reasonable. This assumption permits the algorithm to "fill in" missing data that was not present when the data dispersion model was created. Neither of the non-parametric classifiers observed in this study are able to accomplish this. Given robust spectrally pure training data, the fuzzy ARTMAP may provide slightly higher classification accuracies, but this comes at the expense of considerable complexity. Given classes that are readily spectrally separable, the nPDF algorithm produced reasonable results. In certain situations its performance may be optimal. In general, it is hampered by its inherent projection methodology.

The classification algorithms and methods employed in this study were evaluated under ideal "laboratory" conditions. As such, some comments on transitioning this system to an operational role are warranted. It is obvious that training each algorithm on each class in each image to be classified is overly time consuming. It would be desirable to train the classification algorithms on various target classes of interest and then be able to classify any given image. All of the classification algorithms rely on the digital count values present in an image to distinguish between classes. These digital count values are entirely dependent upon the imaging geometry and atmosphere present at image acquisition time. As such, some method must be employed to remove these effects. Typically these methods either model the contributions of the atmosphere at an imaging time, or more simply scale the digital count values present in one image to match those from the image from which the classifier was trained. As expected, there will be some loss of information or introduction of error when either of the preceding approaches are applied. Therefore, while classification results will not be as accurate as if the classifier

was trained with data from the image to be classified, considerable time will be saved by applying the previously determined classification models.

It is also interesting to note that the classification stage of each algorithm is inherently *parallelizable*. Any problem which can be readily divided and computed separately on multiple processors shares this quality. Since the classification results from the algorithms in this study depend only on the digital count values of the pixel in question, the classification operation can be easily divided across several processing units. This concept is easy to envision. Consider the case where we simply divide the input image by the number of available processors. The resulting subimages could then be classified by each processor and then recombined to form a classification map. As such, a very near linear decrease in classification elapsed time can be realized.

7.1 Suggestions for Future Work

It would be interesting to allow the user to select the starting cluster locations interactively for the fuzzy K-means algorithm instead of selecting them in a pseudorandom fashion. This would give the analyst some control of the resulting cluster centers and make labeling the resulting classes somewhat easier.

While the χ^2 distance measure of the GML classification algorithm allows the extent of classes to be controlled, no individual parameter for each class is provided other than the measure arising from the determinant of the variance-covariance matrix. If individual distance measures were employed, varying class extent in feature space could be controlled and compensated.

The effects of varying complement coding for the fuzzy ARTMAP neural network were not explored. Reduced classification and learning times might be achieved, but their impact on classification accuracy cannot be predicted.

No attempt was made to study the effects of normality on the classification accuracy of the different algorithms. It would be interesting to study this effect by intentionally skewing the distributions of the training and evaluation data.

8.0 References

[1] Richards, J.A. "Remote Sensing Digital Image Analysis: An Introduction" Springer Verlag 1993

[2] Johnson, R. A. and Wichern, D. W. "Applied Multivariate Statistical Analysis" Prentice Hall 1992

[3] Cetin, H. A. and Levandowski, D. W. "Interactive Classification and Mapping of Multidimensional Remotely Sensed Data Using n-Dimensional Probability Density Functions (nPDF)" Photogrammetric Engineering and Remote Sensing Vol. 57, #12, December 1991 1579-1587

[4] Cetin, H. A., "nPDF-An Algorithm for Mapping n-Dimensional Probability Density Functions for Remotely Sensed Data" Proceedings of the 10th Annual International Geoscience & Remote Sensing Symposium IGARSS'90, I:353-356

[5] Rosenfield, G. H. and Fitzpatrick-Lins, K. "A Coefficient of Agreement as a Measure of Thematic Classification Accuracy" Photogrammetric Engineering and Remote Sensing Vol. 52, #2, February 1986 223-227

[6] Foody, G. M. "On the Compensation for Chance Agreement in Image Classification Accuracy Assessment" Photogrammetric Engineering and Remote Sensing Vol. 58, #10, October 1992 1459-1460

[7] Dougherty, E. R. "Probability and Statistics for the Engineering, Computing, and Physical Sciences" Prentice Hall 1990

[8] Carpenter, G. A., Grossberg, S., Markuzon, N., Reynolds, J. H., and Rosen, D. B. "Fuzzy ARTMAP: A Neural Network Architecture for Incremental Supervised learning of Analog Multidimensional Maps" IEEE Transactions on Neural Networks, Vol. 3, #5 September 1992

[9] Grossberg, S., Carpenter, G. A., and Reynolds, J.H. "ARTMAP: Supervised Real-Time Learning and Classification of Nonstationary Data by a Self-Organizing Neural Network" Neural Networks, Vol. 4, 1991

[10] Yager, R. R. and Filev D. P. "Essentials of Fuzzy Modeling and Control" Wiley-Interscience 1994

[11] Dunn, J. C. " A Fuzzy Relative of the ISODATA Process and its Use in Detecting Compact-Well Separated Clusters" Journal of Cybernetics, Vol. 3, #3 1973

[12] Frey, B. "An Examination of Distributional Assumptions in LANDSAT TM Imagery" Unpublished Master of Science Thesis, RIT

9.0 Appendices

Appendix Table of Contents

Source code for the fuzzy K-means AVS module

```
/*=========================================================================


Author:         Nessmiller, Steven W.
File Name:      AVS_fuzzy_K_means.c

This software tool creates a fuzzy K means clusters outputs the "classified"
image and optionally writes the training data to an output port.

Revision history:
        12 May 95 now exports training data via the IPIlib
        20 May 95 input image sampling by offset implemented
        28 June 95 routine can skip zero vectors

=========================================================================*/

#include <stdio.h>
#include <stddef.h>
#include <sys/types.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <sys/file.h>
#include <sys/types.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/flow.h>
#include <avs/avs_data.h>
#include <IPIlib.h>
#include "/cis/src/local/avs/src/IPI/avs_types.h"
#include "memutil.h"
#include "random.h"
#include "index.h"

/*
#define RANDOM_DEBUG
#define CENTER_DEBUG
#define CLASS_DEBUG
#define TRAINING_SET_DEBUG
#define MEMBERSHIP_DEBUG
*/
#define DEBUG

#define MAXSTR 80

/* this structure supports the input image */
typedef struct{
        int num_pixels;
        int num_bands;
        int num_rows;
        int num_cols;
}MSI_IMG;

typedef MSI_IMG *msi_img_ptr;

/* this is the classification map */
typedef struct{
        int **array;
        int num_rows;
        int num_cols;
        int num_valid_pixels;
}CLASS_MAP;

typedef CLASS_MAP *class_map_ptr;

/* this is the membership matrix */
typedef struct{
        float ***array;
        int num_rows;
        int num_cols;
        int num_clusters;
}D3_MAT_STRUCT;

typedef D3_MAT_STRUCT *three_d_matrix_ptr;

/* this will be the output training set */
typedef struct{
        int **array;
        int num_pixels;
        int num_bands;
        int num_classes;
}PIX_LST;

typedef PIX_LST *pix_list_ptr;

/* simple matrix structure */
typedef struct{
        float **array;
        int num_rows;
        int num_cols;
}MAT_STRUCT;

typedef MAT_STRUCT *matrix_ptr;

/* function prototypes */
int build_center_vector(AVSfield_char *input_image, msi_img_ptr input_params,
                        matrix_ptr center, int num_clusters, int zero_vector);
float distance(int row, int col, int cluster_num, AVSfield_char *input_image,
               msi_img_ptr input_params, matrix_ptr center);
int calc_membership(msi_img_ptr input_params, AVSfield_char *input_image,
                    three_d_matrix_ptr membership, matrix_ptr center,
                    float *dist, int num_clusters, int offset, int skip_zeros);

int calc_new_centers(msi_img_ptr input_params, AVSfield_char *input_image,
                     three_d_matrix_ptr membership, matrix_ptr old_center,
                     matrix_ptr center,int num_clusters, int offset,
                     int zero_flag);
int mag_diff(matrix_ptr center, matrix_ptr old_center, int movement_threshold);

int build_output_image(three_d_matrix_ptr membership, class_map_ptr classified,

                       float threshold);
int build_training_set(AVSfield_char *input_image, msi_img_ptr input_params,
                       pix_list_ptr pixel_list, class_map_ptr classified,
                       int *indx, int*arr, AVSfield **train_out, int offset);
int write_training_set(pix_list_ptr pixel_list, int *indx);
```

```
/* initialize the modules */
AVSinit_modules()
{
        int AVS_fuzzy_K_means();
        AVSmodule_from_desc(AVS_fuzzy_K_means);
}

/* interface routine */
AVS_fuzzy_K_means()
{
        int compute_fuzzy_K_means();
        int param;
        int in_port, out_port, train_port;

/* set the module name and type */
        AVSset_module_name("Fuzzy K Means", MODULE_FILTER);

/* create the input port for the training image */
        in_port = AVScreate_input_port("Input Image","field 2D uniform byte",
                                       REQUIRED );

/* create the output port for the clustered image */
        out_port=AVScreate_output_port("Output Image","field 2D uniform float")
;

/* create the output port for the training data */
        train_port = AVScreate_output_port("Training Data", Training_Field);
/* set the title for the window */
        param=AVSadd_parameter("title","string","Clustering Parameters",
                               "","");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* create a widget to get the number of clusters */
        param=AVSadd_parameter("Number of clusters:","integer",4,1,INT_UNBOUND)
;
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to set min clustering threshold */
        param=AVSadd_float_parameter("membership", 0.9, 0.0, 1.0);
        AVSadd_parameter_prop(param,"immediate","boolean",1);
        AVSconnect_widget(param, "dial");

/* create a widget to display the image sample size */
        param=AVSadd_parameter("Sample offset:","integer",1,1,10);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the clustering iteration number */
        param=AVSadd_parameter("Iteration number:","integer",0,0,INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the maximum clustering iteration */
        param=AVSadd_parameter("Maximum iteration:","integer",99,0,INT_UNBOUND)
;
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the limiting cluster shift */
        param=AVSadd_parameter("Cluster shift limit:","integer",4,0,INT_UNBOUND
);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the magnitude of the cluster shift */
        param=AVSadd_float_parameter("Greatest cluster delta:", 0.0, 0.0,
                                     FLOAT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_real");

/* create a widget to display the elapsed time */
        param=AVSadd_parameter("Elapsed time (sec):","integer",0,0,INT_UNBOUND)
;
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to skip zero magnitude vectors */
        param=AVSadd_parameter("Skip zero vectors","boolean",0,0,0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create autogeneration widget */
        param=AVSadd_parameter("Auto-cluster","boolean",0,0,0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to generate the output image */
        param=AVSadd_parameter("Generate clusters", "oneshot", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);

/* free the output data */
        AVSautofree_output(out_port);

/* set the function to create the output */
        AVSset_compute_proc(compute_fuzzy_K_means);
}

/*=============================================================================

M A I N

=============================================================================*/
int compute_fuzzy_K_means(AVSfield_char *input_image,
                          AVSfield_float **output_image,
                          AVSfield **train_out,
                          char *title,
                          int interface_num_clusters,
                          float *interface_threshold,
                          int interface_offset,
                          int iteration,
                          int interface_max_iteration,
                          int interface_limit,
```

```
                        float *interface_delta,
                        int time,
                        int interface_skip_zeros,
                        int auto_run,
                        int run)
{

/* local variables */
pix_list_ptr pixel_list;
msi_img_ptr input_params;
class_map_ptr classified;
matrix_ptr center, old_center;
three_d_matrix_ptr membership;
struct timeval before, after;
register int row, col, cluster;
int dims[2];
int status, converged;
int *indx, *arr;
int write_training_datafile;
float *dist;
float threshold, temp, sendval;

        /* get the threshold value from the dial */
        if( AVSparameter_changed("membership") ){
                /* truncate and update the interface */
                temp = *interface_threshold * 100.0;
                sendval = ( (int)(temp+0.5) )/100.0;
                AVSmodify_float_parameter("membership", AVS_VALUE,
                                          sendval, 0, 0);
        }

        if (run || auto_run)
                AVSmodify_parameter("Generate clusters", AVS_VALUE,0,0,0);
        else{
                AVSmark_output_unchanged( "Output Image" );
                return(1);
        }

#ifdef DEBUG
        printf("AVS_fuzzy_K_means just entered the compute function.\n");
        printf("Number of clusters = %d.\n", interface_num_clusters);
        printf("Membership threshold = %f\n", *interface_threshold);
        printf("Maximum iteration = %d.\n", interface_max_iteration);
        printf("Limiting cluster shift = %d.\n", interface_limit);
#endif

        /* allocate memory for the structures */
        input_params = (msi_img_ptr)malloc(sizeof(MSI_IMG));
        if (input_params == NULL){
                printf("Unable to malloc input image pointer\n");
                return(1);
        }

        classified = (class_map_ptr)malloc(sizeof(CLASS_MAP));
        if (classified == NULL){
                printf("Unable to malloc classified image pointer\n");
                return(1);
        }

        pixel_list = (pix_list_ptr)malloc(sizeof(PIX_LST));
        if (pixel_list==NULL){
                printf("Unable to malloc pixel list pointer.\n");
                return(1);
        }

        center = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (center == NULL){
                printf("Unable to malloc center matrix pointer\n");
                return(1);
        }

        old_center = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (old_center == NULL){
                printf("Unable to malloc old_center matrix pointer\n");
                return(1);
        }

        membership = (three_d_matrix_ptr)malloc(sizeof(D3_MAT_STRUCT));
        if (membership == NULL){
                printf("Unable to malloc membership matrix pointer\n");
                return(1);
        }

#ifdef DEBUG
        printf("All structures were allocated.\n");
#endif

        /* clear out the last elapsed time, iteration, and delta values */
        AVSmodify_parameter("Iteration number:",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            0, 0, 0);

        AVSmodify_float_parameter("Greatest cluster delta:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  0.0, 0.0, 0.0);

        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            0, 0, 0);

        /* define the input image parameters */
        input_params->num_rows = MAXY(input_image);
        input_params->num_cols = MAXX(input_image);
        input_params->num_pixels = input_params->num_rows *
                                   input_params->num_cols;
        input_params->num_bands = input_image->veclen;

        /* allocate memory for the arrays */
        dist = vector(0,interface_num_clusters-1);
        center->num_rows = interface_num_clusters;
        center->num_cols = input_params->num_bands;
        center->array = matrix(0, center->num_rows-1, 0, center->num_cols-1);

        old_center->num_rows = interface_num_clusters;
        old_center->num_cols = input_params->num_bands;
        old_center->array = matrix(0, old_center->num_rows-1,
```

```
                                0, old_center->num_cols-1);

        /* note the division to support sampling! */
        membership->num_rows = (int)(input_params->num_rows/interface_offset);
        membership->num_cols = (int)(input_params->num_cols/interface_offset);
        membership->num_clusters = interface_num_clusters;
        membership->array = f3tensor(0, membership->num_rows-1,
                                     0, membership->num_cols-1,
                                     0, membership->num_clusters -1);

        /* zero out the membership array */
        for(row = 0; row < membership->num_rows; row++)
            for(col = 0; col < membership->num_cols; col ++)
                for(cluster = 0; cluster < membership->num_clusters; cluster ++)
->array[row][col][cluster] = 0;

        classified->num_rows = membership->num_rows;
        classified->num_cols = membership->num_cols;
        classified->array = imatrix(0, classified->num_rows-1,
                                    0, classified->num_cols-1);
#ifdef DEBUG
        printf("All arrays allocated.\n");
        printf("Number of rows in the input image = %d.\n",
                input_params->num_rows);
        printf("Number of columns in the input image = %d.\n",
                input_params->num_cols);
        printf("Number of rows in the output image = %d.\n",
                classified->num_rows);
        printf("number of columns in the output image = %d.\n",
                classified->num_cols);
#endif

        /* randomly sample the data to get the starting centers */
        status = build_center_vector(input_image, input_params, center,
                                     interface_num_clusters,
                                     interface_skip_zeros);
        if(status == 1)
                printf("Initial center vector constructed.\n");
        else{
                printf("ERROR: bad status from build_center_vector.\n");
                return(1);
        }

#ifdef DEBUG
        printf("Fuzzy K means just entered the clustering loop.\n");
#endif

        /* iteration control variables */
        converged = 0;
        iteration = 0;
        gettimeofday(&before, NULL);

        while( (!converged) && (iteration <= interface_max_iteration) ){
                status = calc_membership(input_params, input_image,
                                         membership, center, dist,
                                         interface_num_clusters,
                                         interface_offset,
                                         interface_skip_zeros);
                if (status != 1){
                        printf("ERROR: bad status from calc_membership.\n");
                        return(1);
                }
    #ifdef DEBUG
                printf("Membership calculated for iteration %d.\n", iteration);
    #endif

                status = calc_new_centers(input_params, input_image,
                                          membership, old_center, center,
                                          interface_num_clusters,
                                          interface_offset,
                                          interface_skip_zeros);
membership      if(status != 1){
                        printf("ERROR: bad status from calc_new_centers.\n");
                        return(1);
                }
    #ifdef DEBUG
                printf("Center vectors updated for iteration %d.\n", iteration);
    #endif

                converged = mag_diff(center, old_center, interface_limit);
                iteration +=1;

                /* update interface */
                AVSmodify_parameter("Iteration number:",
                                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                    iteration, iteration, iteration);

        }       /* end of while */

        /* update the elapsed time */
        gettimeofday(&after, NULL);
        time = (int)(after.tv_sec - before.tv_sec);
        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            time, time, time);

        /* check for max iterations and no convergence */
        if( (!converged) && (iteration > interface_max_iteration) ){
                AVSmessage("beta",AVS_Fatal, NULL, "Fuzzy K means","Kill module","Clusters di
d not converge in maximum allowed iterations.");
        }

        printf("The fuzzy K-means algorithm converged in %2d iterations\n",
                        iteration-1);

        /* build the output image */
        printf("building the output image...\n");
        threshold = *interface_threshold;
        status = build_output_image(membership, classified, threshold);
        if(status != 1){
                printf("ERROR: bad status from build_output_image.\n");
                return(1);
        }

        /* allocate then copy the resulting image to output port */
        dims[0] = classified->num_cols;
```

```
        dims[1] = classified->num_rows;

#ifdef DEBUG
        printf("Allocating the %d rows by %d columns output image.\n",
                dims[0], dims[1]);
#endif

        if ( *output_image == NULL ) {
            *output_image = ( AVSfield_float * )
                ( AVSdata_alloc( "field 2D uniform scalar float", dims ));
        }
        else if ( ( dims[ 0 ] != MAXX( *output_image ) ) ||
                  ( dims[ 1 ] != MAXY( *output_image ) ) ) {
            AVSfield_free( (AVSfield *) *output_image );
            *output_image = ( AVSfield_float * )
                ( AVSdata_alloc( "field 2D uniform scalar float", dims ));
        }

        printf("Output field allocated.\n");

        for(row=0; row < classified->num_rows; row++)
                for(col =0; col < classified->num_cols; col++)
                        I2D(*output_image, col, row) =
                                                classified->array[row][col];
#ifdef DEBUG
        printf("Output field written.\n");
#endif

        /* allocate then build the training set */
        pixel_list->num_classes = interface_num_clusters;
        pixel_list->num_pixels = classified->num_valid_pixels;
                        pixel_list->num_bands = input_params->num_bands;
                        pixel_list->array =imatrix(1, pixel_list->num_pixels,
                                                1, pixel_list->num_bands+1);

        indx = ivector(1, pixel_list->num_pixels);
        arr = ivector(1, pixel_list->num_pixels);

        status = build_training_set(input_image, input_params, pixel_list,
                                    classified, indx, arr, train_out,
                                    interface_offset);
        if(status != 1){
                printf("ERROR: bad status from build_training_set.\n");
                return(1);
        }

        /* free all the allocated memory */
        free_vector(dist, 0, interface_num_clusters-1);

        free_matrix(center->array, 0, center->num_rows-1,
                                   0, center->num_cols-1);

        free_matrix(old_center->array, 0, old_center->num_rows-1,
                                       0, old_center->num_cols-1);

        free_imatrix(classified->array, 0, classified->num_rows-1,
                                        0, classified->num_cols-1);
```

```
        free_f3tensor(membership->array, 0, membership->num_rows-1,
                                         0, membership->num_cols-1,
                                         0, membership->num_clusters -1);

        free_ivector(indx, 1, pixel_list->num_pixels);

        free_ivector(arr, 1, pixel_list->num_pixels);


        free_imatrix(pixel_list->array, 1, pixel_list->num_pixels,
                                        1, pixel_list->num_bands+1);
        free(center);
        free(old_center);
        free(input_params);
        free(classified);
        free(membership);
        free(pixel_list);

        /* return success */
        return (1);

}       /* end of main */

/*=========================================================================
function build_center_vector
        takes num_cluster random points from the data matrix and
        utilizes them as the initial centers of the data clusters
=========================================================================*/
int build_center_vector(AVSfield_char *input_image, msi_img_ptr input_params,
                        matrix_ptr center, int num_clusters, int zero_flag)
{

register int row, band;
int index1, index2, status, magnitude;
float x, y;
long idum = (-13);

#ifdef DEBUG
        printf("\nBuilding the random center vector.\n");
#endif

/* initialize the random number generator */
        x = ran1(&idum);

        for (row=0; row < num_clusters; row++){
                x = ran1(&idum);
                index1 = (int)( (input_params->num_rows * x) + 0.5 );
                y = ran1(&idum);
                index2 = (int)( (input_params->num_cols * y) + 0.5 );

                magnitude =0;
                for (band = 0; band < input_params->num_bands; band++){
                        center->array[row][band] =
                           (float)( *(I2DV(input_image,index2,index1)+band) );

                        if (zero_flag == 1)
                                magnitude += center->array[row][band];
```

```
                }       /* end band loop */

                if( (magnitude == 0) && (zero_flag == 1) ){
                        row -= 1;
                        if (row == -1)
                                row = 0;


                }
        }


#ifdef RANDOM_DEBUG
        printf("The random center vector is :\n");
        for (row=0; row < num_clusters; row++){
                printf("Random center[%2d] = ", row);
                for (band = 0; band < center->num_cols; band++){
                        printf("%3f\t", center->array[row][band]);
                }
                printf("\n");
        }
#endif

        status = 1;
        return(status);

}       /* end of build_center_vector */

/*=============================================================================
function distance
        determines the distance between a data point and the different cluster
        centers
=============================================================================*/
float distance(int row, int col, int cluster_num, AVSfield_char *input_image,
               msi_img_ptr input_params, matrix_ptr center){

register int band;
float dist, temp, output, value;

        dist = 0.0;

        for (band = 0; band < input_params->num_bands; band++){
                value = (float)( *(I2DV(input_image,col,row)+band));
                temp = pow( (value - center->array[cluster_num][band]), 2.0);
                dist += temp;
        }

        output = sqrt(dist);
        return (output);

}       /* end of distance */


/*=============================================================================

Function calc_membership
        Calculates the membership value for each of the data points with
        respect to each cluster center.
=============================================================================*/

int calc_membership( msi_img_ptr input_params, AVSfield_char *input_image,
                     three_d_matrix_ptr membership, matrix_ptr center,
                     float *dist, int num_clusters, int offset, int skip_zeros){

register int row, col, band, cluster;
int index, flag, status, image_row, image_col, magnitude, zero_flag;
float temp, factor, small;

        flag = 0;
        small = 1.0;

for (row = 0; row < membership->num_rows; row++){
        image_row = offset * row;
        for (col = 0; col < membership->num_cols; col++){
                image_col = offset * col;

/* see if we have a zero magnitude pixel */
                zero_flag = 0;
                if (skip_zeros == 1){
                        magnitude = 0;

                        for(band = 0; band < input_params->num_bands; band ++)
                                magnitude +=(int)( *(I2DV(input_image,
                                                    image_col,image_row)+band));
                                if (magnitude == 0)
                                        zero_flag = 1;
                }

/* calculate the distance from the point to each cluster center */
        if(zero_flag != 1){
                for(cluster = 0;cluster < membership->num_clusters; cluster++){
                        dist[cluster] = distance(image_row, image_col, cluster,
                                        input_image, input_params, center);
                        if (dist[cluster] < small){
                                flag = 1;
                                index = col;
                        }
                }

/* point is very near a cluster center.  Set its membership to that cluster
   to one and zero all others */

                if (flag==1){
                        for (cluster =0; cluster < num_clusters; cluster++)
                                membership->array[row][col][cluster]=0.0;

                        membership->array[row][col][index] = 1.0;
                        flag = 0;
                }

/* not near any cluster center, so calculate its membership to all cluster */

                else{
                        temp = 0.0;
                        for (cluster =0; cluster < num_clusters; cluster++)
                                temp+=1.0/(dist[cluster]*dist[cluster]);
```

```
                        factor = temp;

                        for(cluster = 0; cluster < num_clusters; cluster++)
                                membership->array[row][col][cluster] =
                                  (1.0/(dist[cluster]*dist[cluster]))/(factor);

                        }

        }       /* end if */

        zero_flag = 0;

        }       /* end of col loop */


}       /* end of row loop */

#ifdef MEMBERSHIP_DEBUG
for (row = 0; row < membership->num_rows; row++){
        for (col = 0; col < membership->num_cols; col++){
                printf("membership[%2d][%2d] = ", row, col);
                for (cluster = 0;cluster < membership->num_clusters;cluster++)
                        printf("%3f\t", membership->array[row][col][cluster]);
        }
                printf("\n");
}
#endif

status = 1;
return(status);

} /* end of calc_membership */

/*=============================================================================

function calc_new_centers
        calculate the new centers of the clusters based on the results from
        the membership function
=============================================================================*/

int calc_new_centers(msi_img_ptr input_params, AVSfield_char *input_image,
                     three_d_matrix_ptr membership, matrix_ptr old_center,
                     matrix_ptr center,int num_clusters, int offset,
                     int skip_zeros)
{

register int band, cluster, row, col;
int status, num_features, image_row, image_col, magnitude, zero_flag;
float denom, image_value, *numer;

        num_features = input_params->num_bands;
        numer = vector(0, num_features-1);

/* transfer the cluster centers from center to old_center */
        for (row=0; row < num_clusters; row++)
                for (band=0; band < num_features; band++)
                  old_center->array[row][band] = center->array[row][band];
```

```
/* Calculate the new centers */
for (cluster = 0; cluster < num_clusters; cluster++){

/* initialize summing variables */
        denom=0.0;
        for (band = 0; band < input_params->num_bands; band++)
                numer[band]=0.0;

/* calculate the denominator term and numerator terms */
        for (row = 0; row < membership->num_rows; row++){
                image_row = offset * row;
                for (col = 0; col < membership->num_cols; col++){
                    image_col = offset * col;

/* check for zero vectors */
zero_flag = 0;

if (skip_zeros == 1){
    for(band = 0; band < input_params->num_bands; band++)
        magnitude += *(I2DV(input_image, image_col, image_row) + band);
        if (magnitude == 0)
            zero_flag = 1;
}

        if (zero_flag != 1){
                denom += pow( (double)membership->array[row][col][cluster],2.0)
;

                    for (band = 0; band < input_params->num_bands; band++){
                        image_value =  *(I2DV(input_image, image_col, image_row)
                                        + band);
                        numer[band] += pow( (double)
                        membership->array[row][col][cluster],2.0) * image_value;

                    }           /* end band loop */

        }       /* end if */

        }       /* end col loop */

        }       /* end row loop */

/* calculate the new center */
        for (band = 0; band < input_params->num_bands; band++)
                center->array[cluster][band] = numer[band]/denom;

}       /* end of cluster loop */

#ifdef CENTER_DEBUG
for (row=0; row < num_clusters; row++){
        printf("New center[%2d] = ", row);
        for (band = 0; band < input_params->num_bands; band++){
                printf("%3f\t", center->array[row][band]);
        }
        printf("\n");
}
```

```
#endif

free_vector(numer,0,input_params->num_bands-1);

status = 1;
return(status);
}

/*=============================================================================

function mag_diff
        calculates the difference between old_center and center to determine
        if the clusters have converged.

=============================================================================*/
int mag_diff(matrix_ptr center, matrix_ptr old_center, int movement_threshold)
{

register int row, col;
int result, num_clusters, num_features;
float temp, difference, max;

        num_clusters = center->num_rows;
        num_features = center->num_cols;
        max = -1.0;

        for (row=0; row < num_clusters; row++){
                difference = 0.0;
                for (col=0; col < num_features; col++){
                        temp = center->array[row][col] -
                               old_center->array[row][col];
                        difference += fabs(temp);

                }       /* end of col loop */

                if (difference > max){
                        max = difference;
                }

        }       /* end of row loop */

        AVSmodify_float_parameter("Greatest cluster delta:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  max, max, max);

        if ( max >  movement_threshold)
                result = 0;
        else
                result = 1;

return(result);
}

/*=============================================================================
function build_output_image
        this function creates the output image from the input image and the
        membership matrix with reguard to the user selected threshold value.
=============================================================================*/
int build_output_image(three_d_matrix_ptr membership, class_map_ptr classified,
                       float threshold){

register int row, col, cluster;
int max_index, status;
float value, max;

#ifdef DEBUG
        printf("Building the %d rows by %d columns output image.\n",
                classified->num_rows, classified->num_cols);
#endif
        classified->num_valid_pixels = 0;
/* scan the membership matrix for the maximum cluster membership value */
        for(row = 0; row < classified->num_rows; row++){
                for(col = 0; col < classified->num_cols; col++){
                        max = -1.0;
                        for(cluster = 0; cluster < membership->num_clusters;
                            cluster++){
                                value = membership->array[row][col][cluster];
                                if (value > max){
                                        max = value;
                                        max_index = cluster + 1;
                                }
                        }       /* end of cluster loop */

                        if (max >= threshold){
                                classified->array[row][col] = max_index;
                                classified->num_valid_pixels += 1;
                        }
                        else
                                classified->array[row][col] = 0;

                }       /* end of col loop */
        }       /* end of row loop */

#ifdef DEBUG
        printf("Of %d total pixels, %d fall within the threshold criteria.\n",
                        classified->num_rows * classified->num_cols,
                        classified->num_valid_pixels);
#endif

#ifdef CLASS_DEBUG
        printf("This is the classified image:\n");
        for (row=0; row < classified->num_rows; row++){
                for (col = 0; col < classified->num_cols; col++){
                        printf("%d\t", classified->array[row][col]);
                }
                printf("\n");
        }
#endif

        status = 1;
        return(status);

}       /* end of build_output_image */

/*=============================================================================
```

```
function build_training_set
        this function creates the training set from the output image and the
        input image.

=============================================================================*/
int build_training_set(AVSfield_char *input_image, msi_img_ptr input_params,
                       pix_list_ptr pixel_list, class_map_ptr classified,
                       int *indx, int*arr, AVSfield **trainfield, int offset){

register int row, col, band, class;
int status, list_row, start_row, set_row, image_row, image_col;
int *length;
int total_training_points;
double *vector;
IPItraining trainchain = NULL;
IPItraining trainset = NULL;

/* allocate and initialize the length vector */
        length = ivector(1,pixel_list->num_classes);
        for (class = 1; class <= pixel_list->num_classes; class++)
                length[class] = 0;

        vector = dvector(0,input_params->num_bands-1);

/* scan the classified matrix for values other than zero */
        list_row = 0;
        for(row=0; row < classified->num_rows; row++){
                image_row = offset * row;
                for(col = 0; col < classified->num_cols; col++){
                        image_col = offset * col;

/* the pixel_list array is ONE based to support the sort */

                if(classified->array[row][col] > 0){
                        /* calculate total pixels and # in each class */
                        list_row += 1;
                        length[ classified->array[row][col] ] += 1;
                        for(band=0; band < input_params->num_bands; band++){
                                pixel_list->array[list_row][band+1] =
                                *(I2DV(input_image,image_col,image_row)+band);

                        } /* end band */

                pixel_list->array[list_row][input_params->num_bands + 1] =
                                                classified->array[row][col];
                        arr[list_row] = classified->array[row][col];

                } /* end if */

                } /* end col */

        } /* end row */

        /* sort the arr vector */
        iindexx(classified->num_valid_pixels,arr,indx);

#ifdef TRAINING_SET_DEBUG
```



```
        printf("This is the sorted pixel list:\n");
                for (row = 1; row <= pixel_list->num_pixels; row++){
                        for (band = 1; band <= pixel_list->num_bands+1; band++){
                                printf("%d\t",pixel_list->array[indx[row]][band]);

                        }
                        printf("\n");
                }
        for (class = 1; class <= pixel_list->num_classes; class++)
                printf("Length of class[%d] = %d\n", class, length[class]);
#endif

        total_training_points = 0;
        set_row =0;
        start_row = 1;
        for (class = 1; class <= pixel_list->num_classes; class++){
                /* initialize the trainset */
                IPIappend_to_training_chain(&trainchain, &trainset);
                IPIset_training_gis(trainset, class);
                IPIset_training_datatype(trainset, IPI_byte);
                IPIset_training_bands(trainset, input_params->num_bands);
                IPIset_training_length(trainset, length[class]);

#ifdef TRAINING_SET_DEBUG
        printf("Set %d appended to chain.\n",class);
        printf("GIS value set to %d.\n",class);
        printf("Data type set to byte.\n");
        printf("Number of training bands set to %d.\n",
                        input_params->num_bands);
        printf("length of training set = %d.\n", length[class]);

                printf("Sending class %d's %d pixels to the training set.\n",
                        class, length[class]);
#endif

                for (row = start_row; row < start_row+length[class]; row++){

                  for(band = 0; band < pixel_list->num_bands; band++){
                    vector[band] = (double)pixel_list->array[indx[row]][band+1];
                  } /* end band loop */

                  IPIset_training_vector(trainset, set_row, vector);

                  set_row += 1;

                } /* end row loop */

                start_row += length[class];
                total_training_points += length[class];
                set_row = 0;

        } /* end class loop */

        /* write the results to the output port */
        IPItraining_to_field( trainchain, trainfield, NULL, 0 );
        printf("Total of %d pixels were written to the training set.\n",
                total_training_points);
```

```
        free_ivector(length,1,pixel_list->num_classes);
        free_dvector(vector,0,input_params->num_bands-1);
        status = 1;
        return(status);

}        /* end of build_training set */

/*=========================================================================

function write_training_set
        this function writes the classified image to a user defined disk file.
        Function will happily cream any disk file with the same name as the
        user inputs

=========================================================================*/
int write_training_set(pix_list_ptr pixel_list, int *indx)
{

/* local variables */
FILE *out_file;
register int row, band;
int status;
char response[MAXSTR];

        status=0;
        fflush(stdin);  /* flush stdin prior to the character read */
        printf("Please enter the name for the training data (.dat) file: ");
        gets(response);
        out_file = fopen(response, "w");

        if(out_file != NULL)
                status=1;

        rewind(out_file);
        fprintf(out_file,"Training data from fuzzy K-means\n");
        fprintf(out_file,"%d pixels\n",pixel_list->num_pixels);
        fprintf(out_file,"%d bands\n",pixel_list->num_bands);

        for(row=1; row <= pixel_list->num_pixels; row++){
                for (band = 1; band <= pixel_list->num_bands+1; band++){
                  fprintf(out_file,"%d\t",pixel_list->array[indx[row]][band]);
                }
                fprintf(out_file,"\n");
        }

        fclose(out_file);
        printf("The training data was successfully written to disk.\n");
        return(status);
}
```

Source code for the Class Statistics and GML Classification AVS modules

```
/*=============================================================================

Filename:       classtat.c
Author:         Nessmiller, Steven W.
Summary:        This AVS modulke computes the mean vector and the variance
                covariance matrix for each training class.

Revisions:      AVS port
                14 April 95
                        slight change to header on statistics file
                5 May 95
                        implemented training data read via IPI.
                29 June 95
                        minor bug fix

=============================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <sys/file.h>
#include <sys/types.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/flow.h>
#include <avs/avs_data.h>
#include <IPIlib.h>
#include "/cis/src/local/avs/src/IPI/avs_types.h"
#include "memutil.h"
#include "imat.h"
#include "lu_dcomp.h"

/* layered debug definitions */
#define DEBUG
#define MEAN_DEBUG
#define VC_DEBUG
#define LU_DEBUG
#define INV_DEBUG
#define DET_DEBUG
/*
#define PIXEL_DEBUG
*/

#define MAXSTR 80

typedef struct{
        int **array;
        int num_pixels;
        int num_bands;
        int num_classes;
}PIX_LIST;
typedef PIX_LIST *pix_list_ptr;

typedef struct{
        float ***array;
        int num_rows;
        int num_cols;
        int num_classes;
        int num_bands;
}D3_MAT_STRUCT;
typedef D3_MAT_STRUCT *three_d_matrix_ptr;

/* function prototypes */
int calc_mean_vectors(pix_list_ptr pixel_list, matrix_ptr mean);
int calc_vc_matrix(pix_list_ptr pixel_list, matrix_ptr mean,
                   three_d_matrix_ptr vc);
int write_output(char *filename, matrix_ptr mean, three_d_matrix_ptr inv_vc,
                 three_d_matrix_ptr vc, float *log_det, float *determinant);

/* initialize the module */
AVSinit_modules ()
{
        int AVS_classtat();
        AVSmodule_from_desc(AVS_classtat);
}

/* interface routine */
AVS_classtat()
{
        int compute_classtat();
        int param;
        int in_port;
        char cwd[MAXSTR];

/* set the module name and type */
        AVSset_module_name("Class Statistics", MODULE_FILTER);

/* create the input port for the training data */
        in_port = AVScreate_input_port("Input Training Set", Training_Field,
                                       REQUIRED );

/* set the title for the window */
        param=AVSadd_parameter("title","string",
                               "Training class statistics","","");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* create a widget for the output statistics filename */
        getwd(cwd);
        if( cwd[strlen(cwd)-1] != '/')
                strcat(cwd,"/");
        param=AVSadd_parameter("Statistics file (*.sta):","string",cwd, 0,
                               ".sta");
        AVSadd_parameter_prop(param, "width","integer",4);
        AVSconnect_widget(param, "browser");

/* allow user to view all file types */
        param = AVSadd_parameter("View all file types", "boolean",0,0,1);
        AVSadd_parameter_prop(param, "width", "integer",4);

/* create a widget to display the number of training sets */
        param=AVSadd_parameter("Number of training sets:", "integer", 0, 0,
                               INT_UNBOUND);
```

```
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the number of training pixels */
        param=AVSadd_parameter("Number of training pixels:", "integer", 0, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the number of bands in the training sets */
        param=AVSadd_parameter("Number of bands:", "integer", 0, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the elapsed time */
        param=AVSadd_parameter("Elapsed time (sec):","integer",0,
                               0,INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create autogeneration widget */
        param=AVSadd_parameter("Auto-generate statistics file","boolean",0,
                               0,0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to generate the statistics file */
        param=AVSadd_parameter("Generate statistics file","oneshot", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);

/* set the function to create the output */
        AVSset_compute_proc(compute_classtat);
}

/*============================================================================
        main compute function
============================================================================*/

int compute_classtat(AVSfield *input,
                     char *title,
                     char *stats_filename,
                     int all,
                     int interface_num_classes,
                     int interface_num_pixels,
                     int interface_num_bands,
                     int time,
                     int auto_run,
                     int run)
{
/* local variables */
pix_list_ptr pixel_list;
matrix_ptr mean;
three_d_matrix_ptr vc, lu_vc, inv_vc;
float d;
float *determinant;
float *log_det;
float *col_vector;
int **index, *length;
int training_bands, flag;
char *answer;
FILE *output_file;
register int class_index, row, col, i, i2, j, band, class;
int status, n, elapsed_time;
struct timeval before, after;
double *train_vector;
int list_row;
IPItraining trainchain = NULL;
IPItraining trainset = NULL;
static int new_file = 0;
static int file_status = 0;
static char old_stats_filename[80];
char *reset_stats_filename = " ";


        /* check for view all parameter filenames */
        if(AVSparameter_changed("View all file types")){
                if (all){
                        AVSmodify_parameter("Statistics file (*.sta):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/",0,"");
                        printf( "Setting maxval to null\n");
                        AVSmodify_parameter("Statistics file (*.sta):",
                             AVS_VALUE, stats_filename, 0, 0);
                }
                else{
                        AVSmodify_parameter("Statistics file (*.sta):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/", 0, ".sta");
                        printf( "Setting maxval to .sta\n");
                        AVSmodify_parameter("Statistics file (*.sta):",
                                        AVS_VALUE, stats_filename, 0, 0);
                }
        }

        /* make sure we've got a file NOT just a directory */
        if(stats_filename[strlen(stats_filename) - 1] == '/'){
                new_file = 0;
                return(1);
        }

        /* see if the statistics filename has changed */
        status = strcmp(stats_filename, old_stats_filename);

        if (status != 0){
                strcpy(old_stats_filename, stats_filename);
                output_file = fopen(stats_filename, "r+");

                if(output_file == NULL){
                        printf("Class statistics file does not exist.\n");
                        new_file = 1;
                }
                else{
                        /* file already exists, close and check for overwrite */
                        fclose(output_file);

                        if (file_status == 0){
                            printf("Statistics file already exists.\n");
```

```
                        answer = AVSmessage("beta",AVS_Warning,NULL,
                            "Class Statistics","Overwrite!Cancel",
                            "Class statistics file %s already exists.\n",
                            stats_filename);

                        if( strcmp(answer,"Overwrite") == 0 )
                            file_status = 1;
                        else{
                            file_status = 0;
                            return(1);
                        }
                    }
                }
        }

        if ( (file_status == 1) || (new_file ==1) )
                printf("The output filename is: %s\n",stats_filename);
        else
                return(1);


/* check if run time */
        if( (run || auto_run) && (file_status || new_file) )
                AVSmodify_parameter("Generate statistics file",AVS_VALUE,0,0,
                                    0);
        else
                return(1);

        printf("Class statistics just entered the compute function.\n");

        /* reset the display */
        AVSmodify_parameter("Number of training sets:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        0,0,0);

        AVSmodify_parameter("Number of training pixels:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        0,0,0);

        AVSmodify_parameter("Number of bands:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        0,0,0);

        AVSmodify_parameter("Elapsed time (sec):",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        0,0,0);

        /* reset the static variables */
        new_file = 0;
        file_status = 0;
        strcpy(old_stats_filename, reset_stats_filename);

        /* allocate memory for the structures */
        pixel_list = (pix_list_ptr)malloc(sizeof(PIX_LIST));
        if (pixel_list==NULL){
                printf("Unable to malloc pixel list pointer.\n");
                return(1);
        }

        vc = (three_d_matrix_ptr)malloc(sizeof(D3_MAT_STRUCT));
        if (vc==NULL){
                printf("Unable to malloc the vc matrix pointer.\n");
                return(1);
        }

        lu_vc = (three_d_matrix_ptr)malloc(sizeof(D3_MAT_STRUCT));
        if (lu_vc==NULL){
                printf("Unable to malloc the LU decomposed matrix pointer.\n");

                return(1);
        }

        inv_vc = (three_d_matrix_ptr)malloc(sizeof(D3_MAT_STRUCT));
        if (inv_vc==NULL){
                printf("Unable to malloc the inverse matrix pointer.\n");
                return(1);
        }

#ifdef DEBUG
        printf("All the structures were successfully allocated.\n");
#endif

        /* Convert the training field */
        IPIfield_to_training(input,NULL,&trainchain,0);

#ifdef DEBUG
        printf("Training field converted.\n");
#endif

        /* get the number of training classes and update the interface */
        pixel_list->num_classes = IPIlength_of_training_chain( trainchain );
        interface_num_classes = pixel_list->num_classes;
        AVSmodify_parameter("Number of training sets:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        interface_num_classes, interface_num_classes,
                        interface_num_classes);

#ifdef DEBUG
        printf("There are %d classes of data.\n", pixel_list->num_classes);
#endif

        /* get number of bands in the training sets and update the interface */
        IPIget_training_bands(trainchain, &training_bands);
        pixel_list->num_bands = training_bands;

#ifdef DEBUG
        printf("There are %d bands in the training sets.\n",
                pixel_list->num_bands);
#endif

        interface_num_bands = pixel_list->num_bands;
        AVSmodify_parameter("Number of bands:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        interface_num_bands, interface_num_bands,
                        interface_num_bands);
```

```
        /* Get all the training set information */
        pixel_list->num_pixels = 0;
        length = ivector(0, pixel_list->num_classes);
        trainset = NULL;
        IPInext_on_training_chain( &trainchain, &trainset );
        printf("Just executed first next_on_training_chain.\n");
        class = 0;
        while ( trainset != NULL ) {
            /* get the pixel list parameters and allocate memory */
            length[ class ] = IPIsize_of_training_set(trainset);
            pixel_list->num_pixels += length[ class ];

            IPInext_on_training_chain( &trainchain, &trainset );
            printf("Executed next_on_training_chain again.\n");
            class += 1;
        }

        printf("Number of pixels per class has been calculated.\n");
        interface_num_pixels = pixel_list->num_pixels;
        AVSmodify_parameter("Number of training pixels:",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            interface_num_pixels, interface_num_pixels,
                            interface_num_pixels);

        /* check to ensure enough pixels in each class! */
        flag = 0;
        for(class = 0; class < pixel_list->num_classes; class++){
                if(length[class] < pixel_list->num_bands)
                        flag = 1;
        }

        if (flag == 1){
                AVSmessage("beta",AVS_Error,NULL,"Class Statistics","Okay",
                           "Not enough pixels in the training classes.  The var
iance covariance matrix will not be invertible.\n");
                return(1);
        }

#ifdef DEBUG
        printf("Total number of pixels = %d.\n",pixel_list->num_pixels);
        for(class = 0; class < pixel_list->num_classes; class++){
                printf("The number of pixels in class %d is %d.\n",
                       class, length[class]);
        }
#endif

        pixel_list->array = imatrix(0, pixel_list->num_pixels-1,
                                    0, pixel_list->num_bands);
        if(pixel_list->array == NULL){
                printf("Could not allocate the pixel list.\n");
                return(1);
        }
        else
                printf("Pixel list allocated.\n");

/* read the training data into the pixel_list structure */
        list_row = 0;
        train_vector = dvector(0,pixel_list->num_bands-1);
```

```
        trainset = NULL;
        for(class = 0; class < pixel_list->num_classes; class++){
            IPInext_on_training_chain( &trainchain, &trainset );
            printf("Reading in class %d.\n", class);

            for(row = list_row; row < list_row + length[class]; row++){
                IPIget_training_vector(trainset, row - list_row, train_vector);

                for(band = 0; band < pixel_list->num_bands; band++){
                    pixel_list->array[row][band] = (int)train_vector[band];
                } /* end band loop */

                pixel_list->array[row][pixel_list->num_bands] = class;

#ifdef PIXEL_DEBUG      /* print out the training pixels */
        for(band = 0; band <= pixel_list->num_bands; band++){
                printf("%d\t",pixel_list->array[row][band]);
        }
        printf("\n");
#endif

            } /* end of row loop */

            list_row += length[class];
            printf("Class %d successfully read in.\n",class);

        } /* end of class loop */

#ifdef DEBUG
        printf("Training data successfully read in.\n");
#endif

        free_dvector(train_vector,0,pixel_list->num_bands-1);

        /* Calculate the mean vectors */
        gettimeofday(&before,NULL);
        status = calc_mean_vectors(pixel_list, mean);
        printf("The mean vectors have been calculated.\n");

        /* Calculate the variance-covariance matrices */
        status = calc_vc_matrix(pixel_list,mean,vc);
        printf("The variance covariance matrix has been calculated.\n");

        /* Accomplish the LU decomposition */
        /* first allocate storage for the LU decomposed matrices */
        lu_vc->num_rows = vc->num_rows;
        lu_vc->num_cols = vc->num_cols;
        lu_vc->num_classes = vc->num_classes;
        lu_vc->num_bands = vc->num_bands;
        lu_vc->array = f3tensor(1, lu_vc->num_classes,
                                1, lu_vc->num_rows,
                                1, lu_vc->num_cols);

        /* ludcmp expects a 1 based array just to make life interesting */
        /* read the vc matrices into the lu_vc arrays */
        for(class_index=1; class_index <= lu_vc->num_classes; class_index++)
                for(row=1; row <= lu_vc->num_rows; row++)
```

```
                        for (col=1; col <= vc->num_bands; col++)
                                lu_vc->array[class_index][row][col] =
                                        vc->array[class_index-1][row-1][col-1];

        /* send each lu_vc array to ludcmp */
        n = lu_vc->num_bands;
        index = imatrix(1,lu_vc->num_classes,1,n);
        determinant = vector(1,n);
        log_det = vector(1,n);
        for(class_index = 1; class_index <= lu_vc->num_classes; class_index++){

                ludcmp(lu_vc->array[class_index], n, index[class_index],&d);
                determinant[class_index]=d;
        }

#ifdef LU_DEBUG
/* debug code - print out the lu decomposed matrices and the index matrix */
        printf("\nThese are the values in the index matrix\n");
        for(row=1; row <= lu_vc->num_classes; row ++){
                for (col = 1; col <= n; col++)
                                printf("%d  ",index[row][col]);
                printf("\n");
        }

       printf("\nThese are the values of the elements in the lu_vc matrix\n\n")
;
        for(class_index=1; class_index <= lu_vc->num_classes; class_index++){
           for(row=1; row <= lu_vc->num_rows; row ++){
                for (col = 1; col <= lu_vc->num_cols; col++)
                        printf("%f  ",lu_vc->array[class_index][row][col]);
                printf("\n");
           }
        printf ("\n");
}
#endif
        /* calculate and store the determinant of each LU decomposed matrix */
        for(class_index=1; class_index <= lu_vc->num_classes; class_index++){
            for(row=1; row <= lu_vc->num_rows; row++)
                determinant[class_index] *= lu_vc->array[class_index][row][row];
            log_det[class_index] = (float) log((double)determinant[class_index]);
        }

#ifdef DET_DEBUG
        printf("These are the determinants:\n");
        for(class_index=1; class_index <= lu_vc->num_classes; class_index++)
                printf("%f    ",determinant[class_index]);
        printf("\nThese are the natural logs of the determinants:\n");
        for(class_index=1; class_index <= lu_vc->num_classes; class_index++)
                printf("%f    ",log_det[class_index]);
#endif

        /* Calculate the matrix inverses from the LU decomposed matrices */
        inv_vc->num_rows = vc->num_rows;
        inv_vc->num_cols = vc->num_cols;
        inv_vc->num_classes = vc->num_classes;
        inv_vc->num_bands = vc->num_bands;
        inv_vc->array = f3tensor(1, inv_vc->num_classes,
                                 1, inv_vc->num_rows,
                                 1, inv_vc->num_cols);
        n = inv_vc->num_bands;
        col_vector=vector(1,n);
        for (class_index = 1;class_index<=inv_vc->num_classes;class_index++){
                for (j=1; j <=n; j++){
                        for(i=1; i <= n; i++)
                                col_vector[i]=0.0;
                        col_vector[j] = 1.0;
                        lubksb(lu_vc->array[class_index], n,
                               index[class_index], col_vector);
                        for(i2=1; i2 <= n; i2++)
                                inv_vc->array[class_index][i2][j]=
                                                        col_vector[i2];
                }
        }

#ifdef INV_DEBUG        /* debug code - print out the inverted matrices */
        printf("\nThese are the inverted matrices:\n\n");
        for(class_index=1;class_index<=inv_vc->num_classes;class_index ++){
                for(row=1; row <= inv_vc->num_rows; row++){
                        for(col=1; col <= inv_vc->num_cols; col++)
                           printf("%f  ",inv_vc->array[class_index][row][col]);

                        printf("\n");
                }
                printf("\n");
        }
#endif

        /* update the display with the elapsed time */
        gettimeofday(&after, NULL);
        time = (int)(after.tv_sec-before.tv_sec);
        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            time, time, time);

        /* write the output to the disk file */
        status=write_output(stats_filename, mean, inv_vc, vc, log_det,
                            determinant);

        /* free all the previously allocated memory */
        free_imatrix(pixel_list->array, 0, pixel_list->num_pixels-1,
                                           0, pixel_list->num_bands);
        free_imatrix(index,1,lu_vc->num_classes,1,n);
        free_ivector(length,0,pixel_list->num_classes -1);
        free_vector(determinant,1,n);
        free_vector(log_det,1,n);
        free_vector(col_vector,1,n);
        free_matrix(mean->array,0,mean->num_rows-1,0,mean->num_cols-1);
        free_f3tensor(vc->array,0,vc->num_classes-1,
                                0,vc->num_rows-1,
                                0,vc->num_cols-1);
        free_f3tensor(lu_vc->array,1,lu_vc->num_classes,
                                   1,lu_vc->num_rows,
                                   1,lu_vc->num_cols);
        free_f3tensor(inv_vc->array,1,inv_vc->num_classes,
                                    1,inv_vc->num_rows,
                                    1,inv_vc->num_cols);
```

```
        return(1);

}       /* End of Main */


/*=======================================================================

        Function calc_mean_vector
        This routine calculates the mean vector and number of elements for each

        training class

=======================================================================*/
int calc_mean_vectors(pix_list_ptr pixel_list, matrix_ptr mean)
{
register int row, col, pix_num;
int status, class_index;

        mean->num_rows=pixel_list->num_classes;
        mean->num_cols=pixel_list->num_bands+1;
        mean->array=matrix(0,mean->num_rows-1,0,mean->num_cols-1);

        /* Initialize the mean matrix with zeros */
        for(row=0;row < mean->num_rows; row++)
                for(col=0; col < mean->num_cols; col++)
                        mean->array[row][col]=0.0;

        for(pix_num=0; pix_num < pixel_list->num_pixels; pix_num++){
                class_index=pixel_list->array[pix_num][mean->num_cols-1];
                for(col=0; col < mean->num_cols-1; col++)
                        mean->array[class_index][col]+=
                                        pixel_list->array[pix_num][col];
                mean->array[class_index][mean->num_cols-1]+=1.0;
        }

        /* Need to normalize the individual vectors */
        for(row=0; row < mean->num_rows; row++)
                for(col=0; col < mean->num_cols-1; col++)
                   mean->array[row][col]/=mean->array[row][mean->num_cols-1];

#ifdef MEAN_DEBUG       /* debug code - print out the mean matrix */
        printf("\nMean matrix with num_elements in each class appended:\n");
        for(row=0; row < mean->num_rows; row++){
                for(col=0; col < mean->num_cols; col++)
                        printf("%f  ",mean->array[row][col]);
                printf("\n");
        }
#endif

        status=1;
        return(status);
}/* end of calc_mean_vectors */

/*=======================================================================

        Function calc_vc_matrix
        This routine calculates the variance-covariance matrix for each
        training class

=======================================================================*/
int calc_vc_matrix(pix_list_ptr pixel_list, matrix_ptr mean,
                three_d_matrix_ptr vc){

matrix_ptr temp, sum, product, norm_vector, norm_vector_trans;
int target_class, status;
register int class_index, row, col, pix_num, band_index;
float denom, value;

        status=0;       /* indicate failure until we know better */

        /* malloc the structure pointers */
        norm_vector = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (norm_vector==NULL){
                printf("ERROR: Unable to malloc norm_vector pointer.\n");
                return(1);
        }

        norm_vector_trans = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (norm_vector_trans==NULL){
                printf("ERROR: Unable to malloc norm_vector_trans pointer.\n");

                return(1);
        }

        product = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (product==NULL){
                printf("Unable to malloc product matrix pointer.\n");
                return(1);
        }

        sum = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (sum==NULL){
                printf("ERROR: Unable to malloc sum matrix pointer.\n");
                return(1);
        }

        temp = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (temp==NULL){
                printf("ERROR: Unable to malloc temp matrix pointer.\n");
                return(1);
        }

        /* Set up storage for the vc matrix */
        vc->num_cols = pixel_list->num_bands;
        vc->num_rows = pixel_list->num_bands;
        vc->num_bands = pixel_list->num_bands;
        vc->num_classes = pixel_list->num_classes;
        vc->array = f3tensor(0, vc->num_classes-1,
                             0,vc->num_rows-1,
                             0,vc->num_cols-1);

        /* initialize the vc matrix */
        for(class_index=0; class_index < vc->num_classes; class_index++)
                for(row=0; row < vc->num_rows; row++)
                        for (col=0; col < vc->num_bands; col++)
                                vc->array[class_index][row][col]=0.0;
```

```
        /* set up the storage for the normalized vector */
        norm_vector->num_rows = vc->num_bands;
        norm_vector->num_cols = 1;
        norm_vector->array = matrix(0,norm_vector->num_rows-1,
                                    0,norm_vector->num_cols-1);

        /* and its transpose */
        norm_vector_trans->num_rows = 1;
        norm_vector_trans->num_cols = vc->num_bands;
        norm_vector_trans->array = matrix(0,norm_vector_trans->num_rows-1,
                                          0,norm_vector_trans->num_cols-1);

        /* MUST set up storage for the matrix multiplication result */
        product->num_rows = vc->num_bands;
        product->num_cols = vc->num_bands;
        product->array = matrix(0,product->num_rows-1,0,product->num_cols-1);

        /* MUST set up storage for the matrix addition result */
        sum->num_rows = vc->num_bands;
        sum->num_cols = vc->num_bands;
        sum->array = matrix(0,sum->num_rows-1,0,sum->num_cols-1);

        /* calculate (x-u)*(x-u)^T */
for(pix_num=0; pix_num < pixel_list->num_pixels; pix_num++){
        target_class=pixel_list->array[pix_num][pixel_list->num_bands];

        for(band_index=0; band_index < vc->num_bands; band_index++){
                norm_vector->array[band_index][0]=
                                    pixel_list->array[pix_num][band_index]
                                    - mean->array[target_class][band_index];

                norm_vector_trans->array[0][band_index]=
                                    norm_vector->array[band_index][0];
        }  /* end of band_index loop */

                mult_matrices(norm_vector, norm_vector_trans, product);

                /* move vc[band] into structure to pass to add_matrices */
                temp->array=vc->array[target_class];
                temp->num_rows=vc->num_rows;
                temp->num_cols=vc->num_cols;

                add_matrices(temp, product, sum);

                /* read the sum back into the appropraite vc array */
                for(row=0; row < vc->num_rows; row++)
                        for(col=0; col < vc->num_cols; col++)
                                vc->array[target_class][row][col]=
                                                sum->array[row][col];

#ifdef SUBVC_DEBUG
        printf("Currently processing pixel number %d.\n", pix_num);
        printf("This pixel belongs to class %d.\n",target_class);
        printf("\nThis is the result of the call to mult_matrices\n");

        for(row=0; row < product->num_rows; row++){
                for(col=0; col < product->num_cols; col++)
                        printf("%f\t", product->array[row][col]);
                printf("\n");
        }

        printf("\n");
        printf("\nThis is the result of the call to add_matrices\n");

        for(row=0; row < sum->num_rows; row++){
                for(col=0; col < sum->num_cols; col++)
                        printf("%f\t", sum->array[row][col]);
                printf("\n");
        }

        printf("\n");
        printf("\nThese are the values of the elements in the vc matrix\n");

        for(class_index=0;class_index<vc->num_classes;class_index++){

                for(row=0; row < vc->num_rows; row++){

                        for(col=0; col < vc->num_cols; col++)
                                printf("%f  ",vc->array[class_index][row][col]);

                        printf("\n");

                }

                printf("\n");

        } /* end of class index loop */
#endif

} /* end of pix_num loop */

        /* divide elements by one less than the num_elements in the class */
        printf("Normalizing the elements of the vc matrix.\n");
        for(class_index=0; class_index < vc->num_classes; class_index++){
            value = (float)mean->array[class_index][pixel_list->num_bands];
            denom = value-1.0;
            for(row=0; row < vc->num_rows; row++)
                  for(col=0; col < vc->num_cols; col++)
                        vc->array[class_index][row][col]/=denom;
        }

#ifdef DEBUG
        printf("\nThese are the vc matrices after normalization:\n\n");
        for(class_index=0; class_index < vc->num_classes; class_index ++){
                for(row=0; row < vc->num_rows; row++){
                        for(col=0; col < vc->num_cols; col++)
                              printf("%f  ",vc->array[class_index][row][col]);
                        printf("\n");
                }
                printf("\n");
        }
#endif

        /* free all the allocated memory, order is important */
        free_matrix(norm_vector->array,0,norm_vector->num_rows-1,
```

A-19

```
                                0,norm_vector->num_cols-1);
        free_matrix(norm_vector_trans->array,0,norm_vector_trans->num_rows-1,
                                0,norm_vector_trans->num_cols-1);
        free_matrix(product->array,0,product->num_rows-1,
                                0,product->num_cols-1);
        free_matrix(sum->array,0,sum->num_rows-1,0,sum->num_cols-1);
        free(norm_vector);
        free(norm_vector_trans);
        free(sum);
        free(product);
        free(temp);

        status=1;       /* We're done, let's jam */
        return(status);

} /* end of calc_vc_matrix */

/*============================================================================

        Function write_output
        this function writes the mean vector matrix, The variance-covariance
        matrix, its determinant, and its inverse to a user defined disk file.
        Function will happily cream any disk file with the same name as the
        user inputs

============================================================================*/
int write_output(char *filename, matrix_ptr mean, three_d_matrix_ptr inv_vc,
                three_d_matrix_ptr vc, float *log_det, float *determinant)
{

FILE *out_file;
register int row, col, class_index;
int status;
char response[128];

        status=0;       /* assume failure */
        out_file = fopen(filename, "w");
        if(out_file == NULL){
                AVSmessage("beta",AVS_Error,NULL,"Class Statistics","Okay",
                        "Could not create: %s\n",filename);
                return(1);
        }

        rewind(out_file);
        /* print the file verification header */
        fprintf(out_file, "training class statistics file for AVS GML\n");
        fprintf(out_file,"%d     bands\n",inv_vc->num_bands);
        fprintf(out_file,"%d     classes\n",inv_vc->num_classes);

        /* print the mean vectors */
        fprintf(out_file,"These are the mean vectors:\n");
        for(row=0; row < mean->num_rows; row++){
                for(col=0; col < mean->num_cols; col++)
                        fprintf(out_file,"%f  ",mean->array[row][col]);
                fprintf(out_file,"\n");
        }

        /* print the inverted vc matrices */
        fprintf(out_file,"These are the inverted matrices:\n");
        for(class_index=1; class_index <= inv_vc->num_classes; class_index ++){

                for(row=1; row <= inv_vc->num_rows; row++){
                        for(col=1; col <= inv_vc->num_cols; col++)
                                fprintf(out_file,"%f  ",
                                        inv_vc->array[class_index][row][col]);
                        fprintf(out_file, "\n");
                }
        }

        /* print out the natural log of the determinants */
        fprintf(out_file,"These are the natural logs of the determinants:\n");
        for(class_index=1; class_index <= inv_vc->num_classes; class_index ++)
                fprintf(out_file,"%f  ", log_det[class_index] );
        fprintf(out_file, "\n");

        /* print out the normalized vc matrices */
        fprintf(out_file, "\nThese are the normalized vc matrices:\n\n");
        for(class_index=0; class_index < vc->num_classes; class_index ++){
                for(row=0; row < vc->num_rows; row++){
                        for(col=0; col < vc->num_cols; col++)
                                fprintf(out_file, "%f  ",
                                        vc->array[class_index][row][col]);
                        fprintf(out_file, "\n");
                }
                fprintf(out_file, "\n");
        }

        /* print out the determinants */
        fprintf(out_file,"These are the determinants:\n");
        for(class_index=1; class_index <= inv_vc->num_classes; class_index ++)
                fprintf(out_file,"%f  ", determinant[class_index] );
        fprintf(out_file, "\n");

        fclose(out_file);
        printf("The statistics file was successfully written to disk.\n");


        /* return success */
        status=1;
        return(status);

}       /* end of write_output */
```

A-20

```
/*=========================================================================

This program performs Gaussian Maximum Likelihood (GML) on an image given a
file with a mean vector and a variance covariance matrix for each class.  This
file can be created with the classtat program (another AVS module).

Author:                 Nessmiller, Steven W.
Date of last revision:  16 April 1995
                        7 May 1995 -
                                support added to read statistics file
                        20 May 95 -
                                more filename handling enhancements
                                handles images that are NOT square
                        29 June 95
                                minor bug fix

=========================================================================*/
#include <math.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <sys/time.h>
#include <avs/avs.h>
#include <avs/avs_data.h>
#include <avs/field.h>
#include <avs/flow.h>
#include "memutil.h"
#include "imat.h"

#define MAXSTR 80
#define HUGEMHD 600

/* layered debug definitions */
#define DEBUG
#define STATS_DEBUG

/*
#define HIGHPROCESS_DEBUG
#define LOWPROCESS_DEBUG
*/

/* this structure supports the input image */
typedef struct{
        int num_rows;
        int num_cols;
        int num_bands;
}IMG_STRUCT;

typedef IMG_STRUCT *image_ptr;

/* this structure supports the classification map */
typedef struct{
        int **array;
        int num_rows;
        int num_cols;
        int num_classes;
}CLASS_MAP;

typedef CLASS_MAP *class_map_ptr;

/* this is the inv_vc matrix */
typedef struct{
        float ***array;
        float *nlog_determinant;
        int num_rows;
        int num_cols;
        int num_classes;
        int num_bands;
}D3_MAT_STRUCT;

typedef D3_MAT_STRUCT *three_d_matrix_ptr;

/* function prototypes */
int get_stats(char *filename, matrix_ptr mean, three_d_matrix_ptr inv_vc);

int process_image(float chi_squared,
                  AVSfield_char *input_image,
                  image_ptr input,
                  matrix_ptr mean,
                  three_d_matrix_ptr inv_vc,
                  class_map_ptr classified,
                  int interface_skip_zeros);

AVSinit_modules()
{
        int AVS_gml();
        AVSmodule_from_desc (AVS_gml);
}

AVS_gml()
{
        int param;
        int in_port, out_port;
        char cwd[MAXSTR];
        int GML_compute();

/* set the module name and type */
        AVSset_module_name("GML Classification", MODULE_FILTER);

/* create the input port */
        in_port = AVScreate_input_port( "Input Image", "field 2D uniform byte",

                                        REQUIRED );
/* create the output port */
        out_port = AVScreate_output_port( "Class Image",
                                          "field 2D uniform float",REQUIRED );

/* set up the input windows and buttons */
        param = AVSadd_parameter("head", "string",
                                 "GML Classification Parameters:", "", "");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* create a window to select the statistics filename */
        getwd(cwd);
```

```
        if( cwd[strlen(cwd)-1] != '/')
                strcat(cwd,"/");

       param=AVSadd_parameter("Statistics file (*.sta):","string",cwd,0,".sta")
;
        AVSadd_parameter_prop(param, "width","integer",4);
        AVSconnect_widget(param, "browser");

/* allow user to view all file types */
        param=AVSadd_parameter("View all file types","boolean",0,0,1);
        AVSadd_parameter_prop(param,"width","integer",4);

/* create a widget to get chi squared value */
        param =AVSadd_float_parameter("Chi squared:", 12.8, 1.0, FLOAT_UNBOUND);
        AVSconnect_widget(param, "typein_real");

/* create an output window to display the number of classes */
        param=AVSadd_parameter("Number of classes:","integer",0,0,INT_UNBOUND);

        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create an output window to display the number of bands */
        param=AVSadd_parameter("Number of bands:","integer", 0, 0,INT_UNBOUND);

        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the percentage of the image that is classified */

        param =AVSadd_float_parameter("Percent classified:", 0.0, 0.0, 100.0);
        AVSconnect_widget(param, "typein_real");

/* create an output window to display elapsed time */
        param=AVSadd_parameter("Elapsed time (sec):","integer",0,0,
                                INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to skip zero vectors */
        param=AVSadd_parameter("Skip zero vectors", "boolean", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to permit autogeneration */
        param=AVSadd_parameter("Auto-generate GML", "boolean", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to generate the GML class map */
        param=AVSadd_parameter("Generate GML map", "oneshot", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);

/* free the output data */
        AVSautofree_output(out_port);

/* set the function to create the output */
        AVSset_compute_proc(GML_compute);
}

int GML_compute(AVSfield_char *input_image,
                AVSfield_float **output_image,
                char *head,
                char *stats_filename,
                int all,
                float *interface_chi_square,
                int interface_num_classes,
                int interface_num_bands,
                float *interface_percent_classified,
                int time,
                int interface_skip_zeros,
                int auto_run,
                int run)
{
/* local variables */
image_ptr input;
class_map_ptr classified;
three_d_matrix_ptr inv_vc;
matrix_ptr mean;
float chi_squared;
register int row, col;
int value, dims[2], status;
struct timeval before, after;
FILE *input_file;
static int first_time_through = 1;
static int valid_stats_file = 0;
static char old_stats_filename[80];
char *reset_stats_filename = " ";

        /* check for view all parameter filenames */
        if(AVSparameter_changed("View all file types")){
                if (all){
                        AVSmodify_parameter("Statistics file (*.sta):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/",0,"");
                        printf( "Setting maxval to null\n");
                        AVSmodify_parameter("Statistics file (*.sta):",
                             AVS_VALUE, stats_filename, 0, 0);
                }
                else{
                        AVSmodify_parameter("Statistics file (*.sta):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/", 0, ".sta");
                        printf( "Setting maxval to .sta\n");
                        AVSmodify_parameter("Statistics file (*.sta):",
                                        AVS_VALUE, stats_filename, 0, 0);
                }
        }

        if(first_time_through){
                /* allocate memory for the structures */
                printf("Allocating structures for statistics file read.\n");

                inv_vc = (three_d_matrix_ptr)malloc(sizeof(D3_MAT_STRUCT));
                if (inv_vc == NULL){
                        printf("ERROR: Unable to malloc inv_vc pointer\n");
                        return(1);
                }
```

```
        inv_vc->array = NULL;

        mean = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (mean == NULL){
                printf("ERROR: Unable to malloc mean pointer\n");
                return(1);
        }
        mean->array = NULL;

        first_time_through = 0;
}

/* ensure that we have a file and NOT just a directory */
if(stats_filename[strlen(stats_filename) - 1] == '/'){
        valid_stats_file = 0;
        return(1);
}

/* see if the statistics file has changed */
status = strcmp(stats_filename, old_stats_filename);

if (status != 0){
    strcpy(old_stats_filename, stats_filename);
    input_file = fopen(stats_filename, "r+");

    if(input_file == NULL){
        AVSmessage("beta", AVS_Error, NULL, "GML Classification",
                "Okay", "Couln not open file %s.\n", stats_filename);
        valid_stats_file = 0;
        return(1);
    }
    else{
        /* free memory if necessary */
        if (mean->array != NULL){
            printf("Freeing the mean matrix.\n");
            free_matrix(mean->array,0,mean->num_rows-1,
                                0,mean->num_cols-1);
        }

        if (inv_vc->array != NULL){
            printf("Freeing the inv_vc matrix.\n");
            free_f3tensor(inv_vc->array,0,inv_vc->num_classes-1,
                                    0,inv_vc->num_rows-1,
                                    0,inv_vc->num_cols-1);
        }

        /* get the mean and vc matrices from the file */
        valid_stats_file = get_stats(stats_filename,mean,inv_vc);

        /* update the interface with the data from the file */
        if(valid_stats_file){
            AVSmodify_parameter("Number of classes:",
        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
        inv_vc->num_classes, inv_vc->num_classes, inv_vc->num_classes);

            AVSmodify_parameter("Number of bands:",
        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
        inv_vc->num_bands, inv_vc->num_bands, inv_vc->num_bands);
```

```
                }
            }
        }

#ifdef DEBUG
        printf("The statistics filename is %s.\n", stats_filename);
        printf("The value of Chi squared is %f\n", *interface_chi_square);
        printf("Number of classes = %d and number of bands = %d.\n",
                inv_vc->num_classes, inv_vc->num_bands);
        printf("run = %d autorun = %d.\n", run, auto_run);
#endif

        /* check if run time */
        if( run && valid_stats_file )
                AVSmodify_parameter("Generate GML map", AVS_VALUE, 0, 0, 0 );

        else {
                AVSmark_output_unchanged("Class Image");
                return(1);
        }

        /* reset the static variables */
        valid_stats_file = 0;
        first_time_through = 1;
        strcpy(old_stats_filename, reset_stats_filename);

        /* clear out the elapsed time  and percent classified from last run */
        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            0, 0, 0);

        AVSmodify_float_parameter("Percent classified:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  0.0, 0.0, 0.0 );

        /* allocate memory for the structure pointers */
        input = (image_ptr)malloc(sizeof(IMG_STRUCT));
        if(input == NULL){
                printf("ERROR: Unable to malloc input image pointer.\n");
                return(1);
        }

        /* set up some variables that we'll need */
        input->num_rows = MAXY(input_image);
        input->num_cols = MAXX(input_image);
        input->num_bands = input_image->veclen;

        /* check to make sure that the parameter file and the image jive */
        if (input->num_bands != inv_vc->num_bands){
                AVSmessage("beta",AVS_Error, NULL, "GML Classification","Okay",
                        "The input image and the statistics file are NOT compatible.");
                return(1);
        }

        classified = (class_map_ptr)malloc(sizeof(CLASS_MAP));
        if (classified == NULL){
                printf("ERROR: Unable to malloc classified image pointer\n");
                return(1);
```

```
        }

#ifdef DEBUG
        printf("All structures allocated.\n");
#endif

        /* set up memory for the classification map */
        classified->num_classes = inv_vc->num_classes;
        classified->num_rows=input->num_rows;
        classified->num_cols=input->num_cols;
        classified->array=imatrix(0,classified->num_rows-1,
                                  0,classified->num_cols-1);

        /* calculate each pixel's class membership */
        gettimeofday(&before, NULL);
        chi_squared = *interface_chi_square;
        status=process_image(chi_squared, input_image, input, mean, inv_vc,
                             classified, interface_skip_zeros);
        if (status!=1){
                printf("Status error in process_image\n");
                return(1);
        }

        gettimeofday(&after, NULL);
        time = (int)(after.tv_sec - before.tv_sec);
        printf("All pixels processed in %d seconds.\n",time);
        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            time, time, time);

        /* set up storage for the output image */
        dims[0] = input->num_cols;
        dims[1] = input->num_rows;

        if ( *output_image == NULL ) {
            *output_image = ( AVSfield_float * )
                ( AVSdata_alloc( "field 2D uniform scalar float", dims ));
        }
        else if ( ( dims[ 0 ] != MAXX( *output_image ) ) ||
                  ( dims[ 1 ] != MAXY( *output_image ) ) ) {
            AVSfield_free( (AVSfield *) *output_image );
            *output_image = ( AVSfield_float * )
                ( AVSdata_alloc( "field 2D uniform scalar float", dims ));
        }

#ifdef DEBUG
        printf("output field allocated\n" );
#endif

        /* copy the classification map to the output port */
        for (row=0; row < classified->num_rows; row++) {
            for (col=0; col < classified->num_cols; col++) {
                I2D(*output_image, col, row) = classified->array[row][col];
            }
        }

#ifdef DEBUG
        printf("Field written\n" );
#endif
```

```
#endif

        /* free all the allocated memory, order is VERY IMPORTANT!!!!*/
        free_imatrix(classified->array,
                     0,classified->num_rows-1,
                     0,classified->num_cols-1);

        free_f3tensor(inv_vc->array,0,inv_vc->num_classes-1,
                      0,inv_vc->num_rows-1,
                      0,inv_vc->num_cols-1);

        free_matrix(mean->array,0,mean->num_rows-1,0,mean->num_cols-1);

        /* now free the structure pointers */
        free(input);
        free(classified);
        free(inv_vc);
        free(mean);

        /* return success */
        return(1);
}

/*=============================================================================

function get_stats
        reads in the stats file and assigns a pointer to it.


=============================================================================*/
int get_stats(char *filename, matrix_ptr mean, three_d_matrix_ptr inv_vc)
{
        FILE *in_file;
        register int row, col, class_num;
        char line[MAXSTR];
        char *valid = "training class statistics file for AVS GML\n";
        int classes, bands, status;


        in_file = fopen(filename,"r");

                if (!in_file){
                AVSmessage("beta", AVS_Error, NULL, "GML Classification",
                     "Okay", "Unable to open statistics file: %s",filename);
                return(-1);
                }

        printf("The requested statistics file was located\n");

        /* check for proper file type */
        (void)fgets(line, sizeof(line), in_file);
        status = strcmp(line, valid);
        if (status != 0) {
                fclose(in_file);
                AVSmessage("beta", AVS_Error, NULL, "GML Classification",
                     "Okay", "%s is NOT a valid GML statistics file",filename);

                return(-1);
```

```
        }

        status = 1;     /* assume success */

        /* read in the rest of the header */
        (void)fgets(line,sizeof(line), in_file);
        (void)sscanf(line,"%d",&bands);
        (void)fgets(line,sizeof(line), in_file);
        (void)sscanf(line,"%d",&classes);

        /* asssign the values to the structure elements */
        mean->num_rows = classes;
        mean->num_cols = bands+1;       /* total number of pixels in class */
        inv_vc->num_rows = bands;
        inv_vc->num_cols = bands;
        inv_vc->num_classes = classes;
        inv_vc->num_bands = bands;

#ifdef STATS_DEBUG      /* Echo the header information just read in */
        printf("Statistics file header information:\n");
        printf("The num rows in the mean matrix = %d\n",mean->num_rows);
        printf("The num columns in the mean matrix = %d\n",mean->num_cols);
        printf("The num rows in each inv_vc matrix = %d\n",inv_vc->num_rows);
        printf("The num cols in each inv_vc matrix = %d\n",inv_vc->num_cols);
        printf("The number of classes in the data = %d\n",inv_vc->num_classes);
        printf("The number of bands in the data = %d\n",inv_vc->num_bands);
#endif

        /* malloc the memory for the mean matrix and check its status */
        mean->array = matrix(0,mean->num_rows-1,0,mean->num_cols-1);
        inv_vc->array = f3tensor(0,inv_vc->num_classes-1,
                                 0,inv_vc->num_rows-1,
                                 0,inv_vc->num_cols-1);
        inv_vc->nlog_determinant = vector(0,inv_vc->num_classes-1);

        if (mean->array == NULL){
                printf("ERROR: unable to malloc mean matrix.\n");
                status = 0;
                return(status);
        }

        if (inv_vc->array == NULL){
                printf("ERROR: unable to malloc inv_vc matrix.\n");
                status = 0;
                return(status);
        }

        if (inv_vc->nlog_determinant == NULL){
                printf("ERROR: unable to malloc ln of determinant matrix.\n");
                status = 0;
                return(status);
        }


        printf("Statistics file file being read into memory\n");
        printf("Processing the mean matrix\n");
        (void)fgets(line,sizeof(line), in_file); /* skip the mean heading */
        /* read in the values */
        for (row=0; row < mean->num_rows; row++)
                for (col = 0; col < mean->num_cols; col++)
                                fscanf(in_file, "%f", &mean->array[row][col]);

        printf("Processing the inv_vc matrix\n");
        /* the file pointer has not seen the EOL yet! */
        (void)fgets(line,sizeof(line),in_file);/* move it to the next line */
        (void)fgets(line,sizeof(line),in_file); /* skip the matrix heading */
        /* read in the values */
        for (class_num=0; class_num < inv_vc->num_classes; class_num++)
                for (row=0; row < inv_vc->num_rows; row++)
                        for (col = 0; col < inv_vc->num_cols; col++)
                fscanf(in_file, "%f", &inv_vc->array[class_num][row][col]);

        /* read in the natural log of the determinants */
        /* the file pointer has not seen the EOL yet! */
        (void)fgets(line,sizeof(line),in_file);/* move it to the next line */
        (void)fgets(line,sizeof(line),in_file); /* skip the matrix heading */
        /* read in the values */
        for (class_num=0; class_num < inv_vc->num_classes; class_num++)
                fscanf(in_file, "%f", &inv_vc->nlog_determinant[class_num]);


        fclose(in_file);
        printf("Training statistics successfully read in.\n");

#ifdef STATS_DEBUG /* Debug code - echo the matrix values to maintain sanity */

        printf("These are the mean vectors:\n");
        for (row=0; row < mean->num_rows; row++){
                for (col = 0; col < mean->num_cols; col++)
                        printf("%2f  ",mean->array[row][col]);
                printf("\n");
        }

        printf("These are the inverted vc matrices:\n");
        for (class_num=0; class_num < inv_vc->num_classes; class_num++)
                for (row=0; row < inv_vc->num_rows; row++){
                        for (col = 0; col < inv_vc->num_cols; col++)
                        printf("%2f  ",inv_vc->array[class_num][row][col]);
                printf("\n");
                }

        printf("These are the natural logs of the determinants:\n");
        for (class_num=0; class_num < inv_vc->num_classes; class_num++)
                printf("%2f  ",inv_vc->nlog_determinant[class_num]);
        printf("\n");
#endif

        return(status);

}

/*=============================================================================

Function process_image
        This function produces the classification map
```

A-25

```
==========================================================================*/
int process_image(float chi_squared,
                  AVSfield_char *input_image,
                  image_ptr input,
                  matrix_ptr mean,
                  three_d_matrix_ptr inv_vc,
                  class_map_ptr classified,
                  int interface_skip_zeros)
{

/* local variables */
matrix_ptr norm_vector,
           norm_vector_trans,
           spectral_vector,
           psi,
           product1,
           product2;
register int row, col, band, class_num, x;
float smallest, mh_dist;
int smallest_index, assigned, num_classified, magnitude;
float percent_classified, percent_done, row_step, stage;
int send_val, time, status;
struct timeval before, after;


        /* set up storage for each matrix structure pointer*/
        norm_vector = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (norm_vector == NULL){
                printf("Unable to malloc norm_vector matrix pointer\n");
                status = 0;
                return(status);
        }

        norm_vector_trans = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (norm_vector_trans == NULL){
                printf("Unable to malloc norm_vector transpose pointer\n");
                status = 0;
                return(status);
        }

        spectral_vector = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (spectral_vector == NULL){
                printf("Unable to malloc spectral_vector transpose pointer\n");
                status = 0;
                return(status);
        }

        psi = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (psi == NULL){
                printf("Unable to malloc psi matrix pointer\n");
                status = 0;
                return(status);
        }

        product1 = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (product1 == NULL){
                printf("Unable to malloc product1 matrix pointer\n");
                status = 0;
                return(status);
        }

        product2 = (matrix_ptr)malloc(sizeof(MAT_STRUCT));
        if (product2 == NULL){
                printf("Unable to malloc product2 matrix pointer\n");
                status = 0;
                return(status);
        }


        /* set up the storage for the normalized vector */
        norm_vector->num_rows = 1;
        norm_vector->num_cols = input->num_bands;
        norm_vector->array = matrix(0,norm_vector->num_rows-1,
                                    0,norm_vector->num_cols-1);
        /* and its transpose */
        norm_vector_trans->num_rows = input->num_bands;
        norm_vector_trans->num_cols = 1;
        norm_vector_trans->array = matrix(0,norm_vector_trans->num_rows-1,
                                          0,norm_vector_trans->num_cols-1);
        /* and the spectral vector */
        spectral_vector->num_rows=1;
        spectral_vector->num_cols=input->num_bands;
        spectral_vector->array=matrix(0,spectral_vector->num_rows-1,
                                      0,spectral_vector->num_cols-1);

        /* MUST set up storage for the first matrix multiplication result */
        product1->num_rows = 1;
        product1->num_cols = input->num_bands;
        product1->array=matrix(0,product1->num_rows-1,0,product1->num_cols-1);

        /* MUST set up storage for the second matrix multiplication result */
        product2->num_rows = 1;
        product2->num_cols = 1;
        product2->array=matrix(0,product2->num_rows-1,0,product2->num_cols-1);


        /* classify the entire input image */
        num_classified = 0;
        stage = 0.10;
        row_step = 1.0 / input->num_rows;
        for(row = 0; row < input->num_rows; row++){
                for(col = 0; col < input->num_cols; col++){

                        /* load each pixel into the spectral vector */
                        magnitude = 0;
                        for(band = 0; band < input->num_bands; band++){
   spectral_vector->array[0][band] = (float)(*(I2DV(input_image,col,row)+band))
;
   magnitude += spectral_vector->array[0][band];
                        }

                /* reset the value of smallest, its index, and assigned */
                        smallest = HUGEMHD;
                        smallest_index=0;
                        assigned = 0;
```

```
if( (interface_skip_zeros) && (magnitude == 0) ){
        smallest = HUGEMHD;
        smallest_index=0;
        assigned = 0;
}
else{
        /* step through each class */
        for(class_num = 0; class_num < inv_vc->num_classes; class_num++){

                /* normalize the pixel with respect to each class */
                for(band=0; band < input->num_bands; band++){
                norm_vector->array[0][band] = spectral_vector->array[0][band]
                                            - mean->array[class_num][band];
                norm_vector_trans->array[band][0]=norm_vector->array[0][band];
                }

                /* move inv_vc[class_num] into structure for mult_matrices */
                psi->array=inv_vc->array[class_num];
                psi->num_rows=inv_vc->num_rows;
                psi->num_cols=inv_vc->num_cols;

                /* calculate (x-u)*psi*(x-u)^T */
                mult_matrices(norm_vector, psi, product1);
                mult_matrices(product1, norm_vector_trans, product2);

                /* this is the mh_dist */
                mh_dist = product2->array[0][0];

                /* account for the ln of the determinant of each class */
                mh_dist += inv_vc->nlog_determinant[class_num];

                /* see if this mh_dist is the smallest */
                if(mh_dist < smallest){
                        smallest = mh_dist;

                        if (smallest < chi_squared){
                            /* classify the pixel, zero is background */
                            smallest_index = class_num+1;
                            classified->array[row][col] = smallest_index;
                            assigned = 1;
                        }
                }

        }       /* end of class_num loop */

}       /* end else loop */

#ifdef HIGHPROCESS_DEBUG /* debug code - echo the classification results */

        printf("\nProcessing spectral vector [%d][%d]\n",row,col);
        for(x = 0; x < spectral_vector->num_cols; x++)
                printf("%f  ",spectral_vector->array[0][x]);

        printf("\n");
        printf("The smallest Mahalanobis distance = %f\n",smallest);
        printf("This value must be less than %f to be classified.\n",
                chi_squared);

        if (smallest < chi_squared){
                printf("It was assigned to class %d.\n",smallest_index);
                printf("This class' mean vector is:\n");

                for(x = 0; x < mean->num_cols-1; x++)
                        printf("%f  ",mean->array[smallest_index-1][x]);
        }
        else{
                printf("This pixel was assigned to the background class.\n");
        }

        printf("\n");
        printf("\n");
#endif


        if (assigned == 1)
                num_classified += 1;

        } /* end of col loop */

        /* report the level of completion */
        percent_done=(float)row/(float)input->num_rows;
        if((percent_done > stage) && (percent_done < stage+(2*row_step))){
                send_val = (int)(100.0 * stage);
                printf("percent complete %d\n",send_val);
                AVSmodule_status("percent complete",send_val);
                stage += .10;
        }

        } /* end of row loop */

        /* report the percentage of the image that is classified */
        percent_classified =  100.0 * ( (float) num_classified /
                              (float)(input->num_rows * input->num_cols) );

        AVSmodify_float_parameter("Percent classified:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  percent_classified, percent_classified,
                                  percent_classified);


        /* free all the allocated memory.  Order is VERY IMPORTANT!!!!!!!!!!*/
        free_matrix(norm_vector->array,0,norm_vector->num_rows-1,
                                        0,norm_vector->num_cols-1);
        free_matrix(norm_vector_trans->array,0,norm_vector_trans->num_rows-1,
                                        0,norm_vector_trans->num_cols-1);
        free_matrix(product1->array,0,product1->num_rows-1,
                                        0,product1->num_cols-1);
        free_matrix(product2->array,0,product2->num_rows-1,
                                        0,product2->num_cols-1);
        free_matrix(spectral_vector->array,0,spectral_vector->num_rows-1,
                                        0,spectral_vector->num_cols-1);
        free(norm_vector);
        free(norm_vector_trans);
        free(spectral_vector);
        free(psi);
        free(product1);
```

```
        free(product2);

        return(1); /* we're done, let's jam */
}
```

Source code for the nPDF Projection and nPDF Classification AVS modules

```
/*=========================================================================
TITLE:  AVS_NPDF.c
AUTHOR: Nessmiller, Steven W.
DATE:   10 Feb 95

SUMMARY: This AVS_module performs the nPDF transformation on a given
input image.  In addition,it allows the user to specify the corners
of the hypercube from which the distance measures are made as well as
the scale factor for the resulting nPDF plot.

=========================================================================*/

#include <math.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <time.h>
#include <limits.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/flow.h>
#include <avs/avs_data.h>
#include "memutil.h"

#define DEBUG
/*
#define DIST_DEBUG
*/

#define MAXSTR 80

typedef struct{
        int dist1;
        int dist2;
}COORDINATE;

/* this structure is used to reference the output height field */

typedef struct{
        unsigned long **array;
        int num_rows;
        int num_cols;
}MAT_STRUCT;


/* Prototypes */
COORDINATE calc_distance(int *spectral_vector,int corner1,int corner2,
                         int range, int num_bands,int num_bits,int scale);

AVSinit_modules ()
{
        int AVS_npdf();
        AVSmodule_from_desc (AVS_npdf);
}

AVS_npdf()
{
        int param;
        int in_port, out_port;
        int npdf_compute();

/* set the module name and type */
        AVSset_module_name("nPDF Projection", MODULE_FILTER);

/* create the input port */
        in_port = AVScreate_input_port( "Input Image", "field 2D uniform byte",
                  REQUIRED );

/* create the output port */
        out_port = AVScreate_output_port( "Output Image",
                  "field 2D uniform float" );

/* set up the input windows and buttons */
        param = AVSadd_parameter("head", "string",
                                 "nPDF projection parameters", "", "");
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "text");

/* add a title to the radio buttons */
        param = AVSadd_parameter("title", "string",
                                 "Select hyper-cube corners:", "", "");
        AVSadd_parameter_prop(param, "width", "integer",4);
        AVSconnect_widget(param, "text");

/* create the radio buttons */
        param = AVSadd_parameter("corner_choice", "choice",
"corner 1 and corner 4","corner 1 and corner 2!corner 1 and corner 3!corner 1 a
nd corner 4!corner 2 and corner 3!corner 2 and corner 4!corner 3 and corner 4",

"!");
        AVSadd_parameter_prop(param, "width", "integer",4);
        AVSconnect_widget(param, "radio_buttons");

/* create an input window for the value of the scale factor */
        param = AVSadd_parameter("Scale factor", "integer", 512, 256, 2048);
        AVSadd_parameter_prop(param, "width", "integer",2);
        AVSconnect_widget(param, "typein_integer");

/* create an output window to display elapsed time */
        param=AVSadd_parameter("Elapsed time (sec):","integer",0,0,INT_UNBOUND)
;
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to permit autogeneration */
        param=AVSadd_parameter("Auto-generate nPDF plot", "boolean", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to generate the nPDF plot */
        param=AVSadd_parameter("Generate nPDF plot", "oneshot", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);

/* free the output data */
```

```
        /* AVSautofree_output(out_port); */

/* set the function to create the output */
        AVSset_compute_proc(npdf_compute);
}

int npdf_compute(AVSfield_char *input_image,
                 AVSfield_float **output_image,
                 char *head,
                 char *title,
                 char *corner_choice,
                 int scale,
                 int time,
                 int auto_run,
                 int run)
{
/* local variables */
int *spectral_vector;          /* This is the pixel               */
MAT_STRUCT height_field;       /* This is the output height field */
int dims[2];                   /* and its dimensions              */
register int row, col, band_index;
int range, num_rows, num_cols, num_bands, num_bits;
struct timeval before, after;
int corner, corner1, corner2, send_val, elapsed_time;
float percent_done, row_step, stage;
COORDINATE point;

        /* check if run time */
        if( run || auto_run )
                AVSmodify_parameter( "Generate nPDF plot", AVS_VALUE, 0, 0, 0 );

        else {
                AVSmark_output_unchanged( "Output Image" );
                return( 1 );
        }

        /* decipher the corner choice */
        corner = AVSchoice_number("corner_choice",corner_choice);

        switch(corner){
                case 1:
                        corner1=1;
                        corner2=2;
                        break;

                case 2:
                        corner1=1;
                        corner2=3;
                        break;

                case 3:
                        corner1=1;
                        corner2=4;
                        break;

                case 4:
                        corner1=2;
                        corner2=3;
                        break;

                case 5:
                        corner1=2;
                        corner2=4;
                        break;

                case 6:
                        corner1=3;
                        corner2=4;
                        break;
        }

        /* clear out the elapsed time from the last run */
        AVSmodify_parameter("Elapsed time (sec):",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        0,0,0);

        /* set up some stuff that we'll need */
        num_rows = MAXY(input_image);
        row_step = 1.0/num_rows;
        num_cols = MAXX(input_image);
        num_bands = input_image->veclen;
        num_bits = 8;
        range = (int)pow(2.0,(double)num_bits) - 1;

        /* set up storage for the spectral vector and the height field array */
        spectral_vector = ivector(0,num_bands-1);
        height_field.array = lmatrix(0,scale,0,scale);
        height_field.num_cols = scale+1;
        height_field.num_rows = scale+1;

        /* zero out the height field array */
        for (row=0; row < height_field.num_rows; row++)
            for (col=0; col < height_field.num_cols; col++)
                height_field.array[row][col]=0;

#ifdef DEBUG
        printf("Immediately prior to entering the image processing loop\n");
#endif

/* Do the nPDF transformation on the entire image */
        gettimeofday(&before,NULL);
        stage = 0.10;
        for (row=0; row < num_rows; row++) {
            for (col=0; col < num_cols; col++) {
                for(band_index=0; band_index < num_bands; band_index ++)
                        spectral_vector[band_index] =
                                *(I2DV(input_image, col, row)+band_index);

                point = calc_distance(spectral_vector, corner1, corner2, range,
                        num_bands, num_bits, scale);

                height_field.array[point.dist1][point.dist2]+=1;
            } /* end col loop */

            percent_done=(float)row/num_rows;
            if((percent_done > stage)&&(percent_done < stage+(2*row_step))){
```

```
                send_val = (int)(100.0 * stage);
                printf("percent complete %d\n",send_val);
                AVSmodule_status("percent complete",send_val);
                stage += .10;
            }

        } /* end row loop */

        gettimeofday(&after,NULL);
        elapsed_time=(int)(after.tv_sec-before.tv_sec);
        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            elapsed_time, elapsed_time, elapsed_time);

#ifdef DEBUG
        printf("All pixels processed in %d (sec).\n", elapsed_time);
#endif

        /* set up storage for the output image */
        dims[0] = dims[1] = scale;
        if ( *output_image == NULL ) {
            *output_image = ( AVSfield_float * )
                ( AVSdata_alloc( "field 2D uniform scalar float", dims ));
        } else if ( ( dims[ 0 ] != MAXX( *output_image ) ) ||
                    ( dims[ 1 ] != MAXY( *output_image ) ) ) {
            AVSfield_free( (AVSfield *) *output_image );
            *output_image = ( AVSfield_float * )
                ( AVSdata_alloc( "field 2D uniform scalar float", dims ));
        }

#ifdef DEBUG
        printf("Field allocated\n" );
#endif
        /* copy the height field to the output port */
        for (row=0; row < scale + 1; row++) {
            for (col=0; col < scale + 1; col++) {
                I2D(*output_image, col, row) = height_field.array[row][col];
            }
        }

#ifdef DEBUG
        printf("Field written\n" );
#endif
        /* free all the allocated memory */
        free_ivector(spectral_vector,0,num_bands-1);
        free_lmatrix(height_field.array,0,scale,0,scale);

        /* return success */
        return(1);

} /* end of main */

/*****************************************************************************

function calc_distance
        This function calculates the two distance measurements from the user
        defined corners of the hypercube.
        INPUTS a num_bands dimensional pixel from the image to be transformed
        OUTPUTS the two nPDF coordinates in the coordinate structure named
        point
*****************************************************************************/

COORDINATE calc_distance(int *spectral_vector,
                         int corner1,
                         int corner2,
                         int range,
                         int num_bands,
                         int num_bits,
                         int scale)
{
        COORDINATE      point;
        int             i;
        double          temp1,temp2;
        double          dist1,dist2;


        /* this matrix holds the coefficients that "turn" the various vector
        components on or off to calculate the distances from the user
        selected corners of the hypercube.  Note that the a and b values are
        complements and that the column number is essentially one less than
        the user selected corner value.  The matrix coefficients are defined
        as follows:  */

        static int      a[6][4] = {{ 1, 1, 1, 1 },
                                   { 1, 1, 0, 0 },
                                   { 1, 0, 1, 0 },
                                   { 1, 1, 1, 1 },
                                   { 1, 1, 0, 0 },
                                   { 1, 0, 1, 0 }};

        temp1=0.0;
        temp2=0.0;

        for (i =0; i < num_bands; i++){
            temp1 += pow(( double )(spectral_vector[i]*a[i][corner1-1]),2.0)+
                     (pow(( double )(range-spectral_vector[i]),2.0)*
                     (1.0-a[i][corner1-1]));
            temp2 += pow(( double )(spectral_vector[i]*a[i][corner2-1]),2.0)+
                     (pow(( double )(range-spectral_vector[i]),2.0)*
                     (1.0-a[i][corner2-1]));
        }

        dist1 = sqrt(temp1);
        dist2 = sqrt(temp2);

        /* we have the two distance measures so now do the nPDF calculations */
        temp1 = ( double )scale * dist1 / (pow(2.0,( double )num_bits)*
                pow(( double)num_bands,0.5));
        point.dist1 = (int)(temp1+0.5);

        temp2 = (double)scale * dist2 / (pow(2.0, (double)num_bits)*
                pow((double)num_bands,0.5));
        point.dist2 = (int)(temp2+0.5);

#ifdef DIST_DEBUG /* Debug code - echo distance values to maintain sanity */
        printf("npdf dist1 = %d and npdf dist2 = %d\n",point.dist1,
```

```
                point.dist2);
#endif

        return(point);

} /* end of calc_distance */


/*****************************************************************************


function write_output
        this function writes the nPDF height field matrix as determined
        by the algorithm to a user defined disk file.  Function will happily
        cream any disk file with the same name as the user inputs.

*****************************************************************************/

void write_output(MAT_STRUCT height_field)
{

FILE *out_file;
register int row, col;
char response[MAXSTR];

        fflush(stdin);  /* clear stdin befor the character read */
        printf("Please enter the name for the nPDF height field (.hf) array: ");
        gets(response);
        out_file = fopen(response, "w");

        /* rewind(out_file); */

        for(row=0; row < height_field.num_rows; row++){
            for (col=0; col < height_field.num_cols; col++){
                fprintf(out_file,"%7d",height_field.array[row][col]);
            }
            fprintf(out_file,"\n");
        }

        fclose(out_file);
        printf("The nPDF height field array was successfully written to disk.\n");
}
```

```
/*=========================================================================

TITLE:  npdf_classification.c
AUTHOR: Nessmiller, Steven W.
DATE:   3 March 1995

SUMMARY:
This program performs image classification by applying the nPDF transformation

and utilizing a previously determined LUT.  The nPDF distance measurements are
calculated for each pixel in the original image and these two values are used
to index the LUT.  If the distances reference a target class in the LUT, the
pixel in the output image is assigned that class' numerical value.  If the
pixel's resulting distances place it outside of any class, it is assigned a
value of zero.  The user must inform the program of the corners of the
hypercube utilized to make the LUT as this information is othewrwise
unavailable.

REVISIONS:
        22 May 95 added LUT support
        22 July 95 added skip zero vector support

=========================================================================*/


#include <math.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <sys/time.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/flow.h>
#include "memutil.h"

/* layered debug definitions */
#define DEBUG
/* #define DIST_DEBUG */

#define MAXSTR 80

typedef struct{
        int dist1;
        int dist2;
}COORDINATE;

/* this structure is used to reference the classification map */
typedef struct{
        int **array;
        int num_rows;
        int num_cols;
}CLASS_MAP_STRUCT;


/* Prototypes */
COORDINATE calc_distance(int *spectral_vector,int corner1,int corner2,
                         int range, int num_bands,int num_bits,int scale);
```

```
AVSinit_modules ()
{
        int AVS_npdf_classification();
        AVSmodule_from_desc(AVS_npdf_classification);
}

AVS_npdf_classification()
{
        int param;
        int in_port, out_port, LUT_port;
        int npdf_compute_class();

/* set the module name and type */
        AVSset_module_name("nPDF Classification", MODULE_FILTER);

/* create the image input port - this one will be leftmost */
        in_port = AVScreate_input_port( "Input Image", "field 2D uniform byte",
                                        REQUIRED );

/* create the LUT input port - this one will be rightmost */
        LUT_port = AVScreate_input_port("LUT Image","field 2D uniform float",
                                        REQUIRED);

/* create the output port */
        out_port = AVScreate_output_port( "Output Image",
                   "field 2D uniform float" );

/* set up the input windows and buttons */
        param = AVSadd_parameter("head", "string",
                                 "nPDF classification parameters", "", "");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* add a title to the radio buttons */
        param = AVSadd_parameter("title", "string", "Select hyper-cube corners:"
          ,"", "");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* create the radio butons to input the values of the corners */
        param = AVSadd_parameter("corner_choice", "choice",
                "corner 1 and corner 4","corner 1 and corner 2!corner 1 and corner 3!corner 1
 and corner 4!corner 2 and corner 3!corner 2 and corner 4!corner 3 and corner 4
", "!");
        AVSadd_parameter_prop(param, "width", "integer",4);
        AVSconnect_widget(param, "radio_buttons");

/* create an input window for the value of the scale factor */
        param = AVSadd_parameter("Scale factor:", "integer",512, 256, 2048);
        AVSadd_parameter_prop(param, "width", "integer",2);
        AVSconnect_widget(param, "typein_integer");

/* create a widget to display the percent classified */
        param = AVSadd_float_parameter("Percent classified:", 0.0, 0.0, 100.0);
        AVSadd_parameter_prop(param, "width", "integer", 2);
        AVSconnect_widget(param, "typein_real");

/* create an output window to display elapsed time */
```

```
        param=AVSadd_parameter("Elapsed time (sec):","integer",0,0,INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to allow zero vectors to be skipped */
        param=AVSadd_parameter("Skip zero vectors", "boolean", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to permit autogeneration */
        param=AVSadd_parameter("Auto-classify image", "boolean", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to generate the nPDF plot */
        param=AVSadd_parameter("Classify image", "oneshot", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);

/* free the output data */
        AVSautofree_output(out_port);

/* set the function to create the output */
        AVSset_compute_proc(npdf_compute_class);
}

int npdf_compute_class(AVSfield_char *input_image,
                AVSfield_float *lut_image,
                AVSfield_float **output_image,
                char *head,
                char *title,
                char *corner_choice,
                int interface_scale,
                float *interface_percent_classified,
                int time,
                int skip_zero_vectors,
                int auto_run,
                int run)
{

/* local variables */
int *spectral_vector;          /* This is the pixel      */
CLASS_MAP_STRUCT class_map;    /* This is the class map  */
int dims[2];                   /* and its dimensions     */
register int row, col, band_index;
int range, num_rows, num_cols, num_bands, num_bits, corner, corner1;
int corner2, target_class, send_val, scale, magnitude, num_classified;
struct timeval before, after;
COORDINATE point;
float percent_done, percent_classified, row_step, stage;


        /* check if run time */
        if( run || auto_run )
                AVSmodify_parameter( "Classify image", AVS_VALUE, 0, 0, 0 );

        else {
                AVSmark_output_unchanged( "Output Image" );
                return( 1 );
        }

/* set up some variables that we'll need */
num_rows = MAXY(input_image);
row_step = 1.0/num_rows;
num_cols = MAXX(input_image);
num_bands = input_image->veclen;
num_bits = 8;
range = (int)pow(2.0,(double)num_bits) - 1;
scale = MAXX(lut_image);

/* update the interface */
AVSmodify_parameter("Elapsed time (sec):",
                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                    0, 0, 0);

AVSmodify_parameter("Scale factor:",
                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                    scale, scale, scale);

/* decipher the corner choice */
corner = AVSchoice_number("corner_choice",corner_choice);

switch(corner){
        case 1:
                corner1=1;
                corner2=2;
                break;
        case 2:
                corner1=1;
                corner2=3;
                break;
        case 3:
                corner1=1;
                corner2=4;
                break;
        case 4:
                corner1=2;
                corner2=3;
                break;
        case 5:
                corner1=2;
                corner2=4;
                break;
        case 6:
                corner1=3;
                corner2=4;
                break;
}

/* set up storage for the spectral vector and the class map */
spectral_vector = ivector(0,num_bands-1);

/* the class map is the same size as the input image */
class_map.array = imatrix(0,num_rows-1,0,num_cols-1);
class_map.num_rows = num_rows;
class_map.num_cols = num_cols;
```

```
#ifdef DEBUG
        printf("Immediately prior to entering the image processing loop\n");
#endif

/* Do the nPDF classification on the entire image */
        gettimeofday(&before,NULL);
        num_classified = 0;
        stage = 0.10;
        for (row=0; row < num_rows; row++) {
            for (col=0; col < num_cols; col++) {
                magnitude = 0;
                for(band_index=0; band_index < num_bands; band_index ++){
                        spectral_vector[band_index] =
                                *(I2DV(input_image, col, row)+band_index);
                        magnitude += spectral_vector[band_index];
                }

                if ( (skip_zero_vectors == 1) && (magnitude == 0) )
                        target_class = 0;
                else{
                        point = calc_distance(spectral_vector,corner1,corner2,
                                        range, num_bands,num_bits,scale);
                        target_class = I2D(lut_image, point.dist2, point.dist1);
                }

                if (target_class != 0)
                        num_classified += 1;

                class_map.array[row][col] = target_class;
            }

            /* report the level of completion */
            percent_done=(float)row/(float)num_rows;
            if((percent_done > stage) && (percent_done < stage+(2*row_step))){
                send_val = (int)(100.0 * stage);
                printf("percent complete %d\n",send_val);
                AVSmodule_status("percent complete",send_val);
                stage += .10;
            }
        }

        /* report elapsed time and percent classified */
        gettimeofday(&after,NULL);
        time=(int)(after.tv_sec-before.tv_sec);
        percent_classified = 100.0 * ( (float)num_classified /
                         (float)(num_rows * num_cols) );

        AVSmodify_parameter("Elapsed time (sec):",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        time, time, time);

        AVSmodify_float_parameter("Percent classified:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        percent_classified, percent_classified,
                        percent_classified);

#ifdef DEBUG
        printf("All pixels classified in %d (sec).\n",time);
#endif

        /* set up storage for the output image */
        dims[0] = num_cols;
        dims[1] = num_rows;
        if ( *output_image == NULL ) {
            *output_image = ( AVSfield_float * )
                ( AVSdata_alloc( "field 2D uniform scalar float", dims ));
        }
        else if ( ( dims[ 0 ] != MAXX( *output_image ) ) ||
                    ( dims[ 1 ] != MAXY( *output_image ) ) ) {
            AVSfield_free( (AVSfield *) *output_image );
            *output_image = ( AVSfield_float * )
                ( AVSdata_alloc( "field 2D uniform scalar float", dims ));
        }

#ifdef DEBUG
        printf("Field allocated\n" );
#endif
        /* copy the classification map to the output port */
        for (row=0; row < class_map.num_rows; row++) {
            for (col=0; col < class_map.num_cols; col++) {
                I2D(*output_image, col, row) = class_map.array[row][col];
            }
        }

#ifdef DEBUG
        printf("Field written\n" );
#endif
        /* free all the allocated memory */
        free_ivector(spectral_vector,0,num_bands-1);
        free_imatrix(class_map.array, 0, class_map.num_rows-1,
                    0, class_map.num_cols-1);

        /* return success */
        return(1);

} /* end of main */

/*****************************************************************************

function calc_distance
        This function calculates the two distance measurements from the user
        defined corners of the hypercube.
        INPUTS a num_bands dimensional pixel from the image to be transformed
        OUTPUTS the two nPDF coordinates in the coordinate structure named
        point
*****************************************************************************/

COORDINATE calc_distance(int *spectral_vector,
                         int corner1,
                         int corner2,
                         int range,
                         int num_bands,
                         int num_bits,
                         int scale)
{
        COORDINATE      point;
```

```
        int             i;
        double          temp1,temp2;
        double          dist1,dist2;


        /* this matrix holds the coefficients that "turn" the various vector
        components on or off to calculate the distances from the user
        selected corners of the hypercube.  Note that the a and b values are
        complements and that the column number is essentially one less than
        the user selected corner value.  The matrix coefficients are defined
        as follows:  */

        static int      a[6][4] = {{ 1, 1, 1, 1 },
                                   { 1, 1, 0, 0 },
                                   { 1, 0, 1, 0 },
                                   { 1, 1, 1, 1 },
                                   { 1, 1, 0, 0 },
                                   { 1, 0, 1, 0 }};

        temp1=0.0;
        temp2=0.0;

        for (i =0; i < num_bands; i++){
            temp1 += pow(( double )(spectral_vector[i]*a[i][corner1-1]),2.0)+
                     (pow(( double )(range-spectral_vector[i]),2.0)*
                     (1.0-a[i][corner1-1]));
            temp2 += pow(( double )(spectral_vector[i]*a[i][corner2-1]),2.0)+
                     (pow(( double )(range-spectral_vector[i]),2.0)*
                     (1.0-a[i][corner2-1]));
        }

        dist1 = sqrt(temp1);
        dist2 = sqrt(temp2);

        /* we have the two distance measures so now do the nPDF calculations */
        temp1 = ( double )scale * dist1 / (pow(2.0,( double )num_bits)*
                pow(( double)num_bands,0.5));
        point.dist1 = (int)(temp1+0.5);

        temp2 = (double)scale * dist2 / (pow(2.0, (double)num_bits)*
                pow((double)num_bands,0.5));
        point.dist2 = (int)(temp2+0.5);

#ifdef DIST_DEBUG /* Debug code - echo distance values to maintain sanity */
        printf("npdf dist1 = %d and npdf dist2 = %d\n",point.dist1, point.dist2);
#endif

        return(point);

} /* end of calc_distance */
```

Source code for the Make ARTMAP and Fuzzy ARTMAP Classification AVS modules and supporting ART and fuzzy set theory algorithms

```
/*=============================================================================

 A V S  F U Z Z Y  A R T M A P  (1 9 9 5)  Rochester Institute of Technology

  File Name: AVS_makenet.c

  ** Original fuzzy ARTMAP code written by Ah-Hwee Tan at Boston University **

  ** ported to ANSI C and the AVS environment by Steven W. Nessmiller **

  This software tool creates a fuzzy artmap neural network from a training
  image.  It then writes the network data to a disk file that can be utilized
  by the classification module.

  Revision history:
  29 March 1995 - ANSI C port
  7 April 1995 - AVS port
  12 May 1995 - implemented reading of training set via the IPI library

=============================================================================*/

#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <sys/file.h>
#include <sys/types.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/flow.h>
#include <avs/avs_data.h>
#include <IPIlib.h>
#include "/cis/src/local/avs/src/IPI/avs_types.h"
#include "struct.h"
#include "memutil.h"
#include "idef.h"
#include "ptypes.h"

#define DEBUG
/*
#define ART_DEBUG
#define PROCESS_DEBUG
#define INTERFACE_DEBUG
#define TRAINING_DEBUG
#define INPUT_DEBUG
*/

#define MAXSTR 80


/* function prototypes */
void remap_pixel(parameter_struct_ptr parameter, float *pixel, int flag,
                 float min, float max);
```



```
/* initialize the modules */
AVSinit_modules()
{
        int AVS_makenet();
        AVSmodule_from_desc(AVS_makenet);
}

/* interface routine */
AVS_makenet()
{
        int compute_makenet();
        int param;
        int in_port;
        char cwd[MAXSTR];

/* set the module name and type */
        AVSset_module_name("Make fuzzy ARTMAP", MODULE_FILTER);

/* create the input port for the training data */
        in_port = AVScreate_input_port("Input Training Set", Training_Field,
                                       REQUIRED );

/* set the title for the window */
        param=AVSadd_parameter("title","string","Fuzzy ARTMAP Parameters",
                               "","");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* create a widget for the parameter filename */
        getwd(cwd);
        if( cwd[strlen(cwd)-1] != '/')
                strcat(cwd,"/");
        param=AVSadd_parameter("Parameter file (*.par):","string",cwd,0,".par")
;
        AVSadd_parameter_prop(param, "width","integer",4);
        AVSconnect_widget(param, "browser");

/* create a widget for the network filename */
        param=AVSadd_parameter("Network file (*.net):","string",cwd,0,".net");
        AVSadd_parameter_prop(param, "width","integer",4);
        AVSconnect_widget(param, "browser");

/* allow user to view all file types */
        param = AVSadd_parameter("View all file types", "boolean",0,0,1);
        AVSadd_parameter_prop(param, "width", "integer",4);

/* create a widget to set min rho for ARTa */
        param=AVSadd_float_parameter("rho for ARTa", 0.90, 0.1, 1.0);
        AVSconnect_widget(param, "typein_real");

/* create a widget to set min rho for ARTb */
        param=AVSadd_float_parameter("rho for ARTb", 1.0, 0.1, 1.0);
        AVSconnect_widget(param, "typein_real");

/* create a widget to display the number of training patterns */
        param=AVSadd_parameter("Number of training patterns:","integer", 0, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
```

```
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the number of training classes */
        param=AVSadd_parameter("Number of training classes:","integer", 0, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the number of bands in the training sets */
        param=AVSadd_parameter("Number of bands:", "integer", 0, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the current learning iteration */
        param=AVSadd_parameter("Learning iteration:","integer",0,0,INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the maximum learning iteration */
        param=AVSadd_parameter("Maximum iteration:","integer", 10, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the elapsed time */
        param=AVSadd_parameter("Elapsed time (sec):","integer",0,0,INT_UNBOUND)
;
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the number of ARTa classification nodes */
        param=AVSadd_parameter("ARTa classification nodes:","integer", 0, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to update the parameter file */
        param = AVSadd_parameter("Update parameter file", "boolean",0,0,1);
        AVSadd_parameter_prop(param, "width", "integer",4);

/* create a recasting widget */
        param=AVSadd_parameter("Recast inconsistent cases","boolean",0,0,0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create autogeneration widget */
        param=AVSadd_parameter("Auto-generate network","boolean",0,0,0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to generate the output file */
        param=AVSadd_parameter("Generate network", "oneshot", 0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);

/* set the function to create the output */
        AVSset_compute_proc(compute_makenet);
}
```

```
/*========================================================================

M A I N

========================================================================*/
int compute_makenet(AVSfield *input,
                    char *title,
                    char *param_filename,
                    char *network_filename,
                    int all,
                    float *interface_Arho,
                    float *interface_Brho,
                    int interface_num_patterns,
                    int interface_num_classes,
                    int interface_num_bands,
                    int learn_iter,
                    int interface_max_iterations,
                    int time,
                    int num_nodes,
                    int interface_update,
                    int interface_recast,
                    int auto_run,
                    int run)
{

/* local variables */
int status, comp_flag, A_winner, B_winner, class_val;
int prediction, times_through_training, result;
float min, max, temp;
register int class, row, band, index, j;
char *answer;
struct timeval before, after;
parameter_struct_ptr parameter;
network_data_ptr network;
netstat_ptr net_status;
data_struct_ptr data;
series_data_ptr series;
double *train_vector;
int list_row, training_bands, send_val, elapsed_time;
int *length;
float percent_done, row_step, stage;
FILE *input_param_file;
FILE *output_file;
IPItraining trainchain = NULL;
IPItraining trainset = NULL;
static char old_param_filename[80];
static char old_network_filename[80];
char *reset_param_filename = " ";
char *reset_network_filename = " ";
static int first_time_through = 1;
static int goodpar = 0;
static int goodnet = 0;

        /* check for view all parameter filenames */
        if(AVSparameter_changed("View all file types")){
                if (all){
```

```
                    AVSmodify_parameter("Parameter file (*.par):",
                        AVS_VALUE|AVS_MAXVAL, "/tmp/",0,"");
                    AVSmodify_parameter("Network file (*.net):",
                        AVS_VALUE|AVS_MAXVAL, "/tmp/",0,"");
                    printf( "Setting maxval to null\n");
                    AVSmodify_parameter("Parameter file (*.par):",
                        AVS_VALUE, param_filename, 0, 0);
                    AVSmodify_parameter("Network file (*.net):",
                        AVS_VALUE, network_filename, 0, 0);
                }
                else{
                    AVSmodify_parameter("Parameter file (*.par):",
                        AVS_VALUE|AVS_MAXVAL, "/tmp/", 0, ".par");
                    AVSmodify_parameter("Network file (*.net):",
                        AVS_VALUE|AVS_MAXVAL, "/tmp/", 0, ".net");
                    printf( "Setting maxval to .par and .net\n");
                    AVSmodify_parameter("Parameter file (*.par):",
                                        AVS_VALUE, param_filename, 0, 0);
                    AVSmodify_parameter("Network file (*.net):",
                                        AVS_VALUE, network_filename, 0, 0)
;
                }
        }

        if(first_time_through){
            printf("Allocating the parameter file.\n");
            parameter=(parameter_struct_ptr)malloc(sizeof(PARAMETER_STRUCTURE));
            if (parameter ==NULL){
                printf("ERROR: unable to allocate the parameter structure.\n");

                return(1);
            }

            first_time_through = 0;

        } /* end if */

        /* see if the parameter file has changed */
        status = strcmp(param_filename, old_param_filename);

        if(status != 0) {
            strcpy(old_param_filename, param_filename);
            input_param_file = fopen(param_filename, "r+");

            if(input_param_file == NULL){
                printf("AVS Make fuzzy ARTMAP needs a valid parameter file.\n")
;
                return(1);
            }

            fclose(input_param_file);
            goodpar = read_parameters(param_filename, parameter);

            if(goodpar){
                /* display the parameter file to stdout */
                status = display_parameters(parameter);
            }
            else{
                printf("ERROR: bad status returned by read_parameters.\n");
                AVSmessage("beta", AVS_Warning,NULL,"Make ARTMAP",
                           "Exit", "Paramater file %s is not a valid file.\n",
                            param_filename);
                return(1);
            }
        }


        /* check for updated rho value from interface */
/*
        if( AVSparameter_changed("rho for ARTa") ){

                temp = *interface_Arho * 100.0;
                send_val = ( (int)(temp+0.5) )/100.0;
                printf("Rho for ARTA reset to %f\n", send_val);
                AVSmodify_float_parameter("rho for ARTa", AVS_VALUE,
                                          send_val, 0, 0);
        }
*/

        /* see if the network file has changed */
        status = strcmp(network_filename, old_network_filename);

        if(status != 0) {
            strcpy(old_network_filename, network_filename);
            output_file = fopen(network_filename, "r+");

            if(output_file == NULL){
                printf("Network file does not exist.\n");
                goodnet = 1;
            }
            else{
                /* file already exists, close file and check for overwrite */
                fclose(output_file);
                printf("Network file already exists.\n");
                answer = AVSmessage("beta",AVS_Warning,NULL,"Make ARTMAP",
                        "Overwrite!Cancel",
                        "Network file %s already exists.\n",network_filename);
                if(strcmp(answer,"Overwrite") != 0){
                        return(1);
                } /* end if */

                goodnet = 1;

            } /* end else */
        }

#ifdef INTERFACE_DEBUG
        printf("The selected network filename is: %s\n",network_filename);
        printf("The selected parameter filename is: %s\n", param_filename);
        printf("The old network filename is: %s\n", old_network_filename);
        printf("The old parameter filename is: %s\n", old_param_filename);
        printf("run = %d auto_run = %d goodnet = %d goodpar = %d.\n",
                run, auto_run, goodnet, goodpar);
#endif

if( (run || auto_run) && (goodnet && goodpar) )
```

```
        AVSmodify_parameter("Generate network",AVS_VALUE,0,0,0);
else{
        return(1);
}

printf("Make ARTMAP just entered the compute function.\n");

/* reset the static variables */
goodpar = 0;
goodnet = 0;
strcpy(old_network_filename, reset_network_filename);
strcpy(old_param_filename, reset_param_filename);
first_time_through = 1;

/* clear out parameters from last run */
AVSmodify_parameter("Number of training patterns:",
                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                    0,0,0);
AVSmodify_parameter("Number of training classes:",
                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                    0,0,0);
AVSmodify_parameter("Number of bands:",
                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                    0,0,0);
AVSmodify_parameter("Learning iteration:",
                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                    0,0,0);
AVSmodify_parameter("Elapsed time (sec):",
                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                    0,0,0);
AVSmodify_parameter("ARTa classification nodes:",
                    AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                    0,0,0);

/* allocate other data structures */
network=(network_data_ptr)malloc(sizeof(NETWORK_DATA_STRUCT));
if(network==NULL){
        printf("ERROR: unable to allocate the network.\n");
        return(1);
}

series = (series_data_ptr)malloc(sizeof(SERIES_DATA_STRUCT));
if (series==NULL){
        printf("Unable to malloc series structure pointer.\n");
        return(1);
}

net_status = (netstat_ptr)malloc(sizeof(STATUS_STRUCT));
if (net_status == NULL){
        printf("Unable to malloc status structure pointer.\n");
        return(1);
}

data = (data_struct_ptr)malloc(sizeof(DATA_STRUCT));
if (data == NULL){
        printf("Unable to malloc data structure pointer.\n");
        return(1);
}
```

```
#ifdef DEBUG
        printf("All structures allocated.\n");
#endif

/* copy the training data to the pixel list structure */

        /* Convert the training field */
        IPIfield_to_training(input,NULL,&trainchain,0);

#ifdef DEBUG
        printf("Training field converted.\n");
#endif

        /* get the number of training classes and update the interface */
        data->num_classes = IPIlength_of_training_chain( trainchain );

#ifdef DEBUG
        printf("There are %d classes of data.\n", data->num_classes);
#endif

        /* get number of bands in the training sets and update the interface */

        IPIget_training_bands(trainchain, &training_bands);
        data->num_bands = training_bands;

#ifdef DEBUG
        printf("There are %d bands in the training sets.\n",
                data->num_bands);
#endif

        /* Get all the training set information */
        data->num_pixels = 0;
        length = ivector(0, data->num_classes);
        trainset = NULL;
        IPInext_on_training_chain( &trainchain, &trainset );
        printf("Just executed first next_on_training_chain.\n");
        class = 0;
        while ( trainset != NULL ) {
            /* get the pixel list parameters and allocate memory */
            length[ class ] = IPIsize_of_training_set(trainset);
            data->num_pixels += length[ class ];

            IPInext_on_training_chain( &trainchain, &trainset );
            printf("Executed next_on_training_chain again.\n");
            class += 1;
        }

#ifdef TRAINING_DEBUG
        printf("Number of pixels per class has been calculated.\n");
        printf("Total number of pixels = %d.\n",data->num_pixels);
        for(class = 0; class < data->num_classes; class++){
                printf("The number of pixels in class %d is %d.\n",
                        class, length[class]);
        }
#endif
        data->dpt = matrix(0, data->num_pixels-1, 0, data->num_bands-1);
        data->label = ivector(0, data->num_pixels-1);
```

```
        data->frequency = ivector(0, data->num_classes+1);


/* read the training data into the data structure */
        list_row = 0;
        train_vector = dvector(0,data->num_bands-1);

        trainset = NULL;
        for(class = 0; class < data->num_classes; class++){
            IPInext_on_training_chain( &trainchain, &trainset );

#ifdef TRAINING_DEBUG
        printf("Reading in class %d.\n", class);
#endif

            for(row = list_row; row < list_row + length[class]; row++){
                IPIget_training_vector(trainset, row - list_row, train_vector);


                for(band = 0; band < data->num_bands; band++){
                    data->dpt[row][band] = (float)train_vector[band];
                } /* end band loop */

                data->label[row] = class;

#ifdef TRAINING_DEBUG       /* print out the training pixels */
        for(band = 0; band <= data->num_bands; band++){
                printf("%d\t",data->array[row][band]);
        }
        printf("\n");
#endif

                } /* end of row loop */

            list_row += length[class];
            printf("Class %d successfully read in.\n",class);

        } /* end of class loop */

#ifdef TRAINING_DEBUG
        printf("Training data successfully read in.\n");
#endif

        free_dvector(train_vector,0,data->num_bands-1);

        /* reset any variables that may have changed */
        printf("Rho for ARTa from the interface = %f\n",*interface_Arho);
        printf("Rho for ARTb from the interface = %f\n",*interface_Brho);
        parameter->min_arho = *interface_Arho;
        parameter->a_length = data->num_bands;
        parameter->b_length = data->num_classes;
        parameter->train_pats = data->num_pixels;
        parameter->max_iterations = interface_max_iterations;
        network->num_att[ARTa] = parameter->a_length*(1+parameter->complement);
        network->num_att[ARTb] = parameter->b_length*(1+parameter->complement);

        network->num_category[ARTa] = 0;
        network->num_category[ARTb] = 0;
        network->rho[ARTa] = parameter->min_arho;
        network->rho[ARTb] = parameter->brho;
        data->num_pixels = parameter->train_pats;
        comp_flag = parameter->complement;

        /* update interface with number of training pattern information */
        interface_num_patterns = data->num_pixels;
        AVSmodify_parameter("Number of training patterns:",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            interface_num_patterns, interface_num_patterns,
                            interface_num_patterns);

        interface_num_classes = data->num_classes;
        AVSmodify_parameter("Number of training classes:",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            interface_num_classes, interface_num_classes,
                            interface_num_classes);

        interface_num_bands = data->num_bands;
        AVSmodify_parameter("Number of bands:",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            interface_num_bands, interface_num_bands,
                            interface_num_bands);

        interface_max_iterations = parameter->max_iterations;
        AVSmodify_parameter("Maximum iteration:",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            interface_max_iterations, interface_max_iterations,

                            interface_max_iterations);

#ifdef DEBUG
        printf("Interface updated.\n");
        printf("rho for ARTa = %f rho for ARTb = %f\n",network->rho[ARTa],
                network->rho[ARTb]);
        printf("Alength = %d Blength = %d.\n", parameter->a_length,
                parameter->b_length);
        printf("Num atributes ARTa = %d Num attributes ARTb = %d.\n",
                network->num_att[ARTa], network->num_att[ARTb]);
        printf("Complement coding = %d.\n", comp_flag);
#endif


#ifdef DEBUG
        printf("Internal ARTMAP memory allocated.\n");
        printf("Number of pixels = %d.\n", data->num_pixels);
        printf("Number of bands = %d.\n", data->num_bands);
        printf("Internal data pointer = %x.\n", data->dpt[0][0]);
        printf("Internal label pointer = %x.\n", data->label[0]);
#endif


#ifdef INPUT_DEBUG
        /* echo the data values and label to maintain sanity */
        for(row = 0; row < data->num_pixels; row++){
                for (band = 0; band < parameter->a_length; band++)
                        printf("%1.0f   ",data->dpt[row][band]);
                printf("%d\n",data->label[row]);
```

```
        }
#endif

        /* allocate the network memory */
        status = alloc_memory(network);
        if(status!=1){
                printf("ERROR: incorrect status returned by alloc_memory.\n");
                return(1);
        }
        else
                printf("Memory allocated for the network.\n");

        net_status->network_loaded = TRUE;
        init_ARTMAP(network);
        printf("ARTMAP initialized.\n");
        init_ARTs(network, parameter);
        printf("init_ARTs just called.\n");
        init_ARTb(network, parameter);
        printf("init ARTb just called.\n");

#ifdef DEBUG
        printf("The fuzzy ARTMAP network has been initialized.\n");
#endif


        /* check for and recast inconsistent data */
        if(interface_recast){
                recast_inconsistent_cases(BY_MAJORITY, data, parameter);
        }

        net_status->got_all_train_right = FALSE;
        net_status->done_training = FALSE;
        times_through_training = 0;

        /* step through the training data and train the net */
        gettimeofday(&before, NULL);

#ifdef DEBUG
        printf("Make ARTMAP just entered the training loop.\n");
#endif

while(net_status->done_training == FALSE){
        series->num_wrong = 0;
        series->num_right = 0;
        series->num_perfect_mismatch = 0;
        series->num_no_prediction = 0;
        net_status->done_training = TRUE;
        net_status->got_all_train_right = FALSE;

#ifdef DEBUG
                printf("Learning iteration %d.\n",times_through_training);
#endif

stage = 0.10;
row_step = 1.0 / data->num_pixels;
/* load the pixel into the network field */
for(row = 0; row < data->num_pixels; row++){
    for(band = 0; band < data->num_bands; band ++)
```

```
            network->pattern[ARTa][band] = data->dpt[row][band];

        /* remap and complement the pixel */
        min = 0.0;
        max = 255.0;
        remap_pixel(parameter, network->pattern[ARTa], comp_flag, min,
                    max);

        /* encode the class */
        class_val = data->label[row];
        for(index=0; index < parameter->b_length; index ++){
            network->pattern[ARTb][index] = 0.0;
            if(comp_flag)
                network->pattern[ARTb][index+(parameter->b_length)]=1.0;
        }
        network->pattern[ARTb][class_val] = 1.0;
        if(comp_flag)
            network->pattern[ARTb][class_val+parameter->b_length]=0.0;

#ifdef PROCESS_DEBUG
        /* echo the values to maintain sanity */
        printf("\nLearning iteration %d operating on pattern %d\n",
                        times_through_training+1,row+1);
        printf("\nThe input to ARTa is :");
        for(band = 0; band < network->num_att[ARTa]; band++)
                printf("%1.3f    ",network->pattern[ARTa][band]);
        printf("\n");

        printf("\nThe input to ARTb is :");
        for(band = 0; band < network->num_att[ARTb]; band++)
                printf("%1.0f    ",network->pattern[ARTb][band]);
        printf("\n");
#endif

        /* refresh the system */
        refresh_system(network, parameter, net_status);

#ifdef ART_DEBUG
        printf("\nDoing ART in the B module.\n");
#endif
        /* do ART in b */
        B_winner = ART(network, parameter, ARTb, TRAIN);
        learn(network,parameter,ARTb,B_winner);

/* this is the ART loop */
        do{

        /* initialize loop control variables */
        net_status->perfect_mismatch = FALSE;

#ifdef ART_DEBUG
        printf("\nDoing ART in the A module.\n");
#endif
                /* do ART in a */
                A_winner = ART(network, parameter, ARTa, TRAIN);

#ifdef ART_DEBUG
        printf("ARTa winner = %d ARTb winner = %d.\n",A_winner, B_winner);
```

```
#endif

        /* see if this node makes a prediction */
        if(network->makes_prediction[A_winner]){
                net_status->predict = TRUE;
                net_status->no_prediction = FALSE;

                /* determine the prediction */
                max = -1.0;
                for(j=0; j < network->num_category[ARTb]+1; j++)
                if(network->IAMI[A_winner][j] > max){
                        max = network->IAMI[A_winner][j];
                        prediction = j;
                }

                if(prediction == B_winner){
#ifdef ART_DEBUG
        printf("expectation confirmed\n");
#endif
                        /* expectation confirmed */
                        net_status->reset = FALSE;
                        net_status->right = TRUE;
                        net_status->wrong = FALSE;
                        learn(network,parameter, ARTa, A_winner);
                        change_confidence(network,parameter,A_winner, 0,TRAIN);
                        }

                else{
#ifdef ART_DEBUG
        printf("expectation mismatch\n");
#endif
                        /* expectation mismatch */
                        net_status->wrong = TRUE;
                        net_status->right = FALSE;
                        net_status->reset = TRUE;
                        net_status->perfect_mismatch = FALSE;
                        IA_reset(network,parameter,net_status,data, series,
                                A_winner,TRAIN,row);
#ifdef ART_DEBUG
        printf("IA_reset just called.\n");
#endif
                        change_confidence(network,parameter,A_winner, 1,TRAIN);
#ifdef ART_DEBUG
        printf("Changed confidence of node %d.\n", A_winner);
#endif
                }

        } /* end if makes prediction */

        else{

#ifdef ART_DEBUG
        printf("no prediction\n");
#endif
                /* no prediction was made */
                net_status->wrong = TRUE;
                net_status->predict = FALSE;
                net_status->no_prediction = TRUE;
                net_status->right = FALSE;
                net_status->reset = FALSE;

                /* learn new patterns */
                learn(network, parameter, ARTa, A_winner);
                learn(network, parameter, ARTb, B_winner);
                learn_prediction(network, A_winner, B_winner);
        }

}while(net_status->predict && net_status->reset &&
                !net_status->perfect_mismatch);

        /* record series performance */
        record_train_performance(net_status, series);

        /* report the level of completion */
        percent_done = (float) row/(float)data->num_pixels;
        if( (percent_done > stage) && (percent_done < stage + (2 * row_step)) ){
                send_val = (int)(100.0 * stage);
                AVSmodule_status("percent complete", send_val);
                printf("percent complete %d.\n", send_val);
                stage += 0.10;
        }

}       /* end of row loop */

        /* update the interface */
        times_through_training += 1;
        AVSmodify_parameter("Learning iteration:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        times_through_training, times_through_training,
                        times_through_training);

        AVSmodify_parameter("ARTa classification nodes:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        network->num_category[ARTa],
                        network->num_category[ARTa],
                        network->num_category[ARTa]);

        /* display training statistics */
        printf("Theses are the training stats.\n");
        report_train_performance(network,parameter,series);

        /* see if we're done */
        if( (times_through_training < parameter->max_iterations) &&
        (series->num_right + series->num_perfect_mismatch != data->num_pixels) )
                net_status->done_training = FALSE;

#ifdef DEBUG
        printf("Training iteration = %d.\n",times_through_training);
        printf("Maximum iterations = %d.\n", parameter->max_iterations);
        printf("Number correct predictions = %d.\n", series->num_right);
        printf("Number of data patterns = %d.\n",data->num_pixels);
        printf("Done training = %d.\n", net_status->done_training);
#endif

}       /* end of while loop */
```

```
        /* update the elapsed time */
        gettimeofday(&after, NULL);
        elapsed_time = (int)(after.tv_sec - before.tv_sec);
        printf("The network was trained in %d seconds.\n",elapsed_time);
        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            elapsed_time, elapsed_time, elapsed_time);

#ifdef DEBUG
        /* display network information to the screen */
        printf("Generated network information:\n");
        display_net(network);
#endif


        /* write the network to a disk file */
        write_net_binary(network, network_filename);
        printf("Network file successfully written to disk.\n");

        /* update the parameter file */
        if(interface_update)
                write_param_file(parameter,param_filename);

        /* free all allocated memory */
        status = free_memory(network);
        if(status != 1){
                printf("Bad status returned from free network memory.\n");
                return(1);
        }

        free_matrix(data->dpt, 0, data->num_pixels-1, 0, data->num_bands-1);
        free_ivector(data->label, 0, data->num_pixels-1);
        free_ivector(data->frequency, 0, data->num_classes+1);
        free(parameter);
        free(network);
        free(series);
        free(data);
        free(net_status);

        /* return success */
        return(1);

} /* end of main */


/*=============================================================================
function remap_pixel

        this function remaps the incoming int pixel to a float pixel varying
        between 0 and 1.  Complement coding is automatically supported if flag
        is set to 1.  Values min and max represent the minimal and maximal
        values before the reamp.

=============================================================================*/
void remap_pixel(parameter_struct_ptr parameter, float *pixel, int flag,
                 float min, float max)
{
float value, remap_value;
register int index;


        for(index=0; index<parameter->a_length; index++){
                value = pixel[index];
                remap_value = (value - min) / (max-min);
                pixel[index] = remap_value;

                if (flag==1) /* complement coding, so calculate complements */
                        pixel[index + parameter->a_length] = 1.0-remap_value;

        }

}/* end of remap_pixel */
```

```
/*=============================================================================

 A V S   F U Z Z Y   A R T M A P  (1 9 9 5)   Rochester Institute of Technology


  File Name: main.c

  ** Original code written by Ah-Hwee Tan  **
  ** ported to ANSI C and the AVS environment by Steven W. Nessmiller **
  This software tool reads in an existing fuzzy ARTMAP neural network
  definition file and performs image classification.

Revision history:
        ANSI C port 15 Feb 1995
        AVS port 19 May 1995
        1 July 95
                support added to skip zero vectors

=============================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <sys/file.h>
#include <sys/types.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/flow.h>
#include <avs/avs_data.h>
#include "memutil.h"
#include "struct.h"
#include "idef.h"
#include "ptypes.h"

/* layered debug definitions */
#define DEBUG
#define LOAD_DEBUG
/*
#define PROCESS_DEBUG
*/

#define MAXSTR 80

/* this structure supports the input image */
typedef struct{
        int num_bands;
        int num_rows;
        int num_cols;
}MSI_IMG;

typedef MSI_IMG *msi_img_ptr;

/* this structure supports the classification map */
typedef struct{
        int **array;
        int num_rows;
        int num_cols;
}CLASS_MAP;

typedef CLASS_MAP *class_map_ptr;

/* function prototypes */
void remap_pixel(float *pixel, int length, int flag, float min, float max);

/* initialize the modules */
AVSinit_modules ()
{
        int AVS_fuzzy_ARTMAP ();
        AVSmodule_from_desc (AVS_fuzzy_ARTMAP);
}

/* interface routine */
AVS_fuzzy_ARTMAP ()
{
        int param;
        int in_port, out_port;
        char cwd[MAXSTR];
        int compute_fuzzy_ARTMAP();

/* set the module name and type */
        AVSset_module_name("Fuzzy ARTMAP", MODULE_FILTER);

/* create the input port for the input image */
        in_port = AVScreate_input_port("Input Image","field 2D uniform byte",
                                        REQUIRED );

/* create the output port for the classification map */
        out_port = AVScreate_output_port("Classified Image",
                                        "field 2D uniform float", REQUIRED );

/* set the title for the window */
        param=AVSadd_parameter("title","string","Fuzzy ARTMAP Parameters",
                                "","");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* create a widget for the parameter filename */
        getwd(cwd);
        if( cwd[strlen(cwd)-1] != '/')
                strcat(cwd,"/");
        param=AVSadd_parameter("Parameter file (*.par):","string",cwd,0,".par")
;
        AVSadd_parameter_prop(param, "width","integer",4);
        AVSconnect_widget(param, "browser");

/* create a widget for the network filename */
        param=AVSadd_parameter("Network file (*.net):","string",cwd,0,".net");
        AVSadd_parameter_prop(param, "width","integer",4);
        AVSconnect_widget(param, "browser");

/* allow user to view all file types */
```

```
        param = AVSadd_parameter("View all file types", "boolean",0,0,1);
        AVSadd_parameter_prop(param, "width", "integer",4);

/* create a widget to set min rho for ARTa */
        param=AVSadd_float_parameter("rho for ARTa:", 0.90, 0.1, 1.0);
        AVSconnect_widget(param, "typein_real");

/* create a widget to display the number of ARTa classification nodes */
        param=AVSadd_parameter("ARTa classification nodes:","integer", 0, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the number of target classes */
        param=AVSadd_parameter("Number of training classes:","integer", 0, 0,
                               INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create a widget to display the percentage of classified pixels */
        param=AVSadd_float_parameter("Percent classified:", 0.0, 0.0, 100.0);
        AVSconnect_widget(param, "typein_real");

/* create a widget to display the elapsed time */
        param=AVSadd_parameter("Elapsed time (sec):","integer",0,0,INT_UNBOUND);
        AVSadd_parameter_prop(param,"width","integer",2);
        AVSconnect_widget(param,"typein_integer");

/* create autogeneration widget */
        param=AVSadd_parameter("Skip zero vectors","boolean",
                               0,0,0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create autogeneration widget */
        param=AVSadd_parameter("Auto-generate classification map","boolean",
                               0,0,0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to generate the classification map */
        param=AVSadd_parameter("Generate classification map", "oneshot",
                               0, 0, 0);
        AVSadd_parameter_prop(param, "width", "integer", 4);

/* free the output data */
        AVSautofree_output(out_port);

/* set the function to create the output */
        AVSset_compute_proc(compute_fuzzy_ARTMAP);
}

/*============================================================================
        main compute function
============================================================================*/

compute_fuzzy_ARTMAP(AVSfield_char *input_image,
                     AVSfield_float **output_image,
                     char *title,
                     char *param_filename,
                     char *network_filename,
                     int all,
                     float *interface_Arho,
                     int interface_num_nodes,
                     int interface_num_classes,
                     float *interface_percent_classified,
                     int time,
                     int interface_skip_zeros,
                     int auto_run,
                     int run)
{

/* local variables */
int status, flag, A_winner, num_classified, magnitude;
float min, max, percent_classified;
register int row, col, band;
int send_val, send_num_nodes, send_num_classes, elapsed_time, dims[2];
struct timeval before, after;
float send_Arho, percent_done, row_step, stage;
msi_img_ptr input;
class_map_ptr classified;
parameter_struct_ptr parameter;
network_data_ptr network;
FILE *input_param_file;
FILE *input_network_file;
static char old_param_filename[80];
static char old_network_filename[80];
char *reset_param_filename = " ";
char *reset_network_filename = " ";
static int first_time_through = 1;
static int goodpar = 0;
static int goodnet = 0;

        /* check for view all parameter filenames */
        if(AVSparameter_changed("View all file types")){
                if (all){
                        AVSmodify_parameter("Parameter file (*.par):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/",0,"");
                        AVSmodify_parameter("Network file (*.net):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/",0,"");
                        printf( "Setting maxval to null\n");
                        AVSmodify_parameter("Parameter file (*.par):",
                             AVS_VALUE, param_filename, 0, 0);
                        AVSmodify_parameter("Network file (*.net):",
                             AVS_VALUE, network_filename, 0, 0);
                }
                else{
                        AVSmodify_parameter("Parameter file (*.par):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/", 0, ".sta");
                        printf( "Setting maxval to .sta\n");
                        AVSmodify_parameter("Parameter file (*.par):",
                                             AVS_VALUE, param_filename, 0, 0);
                        AVSmodify_parameter("Network file (*.net):",
                                             AVS_VALUE, network_filename, 0, 0)
;
                }
```

```
    }

    if(first_time_through){
        /* allocate the parameter and network structure pointers */
        printf("Allocating the parameter structure pointer.\n");
        parameter=(parameter_struct_ptr)malloc(sizeof(PARAMETER_STRUCTURE))
;

        if (parameter == NULL){
            printf("ERROR: unable to allocate the parameter structure.\n");

            return(1);
        }

        printf("Allocating the network structure pointer.\n");
        network=(network_data_ptr)malloc(sizeof(NETWORK_DATA_STRUCT));
        network->IAMI = NULL;

        if(network == NULL){
            printf("ERROR: unable to allocate the network structure.\n");
            return(1);
        }

        goodpar = 0;
        goodnet = 0;
        first_time_through = 0;

    } /* end if first time through */

    /* see if the parameter file has changed */
    status = strcmp(param_filename, old_param_filename);

    if(status != 0) {
        strcpy(old_param_filename, param_filename);

        /* see if it's a file or a directory */
        if  (param_filename[strlen(param_filename) - 1] == '/') {
            goodpar = 0;
            return(1);
        }

        input_param_file = fopen(param_filename, "r+");

        if(input_param_file == NULL){
            printf("Fuzzy ARTMAP needs a valid parameter file.\n");
            printf("Use the Make ARTMAP module to create one.\n");
            return(1);
        }

        fclose(input_param_file);
        goodpar = read_parameters(param_filename, parameter);

        if(goodpar){
            /* display the parameter file to stdout */
            status = display_parameters(parameter);
        }
        else{
            printf("ERROR: bad status returned by read_parameters.\n");
            AVSmessage("beta", AVS_Warning, NULL, "Make ARTMAP",
                       "Okay", "Paramater file %s is not valid.\n",
                        param_filename);
            goodpar = 0;
            return(1);
        }
    }

    /* see if the network file has changed */
    status = strcmp(network_filename, old_network_filename);

    if(status != 0 && goodpar) {
        strcpy(old_network_filename, network_filename);

        /* see if it's a file or a directory */
        if  (network_filename[strlen(network_filename) - 1] == '/') {
            goodnet = 0;
            return(1);
        }

        if(network->IAMI != NULL){
            free_memory(network);
        }

        input_network_file = fopen(network_filename, "r+");

        if(input_network_file == NULL){
            goodnet = 0;
            return(1);
        }
        else{
            /* read in the network from disk and allocate memory */
            network->num_att[ARTa] =
                        parameter->a_length*(1+parameter->complement);
            network->num_att[ARTb] =
                        parameter->b_length*(1+parameter->complement);
            network->rho[ARTa] = parameter->min_arho;
            network->rho[ARTb] = parameter->brho;
            status = load_net(network_filename, network);

            if(status==1){
                    printf("Network file sucessfully loaded.\n");
                    display_net(network);

                    /* update the interface */
                    send_Arho = parameter->min_arho;
                    AVSmodify_float_parameter("rho for ARTa:",
                         AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                         send_Arho, send_Arho, send_Arho);

                    send_num_classes = network->num_category[ARTb];
                    AVSmodify_parameter("Number of training classes:",
                         AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                         send_num_classes, send_num_classes,
                         send_num_classes);


                    send_num_nodes = network-> num_category[ARTa];
```

```
                    AVSmodify_parameter("ARTa classification nodes:",
                        AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                        send_num_nodes, send_num_nodes,
                        send_num_nodes);
                    goodnet = 1;
                }
                else{
                    printf("ERROR: incorrect status from load_net\n");
                    AVSmessage("beta", AVS_Warning,NULL,"Fuzzy ARTMAP",
                            "Okay", "Could not open network file: %s.\n",
                            network_filename);
                    goodnet = 0;
                    return(1);
                }

            } /* end else */
        }

#ifdef DEBUG
        if (goodpar)
            printf("The selected parameter filename is:%s\n", param_filename);
        if (goodnet)
            printf("The selected network filename is:%s\n", network_filename);
#endif

/* check if run time */
        if( (run || auto_run) && (goodnet && goodpar) )
                AVSmodify_parameter("Generate classification map", AVS_VALUE,
                                    0, 0, 0 );

        else {
                AVSmark_output_unchanged("Classified Image");
                return( 1 );
        }

        /* reset the static variables */
        goodpar = 0;
        goodnet = 0;
        strcpy(old_network_filename, reset_network_filename);
        strcpy(old_param_filename, reset_param_filename);
        first_time_through = 1;

        /* reset parameters from earlier runs */
        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            0,0,0);

        AVSmodify_float_parameter("Percent classified:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  0.0, 0.0, 0.0);

        /* allocate memory for the structures */
        input = (msi_img_ptr)malloc(sizeof(MSI_IMG));
        if (input == NULL){
                printf("Unable to malloc input image pointer\n");
                return(1);
        }
```

```
        classified = (class_map_ptr)malloc(sizeof(CLASS_MAP));
        if (classified == NULL){
                printf("Unable to malloc classification map pointer\n");
                return(1);
        }

        /* set up storage for the resulting class map */
        input->num_rows = MAXY(input_image);
        input->num_cols = MAXX(input_image);
        input->num_bands = input_image->veclen;
        classified->num_rows = input->num_rows;
        classified->num_cols = input->num_cols;
        classified->array = imatrix(0,classified->num_rows-1,
                                    0,classified->num_cols-1);

        /* process the image */
#ifdef DEBUG
        printf("Processing the image.\n");
#endif
        flag = parameter->complement;       /* see if we are complementing */
        min = 0.0;                          /* values for remap */
        max = 255.0;
        gettimeofday(&before, NULL);
        num_classified = 0;
        stage = 0.10;
        row_step = 1.0 / input->num_rows;
        for(row = 0; row < input->num_rows; row++){
                for(col=0; col < input->num_cols; col++){
                        magnitude = 0;
                        /*load pixel into pattern */
                        for (band=0; band<network->num_att[ARTa]/2; band++){
                                network->pattern[ARTa][band] =
                                  (float)( *(I2DV(input_image, col, row)+band) )
;
                                magnitude += network->pattern[ARTa][band];
                        }

        if ( (interface_skip_zeros) && (magnitude == 0) ) {
                classified->array[row][col] = 0;
        }
        else {
                        /* remap the pixel to between 0 and 1 */
                        remap_pixel(network->pattern[ARTa],
                                network->num_att[ARTa],flag,min,max);
                        /* refresh the F2 field */
                        refresh_F2(network);
                        /* do ART and find the winning node */
                        A_winner = ART(network, parameter, ARTa, TEST);
                        /* map the winning node to a final classification */
                        if (A_winner == network->num_category[ARTa]){
                                /* found no good winner */
                                classified->array[row][col] = 0;
                        }
                        else{
                                classified->array[row][col]=
                                        network->target_class[A_winner]+1;
                                num_classified += 1;
                        }
```

```
#ifdef PROCESS_DEBUG
        printf("Given input pixel[%d][%d]:\n", row, col);
        for(band=0; band < network->num_att[ARTa]/2; band++)
                printf("%d  ",(int)(*I2DV(input_image, col, row)+band));
        printf("\n");
        printf("The normalized and complemented pattern is:\n");
        for(band=0; band < network->num_att[ARTa]; band++)
                printf("%1.3f  ",network->pattern[ARTa][band]);
        printf("\nARTa's node #%d won.\n",A_winner);
        printf("This node has the following weight vector:\n");
        for(band=0; band<network->num_att[ARTa]; band ++)
                printf("%1.3f  ",network->dn[ARTa][A_winner][band]);
        printf("\nThis node maps to class #%d\n",classified->array[row][col]);
        printf("\n\n");
#endif
        }       /* end else */

        } /* end of col loop */

/* report level of completion */
percent_done = (float)row / (float)input->num_rows;
if( (percent_done > stage) && (percent_done < stage+(2*row_step) ) ){
        send_val = (int)(100.0 * stage);
        printf("percent complete %d\n",send_val);
        AVSmodule_status("percent complete", send_val);
        stage += 0.10;
}

        } /* end of row loop */

        /* update elapsed time and percent classified */
        gettimeofday(&after, NULL);
        elapsed_time = (int)(after.tv_sec - before.tv_sec);
        percent_classified = 100.0 * ( (float)num_classified /
                             (float)(input->num_rows * input->num_cols) );
#ifdef DEBUG
        printf("All pixels processed in %d seconds.\n",elapsed_time);
#endif
        AVSmodify_parameter("Elapsed time (sec):",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            elapsed_time, elapsed_time, elapsed_time);

        AVSmodify_float_parameter("Percent classified:",
                            AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                            percent_classified, percent_classified,
                            percent_classified);

        /* create the output image field */
#ifdef DEBUG
        printf("Allocating the output image.\n");
#endif
        dims[0] = classified->num_cols;
        dims[1] = classified->num_rows;
        if(*output_image == NULL){
                *output_image = (AVSfield_float *)
                        (AVSdata_alloc ("field 2D uniform scalar float",dims));
        }
```

```
        else if ( (dims[0] != MAXX( *output_image )) ||
                  (dims[1] != MAXY( *output_image ) ) ) {
                AVSfield_free( (AVSfield *) *output_image);
                *output_image = (AVSfield_float *)
                        (AVSdata_alloc ("field 2D uniform scalar float",dims));

        }


        /* copy the class map to the output map */
#ifdef DEBUG
        printf("Copying the classified image to the output port.\n");
#endif
        for(row = 0; row < classified->num_rows; row++){
                for(col = 0; col < classified->num_cols; col++){
                    I2D(*output_image, col, row) = classified->array[row][col];
                }
        }

        /* free all allocated memory */
        status = free_memory(network);
        free(parameter);
        free(network);
        free_imatrix(classified->array,0,classified->num_rows-1,
                                       0,classified->num_cols-1);
        free(classified);
        free(input);

        /* return success */
        return(1);
} /* end of main */

/*=========================================================================
function remap_pixel

        this function remaps the incoming int pixel to a float pixel varying
        between 0 and 1.  Complement coding is automatically supported if flag
        is set to 1.  Values min and max represent the minimal and maximal
        values before the reamp.

=========================================================================*/
void remap_pixel(float *pixel, int length, int flag, float min, float max)
{
float value, remap_value;
register int index;


        for(index=0; index<length/2; index++){
                value = pixel[index];
                remap_value = (value - min) / (max-min);
                pixel[index]=remap_value;

                if (flag==1) /* indicates complement coding */
                        pixel[index+length/2] = 1.0-remap_value;

        }

}/* end of remap_pixel */
```

```
/*-------------------------------------------------------------------
  Routine Name : Load_Net
  Function: Load an ARTMAP neural network from a disk file.
-------------------------------------------------------------------*/
int load_net(char *filename, network_data_ptr network)
{
register int i, j;
int status, cat;
FILE   *in_file;

in_file = fopen(filename, "rb");

if(in_file==NULL){
        printf("ERROR: unable to open user requested network file.\n");
        printf("Exiting to system...\n");
        exit(1);
}
        /* read in the number of categories and the number of attributes */
        fread(network->num_att, sizeof(int), 2, in_file);
        fread(network->num_category, sizeof(int), 2, in_file);

#ifdef LOAD_DEBUG /* echo the number of attributes and categories */
        printf("These are absolute values and may include complements.\n");
        printf("Number of attributes for ARTa = %d\n",network->num_att[ARTa]);
        printf("Number of attributes for ARTb = %d\n",network->num_att[ARTb]);
        printf("Number of ARTa categories = %d\n",network->num_category[ARTa]);

        printf("Number of ARTb categories = %d\n",network->num_category[ARTb]);

#endif

        /* allocate all the memory for the network */
        status = alloc_memory(network);
                if (status !=1){
                        printf("ERROR: allocate_memory returned bad status.\n")
;
                        exit(1);
                }

        /* read in the committed vector */
        for (i=0; i<2; i++){
                fread(network->committed[i], sizeof(int),
                                network->num_category[i]+1, in_file);
                /* read in the weight vectors */
                for (j=0; j<network->num_category[i]+1; j++)
                        fread(network->dn[i][j], sizeof(float),
                                network->num_att[i], in_file);
        }

        /* read in the IAM */
        for (i=0; i<network->num_category[ARTa]+1; i++)
                fread(network->IAMI[i],sizeof(float),
                                network->num_category[ARTb]+1,in_file);

        /* read in the confidence vector */
        fread(network->confidence,sizeof(float),
                        network->num_category[ARTa]+1,in_file);

        /* read in the predicted category vector */
        fread(network->makes_prediction,sizeof(int),
                        network->num_category[ARTa]+1,in_file);

        /* calc and store the resulting classification for each ARTa F2 node */

        for(i=0; i<=network->num_category[ARTa]; i++){
                cat  = matching_category(network->IAMI[i],
                        network->num_category[ARTb]);
                network->target_class[i]=cat;
        }

#ifdef LOAD_DEBUG /* echo the net file information */
        printf("Node, weight vector, confidence, predicts category:\n");
        for(i=0; i<=network->num_category[ARTa]; i++){
                printf("%d\t", i);
                for(j=0; j < network->num_att[ARTa]; j++)
                        printf("%1.2f  ",network->dn[ARTa][i][j]);
                printf("\t");
                printf("%1.2f\t",network->confidence[i]);
                printf("%d\t",network->makes_prediction[i]);
                printf("%d\n", network->target_class[i]);
        }
#endif

        fclose(in_file);

        return(status);

} /* end of load_network */
```

```
/*=========================================================================

Filename imp_art.c

=========================================================================*/

#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "idef.h"
#include "struct.h"
#include "ptypes.h"

/*-----------------------------------------------------------------
  Routine Name : setup_index
  Function: Randomize the Indices of first num_pats patterns.
-----------------------------------------------------------------*/
void setup_index(data_struct_ptr data, int num_pats)
{
int firstIndex;
int secondIndex;
int holdingTank;

        for (firstIndex = 0; firstIndex < num_pats; firstIndex++){
                secondIndex = rand()%num_pats;
                holdingTank = data->Index[firstIndex];
                data->Index[firstIndex] = data->Index[secondIndex];
                data->Index[secondIndex] = holdingTank;
        }
}

/*-----------------------------------------------------------------
  Routine Name : pre_scan_data
  Function: Before learning, preprocess the data vectors to determine
                                the frequency of eacn data category.
-----------------------------------------------------------------*/
void pre_scan_data(data_struct_ptr data, network_data_ptr network,
                parameter_struct_ptr parameter, int num_pats)
{
int i, B_winner;

        for(i=0; i<num_pats; i++)
                data->Index[i] = i;

        for(i=0; i<=parameter->num_data_category; i++)
                data->frequency[i] = 0;

        for (i=0; i<num_pats; i++){
 /*     read_pattern (i); in io.c       */
        B_winner = data_category(network->pattern[ARTb],parameter->b_length,i);
                if (B_winner==-1)
                        B_winner = parameter->num_data_category;
                        data->frequency[B_winner]++;
                }

        printf ("Number of Data Category = %d\n", parameter->num_data_category);
        for (i=0; i<parameter->num_data_category; i++)
                printf ("Frequency of Category %d = %d\t %1.1f%%\n", i+1,
                        data->frequency[i],
                        100*data->frequency[i]/(float)num_pats);
        printf ("\n");
}

/*-----------------------------------------------------------------
  Routine Name : Init_Random
  Function: Initialize random number generator.
 -----------------------------------------------------------------*/
void init_random(parameter_struct_ptr parameter)
{
        if (parameter->randomInit == 1){
                if (parameter->trace >= 1)
                        printf ("... initializing random number generator by time\n\n");
                srand(time(NULL));
        }
        else{
                if (parameter->trace >= 1)
                        printf("Initializing random generator using %d\n",
                                parameter->randomInit);
                srand(parameter->randomInit);
        }
}

/*-----------------------------------------------------------------
  Routine Name : Init_ARTMAP
  Function: Initialize ARTMAP network in particular the data structure
                                of ARTa F2 recognition nodes.
 -----------------------------------------------------------------*/
void init_ARTMAP(network_data_ptr network)
{
int initx;
int inity;
float baseline_confidence = 1.0;

        for (initx = 0; initx < network->num_category[ARTa]+1; initx++){
                network->confidence[initx] = baseline_confidence;
                network->makes_prediction[initx] = FALSE;
                network->freeze[initx] = FALSE;
                for (inity = 0; inity < network->num_category[ARTb]+1; inity++)
                        network->IAMI[initx][inity] = 0.0;
        }
}

/*-----------------------------------------------------------------
  Routine Name : Equal_Input
  Function: Given two vectors, determine whether they are identifical.
-----------------------------------------------------------------*/
int equal_input(parameter_struct_ptr parameter, float *v1,float *v2)
{
int j;

        for (j=0; j<parameter->a_length; j++)
                if (v1[j] != v2[j])
                        return (FALSE);
        return (TRUE);
```

```
}

/*-------------------------------------------------------------------
  Routine Name : Present
  Function: Determine whether input is present in an ART module.
-------------------------------------------------------------------*/
int present(network_data_ptr network, int whichART)
{
int present_x;

        for (present_x = 0; present_x < network->num_att[whichART]; present_x++)
                if(network->pattern[whichART][present_x] > 0.00000000000001)
                        return(TRUE);
        return(FALSE);
}

/*-------------------------------------------------------------------
  Routine Name : Perfect_Match
  Function: Test whether a perfect match between the input and the
            template vectors has occurred.
-------------------------------------------------------------------*/
int perfect_match(network_data_ptr network, int A_node)
{
        if (matchFunction(network,ARTa, A_node) == 1.0)
                return(TRUE);
        else
                return(FALSE);
}

/*-------------------------------------------------------------------
  Routine Name : Normalize
  Function: Normalize a vector so that it's norm is 1.
-------------------------------------------------------------------*/
void normalize(float *v, int n)
{
int i;
float  sum=0;

        for (i=0; i<n; i++)
                sum += v[i];

        for (i=0; i<n; i++)
                v[i] = v[i]/sum;
}


/*-------------------------------------------------------------------
  Routine Name : Data_Category
  Function: Given a binary vector and its length, determine the data
                                category that it indicates.
-------------------------------------------------------------------*/
int data_category(float *v, int n, int pc)
{
int  i, num_cat, cat;

        num_cat = 0;
        for (i=0; i<n; i++)
                if (v[i]>0.5){
                        cat = i;
                        num_cat++;
                }

        if(num_cat==1)
                return (cat);
        else{
                printf ("WARNING: Error in class of pattern %d.\n", pc);
                return (-1);
        }
}

/*-------------------------------------------------------------------
  Routine Name : Relabel
  Function: Modify the data category of the current data vector.
-------------------------------------------------------------------*/
void relabel(data_struct_ptr data, parameter_struct_ptr parameter,
        network_data_ptr network, int pc)
{
int  i;

        for(i=0; i<parameter->b_length; i++)
                network->pattern[ARTb][i] = 0.0;
        network->pattern[ARTb][data->label[pc]] = 1.0;

        if(parameter->complement){
                for (i=parameter->b_length; i<network->num_att[ARTb]; i++)
                        network->pattern[ARTb][i] = 1.0;
                network->pattern[ARTb][data->label[pc]+parameter->b_length] = 0.0;
  }
}

/*-------------------------------------------------------------------
  Routine Name : Matching_Category
  Function: Given the inter-ART map field vector and its length,
                                determine the ARTb category that it predicts.
-------------------------------------------------------------------*/
int matching_category(float *v, int n)
{
int    i, max_i;
float  max;

        max = -1.0;

        for (i=0; i<n; i++)
                if (v[i]>max){
                        max = v[i];
                        max_i = i;
                }

  return (max_i);
}

/*-------------------------------------------------------------------
  Routine Name : Destroy
  Function: Reset a node to be uncommitted node.
                                Initialize all attached attributes.
-------------------------------------------------------------------*/
```

```c
void destroy (parameter_struct_ptr parameter, network_data_ptr network,
            int A_node)
{
int destx;
float baseline_confidence = 1.0;


        if (parameter->employ_weighting)
                for (destx = 0; destx < network->num_att[ARTa]; destx++)
                        network->dn[ARTa][A_node][destx] =
parameter->wt[destx%parameter->[illegible]
        else
                for (destx = 0; destx < network->num_att[ARTa]; destx++)
                        network->dn[ARTa][A_node][destx] = parameter->dnInit;

        for (destx = 0; destx < network->num_category[ARTb]+1; destx++)
                network->IAMI[A_node][destx] = 0.0;

        network->confidence[A_node] = baseline_confidence;
        network->makes_prediction[A_node] = FALSE;
        network->freeze[A_node] = FALSE;
        network->committed[ARTa][A_node] = FALSE;
        network->vg[ARTa][A_node] = parameter->min_arho;

        if (parameter->trace >= 2)
                printf ("ARTa category %d destroyed.\n", A_node+1);
}

/*------------------------------------------------------------------
Routine Name : Change_Confidence
Function: Update confidence of a Recognition node with respect to
        prediction success.  This subroutine adjusts the confidence of nodes
        according to whether they just made a correct prediction
        (reduce_confidence = 0) or an incorrect prediction
        (reduce_confidence = 1).
------------------------------------------------------------------*/
void change_confidence (network_data_ptr network,
        parameter_struct_ptr parameter, int A_node, int reduce_confidence, int phase)
{
        if(phase == TRAIN){
                network->confidence[A_node] += parameter->noise_rate *
                        (1.0 - network->confidence[A_node] - reduce_confidence);
                if (network->confidence[A_node] < parameter->noise_tolerance)
                        destroy (parameter, network, A_node);
        }
}

/*------------------------------------------------------------------
Routine Name : Learn_Prediction
Function: Update inter-ART map field to make the association. Set
        make_prediction of the ARTa F2 node to be true.
------------------------------------------------------------------*/
void learn_prediction(network_data_ptr network, int A_node, int B_node)
{
int lpx;

        for(lpx = 0; lpx < network->num_category[1]+1; lpx++)
                network->IAMI[A_node][lpx] = 0.0;
```



```c
        network->IAMI[A_node][B_node] = 1.0;
        network->makes_prediction[A_node] = TRUE;
}

/*------------------------------------------------------------------
  Routine Name : Match_Track
  Function: Increase the vigilance by the amount just enough to cause
  a reset. If new vigilance is greater than 1, indicate perfect mismatch.
------------------------------------------------------------------*/
void match_track(network_data_ptr network, parameter_struct_ptr parameter,
        netstat_ptr netstat, data_struct_ptr data, series_data_ptr series,
        float vg, float match,  int A_node, int phase, int pat_id)
{
        if (vg < match + parameter->epsilon)
                vg = match + parameter->epsilon;

        if (vg >= 1.0 && phase==TRAIN){
                netstat->perfect_mismatch = TRUE;
                printf("WARNING: PERFECT_MISMATCH ON PATTERN %d:(%d)\n",
                        pat_id+1,data->Index[pat_id]+1);

                if (parameter->trace >=1 ){
                        printf ("category %d\n", A_node);
                        display_pattern("Z",network->dn[ARTa][A_node],
                                    network->num_att[ARTa]);
                        display_pattern("I",network->pattern[ARTa],
                                    network->num_att[ARTa]);
                }

                series->num_perfect_mismatch++;

        }
        else{
                if(phase == TRAIN)
                        netstat->got_all_train_right = FALSE;
                        network->F2[0][A_node] = -1;
        }
}


/*------------------------------------------------------------------
  Routine Name : IA_reset
  Function: Initial inter-ART reset by performing matching tracking.
------------------------------------------------------------------*/
void IA_reset(network_data_ptr network, parameter_struct_ptr parameter,
        netstat_ptr netstat, data_struct_ptr data, series_data_ptr series,
        int A_node, int phase, int pat_id)
{
float match;

        match = matchFunction(network,ARTa,A_node);

        if(parameter->ind_vg)
                match_track(network, parameter, netstat, data, series,
                        network->vg[0][A_node],match, A_node, phase,
                        pat_id);
        else
```

```
                match_track(network, parameter, netstat, data, series,
                        network->rho[0], match, A_node, phase, pat_id);
}

/*-----------------------------------------------------------------------
Routine Name : refresh_system
Function: Called before each input presentation.  Initialize variables and
        clear F2 activities.  When network makes first prediction, this is set
        to the index of the predicted ART-B node. It is used to keep track of
        errors by prediction type.  If AB, ART-A receives input first.  After
        an inter-ART mismatch reset, order is set to BA indicating that ART-B
        has identified a winner while ART-A returns to its search cycle. It is
        possible to allow ART-B to initially receive its pattern first by
        setting order to BA here.
-----------------------------------------------------------------------*/
void refresh_system(network_data_ptr network, parameter_struct_ptr parameter,
        netstat_ptr netstat)
{
        network->first_prediction[0] = -1;
        netstat->wrong = FALSE;
        netstat->no_prediction = TRUE;
        network->rho[ARTa] = parameter->min_arho;
        netstat->order = BA;
        refresh_F2(network);
}


/*-----------------------------------------------------------------------
  Routine Name : A_only
  Function: Only ARTa has input. Perform ART learning in ARTa.
-----------------------------------------------------------------------*/
void A_only(network_data_ptr network, parameter_struct_ptr parameter, int phase
)
{
        if (phase == TRAIN)
                learn(network,parameter, ARTa,
                      ART(network, parameter, ARTa, TRAIN) );
}

/*-----------------------------------------------------------------------
  Routine Name : B_only
  Function: Only ARTb has input. Perform ART learning in ARTb.
-----------------------------------------------------------------------*/
void B_only(network_data_ptr network, parameter_struct_ptr parameter, int phase
)
{
  if (phase == TRAIN)
         learn(network,parameter,ARTb,ART(network, parameter, ARTb, TRAIN));
}

/*-----------------------------------------------------------------------
  Routine Name : Recast_Inconsistent_Cases
  Function: Perform preprocessing to remove inconsistency in data
                                vectors.
-----------------------------------------------------------------------*/
void recast_inconsistent_cases(int mode, data_struct_ptr data,
                parameter_struct_ptr parameter)
{
```

```
register int    r, s, i;
int    *recasted, *vote, *num_in_cat;
int    max_v, recasted_label, tot_vote, num_recasted;
float  chance, accum_vote_fraction;

        if (mode==0){
                if (parameter->trace>=1)
                printf ("No recasting ...\n");
                return;
        }

        if (parameter->trace>=1)
                printf ("Recasting inconsistent cases. Please wait ...\n");

        recasted = (int *) malloc (sizeof(int)*data->num_pixels);
        vote = (int *)malloc(sizeof(int)*parameter->b_length);

        for (r = 0; r < data->num_pixels; r++)
                recasted[r] = FALSE;


        num_recasted = 0;

        for (r=0; r<data->num_pixels-1; r++)
                if (recasted[r] == FALSE){
                for (i=0; i<parameter->b_length; i++)
                        vote[i] = 0;

                recasted[r] = EXAM;
                vote[data->label[r]] = 1;
                tot_vote = 1;

                for (s=r+1; s<data->num_pixels; s++)
                        if (recasted[s]==FALSE && equal_input(parameter,

                                recasted[s] = EXAM;
                                vote[data->label[s]]++;
                                tot_vote++;
                        }

                        if (mode==BY_MAJORITY){
                                max_v = -999;
                                recasted_label = -1;

                                for (i=0; i<parameter->b_length; i++){
                                        if (vote[i] > max_v){
                                                max_v = vote[i];
                                                recasted_label = i;
                                        }
                                }
                        }
                        else if (mode==BY_PROBABILITY){
                                chance = (rand()%100)/100.0;
                                accum_vote_fraction = 0.0;

                                for (i=0; i<parameter->b_length; i++){
                                        accum_vote_fraction += vote[i]/(float)tot_vote;
```

```
                        if (chance < accum_vote_fraction){
                                recasted_label = i;
                        break;
                        }
                }
        }

        for (s = r; s < data->num_pixels; s++)
                if (recasted[s] == EXAM){
                        if (data->label[s] != recasted_label){
                                data->label[s] = recasted_label;
                                num_recasted++;
                        }
                recasted[s] = TRUE;
                }
        }


    free(recasted);
    free(vote);

    num_in_cat = (int *)malloc(parameter->b_length*sizeof(int));

    for (i=0; i<parameter->b_length; i++)
            num_in_cat[i] = 0;
    for (i=0; i<data->num_pixels; i++)
            num_in_cat[data->label[i]]++;

    if (parameter->trace>=1){
            printf (">> Total of %d cases recasted.\n\n", num_recasted);

            for (i=0; i<parameter->b_length; i++)
                    printf ("Number in Category %d = %d\n", i, num_in_cat[i]);
            printf ("\n");
    }

    free(num_in_cat);
}

/*-----------------------------------------------------------------
  Routine Name : Record_Train_Performance
  Function: Record prediction performance during training in :
            num_no_prediction, num_wrong and num_right.
-----------------------------------------------------------------*/
void record_train_performance(netstat_ptr netstat, series_data_ptr series){

    if (netstat->no_prediction)
            series->num_no_prediction++;

    if (netstat->wrong)
            series->num_wrong++;

    if (netstat->right)
            series->num_right++;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <math.h>
#include "idef.h"       /* header with various network definitions */
#include "struct.h"     /* header containing data structure definitions */
#include "ptypes.h"

/*
#define FUZZY_DEBUG
*/

/*-------------------------------------------------------------------
  Routine Name : Init_ARTs
  Function: Initialize ARTs vaiables and weight templates.
-------------------------------------------------------------------*/
void init_ARTs(network_data_ptr network, parameter_struct_ptr parameter)
{
int initx;
int inity;
int initz;

        network->rho[ARTa] = parameter->min_arho;
        network->rho[ARTb] = parameter->brho;

for(initx=0; initx<2; initx++)
        for(initz=0;initz < network->num_category[initx]+1;initz++){
                network->committed[initx][initz] = FALSE;
                network->vg[initx][initz] = network->rho[initx];
                if (parameter->employ_weighting && initx==ARTa){
                        for(inity = 0;inity < network->num_att[initx];inity++)
                        network->dn[initx][initz][inity] =
                                parameter->wt[inity%parameter->a_length];
                }
                        else
                        for(inity = 0; inity < network->num_att[initx]; inity++)
                        network->dn[initx][initz][inity] = parameter->dnInit;
        }
}

/*-------------------------------------------------------------------
  Routine Name : Init_ARTB
  Function: Write the patterns to a file in binary format.
-------------------------------------------------------------------*/
void init_ARTb(network_data_ptr network, parameter_struct_ptr parameter)
{
int  i, j;

        refresh_F2(network);

for(i=0; i<parameter->b_length; i++){
        for (j=0; j<parameter->b_length; j++){
                network->pattern[1][j] = 0.0;
                if (parameter->complement)
                        network->pattern[1][j+parameter->b_length] = 1.0;
        }
        network->pattern[1][i] = 1.0;
        if (parameter->complement)
                network->pattern[1][i+parameter->b_length] = 0.0;

        B_only (network,parameter,TRAIN);
}

}

/*-------------------------------------------------------------------
  Routine Name : Refresh_F2
  Function: Clear the ARTa and ARTb F2 activities.
-------------------------------------------------------------------*/
void refresh_F2(network_data_ptr network)
{
int refx;
int refy;

        for(refx=0; refx<2; refx++)
                for(refy = 0; refy < network->num_category[refx]+1; refy++)
                        network->F2[refx][refy] = 0.0;

#ifdef FUZZY_DEBUG
        printf("F2 field refreshed.\n");
#endif
}

/*-------------------------------------------------------------------
Routine Name : MatchFunction
Function: Compute the match of the weight template to the input
pattern. The value of whichART determines which ART module is being queried
and f2 is the index of the F2 node being examined.  Mathematically speaking,
this function returns the degree to which the input pattern is a fuzzy subset
of the weight vector.  The resulting vlaue is used to determine if resonance
occurrs between the input and classification fields.
-------------------------------------------------------------------*/

float matchFunction(network_data_ptr network, int whichART, int f2)
{

int mfx;
float numerator;      /* the numerator of the match value */
float denominator;    /* the denominator of the match value */

numerator = 0.0;
denominator = 0.0;
for(mfx=0; mfx < network->num_att[whichART]; mfx++){
        denominator += network->pattern[whichART][mfx];
        if(network->pattern[whichART][mfx]<network->dn[whichART][f2][mfx])
                numerator += network->pattern[whichART][mfx];
        else
                numerator += network->dn[whichART][f2][mfx];
        }
        return (numerator/denominator);
}


/*-------------------------------------------------------------------
Routine Name : ChoiceOrderFunction
Function: Compute the match of the input pattern to the weight template.  The
```

```
value of whichART determines whether ARTa or ARTb is being querried, and f2 is

the index of the F2 node being examined.  Mathematically, this function returns

the degree to which the weight vector is a fuzzy subset of the input vector.
The result values is used to determine the winning F2 node.
-------------------------------------------------------------------------*/
float choiceOrderFunction(network_data_ptr network,
                          parameter_struct_ptr parameter, int whichART, int f2)
{
register int cofx;
float numerator;   /* the numerator of the choice value */
float denominator; /* the denominator of the choice value */

        if (network->F2[whichART][f2]==-1.0) /* if the node is "sleeping" */
                return (-1.0);
        else{
                numerator = 0.0;
                denominator = 0.0;
                for (cofx=0;cofx < network->num_att[whichART];cofx++){
                        denominator += network->dn[whichART][f2][cofx];
        if(network->pattern[whichART][cofx] < network->dn[whichART][f2][cofx])
                                numerator += network->pattern[whichART][cofx];
                        else
                                numerator += network->dn[whichART][f2][cofx];
                }
                denominator += parameter->Alpha;
                return (numerator/denominator);
        }
}
/*-------------------------------------------------------------------
  Routine Name : ARTreset
  Function: Simulate ART reset by turning the activated F2 node off.
---------------------------------------------------------------------*/
int ARTreset(network_data_ptr network, parameter_struct_ptr parameter,
                                int whichART, int f2)
{

if ((parameter->ind_vg && matchFunction(network,whichART,f2) <
        network->vg[whichART][f2]) ||
        (!parameter->ind_vg && matchFunction(network,whichART,f2) <
        network->rho[whichART])){
                network->F2[whichART][f2] = -1.0;
#ifdef FUZZY_DEBUG
                printf("RESET: ARTreset(%d,%d).\n",whichART,f2);
#endif
                return (TRUE);
        }
        else
                return (FALSE);
}

/*-------------------------------------------------------------------------
Routine name ART
Function returns the winning node foran ART module as determined by the value
of whichART
-------------------------------------------------------------------------*/
```

```
int ART(network_data_ptr network, parameter_struct_ptr parameter, int whichART,

                int phase)
{
float maxChoice;     /* highest choice value found so far */
int mismatch;        /* binary: 0 means no chosen node has matched yet */
int winner;          /* the winning f2 node */
int f2;                 /* the current f2 node under consideration */

        /* detrmine the activations for each f2 node in the ART module */
        for(f2=0; f2<=network->num_category[whichART]; f2++)
                network->F2[whichART][f2]=choiceOrderFunction(network,parameter,whichART,f2);


        /* if testing, set the uncommited F2 node activation to zero. */
        if (phase==TEST){
        for(f2=0; f2<=network->num_category[whichART]; f2++)
                if (!network->committed[whichART][f2])
                        network->F2[whichART][f2] = 0.0;
        }

        do {
                maxChoice = -1.0;
                winner = -1;
                for(f2=0; f2<=network->num_category[whichART]; f2++){
                        if (maxChoice < network->F2[whichART][f2]){
                                maxChoice = network->F2[whichART][f2];
                                winner = f2;
                        }
                }
                mismatch = ARTreset(network, parameter, whichART, winner);
        } while (mismatch);

#ifdef FUZZY_DEBUG
        printf("The F2 activation vector is:\n");
        for(f2=0; f2<=network->num_category[whichART]; f2++)
                printf("%1.3f  ",network->F2[whichART][f2]);
        printf("\n");
        printf("The degree of agreement is %1.3f\n",maxChoice);
        printf("This value must be >= %1.3f to be classified.\n",
                        network->rho[whichART]);
#endif

        if (winner==network->num_category[whichART] && phase==TRAIN){ /* make a new no
de */
                network->num_category[whichART]++;
                realloc_memory(network,parameter,whichART);
        }

        return(winner);
}

/*-------------------------------------------------------------------
  Routine Name : Learn
  Function: Modify ART weight template according to the learning rule.
  Essentially accomplishes
  new_weight = beta*(Input^old_weight)+(1-beta)*old_weight
---------------------------------------------------------------------*/
```

```
void learn(network_data_ptr network, parameter_struct_ptr parameter,
                        int whichART, int f2)
{
int   learn_x;
float cur_rate;

        if (network->committed[whichART][f2])
                cur_rate = parameter->rate;
        else
                cur_rate = 1.0;

        for (learn_x = 0; learn_x < network->num_att[whichART]; learn_x++)
if(network->pattern[whichART][learn_x]<network->dn[whichART][f2][learn_x])
        network->dn[whichART][f2][learn_x]= (1.0-cur_rate) *
                                network->dn[whichART][f2][learn_x] +
                                cur_rate*network->pattern[whichART][learn_x];

  network->committed[whichART][f2] = 1;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include "idef.h"
#include "struct.h"
#include "ptypes.h"
#include "memutil.h"

/*-------------------------------------------------------------------
  Routine Name : display_pattern
  Function: Display a pattern with a header string.
-------------------------------------------------------------------*/
void display_pattern(char *str, float *v, int n)
{
int  i;

        printf ("%s = {", str);
                for (i=0; i<n; i++)
                        printf ("%1.2lf,", v[i]);
                printf ("}\n");
}

/*===================================================================

function read_parameter
This function loads in the user defined parameters for the neural network.

===================================================================*/
int read_parameters(char *filename, parameter_struct_ptr parameter)
{
FILE   *in_file;
int    status;
register int j;
char   line[200];
float float_dummy;


        in_file = fopen(filename, "r");
        if (!in_file){
                        printf("Cannot open requested parameter file.\n");
                        printf("Exiting from program.\n");
                        exit(1);
        }

        (void)fgets(line,sizeof(line),in_file);
        (void)sscanf (line, "%d", &parameter->randomInit);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->orderedSet);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->employ_weighting);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->on_line);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->num_voters);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->num_runs);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->max_iterations);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->complement);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->a_length);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->b_length);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->num_data_category);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->train_pats);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf(line, "%d", &parameter->predict_pats);
        (void)fgets(line, sizeof(line), in_file);
        (void)sscanf(line, "%d", &parameter->test_pats);
        (void)fgets(line, sizeof(line), in_file);
        (void)sscanf(line, "%d", &parameter->on_line_recast);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf(line, "%f", &float_dummy);
        parameter->dnInit = float_dummy;        /* this kluge seems important */
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->Alpha);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->epsilon);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->rate);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->z_bar);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->min_arho);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->brho);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->noise_rate);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->noise_tolerance);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->trace);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->trace_weight);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->display_confusion_matrix);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->incorporate_rules);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->extract_rules);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->quantize_rules);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->quantization_step);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->ind_vg);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%d", &parameter->num_F2_winners);
        (void)fgets (line, sizeof(line), in_file);
        (void)sscanf (line, "%f", &parameter->u_threshold);
        (void)fgets (line, sizeof(line), in_file);

        if (parameter->employ_weighting){
```

```
                parameter->wt = (float *)malloc(parameter->a_length*sizeof(float));
                for (j=0; j<parameter->a_length; j++)
                        fscanf (in_file, "%f", &parameter->wt[j]);
        }

        fclose (in_file);
        status = 1;
        return(status);

}/* end of get_parameters */

/*-----------------------------------------------------------------------
  Routine Name : Display_Parameter
  Function: Display parameters to stdout.
-----------------------------------------------------------------------*/
int display_parameters(parameter_struct_ptr parameter){
int status;

        status=0;
        printf ("\n");
        printf ("Random number seed\t\t\t : %d\n", parameter->randomInit);
        printf ("Ordered set\t\t\t\t : %d\n", parameter->orderedSet);
        printf ("Complement coding\t\t\t : %d\n", parameter->complement);
        printf ("Feature weighting\t\t\t : %d\n", parameter->employ_weighting);
        printf ("On line learning\t\t\t : %d\n", parameter->on_line);
        printf ("On line recasting\t\t\t : %d\n", parameter->on_line_recast);
        printf ("Number of voters\t\t\t : %d\n", parameter->num_voters);
        printf ("Number of F2 winners\t\t\t : %d\n", parameter->num_F2_winners);
        printf ("Number of runs\t\t\t\t : %d\n", parameter->num_runs);
        printf ("Maximum number of training iterations\t : %d\n",
                                parameter->max_iterations);
        printf ("Number of attributes ARTa\t\t : %d\n", parameter->a_length);
        printf ("Number of attributes ARTb\t\t : %d\n", parameter->b_length);
        printf ("Number of data categories\t\t : %d\n", parameter->num_data_category);

        printf ("Number of training patterns\t\t : %d\n", parameter->train_pats);
        printf ("No. of predict patterns\t\t\t : %d\n", parameter->predict_pats);
        printf ("No. of test patterns\t\t\t : %d\n", parameter->test_pats);
        printf ("Initial weight\t\t\t\t : %1.3f\n", parameter->dnInit);
        printf ("Alpha\t\t\t\t\t : %1.3f\n", parameter->Alpha);
        printf ("Epsilon\t\t\t\t\t : %1.3f\n", parameter->epsilon);
        printf ("Learning rate\t\t\t\t : %1.3f\n", parameter->rate);
        printf ("Z-Bar\t\t\t\t\t : %1.3f\n", parameter->z_bar);
        printf ("Minimum ARTa vigilance\t\t\t : %1.3f\n", parameter->min_arho);
        printf ("ARTb Vigilance\t\t\t\t : %1.3f\n", parameter->brho);
        printf ("Noise rate\t\t\t\t : %1.3f\n", parameter->noise_rate);
        printf ("Noise tolerance\t\t\t : %1.3f\n", parameter->noise_tolerance);
        printf ("Trace level\t\t\t\t : %d\n", parameter->trace);
        printf ("Trace weight\t\t\t\t : %d\n", parameter->trace_weight);
        printf ("Display confusion matrix\t\t : %d\n",
                                parameter->display_confusion_matrix);
        printf ("Incorporating rules\t\t\t : %d\n", parameter->incorporate_rules);
        printf ("Extracting rules\t\t\t : %d\n", parameter->extract_rules);
        printf ("Quantizing rules\t\t\t : %d\n", parameter->quantize_rules);
        printf ("Quantization step\t\t\t : %d\n", parameter->quantization_step);
        printf ("Threshold\t\t\t\t : %f\n", parameter->u_threshold);
        printf ("Individual vigilance\t \t\t : %d\n", parameter->ind_vg);
        printf ("\n");

        status=1;
        return(status);

}/*end of display parameter */

/*-----------------------------------------------------------------------
  Routine Name : write_param_file
  Function: writes the parameters to a text file
-----------------------------------------------------------------------*/
void write_param_file(parameter_struct_ptr parameter, char *filename){
FILE *out_file;

        out_file = fopen(filename, "w");
        if (!out_file){
                        printf("Cannot open requested parameter file.\n");
                        printf("Exiting from program.\n");
                        exit(1);
        }

        fprintf (out_file,"%d\t\t\t Random number seed\n", parameter->randomInit);
        fprintf (out_file,"%d\t\t\t Ordered set\n", parameter->orderedSet);
        fprintf (out_file,"%d\t\t\t Feature weighting\n",
                                parameter->employ_weighting);
        fprintf (out_file,"%d\t\t\t On line learning\n", parameter->on_line);
        fprintf (out_file,"%d\t\t\t Number of voters\n", parameter->num_voters);
        fprintf (out_file,"%d\t\t\t Number of runs\n", parameter->num_runs);
        fprintf (out_file,"%d\t\t\t Maximum number of training iterations\n",
                                parameter->max_iterations);
               fprintf (out_file,"%d\t\t\t Complement coding\n", parameter->complement);
        fprintf (out_file,"%d\t\t\t Number of attributes ARTa\n",
                                parameter->a_length);
        fprintf (out_file,"%d\t\t\t Number of attributes ARTb\n",
                                parameter->b_length);
        fprintf (out_file,"%d\t\t\t Number of data categories\n",
                                parameter->num_data_category);
        fprintf (out_file,"%d\t\t\t Number ot training patterns\n",
                                parameter->train_pats);
        fprintf (out_file,"%d\t\t\t Number of predict patterns\n",
                                parameter->predict_pats);
        fprintf (out_file,"%d\t\t\t Number of test patterns\n",
                                parameter->test_pats);
        fprintf (out_file,"%d\t\t\t On line recasting\n", parameter->on_line_recast);
        fprintf (out_file,"%1.3f\t\t\t Initial weight\n", parameter->dnInit);
        fprintf (out_file,"%1.3f\t\t\t Alpha\n", parameter->Alpha);
        fprintf (out_file,"%1.3f\t\t\t Epsilon\n", parameter->epsilon);
        fprintf (out_file,"%1.3f\t\t\t Learning rate\n", parameter->rate);
        fprintf (out_file,"%1.3f\t\t\t Z-Bar\n", parameter->z_bar);
        fprintf (out_file,"%1.3f\t\t\t Minimum ARTa vigilance\n",
                                parameter->min_arho);
        fprintf (out_file,"%1.3f\t\t\t ARTb Vigilance\n", parameter->brho);
        fprintf (out_file,"%1.3f\t\t\t Noise rate\n", parameter->noise_rate);
        fprintf (out_file,"%1.3f\t\t\t Noise tolerance\n",
                                parameter->noise_tolerance);
        fprintf (out_file,"%d\t\t\t Trace level\n", parameter->trace);
        fprintf (out_file,"%d\t\t\t Trace weight\n", parameter->trace_weight);
        fprintf (out_file,"%d\t\t\t Display confusion matrix\n",
                                parameter->display_confusion_matrix);
```

```
        fprintf (out_file,"%d\t\t\t Incorporating rules\n",
                        parameter->incorporate_rules);
        fprintf (out_file,"%d\t\t\t Extracting rules\n", parameter->extract_rules);
        fprintf (out_file,"%d\t\t\t Quantizing rules\n", parameter->quantize_rules);
        fprintf (out_file,"%d\t\t\t Quantization step\n",
                        parameter->quantization_step);
        fprintf (out_file,"%d\t\t\t Individual vigilance\n", parameter->ind_vg);
        fprintf (out_file,"%d\t\t\t Number of F2 winners\n",
                        parameter->num_F2_winners);
        fprintf (out_file,"%f\t\t Threshold\n", parameter->u_threshold);

        fclose(out_file);
}/*end of write_param_file */



/*-------------------------------------------------------------------
  Routine Name : alloc_memory
  Function: Allocate memory for ARTMAP network, including
  weight templates, map field, vigilance, Fab activities
  and ARTa F2 nodes activities, and attached attributes:
  committed, predict, freeze, num_predict, usage, accuracy,
  test_accuracy and confidence.
-------------------------------------------------------------------*/

int alloc_memory(network_data_ptr network)
{
int  i, j, status;

        network->IAMI = (float **)malloc(sizeof(float *)*
                        (network->num_category[ARTa]+1));
                for (i=0; i<network->num_category[ARTa]+1; i++)
                        network->IAMI[i]=(float *)malloc(sizeof(float)* (network->num_category[ARTb]
+1));

        network->pattern = (float **)malloc(sizeof(float *)*2);
        network->pattern[ARTa]=(float *)malloc(sizeof(float)*network->num_att[ARTa]);
        network->pattern[ARTb]=(float *)malloc(sizeof(float)*network->num_att[ARTb]);

        network->F2 = (float **)malloc (sizeof(float *)*2);
                for (i=0; i<2; i++)
                        network->F2[i] =(float *) malloc(sizeof(float)*
                                                        (network->num_category[i]+1));

        network->vg = (float **)malloc (sizeof(float *)*2);
                for (i=0; i<2; i++)
                        network->vg[i] = (float *)malloc(sizeof(float)*
                                                        (network->num_category[i]+1));

        network->Fab = (float *)malloc(sizeof(float)*
                                        (network->num_category[ARTb]+1));

        network->first_prediction = (float *)malloc(sizeof(float)*
                                                                (network->num_category[ARTb]+1));

        network->dn = (float ***)malloc(sizeof(float **)*2);
        for (i=0; i<2; i++){
                network->dn[i] = (float **) malloc(sizeof(float *)*
                                                        (network->num_category[i]+1));
                for (j=0; j<network->num_category[i]+1; j++)
                        network->dn[i][j] =(float *) malloc(sizeof(float)*network->num_att[i]);
        }

        network->confidence = (float *)malloc(sizeof(float)*
                                                                (network->num_category[ARTa]+1));
        network->usage = (float *)malloc(sizeof(float)*
                                                        (network->num_category[ARTa]+1));
        network->accuracy = (float *)malloc(sizeof(float)*
                                                        (network->num_category[ARTa]+1));
        network->target_class = (int *)malloc(sizeof(int)*
                                                                (network->num_category[ARTa]
        network->test_accuracy = (float *)malloc(sizeof(float)*
                                                                (network->num_category[ARTa]+1));
        network->num_predict = (int *)malloc(sizeof(int)*
                                                                (network->num_category[ARTa]+1));
        network->makes_prediction=(int *)malloc(sizeof(int)*
                                                                (network->num_category[ARTa
        network->freeze =(int *)malloc(sizeof(int)*
                                                        (network->num_category[ARTa]+1));
        network->first_prediction = (float *)malloc(sizeof(float)*
                                                                (network->num_categc

        network->committed = (int **) malloc(sizeof(int *)*2);
        for (i=0; i<2; i++)
                network->committed[i] = (int *) malloc(sizeof(int)*
                                                                (network->num_categor
        status=1;
        return(status);

}/* end of alloc_memory */

/*-------------------------------------------------------------------
  Routine Name : Realloc_Memeory
  Function: Allocate new memory when a new ARTa or ARTb F2 node is
                        added.
-------------------------------------------------------------------*/
void realloc_memory(network_data_ptr network, parameter_struct_ptr parameter,
                                                        int whichART)
{
int  i;

        network->F2[whichART]=(float *)realloc(network->F2[whichART],sizeof(float)*
                (network->num_category[whichART]+1));

        network->vg[whichART] = (float *)realloc(network->vg[whichART],
                sizeof(float)*(network->num_category[whichART]+1));

        network->committed[whichART] =(int *)realloc(network->committed[whichART],
                sizeof(int)*(network->num_category[whichART]+1));

        network->F2[whichART][network->num_category[whichART]] = 0.0;
        if (whichART==ARTa)
                network->vg[whichART][network->num_category[whichART]] =

        else
```

```
        network->vg[whichART][network->num_category[whichART]] = parameter->brho;

    network->committed[whichART][network->num_category[whichART]] = FALSE;

    network->dn[whichART] = (float **)realloc(network->dn[whichART],
        sizeof(float *)*(network->num_category[whichART]+1));
    network->dn[whichART][network->num_category[whichART]] =
        (float *)malloc(sizeof(float)*network->num_att[whichART]);

    if (whichART==ARTa && parameter->employ_weighting){
        for(i=0; i<network->num_att[ARTa]; i++)
            network->dn[ARTa][network->num_category[ARTa]][i] =
                                parameter->wt[i%parameter->a_length];
    }
    else
        for (i=0; i<network->num_att[whichART]; i++)
            network->dn[whichART][network->num_category[whichART]][i] =
                                parameter->dnInit;

    if (whichART==ARTa){
        network->IAMI = (float **)realloc(network->IAMI,sizeof(float *)*
                                        (network->num_category[ARTa]+1));
        network->IAMI[network->num_category[ARTa]] =(float *)malloc(sizeof(float)*
                                (network->num_category[ARTb]+1));

        network->confidence = (float *)realloc(network->confidence,sizeof(float)*
                            (network->num_category[ARTa]+1));

        network->usage =(float *)realloc(network->usage,sizeof(float)*
                            (network->num_category[ARTa]+1));

        network->accuracy = (float *)realloc(network->accuracy,sizeof(float)*
                                                (network->num_category[ARTa]+1));

        network->test_accuracy = (float *)realloc(network->test_accuracy,
                                                sizeof(float)*(network->num_category[ARTa]+1));

        network->num_predict = (int *)realloc(network->num_predict,sizeof(int)*
                                                (network->num_category[ARTa]+1));

        network->makes_prediction = (int *)realloc(network->makes_prediction,
                                                sizeof(int)*(network->num_category[ARTa]+1));

        network->freeze =(int *)realloc(network->freeze,sizeof(int)*
                                            (network->num_category[ARTa]+1));

        for (i=0; i<network->num_category[ARTb]+1; i++)
            network->IAMI[network->num_category[ARTa]][i] = 0.0;

        network->confidence[network->num_category[ARTa]] = 1.0;
        network->makes_prediction[network->num_category[ARTa]] = 0;
        network->freeze[network->num_category[ARTa]] = 0;
    }
    else{
        network->first_prediction = (float *)realloc(network->first_prediction,
                                            sizeof(float)*(network->num_category[ARTb]+1));

        network->Fab = (float *)realloc(network->Fab,sizeof(float)*
                                                (network->num_category[ARTb]+1));

        for (i=0; i<network->num_category[ARTa]+1; i++)
            network->IAMI[i] =(float *)realloc(network->IAMI[i],sizeof(float)*
                                                (network->num_category[ARTb]+

        for (i=0; i<network->num_category[ARTa]+1; i++)
            network->IAMI[i][network->num_category[ARTb]] = 0.0;
      }
    }


    /*-----------------------------------------------------------------
      Routine Name : free_memory
      Function: Release all ARTMAP memories (Thanks for the memories....).
    -----------------------------------------------------------------*/
    int free_memory(network_data_ptr network)
    {
        int i, j, status;

        status = 0;

        for (i=0; i<2; i++){
            for (j=0; j<network->num_category[i]+1; j++)
                free(network->dn[i][j]);
            free(network->dn[i]);
        }
        free(network->dn);

        for (i=0; i<2; i++)
            free(network->pattern[i]);
        free(network->pattern);

        for (i=0; i<network->num_category[ARTa]+1; i++)
            free(network->IAMI[i]);
        free(network->IAMI);

        for (i=0; i<2; i++)
            free (network->F2[i]);
        free(network->F2);

        for (i=0; i<2; i++)
            free(network->vg[i]);
        free(network->vg);

        free(network->Fab);
        free(network->first_prediction);
        free(network->confidence);
        free(network->usage);
        free(network->accuracy);
        free(network->target_class);
        free(network->test_accuracy);
        free(network->num_predict);
        free(network->makes_prediction);
        free(network->freeze);

        for(i=0; i<2; i++)
            free(network->committed[i]);
```

```
        free(network->committed);

        status =1;
        return(status);
}

/*-------------------------------------------------------------------
  Routine Name : alloc_series_memory
  Function: Allocate memory for series data on the test patterns.
-------------------------------------------------------------------*/
void alloc_series_memory(series_data_ptr series, parameter_struct_ptr parameter
,                        data_struct_ptr data)
{
        int  i, j, v;

series->right_wrong =(int ****)malloc (sizeof(int ***)*NUM_STAGE);
for (j=0; j<NUM_STAGE; j++){
series->right_wrong[j]=(int ***)malloc(sizeof(int **) * parameter->num_voters);


        for (v=0; v<parameter->num_voters; v++){
            series->right_wrong[j][v]=(int **)malloc(sizeof(int *)* 2);
                for (i=0; i<2; i++)
series->right_wrong[j][v][i] = (int *)malloc(sizeof(int) *
                                parameter->num_data_category);
        }
}

series->sum_rate = matrix(0, NUM_STAGE, 0, parameter->num_voters);

series->sum_sq_rate= matrix(0, NUM_STAGE, 0, parameter->num_voters);

if(parameter->display_confusion_matrix){
        series->p_table = (int ****)malloc(sizeof(int ***)*NUM_STAGE);

        for (j=0; j<NUM_STAGE; j++){
                series->p_table[j] =(int ***) malloc (sizeof(int **) *
                                                parameter->num_voters);
                for (v=0; v < parameter->num_voters; v++){
                        series->p_table[j][v] =(int **)
                        malloc(sizeof(int *) * (parameter->num_data_category+1));

                        for (i=0; i<parameter->num_data_category+1; i++)
                          series->p_table[j][v][i] =(int *)malloc(sizeof(int) *
                                                parameter->num_data_category);
                }
        }

        series->C = f3tensor(0,NUM_STAGE,
                             0, parameter->num_voters,
                             0, parameter->num_data_category);
}

        series->num_votes = matrix(0,data->num_pixels,
                                   0, parameter->num_data_category+1);

        series->actual_category = ivector(0, data->num_pixels);

  printf ("data series allocated.\n");
}

/*-------------------------------------------------------------------
  Routine Name : Init_Series
  Function: Initialize benchmarking statistical variables.
 -------------------------------------------------------------------*/
void init_series(series_data_ptr series, parameter_struct_ptr parameter)
{
int ps_x, ps_y, i, j, k, v;

        series->min_times_through_training = 100;
        series->max_times_through_training = 0;
        series->tot_times_through_training = 0;
        series->tot_committed_nodes = 0;
        series->min_committed_node = 9999;  /* change to 99999 in Unix */
        series->max_committed_node = -1;
        series->acc_num_antecedent = 0;
        series->acc_rule_extracted = 0;
        series->min_rule_extracted = 9999;  /* change to 99999 in Unix */
        series->max_rule_extracted = -1;
        series->min_rate_on_test = 100.0;
        series->max_rate_on_test = 0.0;
        series->tot_no_prediction = 0;
        series->sum_rate_on_test = 0.0;
        series->sum_sq_rate_on_test = 0.0;
        series->num_right = 0;
        series->num_wrong = 0;
        series->num_perfect_mismatch = 0;
        series->num_no_prediction = 0;

        for(j=0; j<NUM_STAGE; j++)
                for (v=0; v<parameter->num_voters; v++)
                        for (i=0; i<2; i++)
                                for (k=0; k<parameter->num_data_category; k++)
                                        series->right_wrong[j][v][i][k] = 0;

        for (j=0; j<NUM_STAGE; j++)
                for (v=0; v<parameter->num_voters; v++){
                        series->sum_rate[j][v] = 0.0;
                        series->sum_sq_rate[j][v] = 0.0;
                }

        if(parameter->display_confusion_matrix){
                for (j=0; j<NUM_STAGE; j++)
                        for (v=0; v<parameter->num_voters; v++)
                for(ps_x = 0; ps_x <parameter->num_data_category+1; ps_x++)
                for(ps_y = 0; ps_y <parameter->num_data_category; ps_y++)
                        series->p_table[j][v][ps_x][ps_y] = 0;
        }
}

/*-------------------------------------------------------------------
  Routine Name : Free_Series_Data
  Function: Release memory for benchmarking statistics.
-------------------------------------------------------------------*/
void free_series_data(series_data_ptr series, parameter_struct_ptr parameter,
                                                        data_struct_ptr data)
```

```
{
        int  i, j, v;

        for (j=0; j<NUM_STAGE; j++){
                for (v=0; v<parameter->num_voters; v++){
                        for (i=0; i<2; i++)
                                free(series->right_wrong[j][v][i]);
                        free(series->right_wrong[j][v]);
                }
                free(series->right_wrong[j]);
        }
        free(series->right_wrong);

        free_matrix(series->sum_rate, 0, NUM_STAGE, 0, parameter->num_voters);
        free_matrix(series->sum_sq_rate, 0, NUM_STAGE,
                                        0, parameter->num_voters);

        if(parameter->employ_weighting){
                free(parameter->wt);
        }

        free_matrix(series->num_votes, 0, data->num_pixels,
                                       0, parameter->num_data_category+1);

        free_ivector(series->actual_category, 0, data->num_pixels);

        if(parameter->display_confusion_matrix){
                for (j=0; j<NUM_STAGE; j++){
                        for (v=0; v<parameter->num_voters; v++){
                                for (i=0; i<parameter->num_data_category+1; i++)
                                        free(series->p_table[j][v][i]);
                                free(series->p_table[j][v]);
                        }
                        free(series->p_table[j]);

                }
                free(series->p_table);

                free_f3tensor(series->C, 0, NUM_STAGE,
                                        0, parameter->num_voters,
                                        0, parameter->num_data_category);
        }
}

/*-------------------------------------------------------------------
  Routine Name : Alloc_Data_Memory
  Function: Allocate memory for input patterns, Index array, label,
                        and frequence information.
-------------------------------------------------------------------*/
void alloc_data_memory(data_struct_ptr data, parameter_struct_ptr parameter)
{
register int  i;

        data->dpt=matrix(0,data->num_pixels-1,0,parameter->a_length);
        data->Index =(int *)malloc(sizeof(int)*data->num_pixels);
        data->train_Index=(int *)malloc(sizeof(int)*(parameter->train_pats+
                        parameter->predict_pats));
        data->label = ivector(0,data->num_pixels-1);
        data->frequency =(int *)malloc(sizeof(int)*(parameter->num_data_category+1));

        printf ("data memory allocated.\n");
}



/*-------------------------------------------------------------------
  Routine Name : Free_Data_Memory
  Function: Release memory for data patterns, Index array, label,
                                and frequence information.
-------------------------------------------------------------------*/
void free_data_memory(data_struct_ptr data, parameter_struct_ptr parameter)
{
int  i;

        free_matrix(data->dpt,0,data->num_pixels-1,0,parameter->a_length);
        free (data->frequency);
        free_ivector(data->label,0,data->num_pixels-1);
        free (data->Index);
        free (data->train_Index);
}

/*-------------------------------------------------------------------
  Routine Name : Display_Weight
  Function: Display the learned weight templates and map field values.
-------------------------------------------------------------------*/
void display_weight(network_data_ptr network)
{
int  j, k;

        for (j=0; j<network->num_category[ARTa]; j++){
                printf ("Category %d ARTa Weight: \n", j+1);
                for (k=0; k<network->num_att[ARTa]/2; k++)
                        printf ("%1.2f ", network->dn[ARTa][j][k]);
                printf ("\n");
                for (k=network->num_att[ARTa]/2; k<network->num_att[ARTa]; k++)
                        printf ("%1.2f ", network->dn[ARTa][j][k]);
                printf ("\n");
                printf ("Category %d Map Field  : ", j+1);
                for (k=0; k<network->num_category[ARTb]; k++)
                        printf ("%1.2f ", network->IAMI[j][k]);
                printf ("\n\n");
        }
}

/*-------------------------------------------------------------------
  Routine Name : weight_features
  Function: Weight the input features based on weighting factors.
-------------------------------------------------------------------*/
void weight_features(network_data_ptr network, parameter_struct_ptr parameter)
{
int i;

        for (i=0; i<network->num_att[ARTa]; i++)
        network->pattern[ARTa][i] = network->pattern[ARTa][i]*
                parameter->wt[i%parameter->a_length];
```

```
}

/*-------------------------------------------------------------------
  Routine Name : Report_Train_Performance
  Function: Report the prediction performance during training.
-------------------------------------------------------------------*/
void report_train_performance(network_data_ptr network,
                                                                    parameter_struct_ptr parameter,
                                                                    series_data_ptr series)
{
        printf ("Total of %d training patterns.\n", parameter->train_pats);
        printf ("%d recognition nodes created.\n", network->num_category[ARTa]);
        printf ("Number of correct prediction : %d\n", series->num_right);
        printf ("Number of perfect mismatch   : %d\n", series->num_perfect_mismatch);
        printf ("Number of wrong prediction   : %d\n", series->num_wrong);
        printf ("Number of no prediction      : %d\n", series->num_no_prediction);
        printf ("Discrimination rate of %1.1f%% on training set achieved.\n\n",
                                100*series->num_right/(float)parameter->train_pats);
}
```

Source code for the Confusion Matrix AVS module

```
/*=============================================================================

TITLE:          conf_mat.c
AUTHOR:         Nessmiller, Steven W.
DATE:           28 March 1995

SUMMARY:
This program computes a confusion matrix, the simple accuracy, the traditional

kappa coefficient, and B and P's kappa coefficient from two input
classification maps.  It then writes the results to a user defined disk file.

=============================================================================*/

#include <math.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/file.h>
#include <avs/avs.h>
#include <avs/avs_data.h>
#include <avs/field.h>
#include <avs/flow.h>
#include "memutil.h"

/* layered debug definitions */
#define DEBUG
/*
#define CONF_DEBUG
*/

#define MAXSTR 80

/* this is the structure to support the "truth" or "class map" image */
typedef struct{
        int num_rows;
        int num_cols;
        int num_classes;
}MAP_STRUCT, *map_struct_ptr;

/* this is the confusion matrix structure */
typedef struct{
        int **array;
        int num_rows;
        int num_cols;
        int num_classes;
        int count;
        float kappa;
        float BandP_kappa;
        float accuracy;
        float weighted_accuracy;
}CONF_MATRIX_STRUCT, *conf_matrix_ptr;

/* function prototypes */
int confusion(AVSfield_float *truth_image,
              AVSfield_float *class_image,
              map_struct_ptr truth,
              map_struct_ptr class_map,
              conf_matrix_ptr conf_matrix);
int write_output(char *filename, conf_matrix_ptr conf_matrix);


/* initialize the modules */
AVSinit_modules ()
{
        int AVS_confusion_matrix();
        AVSmodule_from_desc(AVS_confusion_matrix);
}

/* interface routine */
AVS_confusion_matrix()
{
        int compute_confmat();
        int param;
        int truth_port, class_port;
        char cwd[MAXSTR];

/* set the module name and type */
        AVSset_module_name("Confusion Matrix", MODULE_FILTER);

/* create the input port for the truth image LEFTMOST */
        truth_port = AVScreate_input_port("Truth Image",
                                          "field 2D uniform float", REQUIRED );

/* create the input port for the classification map RIGHTMOST */
        class_port = AVScreate_input_port("Class Image",
                                          "field 2D uniform float", REQUIRED );

/* display truth and classified port location */
        param = AVSadd_parameter("info","string",
                                 "TRUTH                                    CLASSIFIE
D","","");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* set the title for the window */
        param = AVSadd_parameter("title","string",
                                 "Classification accuracy:","","");
        AVSadd_parameter_prop(param,"width","integer",4);
        AVSconnect_widget(param,"text");

/* create a widget for the simple accuracy */
        param = AVSadd_float_parameter("Simple accuracy:", 0.0, 0.0, 100.0);
        AVSadd_parameter_prop(param, "width", "integer", 2);
        AVSconnect_widget(param, "typein_real");

/* create a widget for the weighted accuracy */
        param = AVSadd_float_parameter("Weighted accuracy:", 0.0, 0.0, 100.0);
        AVSadd_parameter_prop(param, "width", "integer", 2);
        AVSconnect_widget(param, "typein_real");

/* create a widget for the kappa coefficient */
        param = AVSadd_float_parameter("Kappa coefficient:", 0.0, 0.0, 100.0);
        AVSadd_parameter_prop(param, "width", "integer", 2);
        AVSconnect_widget(param, "typein_real");
```

```
/* create a widget for the B&P kappa coefficient */
        param=AVSadd_float_parameter("B&P's kappa coefficient:", 0.0,0.0,100.0)
;
        AVSadd_parameter_prop(param, "width", "integer", 2);
        AVSconnect_widget(param, "typein_real");

/* create a widget for the output filename */
        getwd(cwd);
        if( cwd[strlen(cwd)-1] != '/')
                strcat(cwd,"/");
        param=AVSadd_parameter("Output file (*.mat):","string", cwd, 0,".mat");
        AVSadd_parameter_prop(param, "width","integer",4);
        AVSconnect_widget(param, "browser");

/* allow user to view all file types */
        param = AVSadd_parameter("View all file types", "boolean",0,0,1);
        AVSadd_parameter_prop(param, "width", "integer",4);

/* create autogeneration widget */
        param=AVSadd_parameter("Auto-generate confusion matrix","boolean",0,0,
                                0);
        AVSadd_parameter_prop(param, "width", "integer", 4);
        AVSconnect_widget(param, "toggle");

/* create a widget to generate the confusion matrix */
        param=AVSadd_parameter("Calculate confusion matrix","oneshot",0,0,0);
        AVSadd_parameter_prop(param, "width", "integer", 4);

/* set the function to create the output */
        AVSset_compute_proc(compute_confmat);
}

/*============================================================================
function compute_confmat
This function does the confusion matrix computations
============================================================================*/
int compute_confmat(AVSfield_float *truth_image,
                    AVSfield_float *class_image,
                    char *info,
                    char *title,
                    float *accuracy,
                    float *interface_weighted_accuracy,
                    float *kappa,
                    float *BPkappa,
                    char *filename,
                    int all,
                    int auto_run,
                    int run)
{
/* local variables */
map_struct_ptr truth, class_map;
conf_matrix_ptr conf_matrix;
int status;
register int row, col;
float accuracy_val, weighted_accuracy_val, kappa_val, BPkappa_val;
int map_test, map_high, truth_test, truth_high;
char *answer;
FILE *output_file;
static int file_status = 0;
static int new_file = 0;
static char *old_filename[80];
char *reset_filename = " ";

        /* check for view all parameter filenames */
        if(AVSparameter_changed("View all file types")){
                if (all){
                        AVSmodify_parameter("Output file (*.mat):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/",0,"");
                        printf( "Setting maxval to null\n");
                        AVSmodify_parameter("Output file (*.mat):",
                             AVS_VALUE, filename, 0, 0);
                }
                else{
                        AVSmodify_parameter("Output file (*.mat):",
                             AVS_VALUE|AVS_MAXVAL, "/tmp/", 0, ".mat");
                        printf( "Setting maxval to .mat\n");
                        AVSmodify_parameter("Output file (*.mat):",
                                        AVS_VALUE, filename, 0, 0);
                }
        }

        /* see if we have a file and NOT just a directory */
        if(filename[strlen(filename) - 1] == '/'){
                new_file = 0;
                file_status = 0;
                return(1);
        }

        /* see if filename has changed */
        status = strcmp(filename, old_filename);

        if (status != 0){
                strcpy(old_filename, filename);
                output_file = fopen(filename,"r+");

                if(output_file == NULL) {
                        printf("Output file does not exist.\n");
                        new_file = 1;
                }

                else {
                        fclose(output_file);
                        printf("Output file already exists.\n");
                        answer = AVSmessage("beta",AVS_Warning,NULL,
                                 "Confusion Matrix","Overwrite!Cancel",
                                 "Output file %s already exists.\n",filename);

                        if( strcmp(answer,"Overwrite") == 0 )
                                file_status = 1;
                        else {
                                file_status = 0;
                                return(1);
                        }
                }
        }
```

```
        if ( (file_status == 1) || (new_file ==1) )
                printf("The output filename is: %s\n",filename);
        else
                return(1);

/* check if run time */
        if( (run || auto_run) && (file_status || new_file) ) {
              AVSmodify_parameter("Calculate confusion matrix",AVS_VALUE,0,0,0);
        }
        else{
                return(1);
        }

#ifdef DEBUG
        printf("Confusion matrix just entered the compute function.\n");
#endif

        /* reset the static variables for next time */
        new_file = 0;
        file_status = 0;
        strcpy(old_filename, reset_filename);

        /* clear statistics parameters and the elapsed time from other runs */
        AVSmodify_float_parameter("Simple accuracy:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  0.0, 0.0, 0.0);

        AVSmodify_float_parameter("Weighted accuracy:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  0.0, 0.0, 0.0);

        AVSmodify_float_parameter("Kappa coefficient:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  0.0, 0.0, 0.0);

        AVSmodify_float_parameter("B&P's kappa coefficient:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  0.0, 0.0, 0.0);

        /* allocate the structures */
        truth=(map_struct_ptr)malloc(sizeof(MAP_STRUCT));
        if (truth==NULL){
                printf("ERROR: unable to allocate truth structure pointer.\n");

                return(1);
        }

        class_map = (map_struct_ptr)malloc(sizeof(MAP_STRUCT));
        if (class_map == NULL){
                printf("ERROR: unable to allocate class map pointer.\n");
                return(1);
        }

        conf_matrix = (conf_matrix_ptr)malloc(sizeof(CONF_MATRIX_STRUCT));
        if (conf_matrix == NULL){
                printf("ERROR: unable to allocate confusion pointer.\n");
                return(1);
        }
```

```
        }

#ifdef DEBUG
        printf("All structures allocated.\n");
#endif


        /* get the image statistics */
        truth->num_rows = MAXY(truth_image);
        truth->num_cols = MAXX(truth_image);
        class_map->num_rows = MAXY(class_image);
        class_map->num_cols = MAXX(class_image);

        if((truth->num_rows != class_map->num_rows) ||
                        (truth->num_cols != class_map->num_cols)){
                AVSmessage("beta", AVS_Fatal, NULL,"Calculate confusion matrix","Kill module"
, "The number of rows and columns in the truth and classification map do not ag
ree.");
        }

        /* search both images to find the number of classes */
        truth_high = -1;
        map_high = -1;
        for(row=0; row < truth->num_rows; row++){
                for(col=0; col < truth->num_cols; col++){
                        truth_test = (int)I2D(truth_image, col, row);
                        if (truth_test > truth_high)
                                truth_high = truth_test;
                        map_test = (int)I2D(class_image, col, row);
                        if (map_test > map_high)
                                map_high = map_test;
                }
        }

        if(truth_high != map_high){
                printf("Num classes in the truth image = %d",truth_high);
                printf("Num classes in the class map image = %d",map_high);
                AVSmessage("beta", AVS_Fatal, NULL, "Create Confusion Matrix",
                        "Kill module", "The number of classes in the truth and classification map
 do not agree.");
        }

        truth->num_classes = truth_high;
        class_map->num_classes = map_high;

#ifdef DEBUG
        printf("The images are comparable.\n");
        printf("Num rows/columns in the truth image = %d / %d.\n",
                truth->num_rows, truth->num_cols);
        printf("Num rows/columns in the class map image = %d / %d.\n",
                class_map->num_rows, class_map->num_cols);
        printf("Num classes in the truth image = %d\n",truth->num_classes);
        printf("Num classes in the class map image = %d\n",
                class_map->num_classes);
#endif

        /* create the confusion matrix */
        status = confusion(truth_image, class_image, truth, class_map,
```

```
                        conf_matrix);
        if(status != 1){
                printf("ERROR: incorrect status from confusion.\n");
                return(1);
        }
        else
                printf("The confusion matrix has been calculated.\n");

        /* print the conf matrix to stdout */
        printf("This is the confusion matrix:\n");
        for(row=0; row <= conf_matrix->num_rows; row++){
                for(col=0; col <= conf_matrix->num_cols; col++)
                        printf("%d\t",conf_matrix->array[row][col]);
                printf("\n");
        }



        /* update the parameters to display the results */
        accuracy_val =  100 * conf_matrix->accuracy;
        AVSmodify_float_parameter("Simple accuracy:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  accuracy_val, accuracy_val, accuracy_val);

        weighted_accuracy_val = 100 * conf_matrix->weighted_accuracy;
        AVSmodify_float_parameter("Weighted accuracy:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  weighted_accuracy_val, weighted_accuracy_val

                                  weighted_accuracy_val);

        kappa_val =  100 * conf_matrix->kappa;
        AVSmodify_float_parameter("Kappa coefficient:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  kappa_val, kappa_val, kappa_val);

        BPkappa_val =  100 * conf_matrix->BandP_kappa;
        AVSmodify_float_parameter("B&P's kappa coefficient:",
                                  AVS_VALUE | AVS_MINVAL | AVS_MAXVAL,
                                  BPkappa_val, BPkappa_val, BPkappa_val);

        /* write the results to a disk file */
        status = write_output(filename, conf_matrix);
        if(status != 1){
                printf("ERROR: incorrect status returned from write_output.\n");
                return(1);
        }

        /* free all allocated memory */
        free_imatrix(conf_matrix->array,0,conf_matrix->num_rows,
                                        0,conf_matrix->num_cols);
        free(truth);
        free(class_map);
        free(conf_matrix);

        /* indicate success and jam */
        return(1);
} /* end of main */
```

```
/*============================================================================

function confusion
        computes the confusion matrix from the truth and the classification map
        images.  Parameters in the interface are updated in main.

============================================================================*/
int confusion(AVSfield_float *truth_image,
              AVSfield_float *class_image,
              map_struct_ptr truth,
              map_struct_ptr class_map,
              conf_matrix_ptr conf_matrix)
{
/* local variables */
int status, truth_index, class_index, row_sum, col_sum;
int sum_row_marginal, sum_col_marginal;
int send_val;
register int row, col, diag;
float *row_marginal, *row_summation, *col_marginal, *col_summation;
float truth_value, class_value;
float po, pe, class_contribution, element, numerator, denominator;
float row_step, percent_done, stage;

#ifdef DEBUG
        printf("Initializing the matrices.\n");
#endif

        status = 0; /* assume failure */
        row_step = 1.0 / truth->num_rows;
        conf_matrix->count = 0;
        conf_matrix->num_rows = truth->num_classes;
        conf_matrix->num_cols = truth->num_classes;
        conf_matrix->num_classes = truth->num_classes;

/* allocate the confusion matrix.  Must store "background" (class 0) pixels */
        conf_matrix->array=imatrix(0,conf_matrix->num_classes,
                                   0,conf_matrix->num_classes);

        if (conf_matrix->array == NULL){
                printf("ERROR: memory for confusion matrix not allocated.\n");
                return(1);
        }
        else{
                printf("Memory for the confusion matrix allocated.\n");
                status = 1;
        }

        /* allocate memory for the row and column marginals */
        row_marginal = vector(0,conf_matrix->num_classes);
        col_marginal = vector(0,conf_matrix->num_classes);
        row_summation = vector(0,conf_matrix->num_classes);
        col_summation = vector(0,conf_matrix->num_classes);

        /* initialize all the matrices to zero */
#ifdef DEBUG
        printf("Initializing the matrices.\n");
#endif
```

```
        for(row=0; row <= conf_matrix->num_classes; row++){
                for(col=0; col <= conf_matrix->num_classes; col++){
                        conf_matrix->array[row][col]=0;
                }
                row_marginal[row] = 0.0;
                col_marginal[row] = 0.0;
                row_summation[row]= 0.0;
                col_summation[row] = 0.0;
        }

/* step through images, create the confusion matrix, and count all pixels */

#ifdef DEBUG
        printf("Calculating the confusion matrix.\n");
#endif
        stage = 0.10;
        for(row=0; row < truth->num_rows; row++){
                for(col=0; col < truth->num_cols; col++){
                        truth_value = I2D(truth_image, col, row);
                        class_value = I2D(class_image, col, row);
                        truth_index = (int)truth_value;
                        class_index = (int)class_value;
#ifdef CONF_DEBUG
        printf("Incrementing element [%d][%d] of the confusion matrix.\n",
                class_index,truth_index);
#endif
                        conf_matrix->array[class_index][truth_index] += 1;
                        conf_matrix->count+=1;

                }       /* end of col loop */

                /* report the level of completion */
                percent_done = (float)row/(float)truth->num_rows;

                if( (percent_done > stage) &&
                (percent_done < stage + (2 * row_step)) ){
                        send_val = (int)(100.0 * stage);
                        printf("percent complete %d\n",send_val);
                        AVSmodule_status("percent complete", send_val);
                        stage += 0.10;

                }       /* end if */

        }       /* end of row loop */

#ifdef CONF_DEBUG
        printf("This is the confusion matrix.\n");
        for (row=0; row <= conf_matrix->num_rows; row++){
                for (col = 0; col <= conf_matrix->num_cols; col++)
                        printf("%d\t",conf_matrix->array[row][col]);
                printf("\n");
        }
        printf("The total number of pixels = %d\n",conf_matrix->count);
#endif

        /* calculate the row/column marginals and their summation */
        printf("Calculating the row and column marginals.\n");
        sum_row_marginal=0;
```

```
        for(row=0; row <= conf_matrix->num_classes; row++){
                row_sum = 0;
                for(col=0; col <= conf_matrix->num_classes; col++)
                        row_sum += conf_matrix->array[row][col];
                row_marginal[row] = (float)row_sum/(float)conf_matrix->count;
                row_summation[row] = (float)row_sum;
                sum_row_marginal += row_sum;
        }

        sum_col_marginal = 0;
        for(col=0; col <= conf_matrix->num_classes; col++){
                col_sum = 0;
                for(row=0; row <= conf_matrix->num_classes; row++)
                        col_sum += conf_matrix->array[row][col];
                col_marginal[col] = (float)col_sum/(float)conf_matrix->count;
                col_summation[col] = (float)col_sum;
                sum_col_marginal += col_sum;
        }

#ifdef CONF_DEBUG /* Debug code to echo the row and column marginals */
        printf("These are the row marginals.\n");
        for (row=0; row <= conf_matrix->num_rows; row++)
                printf("%f\t",row_marginal[row]);
        printf("\n");
        printf("These are the column marginals.\n");
        for (row=0; row <= conf_matrix->num_rows; row++)
                printf("%f\t",col_marginal[row]);
        printf("\n");
        printf("These are the row summations.\n");
        for (row=0; row <= conf_matrix->num_rows; row++)
                printf("%f\t",row_summation[row]);
        printf("\n");
        printf("These are the column summations.\n");
        for (row=0; row <= conf_matrix->num_rows; row++)
                printf("%f\t",col_summation[row]);
        printf("\n");
#endif

        /* the sum of either the row marginals or the column marginals must
        equal the total number of pixels */

        if((sum_row_marginal != sum_col_marginal) ||
                (sum_row_marginal != conf_matrix->count) ||
                (sum_col_marginal != conf_matrix->count)){
                AVSmessage("beta", AVS_Fatal, NULL, "Calculate confusion matrix","Kill module
","The row and column marginals of the confusion matrix do not agree.");
        }
        else
                printf("Row/column marginals and total pixel count agree.\n");

        /* calculate the simple accuracy and po and pe */
        printf("Calculating the accuracy measurements.\n");
        conf_matrix->accuracy = 0.0;
        conf_matrix->weighted_accuracy = 0.0;
        po = 0.0;
        pe = 0.0;
        for(diag=0; diag <= conf_matrix->num_classes; diag++){
                element = (float)conf_matrix->array[diag][diag];
```

```
                class_contribution = element / col_summation[diag];
                conf_matrix->weighted_accuracy += class_contribution;
                po += element;
                pe += row_marginal[diag] * col_marginal[diag];
        }
        po /= (float)conf_matrix->count;
        conf_matrix->weighted_accuracy /=(float)conf_matrix->num_classes+1;
        conf_matrix->accuracy = po;

        /* calculate both kappa measurements */
        printf("Calculating the kappa measurements.\n");
        conf_matrix->kappa=(po-pe)/(1.0-pe);
        numerator = po-(1.0/(float)(conf_matrix->num_classes+1));
        denominator = 1.0 - (1.0/(float)(conf_matrix->num_classes+1));
        conf_matrix->BandP_kappa=numerator/denominator;

#ifdef CONF_DEBUG /* Debug code to echo the accuracy */
        printf("The value of po is %f\n",po);
        printf("The value of pe is %f\n",pe);
        printf("The simple accuracy = %f\n",conf_matrix->accuracy);
        printf("The kappa coefficient = %f\n",conf_matrix->kappa);
        printf("B and P's kappa coefficient = %f\n",conf_matrix->BandP_kappa);
#endif

        /* free the allocated memory */
        free_vector(row_marginal,0,conf_matrix->num_classes);
        free_vector(col_marginal,0,conf_matrix->num_classes);

        return(status);

} /* end of compute_conf_matrix */

/*****************************************************************************
*

        Function write_output
        this function writes the confusion amtrix and the accuracy measurements
        to a user defined disk file.  The function will happily cream any disk
        file with the same name as the user inputs

*****************************************************************************
/
int write_output(char *filename, conf_matrix_ptr conf_matrix)
{

FILE *out_file;
register int row, col;
int status;

        status=0;       /* assume failure */

        out_file = fopen(filename, "w");
        if(out_file == NULL){
                AVSmessage("beta", AVS_Error, NULL, "Confusion matrix", "Okay", "Could not cr
eate the confusion file:\n%s", filename);
                return(1);
        }

        rewind(out_file);

        /* print the confusion matrix */
        fprintf(out_file,"This is the confusion matrix:\n");
        for(row=0; row <= conf_matrix->num_rows; row++){
                for(col=0; col <= conf_matrix->num_cols; col++)
                        fprintf(out_file,"%d\t",conf_matrix->array[row][col]);
                fprintf(out_file,"\n");
        }

        /* print the accuracy measurements */
        fprintf(out_file,"The simple accuracy = %f\n",conf_matrix->accuracy);
        fprintf(out_file,"The weighted accuracy = %f\n",
                conf_matrix->weighted_accuracy);
        fprintf(out_file,"The kappa coefficient = %f\n",conf_matrix->kappa);
        fprintf(out_file,"Brennan and Prediger's kappa coefficient = %f\n",
                        conf_matrix->BandP_kappa);

        fclose(out_file);
        printf("The confusion matrix file was successfully written to disk.\n");

        status=1;
        return(status);

}       /* end of write_output */
```

Task 1 Confusion Matrices

This is the confusion matrix:

4775	24	578	2284	0	2672	2459
3	152154	0	0	0	0	0
1100	0	43271	2105	145	0	0
0	0	2739	56024	0	2730	0
8762	0	2209	6	15102	0	0
0	0	0	1563	0	59623	6034
4	2081	0	0	0	39	49690

The simple accuracy = 0.910236
The weighted accuracy = 0.837707
The kappa coefficient = 0.885706
Brennan and Prediger's kappa coefficient = 0.895276

This is the confusion matrix:

11858	24	2	0	904	0	4
151	150544	0	0	0	0	1462
3191	0	32354	0	11076	0	0
2039	0	8774	49483	1190	7	0
1854	0	0	0	24225	0	0
1833	0	0	2960	0	52354	10073
549	12	0	0	0	0	51253

The simple accuracy = 0.889747
The weighted accuracy = 0.820863
The kappa coefficient = 0.861089
Brennan and Prediger's kappa coefficient = 0.871372

This is the confusion matrix:

1266	24	944	6311	117	3807	323
31	152116	0	0	0	0	10
4128	0	35112	1751	5630	0	0
6991	0	345	51046	0	3111	0
578	0	3752	0	21749	0	0
2869	0	0	305	0	64033	13
5902	2772	0	0	0	1892	41248

The simple accuracy = 0.876593
The weighted accuracy = 0.776496
The kappa coefficient = 0.843311
Brennan and Prediger's kappa coefficient = 0.856025

This is the confusion matrix:

8700	653	3177	236	3	2341	0
1122	95790	2009	0	6738	0	0
28	11775	82232	0	0	6894	0
668	0	0	61748	0	617	172
4625	128	0	0	82361	0	0
40	0	3071	2062	0	59847	0
3899	0	0	3729	0	0	29211

The simple accuracy = 0.886074
The weighted accuracy = 0.848144
The kappa coefficient = 0.862478
Brennan and Prediger's kappa coefficient = 0.867086

This is the confusion matrix:

1084	4372	1367	3739	1101	1627	1820
33	89578	129	178	15398	196	147
323	6175	89599	718	737	2544	833
4	30	3	51469	3	6	11690
100	75	32	29	86808	27	43
419	983	318	9402	240	53033	625
3	48	10	14	15	33	36716

The simple accuracy = 0.861590
The weighted accuracy = 0.809114
The kappa coefficient = 0.832648
Brennan and Prediger's kappa coefficient = 0.838522

This is the confusion matrix:

1115	4346	5572	523	198	3186	170
6044	85490	13232	0	893	0	0
12042	7061	69596	0	0	12230	0
3784	0	0	55115	0	1281	3025
5957	4423	0	0	76734	0	0
5125	0	534	13317	0	46044	0
8	0	0	1	0	0	36830

The simple accuracy = 0.782745
The weighted accuracy = 0.728336
The kappa coefficient = 0.740041
Brennan and Prediger's kappa coefficient = 0.746536

This is the confusion matrix:

2181	1650	85	3727	123	87
88	39904	2907	317	0	422
5474	947	18878	94	0	0
22	764	0	68426	3780	1041
420	2	0	479	58049	1
1892	1037	0	1363	0	47984

The simple accuracy = 0.800938
The kappa coefficient = 0.871235
Brennan and Prediger's kappa coefficient = 0.877676

This is the confusion matrix:

2415	251	3855	257	1067	8
383	35259	7896	65	0	35
2750	0	22643	0	0	0
256	90	103	62349	11213	22
162	0	0	16	58773	0
5743	827	0	710	33	44963

The simple accuracy = 0.863655
The weighted accuracy = 0.773229
The kappa coefficient = 0.829409
Brennan and Prediger's kappa coefficient = 0.836386

This is the confusion matrix:

1135	2946	1843	100	1681	148
1651	36634	4931	0	58	364
131	381	24881	0	0	0
16319	11624	0	35105	2640	8345
4140	35	6	8532	46235	3
7642	4592	0	1132	1	38909

The simple accuracy = 0.697704
The weighted accuracy = 0.664089
The kappa coefficient = 0.631779
Brennan and Prediger's kappa coefficient = 0.637245

This is the confusion matrix:

6510	965	2050	0	1488	2544
0	115959	9	0	0	0
0	149	29048	0	311	0
4101	0	0	2116	0	1475
0	0	1093	0	39460	2737
0	0	0	0	207	39778

The simple accuracy = 0.931484
The weighted accuracy = 0.885425
The kappa coefficient = 0.903023
Brennan and Prediger's kappa coefficient = 0.917781

This is the confusion matrix:

338	1114	2726	5221	2332	1826
48	113348	1742	271	345	214
7	31	29370	29	43	28
26	108	82	7227	161	88
2	4	3749	91	36215	3229
18	31	49	6736	173	32978

The simple accuracy = 0.877904
The weighted accuracy = 0.781404
The kappa coefficient = 0.829294
Brennan and Prediger's kappa coefficient = 0.853485

This is the confusion matrix:

2624	69	2882	992	2645	4345
320	113625	2023	0	0	0
3016	731	21129	0	4632	0
479	0	0	7213	0	0
8980	0	355	0	31603	2352
1996	0	0	28	3	37958

The simple accuracy = 0.856608
The weighted accuracy = 0.747211
The kappa coefficient = 0.799434
Brennan and Prediger's kappa coefficient = 0.827930

Task 2 Confusion Matrices

This is the confusion matrix:

286600	3	213	0	3	0
2322	11925	0	0	3254	0
206	0	45748	2238	2058	0
615	0	1181	49080	0	0
432	7	481	0	48933	0
7646	0	1191	966	0	8774

The simple accuracy = 0.951852
The weighted accuracy = 0.956570
The kappa coefficient = 0.917434
Brennan and Prediger's kappa coefficient = 0.942223

This is the confusion matrix:

286298	3	0	0	1	517
769	16716	0	0	4	12
3328	0	27000	11017	6210	2695
2384	0	0	43101	0	5391
822	2311	3	28	45878	811
95	0	0	0	0	18482

The simple accuracy = 0.923185
The weighted accuracy = 0.865343
The kappa coefficient = 0.870070
Brennan and Prediger's kappa coefficient = 0.907821

This is the confusion matrix:

262113	642	332	2	0
176	91922	4	81	0
359	861	32393	40	0
220	1430	344	17724	0
7732	0	0	825	976

The simple accuracy = 0.968798
The weighted accuracy = 0.973277
The kappa coefficient = 0.941609
Brennan and Prediger's kappa coefficient = 0.960997

This is the confusion matrix:

262108	644	332	3	2
203	85722	1605	4211	442
739	4936	27740	238	0
2792	5405	0	8476	3045
2545	1	0	59	6928

The simple accuracy = 0.934951
The weighted accuracy = 0.823091
The kappa coefficient = 0.878977
Brennan and Prediger's kappa coefficient = 0.918689

Task 3 Confusion Matrices

This is the confusion matrix:

405060	0	0	0	0	0
0	5572	0	0	0	0
1	104	3515	2	7	14
12	0	8	2010	0	5
0	2	0	9	1245	1
0	0	6	134	3	466

The simple accuracy = 0.999263
The weighted accuracy = 0.976821
The kappa coefficient = 0.988009
Brennan and Prediger's kappa coefficient = 0.999116

This is the confusion matrix:

405060	0	0	0	0	0
0	5521	51	0	0	0
8	0	3602	33	0	0
3	0	29	1997	0	6
0	0	46	9	1074	128
0	0	2	9	0	598

The simple accuracy = 0.999225
The weighted accuracy = 0.959616
The kappa coefficient = 0.987388
Brennan and Prediger's kappa coefficient = 0.999070

This is the confusion matrix:

405060	0	0	0	0	0
0	5572	0	0	0	0
363	0	2408	1	573	298
85	0	1	1915	0	34
62	0	8	0	1182	5
69	0	15	0	9	516

The simple accuracy = 0.996358
The weighted accuracy = 0.877196
The kappa coefficient = 0.939439
Brennan and Prediger's kappa coefficient = 0.995630

This is the confusion matrix:

450664	0	0	0	0
0	8395	0	714	6
22	384	4303	6	6
45	145	0	2671	2
121	3	0	0	6389

The simple accuracy = 0.996932
The weighted accuracy = 0.945095
The kappa coefficient = 0.967534
Brennan and Prediger's kappa coefficient = 0.996165

This is the confusion matrix:

450664	0	0	0	0
2	8890	0	223	0
95	3	4623	0	0
122	19	0	2722	0
0	0	0	0	6513

The simple accuracy = 0.999021
The weighted accuracy = 0.984265
The kappa coefficient = 0.989632
Brennan and Prediger's kappa coefficient = 0.998776

This is the confusion matrix:

450664	0	0	0	0
425	7667	6	1017	0
64	37	4598	21	1
19	33	0	2811	0
99	32	0	0	6382

The simple accuracy = 0.996299
The weighted accuracy = 0.942877
The kappa coefficient = 0.960499
Brennan and Prediger's kappa coefficient = 0.995373

This is the confusion matrix:

255153	0	0	0
0	4090	181	42
0	12	1791	8
0	28	14	825

The simple accuracy = 0.998913
The weighted accuracy = 0.958746
The kappa coefficient = 0.979214
Brennan and Prediger's kappa coefficient = 0.998550

This is the confusion matrix:

255153	0	0	0
12	4257	16	28
0	150	1661	0
14	19	3	831

The simple accuracy = 0.999077
The weighted accuracy = 0.979452
The kappa coefficient = 0.982314
Brennan and Prediger's kappa coefficient = 0.998769

```
This is the confusion matrix:
255153  0       0       0
86      4184    41      2
52      59      1700    0
49      20      15      783
The simple accuracy = 0.998764
The weighted accuracy = 0.986574
The kappa coefficient = 0.976051
Brennan and Prediger's kappa coefficient = 0.998352
```

This is the confusion matrix:

237762	0	0	0	0	0
0	6328	0	0	0	0
0	0	3460	8	85	1
0	0	6	1437	93	0
5	0	45	0	490	9
1	0	8	0	0	262

The simple accuracy = 0.998956
The weighted accuracy = 0.945740
The kappa coefficient = 0.988964
Brennan and Prediger's kappa coefficient = 0.998747

This is the confusion matrix:

237762	0	0	0	0	0
0	6328	0	0	0	0
1	0	3541	1	11	0
2	0	14	1367	153	0
2	0	27	0	509	11
0	0	0	0	0	271

The simple accuracy = 0.999112
The weighted accuracy = 0.950852
The kappa coefficient = 0.990614
Brennan and Prediger's kappa coefficient = 0.998934

This is the confusion matrix:

237762	0	0	0	0	0
0	6328	0	0	0	0
97	0	3366	13	65	13
60	0	4	1401	71	0
86	0	85	7	369	2
1	0	2	0	0	268

The simple accuracy = 0.997976
The weighted accuracy = 0.939378
The kappa coefficient = 0.978400
Brennan and Prediger's kappa coefficient = 0.997571

Plots of classification accuracy metrics

Comparison of accuracy measurements

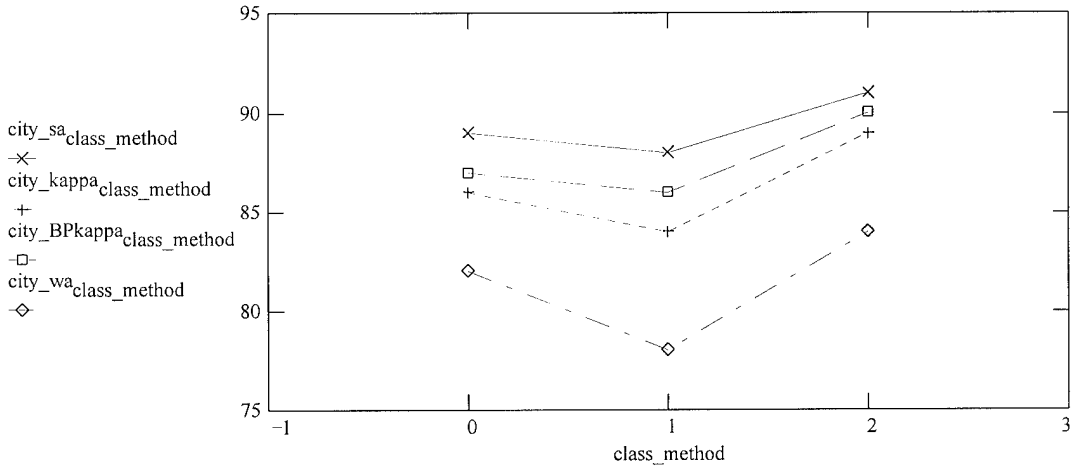
0 = GML 1 = nPDF 2 = ARTMAP class_method := 0, 1 .. 2

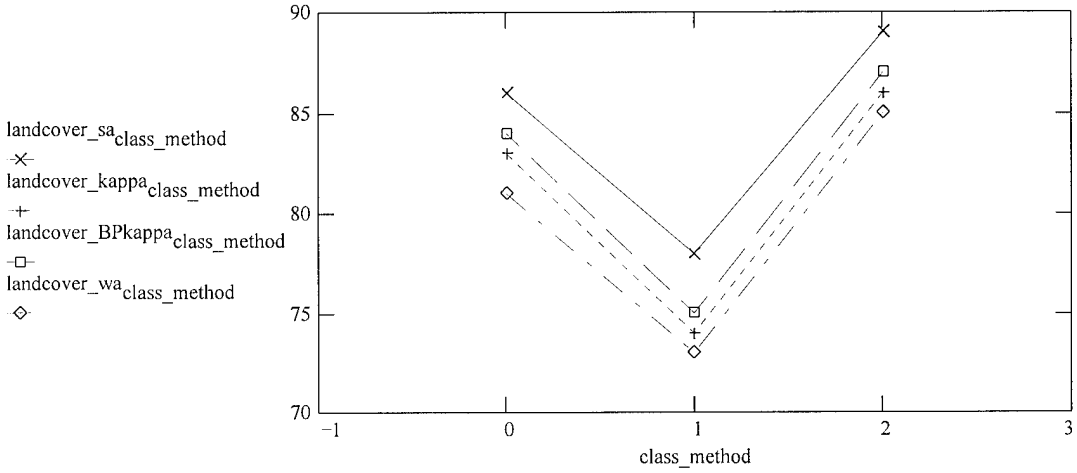
$$\text{city\_sa} := \begin{pmatrix} 89 \\ 88 \\ 91 \end{pmatrix} \quad \text{landcover\_sa} := \begin{pmatrix} 86 \\ 78 \\ 89 \end{pmatrix} \quad \text{rochester\_sa} := \begin{pmatrix} 86 \\ 70 \\ 90 \end{pmatrix} \quad \text{seashore\_sa} := \begin{pmatrix} 88 \\ 86 \\ 93 \end{pmatrix}$$

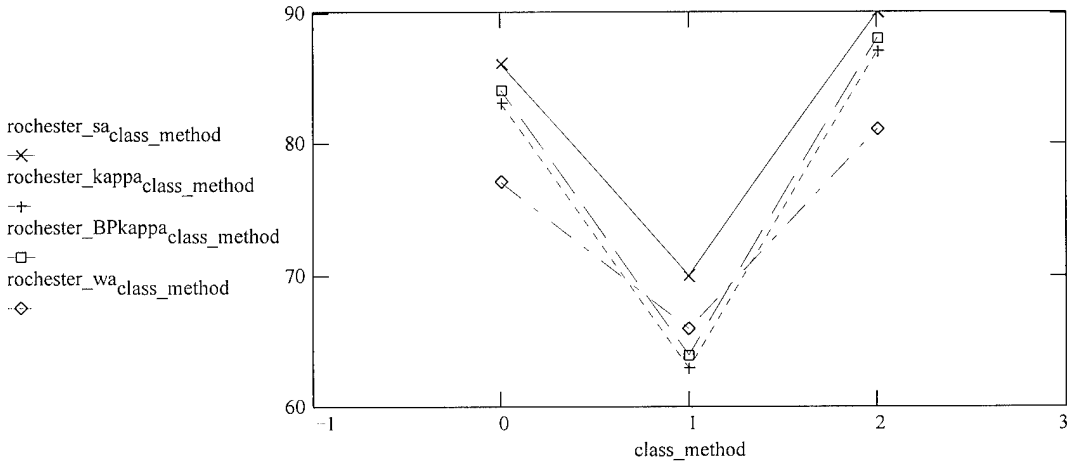
$$\text{city\_kappa} := \begin{pmatrix} 86 \\ 84 \\ 89 \end{pmatrix} \quad \text{landcover\_kappa} := \begin{pmatrix} 83 \\ 74 \\ 86 \end{pmatrix} \quad \text{rochester\_kappa} := \begin{pmatrix} 83 \\ 63 \\ 87 \end{pmatrix} \quad \text{seashore\_kappa} := \begin{pmatrix} 83 \\ 80 \\ 90 \end{pmatrix}$$

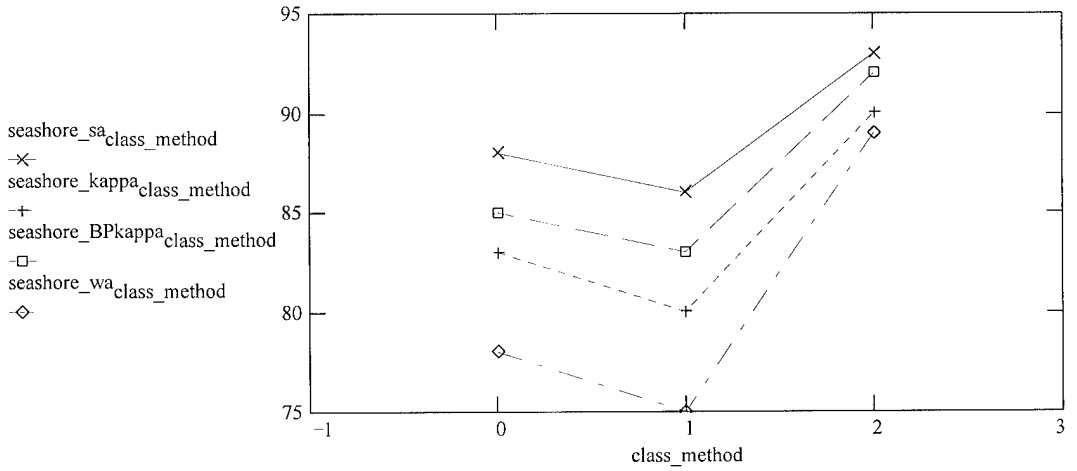
$$\text{city\_BPkappa} := \begin{pmatrix} 87 \\ 86 \\ 90 \end{pmatrix} \quad \text{landcover\_BPkappa} := \begin{pmatrix} 84 \\ 75 \\ 87 \end{pmatrix} \quad \text{rochester\_BPkappa} := \begin{pmatrix} 84 \\ 64 \\ 88 \end{pmatrix} \quad \text{seashore\_BPkappa} := \begin{pmatrix} 85 \\ 83 \\ 92 \end{pmatrix}$$

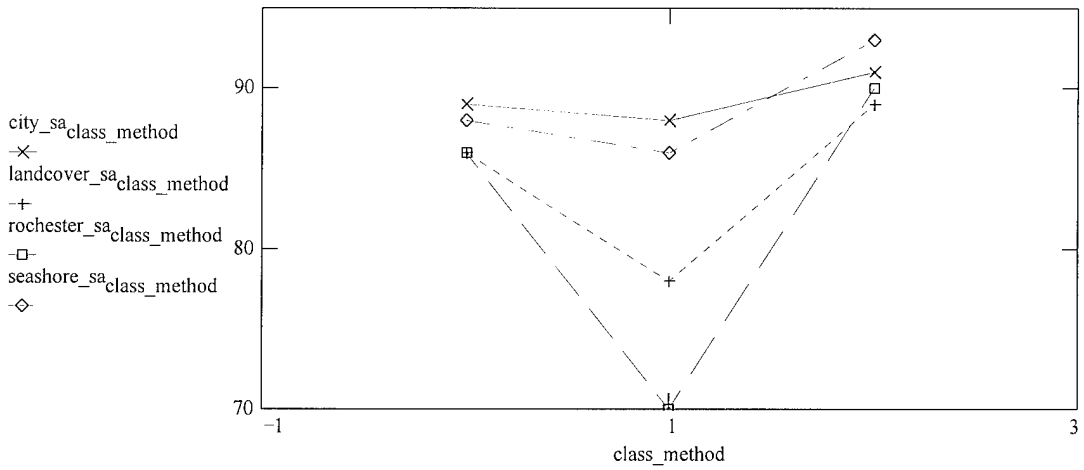
$$\text{city\_wa} := \begin{pmatrix} 82 \\ 78 \\ 84 \end{pmatrix} \quad \text{landcover\_wa} := \begin{pmatrix} 81 \\ 73 \\ 85 \end{pmatrix} \quad \text{rochester\_wa} := \begin{pmatrix} 77 \\ 66 \\ 81 \end{pmatrix} \quad \text{seashore\_wa} := \begin{pmatrix} 78 \\ 75 \\ 89 \end{pmatrix}$$

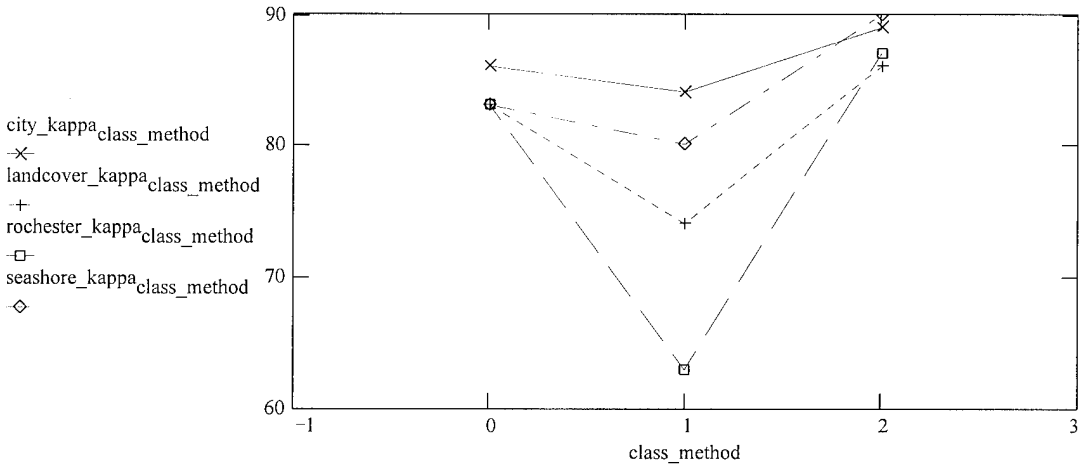


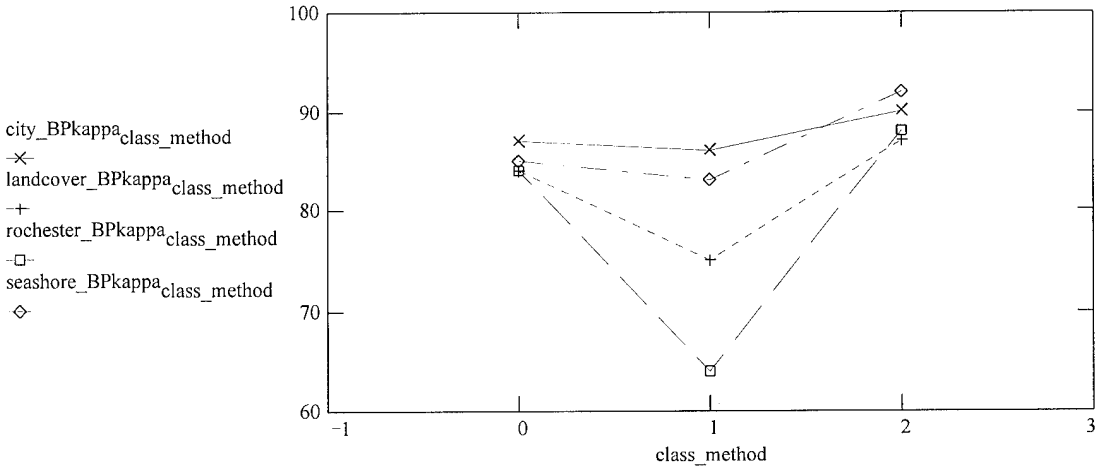

landcover_sa class_method
landcover_kappa class_method
landcover_BPkappa class_method
landcover_wa class_method
90
85
80
75
70
−1
0
1
2
3
class_method

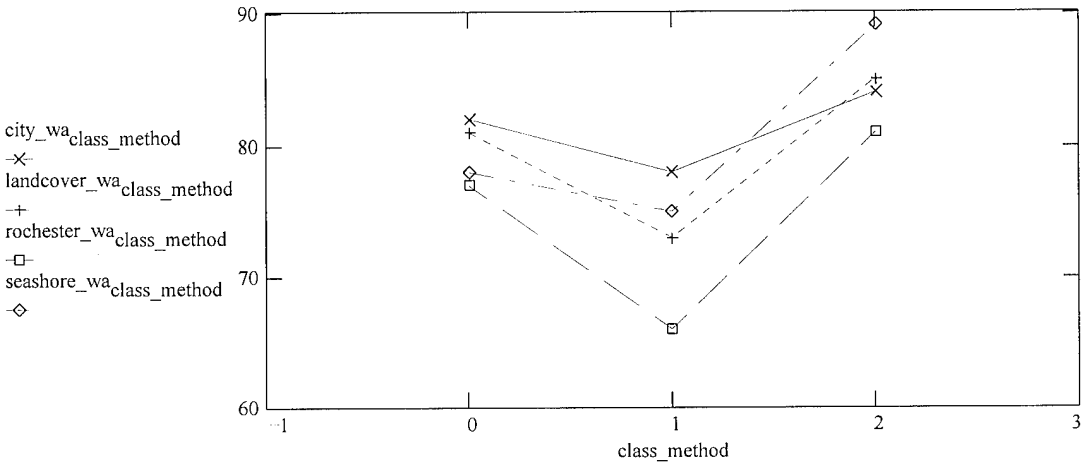

rochester_sa class_method
rochester_kappa class_method
rochester_BPkappa class_method
rochester_wa class_method
90
80
70
60
−1
0
1
2
3
class_method


seashore_sa class_method
seashore_kappa class_method
seashore_BPkappa class_method
seashore_wa class_method
95
90
85
80
75
−1
0
1
2
3
class_method


city_sa class_method
landcover_sa class_method
rochester_sa class_method
seashore_sa class_method
90
80
70
−1
1
3
class_method


city_kappa class_method
landcover_kappa class_method
rochester_kappa class_method
seashore_kappa class_method
90
80
70
60
−1
0
1
2
3
class_method


city_BPkappa class_method
landcover_BPkappa class_method
rochester_BPkappa class_method
seashore_BPkappa class_method
100
90
80
70
60
−1
0
1
2
3
class_method


city_wa class_method
landcover_wa class_method
rochester_wa class_method
seashore_wa class_method
90
80
70
60
−1
0
1
2
3
class_method